Principles of Computer Architecture

Philip Machanick

Principles of Computer Architecture
First edition (incomplete – do not cite as a final work), 2022

Picture credits: all illustrations are either by the author or from public domain sources, as acknowledged in the text.

Author:	Machanick, Philip, 1957-
Title:	Principles of Computer Architecture / Philip Machanick
Edition:	1st ed.
Publisher:	Grahamstown, South Africa : RAMpage Research, 2022.
ISBN:	XXX-X-XXXXXXX-X-X (pbk.)
LoC classification :	QA76

Last typeset 28 July 2022

Contents

List of Figures

List of Tables

1 Introduction

> *Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer* – Thomas Watson Jr., IBM CEO, August 1963
>
> *It seems like Mr. Watson has answered his own question* – Seymour Cray.

COMPUTER ARCHICTURE is a rapidly moving field yet a few things have held good over the last three decades. Before the RISC movement of the late 1970s, much computer architecture was based on gut feel, or poor communication between the hardware and software sides of design teams. For example, the hardware people may decide that an aspect of setting up or undoing up the call stack involves tedious repetition and, to be nice to the compiler people, they roll this all into one instruction. Only, because they didn't correctly understand the requirements of the compiler people, the instruction they design isn't useful and is never used. Then when they try to implement a more aggressive version of the design, they find all the complicated instructions they weren't asked to design by the software people make it hard to produce a more aggressive design. Related to this communication issue is a lack of standards for quantifying improvements. In the 1970s, Gene Amdahl, a former IBM engineer who split with IBM to form his own high performance computing company, formulated a speedup limitation [Amdahl 1967] that became known as Amdahl's Law. In essence this says that to calculate the effect of a speed improvement, you need to take into account the entire run time *including parts that are not sped up*. An

apparently obvious revelation, it's a point often forgotten when extolling some brilliant enhancement (not only to computer systems).

What really created an impetus for improved standards in quantifying performance was a dispute that arose between two schools of computer architecture, the high-level-language oriented approach, and the simplicity-oriented approach. The latter gained credibility as early as the 1960s, when Seymour Cray, at that time working for a relatively small company called Control Data, produced a design that was much faster than the best the market leader IBM could produce – see opening quotes.

However, some designers still argued that a machine instruction set closer to high level languages was more efficient because even though each instruction may be slower than with a design closer to the hardware, you needed fewer instructions. This argument carried some weight when memories were relatively small, and hardware complexity could be replaced by *microcode*, a very low-level instruction set that interpreted the actual machine instructions. Microcode was stored in a ROM (read-only memory) that was many times faster than DRAM (dynamic random access memory, what main memories are made of), so frequent accesses to *microstore* as the cost of fewer instruction fetches was a reasonable trade-off. However, as DRAM speeds caught up with ROM speeds and it became viable to implement caches in fast SRAM (static RAM – more expensive than DRAM because it is lower density), the case was less clear. It took a landmark paper in 1980 [Patterson and Ditzel 1980] to fire up a new movement inspired by Seymour Cray's 1960s designs [Thornton 1963; Pagelkopf et al. 1975] to push the case for simpler instruction sets, and that push led to a more quantitative approach to architecture design and evaluation. In particular, to win the case for simplified instruction sets, the RISC (reduced instruction set computer) movement began a move to more scientific principles in measuring alternative designs, with an emphasis on *repeatable* experiments that were representative of *real workloads*.

Why do we still need to understand all this? A few years back someone asked me what there is still do do in this field because Intel won. With the massive growth in mobile devices using the ARM processors, this claim is not so obvious anymore; since I first wrote this, Apple has shifted to ARM for their Mac line up. Also, a very large number of computer parts is sold as part of another machine – an embedded system. Most embedded CPUs are not Intel parts and ARM, while a big player in this market, does not have the dominance they have in the mobile market. There are also other niches such as graphics processing units (GPUs) that have different design issues. Even if the instruction set design is fixed, implementation

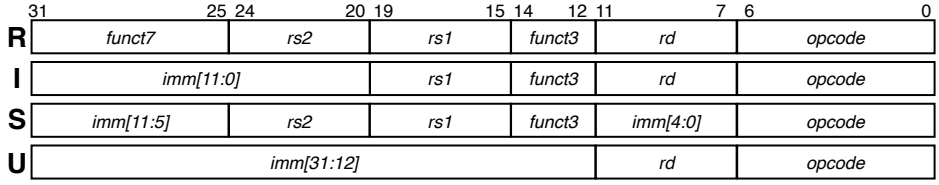


Figure 1.1: RISC-V base formats. *There are other variants but these cover much of the base 32-bit instruction set.*

techniques are wide open to research and understanding design principles aids in choosing between alternative available designs.

In this book we learn about tools and techniques for measuring performance, how architectures are designed and what factors are useful to consider when comparing performance of alternative designs. I relate these issues to a range of different areas of architecture design: memory hierarchy, instruction set design, input and output, and parallelism in various forms.

To provide a focus, I base the discussion on design principles of RISC-V, a free design based on experience of earlier RISC architectures [Asanović and Patterson 2014]. A free design is suitable for research without concern about licensing, and RISC-V is a simplification of previous designs, with a simple minimal 32-bit integer subset that can be used to illustrate basic principles, with optional add-ons that can illustrate any degree of complexity that is reasonable to cover. RISC-V has another significant benefit: the instruction set design is documented, providing reasons for design decisions. I review these decisions as a way of understanding how to understand design choices.

I use MIPS as a basis for comparison, since it is one of the earlier, simpler RISC designs and is easy to use in courses, thanks to the simple SPIM emulator [Larus 1990] – considerably updated since it was an undergraduate project and in wide use.

Figure 1.1 illustrates the way a RISC instruction set has relatively simple variations on instruction types.

1.1 Measurement

Computer architecture measurement falls into two broad categories: evaluating existing designs and implementations, and evaluating design alternatives. In the first category, we can run standard benchmarks (software that represents a workload of interest) and we can also use simulations so as to produce repeatable

run times. In the second category, we mainly rely on simulations because it is too expensive (even using reconfigurable hardware, such as FPGAs) to create multiple real variations in a design to check the effect of changes in design parameters.

Whichever approach we use, we try to adhere to a few essential principles of the scientific method:

- *repeatable* – running the same experiment twice should give the same result and we should report enough detail so others can redo the experiment
- *separation of variables* – where more than one factor can influence performance in a way that cannot be separated out, only vary one such variable at a time
- *representativity* – the experiment should represent something real to those interested in the evaluation, not an artificial exercise that will not rank alternative designs the same way as would real usage

While these principles seem obvious, the quantitative approach was novel enough at the time that two senior academics, David Patterson at the University of California, Berkeley and John Hennessy at Stanford, felt the need to codify the principles in an academic text in 1990 [Hennessy and Patterson 1990] that has subsequently entered its sixth edition [Hennessy and Patterson 2017], and these principles are now routinely observed in mainstream computer architecture research[1], which was not the case when I first became interested in the field in 1980.

The fact that we now have well-established scientific principles of architecture measurement doesn't mean that the field is devoid of creativity or innovation. However, that innovation now has to be based on reasonably sound principles. Even so, some large mistakes are still possible, for example, the attempt by Intel to break out of their IA32 architecture with the IA64 (Itanium) design, which failed to achieve its performance goals or wide market acceptance.

A scientifically sound basis for measurement only allows us to be accurate about making comparisons: it does not remove the requirement of thinking up innovations, because someone has to derive the new ideas to compare with old.

1.2 Design Approaches

Given all that, how do we arrive at innovations?

[1] Well, mostly – I see some research on general-purpose programming on GPUs that has fallen back to the bad old practice of evaluation using code that is mostly one algorithm.

Much of the early computer architecture work up to the 1970s set the scene for widely accepted design alternatives today. A fair fraction of innovation today involves rediscovering old ideas that worked well in a different form factor, and finding that technology today makes those ideas work well once more. Much of Seymour Cray's work in the 1960s was in essence reinvented by RISC designers in the 1970s through to around 2000, as it progressively became possible to fit more of the features of his multichip designs onto a single-chip CPU. Remarkably, very little in modern designs wasn't found in his landmark CDC 6600 of 1962, including hardware to support multiple instructions per clock cycle and out-of-order execution. By the end of the decade, the successor to the 6600, the CDC 7600, had a pipeline as well [Pagelkopf et al. 1975].

A good starting point for looking our for potential for innovation is to examine various trend lines and work out when new design trade-offs become possible. Possibly the most famous of these trends is *Moore's Law*, an observation that the number of transistors at a given price doubles about every 2 years [Moore 1965]. There are others, like the quartering of the cost of DRAM every 3 years, and the much slower speed improvement of DRAM. Understanding how long these trends can persist and when a change in technology is predictable opens up opportunities for architecture research. For example, in the 1990s, I observed that the speed gap between DRAM and CPUs was heading for similar numbers as measured by lost instruction execution opportunity to the speed gap between CPUs and paging devices when virtual memory was first invented. That led to the *RAMpage* project, of which I cite a fraction of the research outputs here [Machanick et al. 1998; Machanick 2000; Machanick and Salverda 1998; Machanick 2004].

In another breakthrough, which led to a major change in the whole industry, a Nigerian academic at Stanford University Kunle Olukotun [Olukotun et al. 1996] made a case for replacing very aggressive single-CPU designs by what are now known as multicore designs. In essence his argument (backed up by design studies and simulations) is that the potential for speedup of a single core design is limited by how much instruction-level parallelism is available, whereas a multicore design can gain speed from several dimensions. A clever compiler can convert instruction-level parallelism into threads, code already designed to run threads or multiple processes can speed up, and multiprogramming workloads (as on a typical operating system where there may be dozens of processes, many not visible to the non-technical user) can also see a speed gain. Multicore designs have in recent years also gained in utility because they create more options for scaling both performance and energy use. The more complex a design is (and this

also applies to miniaturization) the more energy is wasted to *leakage*: transistors leak current even when they are turned off, and this becomes an increasing factor as components scale down and circuits consequently become more complex [Kim et al. 2003].

An example of the kind of design trade-offs possible is ARM multicore designs that have CPUs of varying complexity and hence speed and energy use. In a "big-little" design (officially "big.LITTLE"[2]), when high speed is required e.g. with user interaction that requires rapid responses, the process or thread providing that response can run on a "big" core, while the "little" cores can be used for lower-priority tasks. When no high priority task is active, "big" cores can be put to sleep. In a workload that aims to maximize throughput, deciding how to split work across heterogeneous cores is an interesting challenge [Wang et al. 2019].

Yet another approach to looking for breakthroughs in architecture is studying roadmaps of predicted future technology[3]. In one example, Trever Mudge at the University of Michigan picked up the likelihood that vertical stacking of dies (a die is a chip without the packaging) was on the horizon, and he explored the implications of this technology for making a package tightly integrating DRAM and a multicore CPU design. The resulting design has a number of advantages. Because through-chip *vias* (conductors) can be as fast as within-chip communications and wide buses are practical to construct in this form, the CPU-DRAM speed gap can be considerably reduced. Since the CPU wastes less time waiting for DRAM, a given level of performance can be achieved with a slower clock, reducing he problem of heat dissipation out of a compact package. The resulting PicoServer design [Kgil et al. 2006] and its successor Centip3De [Fick et al. 2012] may at some stage emerge as a commercial product; even if it does not, it is a good illustration of looking out for technologies that may later become viable. A design compromise is to do a multilayer design mostly consisting of DRAM but with a logic layer to speed access; this is how HMC (Hybrid Memory Cube) RAM is designed [Courtland 2014].

A more radical take on going 3-dimensional is 3D Xpoint (pronounced "cross-point") RAM, a kind of nonvolatile RAM designed to be a lot faster and more durable than flash. The earliest product scheduled for launch is Intel's Optane drive [Bourzac 2017]. Details are sketchy but patents are a pointer to some details indicating that the internal structure is truly 3-dimensional rather than layers of

[2] `https://www.arm.com/technologies/big-little`

[3] International Technology Roadmap for Semiconductors `http://www.itrs2.net` is a good example.

2-dimensional slices [Reinberg and Zahorik 2004; Lowrey 2002].

1.3 Performance Factors

When considering performance, we need to take into account several axes. Depending on the target application or market, different axes may be more important. The most significant ones are:

- *cost* – not only of one component such as the CPU, but overall packaging and environment costs
- *speed* – again, not only one component contributes to speed (remember Amdahl?)
- *energy* – in some applications like mobile computing, energy is a first-class concern but even in large-scale computing, energy is a limiting factor
- *scalability* – a design that works at many scales means early expensive versions can be sold into high-margin markets like high-end servers, while older designs can be sold into high-volume markets to maximise amortisation of costs
- *longevity* – one of the most classic errors of hardware designers is to fail to take into account the rate at which technology improves: too small an address space is one of the most common reasons once-successful architectures have had to be abandoned

Cutting across these axes are two aspects of performance that can be in conflict:

- *latency* – time to complete a *specific* operation or service
- *throughput* – average rate of work completion

Low latency is what the user desires; high throughput is what the accountants want. Low latency means you have a responsive system, but that responsiveness can be bought at the expense of lowering throughput, by ensuring that the system is not busy when you want a response.

In this book I examine case studies of performance covering a range of these axes, using RISC-V as an example and also as a contrast to other designs, as noted earlier.

1.4 Architecture Areas

Computer architecture is broadly speaking design principles of any area of the computer system including the hardware and any area where hardware and

software interface. Architecture is usually narrowed to mean aspects of design that do not change the programming model. It's convenient to divide architecture into different areas, though these necessarily interact. For example, the memory hierarchy includes components that use the IO (input-output) system, and efficient implementation of IO requires design with the memory hierarchy in mind. So, as I divide the architecture world for clarity, remember that the division is not absolute.

1.4.1 Memory Hierarchy

The need for a memory hierarchy arises from the fact that memory components fast enough to keep up with the CPU are many times more expensive than slower memories. Fortunately, the *principle of locality* helps here: a program uses a relatively small part of its address space at a time. Locality is generally divided into two types:

- *temporal locality* – a memory location that is referenced is likely to be referenced again soon
- *spatial* – a memory location near a location that is referenced is likely to be referenced soon

The definitions of "soon" and "near" depend on how big the speed gap is between layers. If the speed gap is big, we stretch the definitions out to longer in time and space, because we can less afford the penalty of accessing slower memory.

In an operating systems design, we focus on locality as it applies to virtual memory; here we also consider hardware layers of the memory system, including caches, the TLB (translation lookaside buffer: a small cache of recent page translations) and registers.

1.4.2 Hardware Layers

It is useful to divide computer hardware into logical layers. As seen by the user (or, in today's world, the compiler and related tools like the linker), there is the machine code layer. This layer cannot change much in basic functionality without losing the user base. If you have to recompile or relink your code to run on a new generation of a particular vendor's design, that takes away a reason to stay loyal to that vendor. The *instruction set architecture* or *ISA* is such an important part of a design's identity and its ability to retain a user base that the ISA is often referred to as the architecture (the IA32 architecture, the ARM architecture, etc.).

The ISA is not only characterised by a set of instructions but also by the

available machine registers, the memory bus size and instruction modes such as supervisor and user mode that implement protection. The idea of an ISA originated with the IBM 360 series of the 1960s [Amdahl et al. 1964], which was the first to feature a whole family of designs launched at once that could run the same programs, subject only to resource limits not differences in the type of code that could execute.

The ISA can be implemented many different ways and remains the same ISA as long as the same programs can run (give or take constraints like memory size and available peripherals).

Some details that can vary include the pipeline, extra copies of the registers to support implementation details like hardware multithreading support and out-of-order execution, branch prediction and variations in the memory hierarchy. All these variations are below the level of the ISA because they are hidden (other than performance impacts) from user-level code. This lower layer is the *microarchitecture*.

Below the micoarchitecture is fabrication technology (fab tech): the way a chip (more correctly, a die) is actually made. Fab tech governs how many features there are per unit area, clock speed, heat and power consumption. A die is so named because chips are made in a big circular sheet called a "wafer" that is "diced" into dies, each of which is packaged as a chip into a container with external connections. Each new generation of fab tech makes it possible to make the same functionality cheaper as a smaller die is less expensive or to make a more complex design in the same area. Smaller is cheaper for two reasons. More dies off the same wafer means a lower average cost. Also, more dies means a lower fraction is lost to defects, assuming the average rate of defects is the same.

One area that is not obvious to the user (even a compiler writer) that is hard to change in practice is hardware support for virtual memory (VM). If this changes, unless the old approach is maintained for backward compatibility, every operating system using the new design will need to be modified, since hardware support for VM is tightly integrated into the software side of VM implementation.

1.4.3 Hardware-Software Layers

The operating system provides a layer of abstraction that hides the bare metal from the user, and some parts of the system architecture may involve hardware and software components. The most obvious of these is the virtual memory system that cannot be implemented effectively without hardware support (otherwise,

every memory reference would take several times as long as without VM, since it must be looked up and translated, as well as checked for validity).

There are other aspects of the system where hardware and software play a role. In some earlier microprocesor designs including some RISC designs and some implementations of the Intel IA32, significant speed gains could be had from reordering instructions. Such reordering required recompilation in most cases, and was seldom done for the very good reason that the next generation had a different optimal ordering of instructions.

In yet another area, IO involves hardware-software cooperation. IO is very slow compared with the CPU and RAM and that speed gap has to be hidden. An operating system typically schedules IO-bound processes with higher priority than CPU-bound processes for two reasons. If CPU-bound processes run to completion while there are still IO-bound processes in the system, there is no work to be done while waiting for IO. Secondly, if IO-bound processes are able to use the CPU, it's best to give them more time than CPU-bound processes so they can make progress. An operating system has a range of strategies to hide the latency of IO in addition to scheduling policy. Here is a quick summary:

- *scheduling* – IO-bound processes have higher priority than CPU-bound processes
- *buffering* – slightly different effects for input and output:
  - *input* – read more than absolutely needed, relying on spatial locality not to waste the extra IO because it's usually more efficient to transfer in large blocks
  - *output* – don't wait for writes to complete: dump output to memory and let the device empty the buffer in its own time
- *cacheing* – keep data (or code) in a faster layer of memory as long as possible; buffering can be a form of caching if the contents are available for repeated use
- *spooling* – for devices that have to accept a job to completion, spooling (simultaneous peripheral operations online) is a specialist kind of buffering that stores the data until it's that job's turn (most often used for printing)
- *specialist IO hardware* – in some systems IO is hived off to a separate specialist CPU relieving the main CPU of the detail of IO

These represent some of the strategies used by an operating system; we do not cover much detail here. If you want to know more, you need an OS text.

1.4.4 Hardware-Software Interaction

Given overlaps in hardware and software, how important is it for software to be aware of hardware, and vice-versa? In addition to the issues raised above of communication between parts of the design team, users of a design can benefit from knowing how their software interacts with hardware.

Some areas where this knowledge can apply include:

- *memory-sensitive algorithms design* – understanding of how the cache and VM subsystems work can have a large effect on performance [Lam et al. 1991; Machanick 1996; Xiao et al. 2000; Rahman and Raman 2000]. *But* it is important to reevaluate design trade-offs against current systems as caches are much larger than they were when early studies were conducted.
- *balancing VM use and IO* – in an experiment, I ran quicksort on randomly generated data varying the size until I ran out of RAM and page faults occurred. I rewrote the code so it sorted a section at a time, storing most of the data on disk, using mergesort to merge only as much as would fit into RAM at one time. This ran a lot faster than relying on VM. No big surprise. But what was a bit surprising was that it was not significantly slower than quicksort on data that did fit into RAM.
- *efficient use of shared memory* – with multithreaded code or processes with shared memory, a clear understanding of cacheing makes a big difference to performance [Machanick 1996]; since the 1990s when multiprocessor systems were relatively expensive, some hardware-software interaction concerns have found their way to the mass market because of the proliferation of multicore designs
- *role of VM hardware support* – understanding how VM is supported in hardware can also make a big difference to performance [Machanick 1996]

In this book I explore some of the issues.

1.4.5 Instruction Set Design

Instruction set design used to be a core area of computer architecture. It is less so now that it's clear that RISC is fundamentally a good idea, but the Intel IA32 architecture isn't going to go away despite this. The fact that ARM dominates the mobile space despite breaking a fair number of the rules of a clean RISC design also is a hint that getting the design right is not a guarantee of success. That does not however mean we should not understand the principles of good design so that future designs can draw on past lessons.

Styles of Instruction Set

Prior to the RISC (reduced instruction set computer) movement, there were two major schools of design:

- *short-term design goals* – base design trade-offs on obvious requirements like fitting into a small memory, for example, make common instructions shorter than less common ones to reduce memory footprint at the expense of making instruction fetch and decoding harder
- *high-level language oriented* or *HLL* – design the instruction set to be easier for compiler writers to generate code

In the first category, we have some of the most enduring designs. The Intel IA32 developed out of a processor with a 16-bit address space, the Intel 8086, which was upgraded to a 32-bit address space with the 80386 in 1985 and now includes 64-bit implementations. The IA32 has endured because it was adopted for IBM's PC design, which developed a massive market, and also because Intel was able not only to throw massive resources at improving its performance against the odds, but had very skilled engineers working around the inherent flaws in the design. The IBM 360 architecture [Amdahl et al. 1964; Gifford and Spector 1987] is another that endured for decades despite clear flaws (in terms of subsequent knowledge on how to design for performance). The 360 endured because IBM was one of the first computer companies to sell on service rather than purely on technology, and because the design had a few key things right: it was designed for 32-bit addressing ahead of much of the competition, and had an adequate number of registers, a critical feature for achieving high performance. IBM, like Intel, had very skilled engineers able to work around inherent flaws in the design.

Intel and IBM made design choices that may have seemed optimal at the time. The Intel 8086 was designed at a time when PCs had a few thousand bytes of RAM, so decisions that today make it hard to scale up performance made sense at the time. One of the hardest problems in architecture is balancing forward thinking with immediate design trade-offs applicable to the current generation of hardware.

In the second category, one of the more successful examples is the Burroughs B5000 architecture [Mayer 1982], which used a stack-based instruction set and had hardware support for arrays including bounds checking (a hardware data structure called a *descriptor* stored details of each dimension of the array). Memory was *tagged* with extra bits representing the type of contents of a machine word, further supporting error checking. Since the hardware could determine the

type from the tag bits, there was only one instruction for each basic operation like addition (not a separate instruction for floating point, integer and various precision alternatives). The instruction set was very compact, since stack instructions do not need register names let alone memory addresses except to move data onto or off the stack and, in an exception to common practice, the hardware and software teams worked in close collaboration. Unusually for its time, the system software was written in a high-level language (a version of Algol 60, a language with some following in academia), and the operating system was distributed as source code. The Burroughs machines were not particularly fast if you measured the run of a single program but had a very efficient VM system and, with real workloads, could outperform machines that were a lot faster on paper. Unfortunately, Burroughs designed their array support assuming the Algol 60 approach of storing multidimensional arrays in row major order, whereas the scientific community mostly chose to use FORTRAN, which requires arrays to be stored in column major order, causing significant complications in generating efficient FORTRAN code.

The Burroughs example illustrates one of the hazards of HLL-oriented design: high-level languages differ enough that it's hard to do a design that's good for one without serious compromises for implementing other languages.

One of the less successful examples of HLL designs is the Intel 432 [Organick 1983]. The 432 had very fine-grained protection, supposedly to support object-oriented coding, but had very poor performance [Colwell et al. 1988], and didn't ever gain significant market share.

The IA432 illustrates another hazard of HLL-oriented design: it can result in poor performance, especially when insufficient attention is paid to any of practicalities of hardware implementation and usability of features in compilers.

More recently, hardware support for executing Java bytecode has emerged. However, just in time (JIT) compilers reduce the advantage of a Java machine. One implementation of partial hardware support for Java targets small devices with real-time requirements [Schoeberl 2008]. Some niches may justify specialist designs though on the whole it's easier to use a language that's a better fit to the problem than to design hardware to work around limitations of a language. For example, garbage collection makes for unpredictable execution times, a problem for real time – to fix this, rather use a language without garbage collection or fix the garbage collector than design special hardware.

By contrast, the RISC movement specifies a very simple regular approach to instruction set design:

- *fixed instruction length* – all instructions are the same length, making it easy to fetch and decode multiple instructions in parallel[4]
- *load-store architecture* – all memory references are loads (copy memory contents to a register) or stores (copy contents of a register to memory); arithmetic and logic unit (ALU) operations are always on registers
- *standard operands* – ALU operations always operate on 1 or 2 source operands (immediates or registers) and one destination register
- *bounded execution time* – with the exception of excursions down the memory hierarchy (limited by design to instruction fetches and data movements only in loads and stores), instructions have clearly defined execution time
- *simple control* – the number and type of control transfer instructions is limited (usually unconditional jump, a jump and link that preserves a return address and a few conditional branches)
- *large general-purpose register file* – a small number of registers, registers with specific purposes or setting logic conditions in condition codes makes it harder for compiler writers to generate code, and harder for hardware to reorder instructions

Some RISC designs compromise on details, e.g., a number do use condition codes. Nonetheless RISC architectures are generally very similar, unlike other classes of ISA design that differ widely.

We see next why the RISC movement claims advantages over the other approaches. At this point, note that even with the Burroughs design where the hardware and software teams did work in close collaboration, the fact that their pesky customers chose to use a different language for programming meant that much of their good work was wasted.

Design for Performance

The RISC movement is based on a few key observations of how performance is achieved. First, to make the overall system fast, you need to have the highest possible clock speed and rate of instruction flow through the system. The latter works best if you can implement an efficient pipeline. A pipeline is inherently inefficient if the stages are not all the same length (the longer stages will force the

[4]There are versions of RISC designs with *compressed* modes, in which some instructions are shorter. However, these variants are usually based on a fixed-length design, with shortened instructions in particular implementations designed for small-memory applications.

shorter ones to idle). To implement a fast clock speed, relatively short pipeline stages help. If there are many variations in type and size of instruction, these things become harder to achieve.

Second, an important principle is *make the common case fast*. This seems contrary to the lesson of Amdahl's Law that the best speed gains arise from making everything faster. However if you calculate speed improvement based on accurate performance measurement, you can quantify this effect. For example, having to run 50% more instructions as the price for doubling clock speed is probably a win (though you need a comprehensive measurement that takes into account other effects like changes in memory hierarchy use). By contrast, introducing a few special-case instructions that are rarely used but make it hard to scale up the clock speed is seldom a win. Again, quantifying makes the case, not gut feel or hand waving.

To take an example, RISC architectures generally implement a function or method call with several (general-purpose) instructions to set up parameters and local variable space, rather than using a special instruction to set up a stack frame and pass parameters. While this increases the instruction count, the absence of special instructions makes it easier to implement an aggressive pipeline. At the cost of of occasionally using more instructions, the overall system is faster. We can quantify this effect if we know how much faster the clock speed can be made, or how the pipeline can be improved in other ways by simplifying the instruction set design, and calculate the net gain. The need to do this sort of calculation to convince RISC sceptics provided impetus to the modern approach to quantitative design.

1.4.6 Input and Output

IO is an important part of systems design because most IO devices are so much slower than the rest of the system. A disk for example may take of the order of 10ms to do a seek (move the head to the right track). Flash is about 1000 times faster and DRAM access is about a million times faster. A 2GHz CPU, if executing only one instruction per clock, takes 0.5ns per instruction, which is 20-million times faster than disk seek time. If you have an aggressive pipeline executing several instructions per clock the speed gap is even greater (before you consider keeping up with multiple cores).

In this book I do not cover IO in detail; many of the performance issues are dealt with in operating systems design. At the hardware level we can consider

various modes of interfacing, the relationship between latency and throughput (or bandwidth in the IO context) and trends in device technology.

1.4.7 Parallelism

Every now and then when progress in a given approach to technology appears to be heading for a dead end, parallelism appears as the solution. What usually happens is a new approach to sequential programming appears, and all the complexities of parallel programming lose their attraction. Over time many models of parallelism have appeared, and some have proved enduring, while others keep resurfacing as packaging trade-offs change, and the reason they were abandoned is forgotten.

As long as Moore's Law was effectively delivering double the performance every 2 years or so, there was little benefit in writing parallel code for performance unless you could afford a large-scale system. Any system that achieved a speedup over a serial implementation (measured as $\frac{time_{serial}}{time_{parallel}}$) of less than 4 would be overtaken by faster hardware in a year or two, sometimes sooner than the time it took to achieve an efficient parallelisation.

Instruction-Level Parallelism

Before the multicore era, the most common form of parallelism was instruction-level parallelism (ILP), because it took no effort from the programmer. Provided the hardware can find more than one instruction ready to run at the same time, ILP provides speedup at the expense of hardware complexity, a trade-off increasingly justifiable as the number of transistors per chip at a given price point increases.

ILP however has some inherent limits. There's a limit to how much inherent parallelism that exists at instruction level because of dependencies between instructions [Wall 1991; Lam and Wilson 1992; Postiff et al. 1998], and there are limits to the extent to which practical architectures can find available parallelism (e.g., instructions with no dependency between them may be relatively far apart). A problem that has arisen more recently is that the increasing complexity required for more aggressive ILP has a high cost in energy use [Yeap 2002] and hence also heat.

Another limitation to pursuing performance using more and more aggressive ILP with higher and higher clock speeds is the growing gap between the speed of CPUs and DRAM, resulting in limited gains as a higher fraction of the CPU's

time is spent waiting for DRAM, a problem called the *memory wall*, predicted in 1995 [Wulf and McKee 1995].

Multiprocessor and Multicore

In the past multiprocessor architectures differed widely in characteristics. Some emphasised data parallelism (the same instructions on several or many different data items in registers or memory), others instruction parallelism (different instruction streams on each ALU). Memory organization also varied. A distributed-memory architecture had no shared memory and communication primitives like messaging were used. A shared-memory architecture had one global memory and required mechanisms to ensure consistency of caches. A distributed shared memory system [Bennet et al. 1990; Dwarkadas et al. 1993; Bordawekar 2000] is physically distributed but gives programmers as model that looks like shared memory. There are also programming tools and libraries like MPI that hide some of the detail of the memory model, but some understanding of the memory model is essential to achieve performance.

More recently, as limits of ILP and scaling up the clock speed (in part because of the memory wall but also because of limits to ILP and the increasing cost of hot high-energy consumption designs), multicore designs have become popular, and these generally are shared-memory designs, often with a shared lower-level cache. Writing parallel code is now an option for the mass market, so the specialist skills needed for programming big iron in the 1990s [Cheriton et al. 1991, 1993; Machanick 1996] now apply more widely, but the problems are no easier. A good understanding of how the memory system works in shared-memory multiprocessors is even more important to achieving good performance than with a uniprocessor.

GPUs

The conversion of specialist processors designed for graphics to general-purpose computing is not a new idea. The Intel i860, marketed as a general-purpose CPU with graphics support [Grimes et al. 1989] was clearly designed more with graphics support than general purpose use in mind. Among other things, it suffered very high latency for context switches, and VM support was minimal. On a page fault, it only reported that a fault occurred, not whether it's a read, write or instruction fetch, meaning the page fault handler has to reconstruct the cause, in effect interpreting the instruction that caused the fault to work out what happened

[Anderson et al. 1991]. The i860 was reasonably successful as a graphics processor in the days when a high-end graphics system was a multichip design, with features presaging vector extensions to the IA32 line. Overall the i860, despite being deployed on some large-scale supercomputer designs [Berrendorf et al. 1994], was not a great success as a high-performance CPU.

So is the general purpose GPU (CPGPU) concept reasonable, given that specialist processors have not historically been a win in general-purpose applications? There are arguments for and against. Against: Amdahl's Law tells us that a 100× speedup of a small section of our code will have a small overall effect on run time, and coding for these specialist processors, even with high-level tools like NVIDIA's Cuda [Wynters 2011][5] and OpenCL, is hard. On the for side, the massive market for GPUs means that there's a lot more critical mass behind this movement than other attempts at using specialist processors for more general purposes than originally intended. Computers with powerful GPUs are increasingly ubiquitous in the mass maket, making it likely that large-scale computation using such GPUs will continue into the future, whereas past specialist designs lacked that critical mass.

Warehouse-Scale Computing and the Cloud

One of the ongoing debates in computer architecture is whether large-scale computing is best achieved with massive numbers of inexpensive boxes with redundancy to mask failure, or dedicated highly-scalable designs. One of the earlier ideas of this type is *RAID*, originally *redundant array of inexpensive disks* [Patterson et al. 1988] (now usually "independent" instead if "inexpensive", possibly so manufacturers can claim their disks are "enterprise grade" and hence not inexpensive).

An example of the extension of this idea to a *redundant array of inexpensive computers* (no one uses RAIC as a name for some reason) is Google's approach or warehouses full of inexpensive computers, with many fail-safes to allow for hardware and software faults [Barroso et al. 2003]. This kind of infrastructure is becoming increasingly important as the Internet expands to ever new services including some that might for a brief period require hundreds or even thousands of servers, then settle back to more modest requirements [Liu and Wee 2009]. How best to put these services together is still a work in progress, and there is no doubt that expertise in this field will be useful for some time ahead.

[5] More at `http://developer.nvidia.com/nvidia-gpu-computing-documentation`.

The "cloud" term is somewhat vague in meaning, and is really marketing speak for distributed services, sometimes storage, sometimes computation, sometimes both. The key feature of a *distributed* system, as opposed to a *networked* system, is naming or location *transparency*, i.e., you don't know (or need to know) whether data or a process is running locally, over a network or even over several computers – that is a performance detail. By contrast, a networked service requires naming the location where the service occurs. What really distinguishes the cloud from earlier distributed services is that the infrastructure is provided on a closed proprietary system, rather than as a file system or operating system that installs on your own computer. While some such services like Google Drive or DropBox occupy file space on a computer on which you use the service, they do not integrate cleanly. Can you put a DrobBox folder into a Google Drive folder? If so, will this still work next week? Can you mix any of these with Amazon's AWS or Apple's iCloud?

Large-scale cloud services are linked to warehouse-scale computing in that they need large highly-scalable geographically dispersed implementations. That's not to say all implementations use the same infrastructure, or that someone won't find a better way. This is a relatively new area and one with a lot of potential for innovation – even though the core concept of a distributed system is quite old, with some of the theory dating back to the 1970s [Lamport 1978].

1.5 The Other Edge

Much of the previous discussion assumes we want the fastest possible system, constrained by cost, power consumption etc. However, Moore's Law can be read the other way. As hinted at by Gordon Bell, new classes of computer become viable as a given level of functionality becomes available at a price point [Bell 2008].

In the 1970s, a personal computer capable of doing interesting work – including the first spreadsheet, VisiCalc [Bricklin and Frankston 1979], became viable because a single-chip microprocessor powerful enough to run an elementary operating system and programming tools reached an affordable price point.

Since then, other breakthroughs have included:

- *scalable PC* – the original IBM PC was not a huge advance on the previous generation but Intel's ability to add enhancements like 32-bit addressing as new thresholds in the number of affordable transistors were crossed meant the CPU remained viable – even if major OS rewrites and recompiles were

needed in the transition to 32 bits (and less so to 64 bits)

- *Linux* – when the IA32 became powerful enough to support a UNIX-like operating system it was only a matter of time before a free UNIX emerged; Linux was the first (1991) but there are others like FreeBSD (which appeared only two years later)
- *RISC* – the ability to implement a version of Seymour Cray's 1960s ideas on a single chip, starting from the 1980s, culminating in the collapse of most medium to large-scale computing options that didn't use microprocessors
- *mobile devices* – starting from increasingly sophisticated notebook computers, mobile devices today include smart phones and tablets. Each new form factor derives from another step in the amount of functionality available at a lower price point – and able to run longer between charges or on a smaller and hence cheaper battery
- *pico-PCs* – a likely development out of smart phone parts is ultra-small PCs, of which the Raspberry Pi is an example

In some ways the "other edge" space is even more exciting than the big iron end of the design space because it creates the potential to transform the lives of many people, always remembering that technology is only a tool, and a tool only works if competently applied and to the right problem.

1.6 Structure and further reading

In the remainder of this book I examine the above topics in more detail. First, I use RISC-V to illustrate general principles of instruction set design, by contrasting its design with other common instruction set designs. Then I go on to using memory hierarchy as a starting point for understanding quantitative principles of system design and research, as well as trends and how to analyse their long-term effects. Next I examine parallelism in its various forms, starting with instruction-level parallelism. This is a large topic on its own encompassing pipelines, out-of-order execution, minimizing delays from branches and the rationale behind the multicore movement. Next I look at alternative models of parallelism including data parallel architectures and more specifically GPUs. I go on then to examine thread-level parallelism and how it relates to areas previously covered including memory hierarchy and multicore designs. Finally I look at the two ends of the scale: warehouse-scale computing and emerging small-scale systems as representing two very different consequences of technology trends.

Hennessy and Patterson [2012, 2017] is a comprehensive advanced architecture text and worth reading for full coverage of the field. This shorter work draws on my own research experience and does not aim to be as comprehensive as theirs.

Exercises

1. Does high-level language-oriented design seem like a good idea to you? Consider historical examples and compare with current possibilities.
2. If Intel has features in their current instruction set architecture that made sense in the 1980s and less so now, why is Intel still successful?
3. Contrast the RISC strategy with any design that differs from the core concepts: how much do these differences make it hard to achieve:
   (a) Speed
   (b) Low-energy design?
4. Parallel architectures are becoming mainstream after being confined to narrow niches like supercomputers. Explain why.
5. How does warehouse-scale computing change design options for computer architects?
6. If you had to design a new instruction set from scratch, what factors would you take into account and how would the likely area of application influence design choices?

2 Principles of Instruction Set Design

INSTRUCTION SET DESIGN is a moving target because design trade-offs change over time. Memory sizes and relative speeds change, types of problems of interest change as technology prices and packaging options change. Another big challenge is the scale difference of systems is growing rapidly. As larger and larger-scale systems become viable, very small system with volume markets also become viable. Designs that cover a wide range of scales are desirable so economies of scale can be realized.

As an example of an extremely small system, WiFi-enabled flash cards designed to allow access to photographs in a camera in some cases include a web server. An entire computer system is implemented on a flash card, only visible to the camera (assuming it does not interface to the card functionality) as a higher power draw. For such a system to be easy to implement, it should ideally run a standard operating system that can support a web browser and WiFi base station. And that would be easiest if this WiFi-enabled flash card had the same ISA as a larger, widely-used system with a suitable OS and tool chain.

In addition to scalability, I examine how apparent complexity can simplify hardware design, illustrating the important of hardware-software co-design, i.e., combining the expertise of hardware architects and software designers. This is an old idea [De Michell and Gupta 1997], but is still good.

Finally, I do a case study of how design trade-offs result in familiar features of common instruction sets, such as the number of registers and how load and store addressing is implemented.

2.1 Scalability

One of the most common reasons for an ISA dying out in the early days of computing was insufficient address bits. The PDP-11 was a very successful minicomputer in the days when that was a category, but it had only 16 bits of

addressing. The manufacturer, Digital, replaced it by a very different 32-bit architecture, the VAX. Intel, on the other hand, has progressively scaled the x86 architecture up from 16 bits to 64 bits (the last step led by AMD).

Are we ever going to need more than 64 bits? A 64-bit address space covers about 18-million terabytes. That seems to be more than anyone could ever need, but there could be applications needing a huge address space like a distributed database implemented in a single address space.

A different requirement for scalability is designs that work for very small as well as very large systems. Some design compromises like variations in the ISA or addressing bits are necessary. Intel's IA32 design is most competitive in relatively aggressive designs because its complex instruction set makes it hard to implement simplified versions with really low power consumption. At the other end of the scale, ARM focused on design for low power and struggled to break into the high-end market though they are now starting to appear in high-end systems [Maqbool et al. 2015; Harris et al. 2015].

Let us see how RISC-V addresses scalability.

RISC-V has a core 32-bit integer instruction set that covers a wide range of uses. It has extensions like floating point and variants like a 64-bit version and a compressed instruction set for small memory footprint-sensitive applications. The usual instructions are 32 bits and are stored 4 byte-aligned. The compressed ISA has instructions in 16-bit parcels, so instructions have to be 2 byte-aligned. What makes the design interesting is that all these variants have been designed together.

An example of how RISC-V takes the different options into account in its design is immediate operand bits in control flow instructions. In the MIPS 32-bit ISA, immediates of this kind are at a 4-byte granularity: the low two bits are not stored since they are always zero. In RISC-V, only the lowest bit is not stored for a 2-byte granularity since this is needed for the compressed instruction set. While this means a bit is wasted (in 32-bit instruction mode, the lowest immediate bit is always 0), differences between compressed and non-compressed modes are minimized.

2.2 Hardware Simplicity

Hardware simplicity is critical for implementing aggressive pipelines and slightly less so for simple low-energy designs. Instruction set design plays to both concerns in different ways. RISC-V provides a number of examples that illustrate how simplicity by design is easier to achieve with hindsight from previous ISAs.

Table 2.1: Condition Codes and alternatives

Condition Code	Result to Register	Test and Branch
`sub R1, R2, R3`	`sub R1, R2, R3`	`blt R2, R3, target`
`bneg, target`	`bltz R1, target`	

2.2.1 Condition Codes

An illustrative example is the use of condition codes. Condition codes are typically set by most ALU operations and make it possible to test an outcome without an explicit place to store a test outcome. Superficially, condition codes are a very minor hardware complication since the logic to set a code bit can be added on to each ALU operation at relatively low cost.

The problem comes when you try to implement an aggressive pipeline that reorders instructions dynamically.

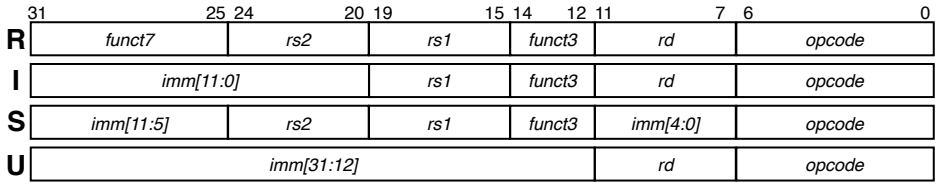
Condition codes are bad for hardware instruction reordering because almost all ALU operations set one or more condition codes so an instruction setting a condition code that is used for example in a later branch creates a dependency that hardware has to detect and differentiate from other instructions that also set condition codes.

So this apparently minor hardware complication turns out to be a major complication in the design of an aggressive implementation.

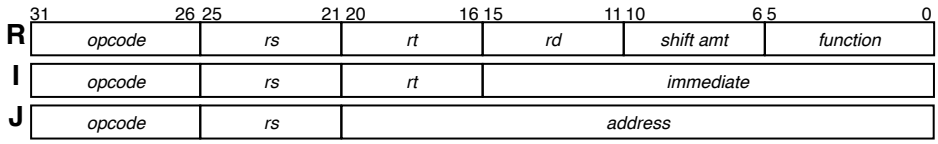
ARM uses condition codes and this has not been an obstacle to implementing low-energy designs. However, ARM has battled to produce higher-performance designs to compete in the desktop and server space. This cannot be the only obstacle since the SPARC architecture, which also uses condition codes, does scale up to relatively fast design – but every additional complication adds to the difficulty of making a fast version. An obstacle isn't an absolute barrier as Apple's latest Macs use very competitive ARM-based CPUs (M1 and M2). However, Apple gains part of their speed improvement by system-on-chip (SoC) packaging, meaning that even if parts are separately fabricated, the tight packaging disguises this fact [Mattioli 2022].

The alternative used in some other RISC instruction sets including MIPS and RISC-V is explicit condition-checking instructions that store a result in an integer register or combine the condition check with a branch instruction. Another option used in IBM's POWER architecture is multiple condition code registers, which breaks the tight dependence between many ALU operations and branches.

Table 2.1 shows some variants on testing whether the contents of register `x2`



(a) RISC-V base formats



(b) MIPS basic instruction formats

Figure 2.1: Variations on RISC formats. *MIPS and RISC-V are conceptually similar with some significant differences.*

is less than x3. The condition code variant has the advantage that more bits are available for the branch offset since no register is named, but a piece of arithmetic is required whether the answer is needed or not, possibly wasting a register. If the arithmetic is done anyway, that objection falls away. The "Result to register" option is similar to MIPS code. This again requires an extra register because MIPS has a limited number of conditional branch instructions, so I in this case convert the condition test to a subtract and a compare less than zero. The final column, "Test and branch", is the RISC-V variant. In this small code snippet, the RISC-V version saves one instruction, but that may not always be the case because the arithmetic required may be needed for other purposes. In the RISC-V design, including the test in the branch instruction does not add significant complication as compared with the MIPS approach of comparing against zero.

2.2.2 Big Gains from Minor Complications

Avoiding complication is a good starting point but when a small amount of complication is a big win, it is a good investment. RISC-V again provides some examples.

It is constructive to examine the differences between basic MIPS and RISC-V instruction formats, as illustrated in figure 2.1. Superficially, the RISC-V basic formats look more complex, but they are designed to simplify implementation. In broad terms these basic formats correspond to (all integer operations):

- R-type – register-register

- I-type – register-immediate (e.g. arithmetic with immediate values, loads)
- S-type – register-register, with immediate and no destination register (e.g. stores)
- U-type – single immediate 20-bit operand (e.g. `jal` instruction: the destination register is the return address)

Here are a few details to note:

- *position of destination register*
  - it looks as if the MIPS instruction set always has the destination register in the same location but this is not true; load and most other I-format instructions only need 2 registers, one of which is a destination, register `rt`. A store instruction also uses *I* format and register `rt` *is not* a destination
  - the RISC-V instruction set, at the cost of sometimes having to split an immediate into parts, always uses the same field for the destination register; the layout of a load and a store for this reason is not the same
- *position of immediates* – in RISC-V instructions, the high bits of an immediate are *always* at the high end of the instruction word, which make sign extension simpler (since the sign bit is always at the highest bit of the word)
- *position of opcode* – The RISC-V opcode is at the low end of the instruction word, while the MIPS opcode is at the high end of the instruction word. The RISC-V arrangement makes it possible to mix 16 and 32-bit instructions in a compressed mode: the low 2 bits determine whether a 32-bit parcel is one instruction or two.

Unscrambling immediates (RISC-V has some formats more complex than the "S" format example given here) is easy in hardware – it is a simple matter of routing wires. On the other hand, having two potential destination register fields means any aggressive pipeline featuring out of order execution must decide which alternative register is the destination early. By contrast, a RISC-V pipeline only needs determine that it is not an "S" format (or others not illustrated here without a destination register). An aggressive implementation could assume bits 7..11 encode the destination register at the early stage of setting up dependences, then drop this dependence if it is not an instruction that writes a register. The MIPS design, with a similar strategy, would require treating both the `rd` and `rt` fields as a potential target register before determining which or both to drop as a dependence.

Both MIPS and RISC-V have function fields that add to the opcode as a way of differentiating instructions and both present similar performance issues. If the

function fields are extracted early in parallel with interpreting the same bits as immediates, some work is thrown away once the opcode is decoded sufficiently to decide on the purpose of the bits but no new dependences are created.

Of these design choices, the ones of most significance are positioning the immediate (very easy to find the sign bit; sign extension can start before knowing whether it's needed) and keeping the destination register in a fixed place (only one possible register write dependence to resolve).

What these examples illustrate is that a good understanding of hardware design can guide instruction set design to non-obvious choices that superficially add complexity but in fact simplify implementation.

2.2.3 Summary

A general principle of instruction set design is to keep things as simple as possible – but also with an eye to scalability up and down. New designs, if capable of scaling up to much more aggressive implementation or down to much less aggressive implementations (e.g., for embedded systems), will have a longer life and wider applicability than designs that neglect these concerns.

Even designs that are not particularly well thought through in terms of longevity can be rescued by smart engineering, though inherent design limits make taking implementation a particular direction – much faster, lower energy use, etc. – more difficult.

2.3 Design Trade-off Case Study

MIPS and RISC-V both have 32 integer registers, requiring 5 bits to encode the registers, and though the size of immediate operands differs between the two ISAs. In both instruction sets, the immediate size in an I-type instruction also determines branch offset size and offsets in load and stores. However, a store instruction in RISC-V is a slightly different format to respect the fact that the destination register field is always in the same place (a store doesn't have a destination register). In this instruction format, the opcode is 7 bits in RISC-V and 6 bits in MIPS. This is a small difference and changing the number of bits to encode a register would have a bigger effect, as I format in both contains two register fields.

Let us consider two questions:

- what if we increase the number of registers?

- why is the offset in a load or store embedded in the instruction (as an immediate operand), rather than another register?

2.3.1 Number of Registers

Increasing the number of registers superficially seems relatively cheap in I-format. If we double the number, the register bits increase by two, since we have two register fields. We could do this by reducing the opcode by 1 bit in RISC-v; this would reduce it to the size of the MIPS opcode and by taking a bit off the immediate. In the RISC-V design, this would reduce the immediate to only 11 bits. Is this sufficient?

How about MIPS? In MIPS the immediate is 16 bits. If we reduce it by 2 bits to 14 bits, it is still bigger than the RISC-V immediate, before our alteration.

Comparing the two designs, it is hard to see that the absolute number of bits in an immediate instruction is cast in stone. However, it would be necessary to do studies of real code to determine the range of immediate operands in common use. The MIPS design allows a 32-bit value to be created relatively easily because the load upper immediate (`lui`) instruction can set the high 16 bits and the low 16 bits can be set by an or immediate (`ori`) instruction. One study for example shows that 70% of immediate operands require only 3 bits [Li 2019]. This sort of design study guides architects on details such as how long immediates should be.

The size of an immediate operand is only one input to the issue of how big register field should be and hence how many registers there should be. All other instruction types that user registers should also be considered. Is doubling (or an even bigger multiple) the number of registers possible without major compromises in other aspects of instruction set design?

However, there are other costs to increasing the number of registers. Each time the operating system switches to a new process, it must save the stalled process's registers and restore the restarted process. Doubling the number of registers doubles the time that this takes.

2.3.2 Memory addressing

Both RISC-V and MIPS use I-format instructions or similar for memory access (loads and stores; RISC-V's S-format for stores is similar in terms of number of bits for each purpose though the layout differs). In both, one register contains the value (source for a store; destination for a load) and another, the base address. The

offset is added to the base address.

For some use cases, this is a good design. If you address an element of a structured type (`struct` in C or object in an object-oriented language), you (meaning the compiler in most cases) should know how far into the data item the element is, so the offset can be embedded into the instruction. Similarly for accessing variables or spilled registers relative to a known starting point like the global variable space or stack frame, you should know the offset and be able to build it into the code.

However: array access is different. At machine level (or in C, which is much the same thing), an array is represented as its start address and the indexing operation is an offset from the start address, scaled by element size. For example, if an array contains 4-byte integers, each element i is at position $i \times 4$ from the start address. Since i, in most cases, can only be only be known at run time, an immediate offset is not possible – and also not desirable as an array offset can potentially be bigger than the immediate field.

How about introducing a new format for load and store instructions based on R format (essentially the same in MIPS-V and RISC though layout differs: three registers, one for destination, two for source operands). Using R format, you could add two registers to obtain the array element address: the start address and the offset. We will ignore the issue for now that RISC-V wants the destination register field to have this purpose in all instructions; should this prove to be a good idea, we will address it then.

Is this desirable? It would seem so but compiler writers have learnt a lot about coding loops tightly and it is unlikely that the extra overheads of setting up an additional register would pay off. In a loop where the loop counter is needed as a separate variable, a separate index register can be set up that is scaled up for element size. In the proposed R-format load or store, this extra register could be used in combination with the base address register. However, it is simple to create the same effect by having an additional register that starts with the base address and is incremented by element size each time through the loop. If the base address of the array isn't needed, the register containing the base address can be incremented each time through the loop instead.

To illustrate this, consider the following MIPS code that iterates through an array and uses a separate register to contain the scaled index to use as an offset:

```
#// find biggest element in array size N and return its index
#// if duplicates, the first biggest item is found
# leaf function with minmal variables we can keep in t registers
```

```
# so no need for a stack frame; keep parameters in $a0, $a1
# other registers:
#  i       $t0
#  imax    $t1
#  max     $t2
#  temps   $t3, $t4
#int arraymax (int data [], int N) {
#  int i;              // loop counter
#  int imax = 0;       // biggest element index so far
arraymax: li $t1, 0

#  int max = data[0];  // biggest element so far
          lw $t2, 0($a0) # $a0 is address of 1st element

#  for (i = 1; i < N; i++) {
      li $t0, 1          # initialise loop counter
      j Ftest01          # test before 1st iteration
Fbody01:                 #   body of loop here
#     if (data[i] > max) {
      sll $t3, $t0, 2  # scale index
      add  $t4, $a0, $t3 # find ith item
      lw $t3, 0($t4)     # $t3 = data[i]
      ble $t3, $t2, Idone01 # invert condition
#       max = data[i]; // update biggest
      move $t2, $t3
#       imax = i;      // update biggest's index
      move $t1, $t0
#     }
Idone01:    add $t0, $t0, 1      # increment loop counter
Ftest01: blt $t0,$a1, Fbody01 # not done? Go again
#  }
#  return imax;
      move $v0, $t1
      jr $ra
#}
```

Assume now that we have load and store instructions that use an extra register for the offset instead of an immediate value. In this case, we only use a load instruction, since the code is finding a value in an array and not modifying an array element. Call the new instruction `lwr` for load word R-format. The instruction looks like this: `lwr` R_D, $R_{offset}(R_{base})$. Let‘s rewrite the loop body code using this new instructoin:

```
#     if (data[i] > max) {
```

```
      sll $t3, $t0, 2  # scale index
      lwr $t3, $t3($a0)     # $t3 = data[i]
      ble $t3, $t2, Idone01 # invert condition
#     etc.
```

What have we gained? The code removes the need for one temporary register (`$t4`) and an add. This would appear to be a win. However this case only applies where we need the value of the start address of the array later in the code. If we think it through, maybe we can remove even this advantage.

Since this is a short function, we can clearly see that the start address passed in (using register `$a0` – the convention in MIPS for passing parameters is to use registers `$a0`...`$a3`) is not used again so we can simplify the loop body to:

```
      lw $t4, 0($a0)     # $t4 = data[i]
      ble $t4, $t2, Idone01 # invert condition
```

and the loop counter increment needs another step, incrementing the array address as well as the loop counter:

```
Idone01:    add $t0, $t0, 1      # increment loop counter
            add $a0, $a0, 4      # increment array element address
```

This version of the code is the same number of instructions as the version with the R-format load (one fewer on the body; one more in the increment step). Even if the start address of the array *is* needed later, you can still use this version of the code but you need to copy `$a0` to another register and increment that register instead of `$a0`. This extra register only needs to be set up once and the critical thing for increasing performance is the length of code inside a loop. Even if this code is an inner loop and there the extra instruction is in an outer loop, inner loops are generally where most time is spent.

We can work through the issue with an example that includes stores; the result should be the same.

In a real design exercise, a wider range of use cases would be considered, as well as analysis of real code and capabilities of typical compilers.

Tbe bottom line? The additional logic complexity of adding a new class of loads and stores is unlikely to be justified when the same effect can easily be created with the existing load and store model – particularly when it is not hard to find a code alternative that is no longer than that of the proposed new instruction or at worst adds code outside the innermost loop.

2.3.3 Summary

Design trade-offs can be very complex and sometimes paper exercises are not sufficient – simulations or at least code instrumentation is necessary to quantify alternatives.

If there is a choice between complexity and simplicity, simplicity should generally win unless there is a quantifiable win for choosing complexity and a clear case that the win will not inhibit future designs (for higher speed, lower power or lower cost).

Complexity needs to be judged in all dimensions, taking into account what is practical to code (particularly by a compiler as few humans code at machine level) and what is practical to implement in hardware. Combining these issues in hardware-software co-design is likely to result in better outcomes than only considering the problem from one angle.

2.4 Lessons

Design trade-offs can be hard. Many of the good (or bad) decisions have already been made. It is useful to understand these decisions and how to make them in case you need to do a new hardware design but also to understand how to code efficiently at machine level and how to compare different designs.

The RISC and CISC approaches are very different. It is worth studying some of the earlier research that led to the RISC idea to understand better why it is (or is not) a better approach.

Exercises

1. Look for instruction set manuals for MIPS, RISC-V and Intel-64 (the current architecture).
   (a) How much similarity do you find between them?
   (b) Could you learn one of the others easily if you knew just one other of these instruction sets? If so, which would you start from?
   (c) Compare RISC-V and MIPS: to what extent is it true that the RISC-V designers learnt from past errors, as claimed?
2. RISC-V always uses the R_d field as a destination for instructions that require one register to be updated. Compare with MIPS and comment on why RISC-V is designed this way.

3. RISC-V in some instruction formats has immediate operands that are split into different parts of the instruction word. Explain why the designers made this choice and discuss whether this has positive or negative consequences for performance.
4. Explain why condition codes are a problem for scaling up to more aggressive pipelines.
5. ARM is branching out from the mobile and embedded markets to high-performance computation. Discuss problems they may run into with this move and any advantages they may have over established competitors.
6. The VAX had a single instruction to set up the call stack yet a sequence of simpler instructions ran 20% faster [Patterson 1985]. Explain what we can learn from this example.

3 Memory and Quantitative Design

MEMORY HIERARCHY is a critical part of computer system design because a memory large enough to contain a whole program and its data, and also fast enough not to stall the CPU, in most cases would be prohibitively expensive and almost certainly physically impossible to design. While we can rely on the principle of locality as outlined in Chapter 1 in general terms, we cannot set the size and organization of the various layers of the memory system with reasonable precision (to achieve a required cost-performance trade-off) without measuring variations.

In this Chapter, I present a range of design alternatives and techniques for measurement focused on evaluating the design alternatives for memory. These same techniques can apply with differences in detail to measuring differences in design alternatives in other areas of system design.

3.1 Memory Systems

Memory systems encompass the biggest range in performance difference of any one logical component of a computer system. For this reason, there are different organization details at each layer, though there are common principles. First I present these common principles, then illustrate how they apply at each level.

3.1.1 Organization Principles

Aside from obvious classifications like speed, size and cost, memory systems are generally organised by how they can be accessed and managed. The following in general terms apply to all memory systems, with significant variations in the detail (summarised in Table 3.1):

- *naming* – some kinds of memory have unique names (generally this applies to registers), others use an addressing scheme where a location is identified

term	**definition**
block	unit of storage or management *caches*: also called *line* *VM*: fixed-size *page* (older systems had variable-sized *segments*)
hit	block is found at the requested level
miss	block is *not* found at the requested level
replacement	if there is no vacant block to place a miss another must be *evicted*
victim	block to be replaced
dirty	block modified with respect to one or more lower layers
write through	writes reflected at next layer down
write back	dirty block copied only on replacement
associativity	measure of how many different locations a block can occupy: *direct-mapped*: only 1 location for any block *n-way set associative*: *n* different locations for a block *fully-associative*: a block can be placed anywhere

Table 3.1: Common terminology. *There is some variation across layers but these terms generally apply.*

by a numeric offset from the start

- *accessible unit* – some kinds of memory are accessible in fixed-size units (registers though nominally the size of a machine word have variants like single and double precision and part-word operations with one register like byte or half-word – usually 16 bits) whereas others can be accessed at various granularities such as a byte, two bytes, etc. The latter category may have alignment restrictions (e.g., if memory addresses refer to bytes, a two-byte access must start at an even address) and preferred sizes (a machine word is usually the width of the data bus)
- *transfer unit* – some kinds of memory only transfer to the next layer up or down in fixed size units (e.g., a cache typically has a *block*, sometimes called a *line* of fixed size; a VM system has a usually fixed page size)
- *management unit* – some kinds of memory are managed in fixed size chunks, including issues like *protection*, recording whether the contents is *modified* (sometimes called *dirty*), *valid* meaning that the unit of memory can be used without generating an interrupt, *present* meaning that the unit of memory is available at that level of the hierarchy or *shared*, meaning that more than one way exists to access that memory (usually a property of

multiprocessor systems)
- *replacement* – how do we determine which unit to evict if we run out of space? If we do so, what is the policy on writing dirty data to the level below?

As we examine levels of the hierarchy we will see how these properties apply and differ. Computer architects consider faster elements of the hierarchy to be "higher" and if the same kind of memory is split into more layers, the highest level is numbered 1.

3.1.2 Levels of the Hierarchy

In considering levels of the hierarchy, it is logical to start from the top and work down. When a program starts executing, the first thing that happens is the program counter (PC) register is loaded with the start address (actually the last thing from the point of view of the software that loads the program). The ALU then attempts to fetch the instruction from the L1 cache – but only after translating the address (on a VM machine) using the TLB, a level above the L1 cache in terms of speed. Levels below these are only accessed if the required data, page translation or instruction is not available at the topmost level. For this reason I describe the hierarchy from the top (fastest) down, though I defer discussion of some of the more complex strategies to the lower layers, since the interface between the very slowest layers and the next level up justifies sophisticated strategies to minimise access to the slowest levels of the hierarchy.

Registers

The top level of the hierarchy is registers. Registers are tightly integrated into the ALU and pipeline, and can usually be accessed in a fraction of a clock cycle. In terms of our universal principles:
- *naming* – register names are encoded into machine instructions, and generally can't be computed at run time
- *accessible unit* – registers are a fixed size though they may sometimes support precision variations (e.g., single, double, part-word) and vector ISAs allow registers to be treated like fixed-length arrays
- *transfer unit* – registers only transfer values in fixed sizes up to their widest precision (times vector length, if applicable)
- *management unit* – registers are sometimes collectively managed in hardware, e.g., if there is hardware support for multithreading, each hardware

context has its own copy of the registers. More commonly, detailed management of registers is in software: the compiler manages what is within them within a single process, and the OS manages saving and restoring registers between context switches (some older designs have hardware support for context switches)

- *replacement* – deciding which register to spill is usually totally under software (in practice, the compiler or, on a context switch, the operating system) control

TLB

The next level of the hierarchy is the *translation lookaside buffer* or *TLB*, which contains recent page translations. The TLB is usually integrated into the pipeline and can be accessed in a fraction of a clock cycle. A TLB is often organised as an *associative memory*, in essence a hardware hash table that doesn't allow collisions. The key being looked up is based on part of the address, the virtual page number. A TLB is in the critical path of logic: if a page translation can be found, it is used immediately to check if the memory location is represented in the L1 cache. In some architectures, virtually addressed caches [Inouye et al. 1992; Wheeler and Bershad 1992] are used, making TLB speed less critical, possibly completely eliminating the need for a TLB [Kang et al. 2011].

- *naming* – virtual page numbers identify entries
- *accessible unit* – each item in the table is a pair: a virtual page number (to check compare against when indexing) and a physical page number
- *transfer unit* – the TLB is generally filled and replaced in units of 1 page translation though it is possible to flush it (depending on the system, this may be necessary on a context switch)
- *management unit* – as with transfers, TLBs are usually managed per entry. With a virtually-addressed cache, if a TLB is present, it will need to be tagged with process IDs or be flushed on a context switch
- *replacement* – TLB replacement can in theory encompass the range of possibilities used in page replacement polices (see below: least recently used, first in first out, etc.) but in practice since the TLB is in the critical path for performance, a strategy that is fast to implement such as random replacement has some appeal

Any machine that is designed to achieve reasonable performance with VM needs hardware support for page table lookups to speed up handling TLB misses [Jacob

and Mudge 1998]. For example, Intel's IA32 architecture has a hardware page table walker that assumes a 2-level page table, reducing the time to handle a TLB miss to data references and no code in routine cases. Hardware page table walkers limit OS designers' ability to experiment with new strategies for page table design. In the worst case, a page table lookup, even with hardware support, can involve a trip to backing store, since some systems allow parts of the page table to be swapped out.

Minimising TLB misses is an aspect of performance tuning that is often neglected, and the consequences can be high. Assume an average TLB miss adds 50 cycles execution time (miss penalty). That is not an unreasonable assumption given the cost of accessing DRAM *vs.* CPU cycle time. Then if 1% of instructions result in a TLB miss on a machine that would otherwise execute 1 instruction per clock cycle, average execution time becomes

$$t_e = 1 + 0.01 \times 50$$

or 1.5 cycles, a significant drop over 1 cycle per instruction.

How can a high TLB miss rate be avoided?

A TLB represents one page translation. If you have a memory access pattern that spends very little time on one page, you will access many pages without accessing a high fraction of total memory. For example, if a page is 4KiB (the most common size), and you have a loop that looks like this:

```
for (i = 0; i < 1024 * 1024 * 1024; i+= 4 * 1024)
  a[i] = 42;
```

each assignment updates memory on a different page. This is of course a contrived example, but it's possible to write code that scatters data references around memory if not in quite such an extreme way. For example, object-oriented code with many small objects that are not referenced in the order they are placed in memory can exhibit this problem [Machanick 1996].

Caches

The next level of the hierarchy is caches. A cache is usually made of static RAM (SRAM), which uses transistors as its building blocks and hence draws on the same technology advances as CPUs. SRAM does not have any significant delay for an access over and above than the time to transfer its contents, so there is no special advantage to doing access in large units. A wide bus will deliver contents

faster than a narrow bus because it can do so in fewer transactions, but there is no lengthy setup time to amortize.

The *top-level* (*level 1*, or *L1*) cache is usually in recent designs tightly integrated into the pipeline and can be accessed in one clock cycle. To continue with our logical progression down the hierarchy, I describe caches before virtual memory, though VM is the natural place to describe some of the more complicated strategies since VM is closer to the operating system and hence has a bigger software component.

- *naming* – cache contents is generally *tagged* with a value representing the machine address of the cache block
- *accessible unit* – when accessing a cache, the CPU uses the same units of addressing as apply to main memory
- *transfer unit* – caches contents are moved or copied in blocks (also called lines) that are typically multiple machine words long. Typical values are 32 to 128 bytes. Some caches feature *critical word first*, in which the part of the block that caused the miss is transferred first to reduce the time the CPU is stalled [Zivkov et al. 1994; Moudgill et al. 1999; Aasaraai and Moshovos 2010]
- *management unit* – caches have tags representing the enough of address of the contents to uniquely identify a given block and state (modified, valid, etc.) for each block
- *replacement* – cache replacement policy depends on how the cache is organized:
  - *direct-mapped* – a given address can only go into one location so if that location is already occupied, whatever is there is replaced: very simple to implement
  - *n-way set associative* – a given address can go in one of n locations, so if none of those is available, one has to be selected for replacement; given the relatively high speeds involved, cache replacement strategies tend to be simple, though some have investigated software-based approaches [Cheriton et al. 1986] that approach the sophistication of virtual memory page replacement; for small n, hardware is still reasonably simple
  - *fully associative* – some have proposed making the lowest level of cache look more like virtual memory, and hence advocate approaches that approach the sophistication of virtual memory page replacement [Machanick et al. 1998; Hallnor and Reinhardt 2000], including

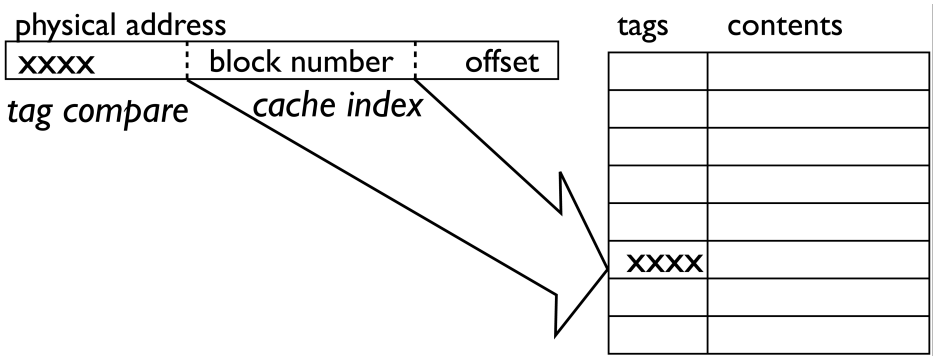


Figure 3.1: Cache addressing. *The low-order bits of the address are used to find the right byte or bytes within the cache block. The next-lower bits are used as an index into the cache, and the high-order bits in excess of those needed to identify a cache block are stored in the tag to identify which of the possible blocks is actually in the cache. For higher associativity, cache indexing produces more than one result and a hit is detected by checking if any of the stored tags matches the required block.*

> allowing a cache block to be placed anywhere in the cache: to implement full associativity purely in hardware is expensive and impractical for a large cache since every location has to be searched to compare the address tag with the request

In Figure 3.1, I illustrate how a machine address is broken up to check for a hit in a cache. It should be clear why the lowest bits identify the byte within a block – the *offset bits*, indicating how far into a block (or line) the required data or code is found. The next-highest bits (*block number*) index into the cache and the highest bits are used in the *tag* to identify which block is actually present. The highest bits are sufficient to store in the tag because the block number is used to find the right position in the cache and need not be stored and the offset bits are not used to identify the block but how far into the block the address specifies. Using lower bits (the block number) to index ensures that there is a reasonably good chance that items close in memory will not map to the same block.

In the event of a hit, the cache returns the required memory items (or in the case of a write, overwrites the portion of the cache block pointed at by the address). In the case of a miss, the cache controller must identify a victim (in a direct-mapped cache, that's always the same location; in other organisations, a victim may be picked at random since time is short at this level of the hierarchy), and request the block from the next level down. If the victim block is *modified* (or *dirty*), it must be written back to the next level down (*cleaned*). Caches can have one of two write policies:

- *write through* – all writes immediately are reflected at the next level down: seldom used because memory traffic using this approach is high
- *write back* – a dirty block is cleaned on replacement

In addition to the address tag, cache blocks have status bits. These can vary but common examples include:

- *modified* – a bit indicating the block is dirty
- *exclusive* – a bit indicating the block is not shared with another CPU or core
- *shared* – a bit set if the block is shared across more than one CPU or core
- *invalid* – a bit set if the block does not have valid contents

A system with this exact set of status bits is referred to as using the *MESI protocol*[1]. You may wonder why you need a shared and an exclusive bit. If a block is not initially shared, setting the exclusive bit makes this clear. We go more into shared caches when considering multiprocessor systems.

In most systems currently available, there is more than one level of cache. The L1 cache is relatively small and tightly integrated into the ALU so it can keep up with the pipeline. The L2 cache is larger and not as fast; some systems have 3 or even more levels of cache, on the principle that as much cache as possible is good but a large one cannot be fast without high costs in energy, a significant factor in design.

In aggressive ILP designs, a cache miss can cause a major slowdown. With a clock speed of 2GHz, one cycle is 0.5ns. If you can execute 4 instructions on one clock, the average time per instruction is 0.125ns so even if your second-level cache is very fast with hits taking only 1ns, a miss costs a delay of 8 instructions. To address this problem, non-blocking caches allow any instructions that are ready to go to continue without waiting for a cache miss [Chen and Baer 1992; Belayneh and Kaeli 1996; Aasaraai and Moshovos 2010]. If your ILP design already includes out of order execution, likely with an aggressive design, support for non-blocking caches is a relatively cheap addition.

For multicore systems, a common approach is to have an L1 cache that is local to each core and a shared lowest-level cache (LLC). Higher-level caches are usually local to each core. Shared caches is an idea explored in research into high-end systems in the past [Cheriton et al. 1988, 1989; Nayfeh and Olukotun 1994] – illustrating the value of a thorough understanding of technology history. Technology change can make it possible to package old high-end ideas at new affordable price points. The classic example: the reinvention of Cray's ideas once

[1] A few other details, specifically restrictions on the allowed combinations, apply to the definition of MESI.

sophisticated single-chip CPUs became viable.

Main Memory

The main memory in current systems is generally made of DRAM. DRAM uses a capacitor as its storage element. Unlike SRAM, DRAM has to be refreshed periodically because a capacitor's charge drains. Because the underlying technology is different, DRAM has its own price-performance trend, and that is driven more by price per bit than by speed. Hence, DRAM speed improvement lags CPU speed improvement (less so since the move from aggressive ILP and higher clock speeds to multicore, but the speed of multicore designs is still growing faster than DRAM speed, if you aggregate the rate at which memory requests occur across the cores). Also, unlike with SRAM, there is a lengthy delay before the contents can be accessed, so most current DRAMs have streaming modes where, once an access is set up, further sequential accesses moving along from that location are a lot faster. For this reason, moving to or from DRAM in large units is attractive if it does not cause other delays.

- *naming* – a memory address usually refers to a byte, numbered from the start; many machines require aligned access for large units (e.g., to do a 2-byte access, you must start on an even address)
- *accessible unit* – most DRAM systems are accessible at the byte level though, in practice, to handle cache misses, write-backs and write-throughs, a larger unit is transferred
- *transfer unit* – the transfer unit is the same as the access unit in practice, since most DRAM access are via the cache.
- *management unit* – at the low level, DRAM can be managed down to the byte level but in practice, with a VM system, what is in the DRAM or not is managed in pages
- *replacement* – replacement strategy in VM is complex and must take into account the mix of processes, other IO (since paging uses an IO device) and the extremely high latency of backing store (also often called *swap*). Some strategies include:
  - *least recently used* or *LRU* – the page used longest ago is evicted
  - *first in first out* or *FIFO* – the oldest page is evicted
  - *working set* or *WS* – each process is limited to pages it used over some fixed time period

- *clock* – a way of approximating LRU by systematically marking pages as unused, working around the list of pages in the style of a clock hand, and selecting a victim that is not marked as used (indicating the page was not used since the clock hand last passed that page)
- *page standby list* – a list of pages recently target for eviction [Russinovich 2007]

In some systems there may be a mix of *global* and *local* policies: a global policy balances DRAM use across processes, while a local policy attempts to ensure that a given process has enough DRAM to make progress. A local policy generally attempts to implement the *working set* principle: a process generally only access a subset of its pages for a reasonably long time before shifting to another part of its code or data address space. While the working set concept is quite old [Denning 1968], the principle still applies and will as long as memory has a hierarchy with several orders of magnitude difference in speed. A global policy may sometimes simply shut down processes if there is insufficient RAM (in the worst case, terminate them).

A complete coverage of virtual memory properly belongs in an operating systems course since it's at the interface between hardware and software, and software plays a much larger role than in higher levels of the hierarchy.

Paging Device

Paging devices historically have been mechanical magnetic storage devices of various forms. Early paging devices were dedicated magnetic *drums*, conceptually the same as a disk but with the recording surface on the outside of a cylinder. The earliest commercial VM system, the British Ferranti Atlas [Lavington 1978], had a drum memory with rotational time of 12ms (and thus an average rotational delay of 6ms), and no seek time since the heads were fixed, making it competitive with technology of 50 years later on speed if not capacity. The basic cycle time of the CPU was 2μs, only about 10^3 faster, compared with today's speed gap of a factor of over 10^6. It is the observation that in the late 1990s the delay in handling a cache miss to DRAM was approaching 3 orders of magnitude slower than CPU cycle times that led me to starting the RAMpage project, in which I move the virtual memory system up a layer to handle misses from SRAM to DRAM [Machanick et al. 1998] – so knowing a bit of history is useful.

Today paging is usually on standard drives (disk or, increasingly commonly, flash – solid state drives, or SSDs). There are two major variants: the traditional

UNIX approach of a swap partition, and using free space in the boot partition. Mac OS X uses the latter; Linux can use either. On iOS devices, which use a relatively small flash drive, paging is limited to evicting easily recreated content such as code from RAM. There are two reasons for this strategy: flash is small on these devices and repeated modification of the same bits in flash can wear them out. Programmers of iOS apps are advised by Apple to accept low memory messages and reduce their memory footprint as required [Apple 2012]. Bigger devices that use flash drives use flash much the same way as disk for paging. Reducing the tendency to wear out over-used bits using *wear levelling* [Chang 2007] may be easier with Apple's strategy of sharing the file system with backing store rather than using a separate swap partition.

- *naming* – a page on backing store can be anywhere on the device and is identified by a page table, using the virtual address (or more properly the virtual page number) as an index
- *accessible unit* – a VM system usually deals in whole pages
- *transfer unit* – pages may be transferred singly or the OS may move several contiguous pages to reduce overall latency
- *management unit* – pages are managed as a unit but also by process; if a process completes or dies, all its pages are freed
- *replacement* – since this is the bottom of the hierarchy, there is no replacement until a process exits the system; however, some systems do not keep pages on backing store if they exist in RAM and in that sense pages may not always exist on swap.

In a difference from caching terminology, a miss is called a *page fault*. In most real systems, a page fault results in a *context switch*: there is no point stalling the CPU for millions of cycles so despite the fact that a context switch has other significant costs like losing contents of caches, it is faster overall to allow another process to use the CPU while waiting for a page fault to be processed.

Having wended our way all the way down from the world of registers and TLBs that are accessible in a fraction of a clock cycle to paging devices that area accessible in millions of cycles, let's see how we measure the effects of all of this.

3.2 Measurement

There are many levels at which we can measure computer systems performance. We can measure individual components, we can measure times taken by small blocks of code, we can time a whole program, and we can time a workload of

interest. Aside from timing overall, we can apportion costs, so as to work out what to improve. Then in addition to timing, we can measure other attributes of interest like energy use, memory requirements if we change some detail (e.g. simplify the instruction set) and frequency of use of specific features.

3.2.1 Architecture-Oriented Measures

Depending on what we are measuring and how much detail we want, there are many variations, including:

- *logic-level simulation* – useful for checking design details like timing and energy use, but too slow to measure non-trivial program runs though work on speeding up such simulations may make larger runs viable [Chatterjee et al. 2009; Mironov et al. 2010]
- *execution-driven simulation* – a program runs on a simulator which can measure at a particular (sometimes parameterizable) level of detail including
  - *cycle-accurate simulation* – simulation run in software designed to give an accurate representation of machine time or energy use [Simunic et al. 1999]; slow for large runs though more recent enhanced techniques make such methods more viable for whole workloads [Lee et al. 2008]
  - *whole-system simulation* – while not necessarily cycle-accurate, these simulators are fast enough to evaluate whole workloads
- *trace-driven simulation* – a record of memory accesses (usually classified as read, write or instruction fetch) is read by these simulators, allowing memory system variation to be modelled (instruction variation can only be modelled in a limited way since the actual instructions are not recorded, and changes in execution order or instruction timing cannot easily be modelled)
- *emulation* – emulation differs from simulation in that it only aims to run a non-native instruction set rather than to provide accurate performance data
- *profiling* – measurement of relative times spend on different parts of a program; profiling can be implemented as a feature of a simulator [Cmelik and Keppel 1994] but it is more commonly implemented by instrumenting code [Reddi et al. 2004]
- *back of the envelope* – quick calculations that quantify relatively simple effects; limited in applicability since a whole system includes complex interactions between all influences on performance

From the difference in goals of emulation and simulation arises an interesting question: is it possible for a simulation to be too good? While real systems have variations in execution time that can't be eliminated arising from interactions between processes and interactions with external events, to produce repeatable results for scientific investigations, you need repeatable measurement. For an emulator, you care less about repeatable measurement and more about both accurate implementation of the target system as well as speed and minimal resource requirements. For a simulator, while those factors are important, it may be reasonable to sacrifice a little accuracy or speed for repeatability. In a sense then it *is* possible for a simulation to be *too good*.

3.2.2 Benchmarking

When we are really only concerned with comparing competing systems, rather than pinning down where the time is spent, benchmarking – comparing standard program runs against competing systems – is popular. Benchmarks fall into two broad categories:

- *kernels* – useful for testing how some very specific feature compares across architectures, e.g., floating point multiplication
- *full workloads* – programs that exercise the whole system including the file system, the memory hierarchy and even the network in ways representative of one or more classes of real programs; some examples include
  - *SPEC* – divided into integer and floating point scores [Henning 2006] and widely used specially in the UNIX space to compare systems
  - numerous other benchmark suites to evaluate web server performance (e.g. SPECweb – discontinued in 2012), database scalability (e.g. TPC benchmarks [Nambiar et al. 2011]), energy [Poess et al. 2010], embedded systems [Guthaus et al. 2001; Schoeberl et al. 2010] and other specific kinds of workload

One of the hot issues in benchmarking is gaming the system. For example, creating a compiler that recognises a specific benchmark and inserts hand-tuned code that no compiler could generate automatically, or including a special instruction that is hard to use in general are tricks used in the past. Kernels have to some extent fallen into disuse because they are so easy to hand-tune or otherwise arrive at fake results that do not predict real system performance. Even with SPEC benchmarks, which are whole programs of the size of a compiler run, I've had the experience of running my own code on two machines one of which had double

term	**definition**
miss rate	fraction of references at a level that miss
global miss rate	miss rate over all references
local miss rate	miss rate at a given level
miss penalty	extra time arising from a miss
hit cost	time for a hit

Table 3.2: Performance parameters. *The most important thing is elapsed time; minimising miss rate for example is not an end in itself.*

the SPEC rating of the other, and my own code reversed this to the extent of the "slower" machine on published SPEC results running in half the time of the "faster" machine.

In my experience the best benchmark is the workload of interest to you, run under conditions representative of your usual work (e.g., running an installer or compiling it yourself, then running it on a system loaded the way you usually run).

Since benchmarks are most useful for comparing competing machines rather than elucidating performance details of system components, I do not include them as an example for measuring memory systems.

3.3 Putting it All Together: Measuring Memory Systems Performance

Since memory references occur at least once for each instruction (an instruction must be fetched from memory and may also move data to or from main memory), an accurate simulation of memory systems performance making it possible to compare different options should really simulate most aspects of the pipeline. I examine here the variations that can be useful, starting from those that simulate the least detail.

Table 3.2 lists some terminology of use when evaluating memory performance. At the top of the hierarchy, the cost of a hit is often absorbed into a pipeline stage and hence not counted. At lower levels, we usually count the hit cost as part of the miss penalty for the layer above. When evaluating memory system alternatives, we care most about overall run time. Minimising miss rate for example may seem like a good idea, especially if as big speed gap is involved, but if doing so slows down the faster layer, there may not be an overall win.

3.3.1 Back of the Envelope Calculation

To get a quick feel for the effect of design parameters we can do simple calculations of the likely effect, remembering always that such calculations can be misleading because they do not take into account the full range of interactions of components. For example, with an aggressive pipeline that allows instructions to continue through the pipeline when others are stalled waiting for a cache miss, a simple calculation of the effect of increasing or decreasing the miss rate is at best a crude approximation.

Let's nonetheless look at an example in detail and at the same time introduce some terminology for speed comparison.

In Table 3.3 I list common measures of speed improvement. A speedup greater than 1 means you are doing better; a speed improvement greater than 0 means you are doing better. Speed improvement is a risky measure to use because "50% faster" doesn't sound nearly as impressive as "150%" faster so many people especially in marketing forget to subtract the 100%. You also get a very different answer if you look at improvement relative to the faster rather than the slower system. When quantifying speed improvement, make sure you define your terms.

To calculate the effect of misses, we need an execution time formula, which I generalize to allow more than one level of cache (and the main memory could also simply be counted as another level; going to a paging device is more complicated because the operating system is involved and hence a simple miss penalty does not apply):

$$t_e = t_{h_1} + \sum_{i=1}^{n} p_{m_i} \times r_{m_i} \tag{3.1}$$

Instructions per clock (*IPC*) is an average that depends on the workload (how much local parallelism there is) and the CPU (how many instructions can where t_e is *relative execution time*, normalized to 1=no misses; actual execution time is $t_e \times IPC \times IC \times t_{clock}$, where p_{m_i} is the penalty of misses from level i and r_{m_i} is

measure	**definition**
speedup	$\frac{time_{original}}{time_{new}}$
improvement	$\frac{time_{original} - time_{new}}{time_{original}}$

Table 3.3: Performance improvement measures. *Improvement is often given as a percentage. Dividing by t_{new} is very misleading and greatly exaggerates the % improvement.*

level	**hit or miss**										penalty
1	h	h	h	h	m	h	h	h	h	m	10
2					h					m	100
3										h	

Figure 3.2: Example of miss rate calculation. *We need to account for misses to L2 and L3, since there are no misses from L3. Assume hits in level 1 take 1 time unit, and penalties are relative to that.*

the rate of misses from level i (we use a global miss rate here since we want to quantify the effect on overall run time). It is useful to leave out the instruction count IC because that way we can compare scenarios where we don't vary the instruction count, without needing to know exactly how many instructions were executed. We also leave out the clock cycle time t_{clock} since that allows us to compare scenarios of similar clock speed without needing to fix the clock cycle time. This general formula can be adapted to include other causes of stalls.

Instructions per clock (IPC) is an average the depends on the workload (including how much local parallelism there is) and the CPU (how many instructions it can complete in one clock cycle). For simple ballpark comparisons we can take $IPC = 1$.

For example, in Figure 3.2, we have miss rates from L1 of 0.2 and from L2 of 0.1 (global miss rate). Applying the formula with the penalties given (and the noted approximations) results in:

$$\begin{aligned} t_e &= 1 + 10 \times 0.2 + 100 \times 0.1 \\ &= 13 \end{aligned}$$

In a real system, you would expect much lower miss rates than this, especially to the lower level (and slower) parts of the hierarchy.

For simplicity I assume that the L1 hit time accounts for all execution time, which is true in the case of a pipelined architecture. There is the possibility of misses for both data and instruction references, and we also need to ensure that we do not double-count hit time at level $i+1$ so we should not we treat an access at that level as part of the miss penalty of level i if we count it as part of the hit time at level $i+1$ – or vice-versa (see Appendix B, page 172 – my simplified simulator counts hit time below L1 as part of the miss cost for the level above). However you do this make sure you make it completely clear what you are including in the calculation and why.

Case Study

A simple example illustrates design trade-offs. Assume we have two ways of designing a cache. A direct-mapped cache has very simple logic (a given address can only map to one block in a cache) but has the drawback that it can have a high miss rate, since some combination of addresses used repeated close together in time that coincidentally map to the same block can evict each other when the cache is nowhere near full. A 4-way associative cache (4 different ways you can place any given address) can avoid this problem at the cost of slower cache reference time. Assume:

- *effect on hits* – the 4-way associative hit time is 10% slower than the direct-mapped hit time
- *effect on miss rate* – the 4-way associative cache has 20% fewer misses
- *miss penalty* – a miss from this level of cache costs 100× a hit in the direct-mapped cache

Calculate the miss rate in the direct-mapped cache at which the two caches have the same performance, and hence the point at which it becomes useful to use the 4-way associative cache.

Solution

We assume that the miss rate r_m is *relative* to this level of cache since we don't know anything about the rest of the hierarchy. We don't know absolute times so make the direct-mapped hit time t_d and base everything on that:

- *hit time at level* $i \equiv t_{h_i}$; for this example:
  - *direct mapped hit time* $\equiv t_d$
  - *4-way associative hit time* $\equiv t_4 = 1.1t_d$
- *miss rate at level* $i \equiv r_{m_i}$; for this example:
  - *direct mapped miss rate* $\equiv r_d$
  - *associative miss rate* $\equiv r_4$
- *miss penalty from level* $i \equiv p_{m_i}$; for this example, only one level with $p_m = 100t_d$

To find the break-even point, we can adapt the execution time formula (3.1) to make it easier to compare our two cache variants without excessive notation. For this example I need only 1 level and drop the i subscript, and derive variants for each case, which I need to set equal to find the break-even point:

$$t_{e_d} = t_d + p_m \times r_d \tag{3.2}$$

$$t_{e_4} = t_4 + p_m \times r_4 \tag{3.3}$$

We know that the 4-way associative cache has 20% fewer misses, and its hit time is 10% slower than the direct-mapped cache, so we can rewrite Equation 3.3 as follows:

$$t_{e_4} = t_d \times 1.1 + p_m \times r_d \times 0.8 \tag{3.4}$$

and the miss rate at which the two equations have the same execution time occurs when Equation 3.4 = Equation 3.2. So we need to solve for r_d in:

$$t_d + p_m \times r_d = t_d \times 1.1 + p_m \times r_d \times 0.8 \tag{3.5}$$

Put all the r_d terms on one side, and put everything in units of direct-mapped hit time t_d, noting that $p_m = 100t_d$:

$$p_m \times r_d - p_m \times r_d \times 0.8 = t_d \times 1.1 - t_d$$

and simplify:

$$100t_d \times r_d(1-0.8) = t_d(1.1-1)$$

$$0.2 \times 100t_d \times r_d = 0.1t_d$$

$$20t_d \times r_d = 0.1t_d$$

So the break-even point is where

$$r_d = 0.005 \tag{3.6}$$

To put the answer in English, in this scenario, we need at least 0.5% of the accesses to the direct-mapped cache to be misses before changing the design to a 4-way associative cache is a win.

Is this result surprising?

Having done a calculation like this, look back at the numbers to see if the answer makes sense. A miss penalty of 100 is pretty big in relation to the penalty of 10% slower hits for the 4-way associative cache so it shouldn't take a high number of misses for a reduction of 20% to be a win even given a small increase in hit time. The answer therefore looks plausible. Now go back to Equation 3.5 and check that $r_d = 0.005$ does indeed make the two sides equal and that a larger value of r_d does make the direct-mapped formula (Equation 3.2) for run time bigger than the 4-way associative formula (Equation 3.3).

Another way to check this sort of calculation is to see if you end up with the right units. We want a number expressed as a fraction without units like seconds or

number of instructions executed, since a miss rate is just a dimensionless fraction. If you end up with something that has the wrong units, you've probably forgotten to cancel something out or made a mistake in moving terms around.

In practice, most CPUs have two or more levels of cache to reduce the need for this sort of design trade-off. The L1 cache can be as fast as possible, and the L2 cache can be designed with a few compromises on raw speed to reduce miss rate.

3.3.2 Profiling

Profiling is most useful to ascertain where time is spent on an existing architecture for a given workload, and is most often used as a tool to tune performance of a given program or set of programs rather than as an architecture design tool. The reason for this is that profiling does not allow the option of varying design parameters on a real system, and there is little point in doing profiling at the application level on a simulator, since you can instrument the simulator.

That said an understanding of architecture can inform your approach to profiling. If you understand the role of various system components like caches and the TLB, you are in a better position to understand where to look for improvements.

3.3.3 Trace-Driven Simulation

A trace-drive simulation takes as input a *trace file*, containing addresses tagged as one of a *read*, *write* or *instruction fetch*. It is possible to simulate multitasking workloads by interleaving traces, including traces simulating operating system functions, though the OS component necessarily must be an approximation.

Given speed improvements in direct execution simulation, trace-driven simulation is not as popular as it used to be [Borg et al. 1990; Uhlig and Mudge 1997; Engblom and Ermedahl 1999], though there is still a fair amount of research conducted using traces. It is nonetheless a useful tool for testing new ideas independently of CPU details. It is not very hard to create a simple trace-driven simulation, and there are tools to generate traces (e.g., Pin [Reddi et al. 2004; Bach et al. 2010]).

To measure memory system variation, the same trace file can be run through different models of the memory hierarchy (e.g., different sizes, organisations and speeds of caches). A simulation may also be sped up by starting the trace at a

point of interest in the code (e.g., skipping initialization). Although there is some inaccuracy, you aim to make that inaccuracy minimal as a fraction of the total run. With the aid of profiling it may be possible to isolate out parts of a program that contribute most to run time and focus on those, not forgetting that effects of the parts of the program not measured can perturb the results.

Tracing also misses the complications of CPU interactions with the memory system (pipeline behaviour etc.) but can give a good first approximation to the effect of memory hierarchy variation and is particularly useful when CPU design is incomplete.

3.3.4 Whole-System Simulation

Since performance of direct-execution simulations improved so that they run at a reasonably small slowdown over running on real hardware, it has become increasingly common for such simulators to support running a full system including an operating system, making for higher accuracy in measuring inter-process and system influences on performance. A good example of an academic project for full-systems simulation is M5 [Binkert et al. 2003] from University of Michigan and its successor gem5 [Binkert et al. 2011]. Gem5 has full-system support for the Alpha, ARM, SPARC and Intel x86 instructions sets. Alpha historically was a popular architecture for research because it is one of the cleaner RISC designs, though it is no longer in production.

A full-system simulation allows not only detailed variation of the cache architecture but also parameterization of memory system performance down to disk and even network layer, and potentially changing the page table structure, if you have the fortitude to rewrite the operating system interface to the hardware.

A factor that mitigates against the slowdown of full-system simulation is ubiquitous PCs capable of running Linux. Rather than run one simulation faster (as you can do with a less detailed model), you can run many instances of the simulation with different parameters if you are lucky enough to be in a university with large numbers of PCs in student labs that researchers can take over at off-peak times.

3.3.5 More Detailed Approaches

It is seldom that low-level cycle-accurate simulation is necessary for evaluating memory system variations. The biggest performance effects are excursions down

the hierarchy, rather than at the level of registers or the pipeline, so small inaccuracies in timing at those levels have an insignificant effect compared with a small change in miss rate. If you are checking a design for correctness, that's a different matter and cycle-accurate simulation as well as mathematical approaches to formal verification play a significant role.

3.3.6 Summary

For most research today, a full system simulation is the approach of choice. For classroom examples, we do paper exercises. For small-scale design studies, trace-driven simulation still has a lot to recommend it. We seldom need more detailed simulations purely to evaluate overall system performance but if producing a new design, we may want to do cycle-accurate simulation to check design assumptions, e.g., for the time a specific implementation should take for given operations (as well as to validate the design, as I describe above). For example, you need to at least work through the timing of the extra logic needed for a 4-way associative cache to know what percentage slower it is than a direct-mapped cache (the 10% number in the case study is not based on a real example).

Learning about a few publically available research tools is useful: do a search to build on the examples listed here. Also practice at examples of back of the envelope calculation. These are useful to build an appreciation of how performance trade-offs work, even if they are poor indicators of overall system performance.

Exercises

1. You are investigating a new page table organization that reduces page faults by 1% of total memory references; to implement this you will lose hardware support for TLB misses. Assume:
   - a page fault takes 1-million cycles to handle
   - cycles for handling a TLB miss are:
     - 50 with hardware support
     - 100 without hardware support
   - the only change in number of misses is the number of page faults

   (a) What is the net speed gain (or loss) of the new page table design if 0.1% of TLB references are misses?

   (b) Is this a useful calculation for a real system? Consider what a real system does on a page fault.
2. Assume we have a machine that in the absence of misses executes on average 2 instructions per cycle. Such a machine would have a higher peak throughput but would be limited by other limits on ILP such as branches.
   (a) Redo the calculation of the case study (3.3.1) under this assumption.
   (b) Now allow for a non-blocking cache that can avoid a stall on average for 5 instructions before having to stall.
   (c) Is a non-blocking cache a useful improvement given the miss cost of this example? When might change your answer?
3. Look up memory specifications on a current design. Have issues moved on much from the references cited in this chapter?

4 Pipelines and ILP

PIPELINES ARE AT THE CORE of instruction-level parallelism so I discuss the two together. A pipeline, sometimes pipe for short, is based on the same principle as assembly-line mass production. If you break a task down into smaller tasks, each requiring the same time to complete, you can dramatically speed up overall operation, even if completing one task is not sped up, because you overlap multiple tasks each at a different stage of the pipeline (or production line).

The key to pipeline performance is *balanced stages*. If one stage takes a lot longer than the others, that stage determines performance. Another consideration is overheads in moving from one stage to the next, which limits how deep a pipeline is practical. Another limitation on how deep a pipeline is practical is the cost of flushing the pipeline when instructions at various stages turn out not to be needed, usually on a branch instruction.

Instruction-level parallelism (ILP) builds on pipelining by adding options of out-of-order execution and more than one instruction per clock. These additions, as noted in Chapter 1, go back to the early work of Seymour Cray in the 1960s. Because RISC architectures lend themselves naturally to aggressive pipelines, some commentators erroneously label such features as "RISC-like", including in versions of the Intel IA32 (and of course IA32-64) architecture, which clearly does not have the attributes of typical RISC ISA. A RISC architecture makes aggressive ILP easier to design, but there is no reason in principle that any other ISA should not also feature an aggressive ILP implementation.

In this chapter, I review basics of pipelining and go on to show how ILP can be added onto a basic design. Much of the discussion is based on pipelines that complete at most a single instruction per cycle and that have the same total execution time. Pipelines that allow multiple instructions per cycle (*superscalar* pipelines) and floating-point pipelines with instructions that have multiple execute cycles considerably increase complexity.

4.1 Simple Pipelines

Pipelines can be organized with many variations on the number and type of stages. To keep things simple, I start out with a 5-stage pipeline that is relatively easy to implement for integer instructions using a RISC ISA. The stages are (in some cases, allowing for variations in instruction types):

1. *instruction fetch* (*IF*) – use the program counter register (PC) to load the next instruction and increment the PC
2. *instruction decode* (*ID*) – decode the instruction and also read register values from source operands; compute the branch (or jump) target address; sign-extend immediate operand values
3. *execution* (*EX*) – complete ALU operations using previously prepared operands including:
   (a) *memory reference* – add the offset to the base address
   (b) *branch* – determine branch outcome
   (c) *register-register ALU operation*
   (d) *register-immediate ALU operation*
4. *memory access* (*MEM*) – for a load instruction, fetch the data from memory; for a store, send the data to memory from the register whose value is to be stored
5. *write-back* (*WB*) – for ALU operations and memory loads, copy the result to the destination register

In a RISC ISA, much of this is radically simplified. For example, in IF, we can do all the possible options simultaneously and drop any not needed, because register operands are always in the same place in an instruction[1]. We need sign extension on immediate operands because a negative value has all 1s in any added most significant bits if we extend the precision (assuming 2s complement representation of integers). An immediate operand is built into the instruction and is therefore smaller than a machine word.

We see the value of the load-store architecture of a RISC ISA here. Because no instruction does both a memory reference and an ALU operation, a single pipe stage can do any part of either kind of memory operation.

This design does not complete all instructions in uniform time. A branch can complete in the third stage, a store in the fourth and all other instructions need all five stages. Nonetheless it is a simple design and easy to pipeline simply by

[1] Almost – remember how MIPS spoils this by sometimes using a different register as a destination?

	clock number								
instruction no.	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
$i+1$		IF	ID	EX	MEM	WB			
$i+2$			IF	ID	EX	MEM	WB		
$i+3$				IF	ID	EX	MEM	WB	
$i+4$					IF	ID	EX	MEM	WB

Figure 4.1: Progress through a 5-stage pipeline.

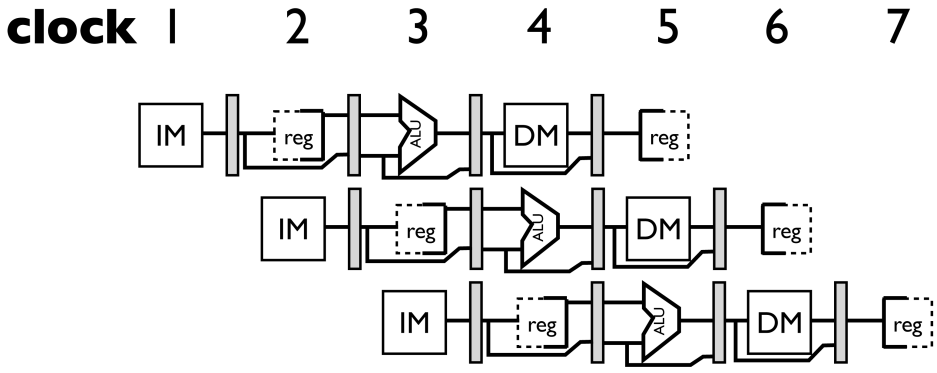


Figure 4.2: Pipeline progress with datapaths. *The register file is repeated: the first time it appears where it's read, the second time where it's written. The grey boxes represent inter-stage registers. Instruction (IM) and data (DM) memory are represented separately though they are logically in the same address space, capturing the property of most L1 caches that are divided between instruction (I) and data (D) caches.*

starting an IF on every clock, provided nothing interferes with simple sequential execution.

In Figure 4.1, I illustrate progress through a pipeline, assuming each instruction can start without delay. This is a common notation for illustrating progress through a pipeline and counting up total elapsed clock cycles. In this example, each instruction can start immediately and continues for all five stages without a break. In real examples, instructions my *stall* for various reasons, adding a *bubble* to the pipeline. A more realistic example should take into account *dependences* between instructions, e.g., if one instruction creates a value, a following instruction cannot enter the pipe stage where it needs that value until it's ready.

Another notation used to illustrate a pipeline uses a picture of a datapath, repeated starting once for each stage, showing the components active at each stage. The advantage of this notation is that it's easy to visualise dependences

between stages. Figure 4.2, based on the style used by Hennessy and Patterson [2012], illustrates how the datapath can be visualised in this time-shifted way. The grey boxes between stages represent *pipeline registers*, which pass values between stages. Because the register file is accessed at two different stages, it appears twice, with a broken line on the left if it's being read and on the right if it's being written. You can clearly see with this notation if a dependency may exist because the pipe stages where registers are accessed are explicit. The notation for the register file is useful because it contains a hint that a modification to a register and a read may be possible on the same cycle if the modification happens in the first half of the cycle and the read in the second half.

4.1.1 Pipeline Limitations

Our 5-stage pipeline isn't the only organisation possible. Some designs have fewer stages and the later versions of the Pentium 4 architecture had as many as 31 [Zukowski et al. 2006]. Very deeply pipelined machines are sometimes referred to as *superpipelined*. The theoretical gain from a deeper pipeline – more instructions in parallel hence theoretically greater speedup – is offset by various costs. These include:

- *clock skew* – longest delay between the clock arriving at any pair of registers
- *propagation delay* – the pipeline registers are fast but each new stage adds delay
- *cost of pipeline flushes* – the deeper the pipeline the more instructions are lost when the wrong instructions are in the pipeline; this adds not only to the cost of branches but also of context switches

In general super-deep pipelines have been explored and not had big enough performance wins to remain in the mainstream.

Another complication in pipeline design is floating-point instructions. For practical purposes, it is not possible to complete some of the more complex operations like floating point divide in one cycle, meaning that the clean simplicity of a RISC pipeline with uniform instruction handling is broken.

4.1.2 Pipeline Performance

Once we've worked out the number of stages and any delays between stages, we can work out a theoretical peak execution rate, which is just the clock rate. The clock rate is limited by the time of the longest pipe stage plus overhead. A 5-stage

pipeline can at most result in a speed up of 5 over a non-pipelined machine. A real machine though will have *bubbles* in the pipeline induced by *stalls* and therefore not achieve its theoretical peak throughput.

Case Study

Let's look at an example. The timing for each stage has to be worked out by doing a proper logic design and working out the longest logic path at that stage. Here, I use invented numbers to illustrate the principle. Assume inter-stage logic has an overhead of 0.1ns and the following times for each stage:

1. IF – 0.5ns
2. ID – 0.4ns
3. EX – 0.3ns
4. MEM – 0.5ns
5. WB – 0.2ns

The longest stage takes 0.5ns and overhead is 0.1ns, so this sets cycle time at 0.6ns (1.67GHz; to convert between GHz and ns: GHz = $\frac{1}{ns}$). How much speedup is this over a non-pipelined implementation? Superficially, we can add the cycle times of the nonpipelined machine, but we should also take into account the fact that some instructions don't use all stages and a nonpipelined implementation could possibly be designed to finish faster. In our 5-stage pipeline, only memory operations need all 5 stages (other instructions are idle in the MEM stage). To work out what average instruction execution time a non-pipelined machine takes, we need an *instruction mix*. Assume instructions break down as follows (as a fraction of all instructions executed in a particular workload):

- load – 20%
- store – 10%
- branch – 20%
- ALU operation – 50%

We can now work out an average for a non-pipelined instruction, in which 30% (loads plus stores) use all 5 stages and the rest skip the MEM stage:

$$\begin{aligned} t_{nopipe} &= 0.7 \times (0.5+0.4+0.3+0.2) + 0.3 \times (0.5+0.4+0.3+0.5+0.2) \\ &= 0.7 \times 1.4 + 0.3 \times 1.9 \\ &= 0.98 + 0.57 \\ &= 1.55ns \end{aligned}$$

So our actual speedup is $\frac{1.55}{0.6} = 2.58$, significantly less than a speedup of 5 that

you would predict from a superficial understanding of pipelining.

It is tempting given the numbers in our example to split the pipeline stages. Assuming we can split each longer stage into two stages, each half the size of the original (of course with overhead as before, but now for more stages), can we do better? Let's work the numbers, aiming for a new maximum stage of 0.25ns:

1. IF1 – 0.25ns
2. IF2 – 0.25ns
3. ID1 – 0.2ns
4. ID2 – 0.2ns
5. EX1 – 0.15ns
6. EX2 – 0.15ns
7. MEM1 – 0.25ns
8. MEM2 – 0.25ns
9. WB – 0.2ns

We how have 9 stages and the longest is 0.25ns, so our cycle time is 0.35ns with overheads, a speedup of 4.4 over the non-pipelined design and 1.7 over the 5-stage pipeline. That looks worthwhile but as we will see later, this is not the whole pipeline story and we need to take into account pipeline stalls before declaring a clear win.

What if we take this to the limit and make each stage 0.1ns, the same as the overhead? In this case, we have 19 stages and the cycle time is 0.2ns, a speedup of 3 over the 5-stage pipeline and 7.8 over the non-pipelined design. However, we have thrown a lot more hardware at the problem and we incur other significant costs, e.g., as we see when we deal with branches, we have significant costs of having the wrong instructions in the pipeline. With these numbers, it should be clear that further reducing the stage size has little benefit.

Hazards

Now we hit the hard part of pipelining, quantifying the costs when we have bubbles in the pipeline. A pipeline has an empty time slot when it can't proceed because of a dependency or resource constraint, generally called a *hazard*. Hazards fall into three categories:

- *data hazards* – data dependences prevent progress, divided into:
  - *read after write* or (*RAW*) – any use of a data value after its changed including registers and memory locations, though mostly registers in our examples: the main challenge is ensuring the updated value is read

  - *write after write* or (*WAW*) – any attempt to change a data value after another change: making sure the last change sticks is the main challenge
  - *write after read* or (*WAR*) – this hazard in less aggressive designs can be avoided by writing to registers and memory in a late stage; see Figure 4.2 for example where the "DM" box representing the MEM pipeline stage where movement of data between memory and registers happens and the second "Reg" box representing the WB pipeline stage are the two latest pipeline stages
- *control hazards* – a change (or possible change) in order of execution prevents progress
- *structural hazards* – a limit on hardware resources prevents progress (e.g., a functional unit is not available to two instructions that need it on the same cycle, something not a problem with our simple pipeline)

In general terms, *pipeline interlock* is any condition where a hazard stalls an instruction including waiting for memory [Callahan et al. 1988]; interlock logic can be anything that detects hazards that require a stall including control hazards [Hennessy and Patterson 2012, p C-65]. It is necessary for hardware to detect hazards, otherwise compiler writers would have a hard time ordering instructions for correct execution, as this could change every time a new pipeline is designed. Some early RISC designs without interlock logic did place this burden on compiler writers but the value of doing so quickly became diminished by new pipeline designs. An example of this is the *branch delay slot* in the MIPS design, where the instruction after a branch is always executed, whether the branch is taken or not. If the compiler can find an instruction that can safely be executed whatever the outcome of the branch, it can be placed there. If not, a `nop` (no operation – null instruction) is placed in the delay slot. However, in a later pipeline design, this feature was of no practical use but had to be retained for backwards compatibility [Yeager 1996].

To quantify simple examples, we need a machine code instruction set. We base ours on a generic RISC architecture, with ALU operations that either take two register source operands and one destination, or the source operands can include an *immediate* operand, a value encoded into the instruction. Memory data references are all either loads (copy from memory to register) or stores (copy from register to memory). We assume 32 registers (named `R0`...`R31`, with `R0` always the value 0) and a 32-bit instruction word.

A few things to note:

instruction	**effect**
`lw` R_d`,offset(`R_{s1}`)`	$R_d \leftarrow mem[R_{s1}+\text{offset}]$
`sw` R_{s2}`,offset(`R_{s1}`)`	$mem[R_{s2}+\text{offset}] \leftarrow R_{s1}$
`add` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} + R_{s2}$
`addi` R_d`,`R_{s1}`,`*`value`	$R_d \leftarrow R_{s1} + value$
`sub` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} - R_{s2}$
`mult` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} \times R_{s2}$
`multi` R_d`,`R_{s1}`,`*`value`	$R_d \leftarrow R_{s1} \times value$
`div` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} \div R_{s2}$
`divi` R_d`,`R_{s1}`,`*`value`	$R_d \leftarrow R_{s1} \div value$
`and` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} \wedge R_{s2}$
`or` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} \vee R_{s2}$
`xor` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} \oplus R_{s2}$
`lshift` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} << R_{s2}$
`rshift` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} >> R_{s2}$
`cmpeq` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} = R_{s2}$
`cmpne` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} \neq R_{s2}$
`cmplt` R_d`,`R_{s1}`,`R_{s2}	$R_d \leftarrow R_{s1} < R_{s2}$
`breq` R_{s1}`,`R_{s2}`,`*`offset`	$R_{s1} = R_{s2}$? $PC \leftarrow PC + offset << 2$
`brne` R_{s1}`,`R_{s2}`,`*`offset`	$R_{s1} \neq R_{s2}$? $PC \leftarrow PC + offset << 2$
`jal` R_d`,`*`address`	$R_d \leftarrow$ PC+4; $PC \leftarrow$ PC + (`address` $<< 2$)
`jalr` R_d`,`R_{s1}	$R_d \leftarrow$ PC+4; PC$\leftarrow$ PC + R_{s1}

Table 4.1: Simple instruction set for examples. *Both `offset` and `value` are signed 12-bit values. All instructions operate on a 32-bit integer word and a "`i`" suffix implies an immediate operand. We don't need both $<$ and $>$ tests because we can reverse the operands. You can obtain a logical negation by using* `xor` R_d`,`R_{s1}`,`R_0. *You can check for negative values by* `cmplt` R_d`,`R_{s1}`,`R_0. *To keep the notation consistent with an assignment, the destination operand is always written first (except for a store instruction, where the register and address are in the same order as a load). A jump instruction without saving the return address (*`j` *or* `jr` *is just the* `jal` *equivalent with the return address register* `x0`*, the RISC-V zero register, which cannot be altered.)*

- *unsigned operands* are specified in the instruction as "`u`" after the operand name
- *immediate* operands are encoded into the instruction and limited to 12 bits so, to extend the range of possible values, when they are used as address offsets for aligned access, the low bits are not present (which is why the "$<< 2$" calculation is used before adding them to a word address); immediate operand instructions are written with a "`i`" suffix
- *register* operands are 32 bits wide and can potentially generate unaligned accesses, which are trapped by hardware since these are errors for this architecture
- *branch* instructions generally are relative to the current program counter (PC); in assembly language for convenience we use symbolic labels to indicate the branch target but, in machine code, the target is a signed offset

- *jump* instructions are unconditional and usually allow longer addresses than the short offsets allowed in branches but are still relative addresses, unless you use the "`r`" variants
- RISC-V combines `jal` and `j` instructions: by setting the the return address to register `x0`[2], which is hardwired to zero, you get the effect of a jump instruction that does not save the return address

I only include word-length instructions with signed operations in the Table 4.1; an example of another variation, an unsigned add of 1 half-word ("s" for "short") is:

```
addsu R6,R5,R4
```

This is a very simple instruction set; simpler in some ways even than the MIPS instruction set, one of the more regular RISC examples[3] and a small sampling of the RISC-V instruction set[4].

Let's look at a simple code snippet, translated to assembly language in our notation and see how it proceeds through the pipeline:

```
for (int i = 0; i < N; i++) {
    a[i] += b[i] - 42;
}
```

To translate to our machine instruction set is reasonably straightforward. We need to note a few things:

- word size is 4 bytes so we need to go up in steps of 4 to iterate through an array
- the variable `i` is local to the loop and only used in array references, so we can replace it by an offset incrementing in steps of 4
- we need to test the stop condition before the first iteration to be consistent with the definition of a C-style `for` loop

In assembly language it looks something like this, with the original code interleaved as comments:

```
# Registers:
#      N: R1
```

[2]To keep things simple, I call my registers in examples `R0` ... `R31`, with `R0` in the role of `x0`. What a register is called doesn't matter and `R` for register is a bit more obvious than `x`.

[3]See `http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html` for some details of the MIPS instruction set.

[4]See `https://rv8.io/isa` for a summary of the RISC-V instruction set.

```
#      base address of a: R2
#      base address of b: R3
#     for (int i = 0; i < N; i++) {
       multi R8,R1,4 # loop end point (N scaled by 4)
       add R4,R0,R0 # i = 0 // scaled by 4 below
       j test
body:  add R5,R2,R4     # address of a[i]
       add R6,R3,R4     # address of b[i]
#            a[i] += b[i] - 42;
       lw R7,0(R6)      # get b[i]
       addi R7,R7,-42    # b[i]-42
       lw R10, 0(R5)    # get a[i]
       add R10, R10, R7 # calculate a[i]+b[i]-42
       sw R10,0(R5)
incr:  addi R4,R4,4     # advance by 4 because word = 4 bytes
test:  blt R8,R4,body   # < end of loop test
#     }

       multi R8,R1,4
       add R4,R0,R0
       j test
body:  add R5,R2,R4
       add R6,R3,R4
       lw R7,0(R6)
       addi R7,R7,-42
       lw R10, 0(R5)
       add R10, R10, R7
       sw R10,0(R5)
incr:  addi R4,R4,4
test:  blt R8,R4,body
```

This serves to illustrate progress of code through a pipeline and gives us a simple example to explore control hazards. To start with, we will only look at the body of the loop without conditional code, to see how data hazards arise. The body of the loop on its own is as follows:

```
       add R5,R2,R4 # address of a[i]
       add R6,R3,R4 # address of b[i]
```

	clock number									
instruction	1	2	3	4	5	6	7	8	9	10
`add R5,R2,R4`	F	D	X	M	W					
`add R6,R3,R4`		F	D	X	M	W				
`lw R7,0(R6)`			F	D	X	M	W			
`addi R7,R7,-42`				F	D	X	M	W		
`sw R7,0(R5)`					F	D	X	M	W	
`addi R4,R4,4`						F	D	X	M	W

Figure 4.3: Our code without pipeline bubbles. *I mark registers modified in previous steps in* `red`. `R5` *is set up far enough ahead that we need not consider it. We now have to work out where stalls should occur. For brevity I shorten the stage names to 1 letter.*

	clock number																	
instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
`add R5,R2,R4`	F	D	X	M	W														
`add R6,R3,R4`		F	D	X	M	W													
`lw R7,0(R6)`			F	–	–	–	D	X	M	W									
`addi R7,R7,-42`							F	–	–	–	D	X	M	W					
`sw R7, 0(R5)`											F	–	–	–	D	X	M	W	
`addi R4,R4,4`															F	D	X	M	W

Figure 4.4: Our code with stalls (marked as "–") causing pipeline bubbles.

```
lw R7,0(R6)
addi R7,R7,-42
lw R10, 0(R5)    # get a[i]
add R10, R10, R7 # calculate a[i]+b[i]-42
sw R10,0(R5)
addi R4,R4,4 # advance by 4 because word = 4 bytes
```

To see what dependences there are, let's write out a timing diagram then refer back to our definition of timing in the pipeline. To fit on a page, I skip a few instructions in the above sequence since they add nothing to the principle of the example. In Figure 4.3, I list the instructions without bubbles in the pipeline but instructions that depend on previous instructions highlighted. The `sw` instruction also depends on `R5` but only the delay caused by `addi` matters, since it is more recent that the calculation of `R5`. In our simple pipeline, a load and ALU result is available in the target register at the end of the WB cycle and an ALU operation needs a register value in the ID stage. That means we must stall the pipeline for three cycles in each case where an ALU or store operation follows another instruction that changes a register it needs.

The result as illustrated in Figure 4.4 is an increase from 10 to 19 cycles to complete the sequence of code.

That's a rather large slowdown[5]: $\frac{10}{19} = 0.53$. Can we do better? Waiting for the end of a cycle when a result is written to a register is not really necessary if we can write a register in the first half of a cycle and read it in the second half. Also, we can go a step further and add hardware resources to determine that a value is needed so we can *bypass* the register file, an approach also called *forwarding*. By making the first improvement, we can reduce each stall by 1 cycle. If we introduce forwarding hardware, we can use each result as soon as it's ready rather than routing it via the register file. In the case of an ALU operation, it is ready the cycle after EX. In the case of a load, it is ready after the MEM stage. Also, we can route the result at the cycle it's needed rather than the cycle before, e.g., for an ALU operation, if the result is ready before EX, forwarding can make it available at the start of EX even if it's not available at the start of ID. A store instruction only needs its value at the start of MEM. I illustrate a minimal version of stall reduction in the top half of Figure 4.5 and a more aggressive version using forwarding in the lower half.

In this example, we are able to eliminate all but one stall by aggressive use of forwarding. The cost of forwarding is a more complex decode stage, which must determine if any needed registers are pending results and if so set up bypass logic, which can include receiving values from the ALU or from a memory read. It is this kind of detail that illustrates the benefit of the extremely regular design of a RISC architecture. Register operands are always encoded the same way, so relatively little effort is required to determine which registers need values in the decode stage. In Figure 4.6, I illustrate why all stalls can be eliminated except for the add immediately following a load.

[5]Technically, this is a "speedup" though the word looks wrong applied to a case of slowdown.

	clock number														
instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`add R5,R2,R4`	F	D	X	M	W											
`add R6,R3,R4`		F	D	X	M	W										
`lw R7,0(R6)`			F	–	–	D	X	M	W							
`addi R7,R7,-42`						F	–	–	D	X	M	W				
`sw R7, 0(R5)`									F	–	–	D	X	M	W	
`addi R4,R4,4`												F	D	X	M	W
`add R5,R2,R4`	F	D	X	M	W											
`add R6,R3,R4`		F	D	X	M	W										
`lw R7,0(R6)`			F	D	X	M	W									
`addi R7,R7,-42`				F	D	–	X	M	W							
`sw R7, 0(R5)`					F	–	D	X	M	W						
`addi R4,R4,4`						–	F	D	X	M	W					

Figure 4.5: Approaches to reducing stalls. *The example above the line shows the effect of being allowed to read a register in the second half of the cycle when it's written. The version below the line illustrates the benefit of forwarding.*

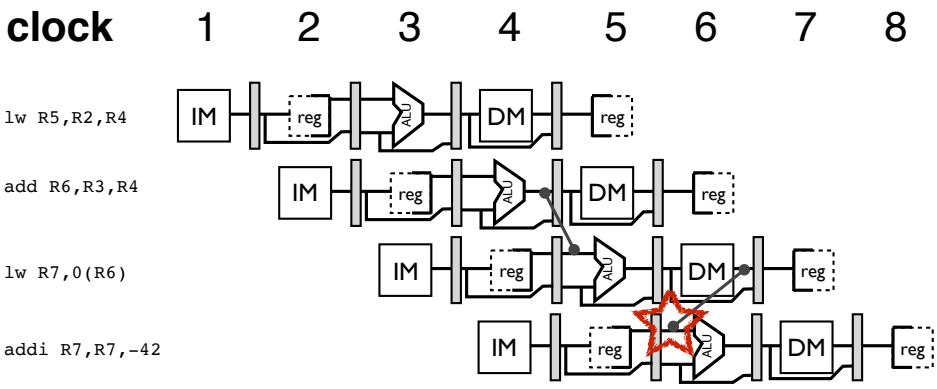


Figure 4.6: Limits of forwarding. *The connecting lines show how values can be forwarded; the star shows where forwarding would require sending a value back in time, since the load result is not ready in time for the next ALU operation.*

What of the branches? We have only so far considered data hazards. There are two places where control hazards occur: the test at the end of the loop and the jump at the start. We consider only the first example. The second is useful as an exercise for later. In this case (Figure 4.7) it is not strictly necessary to stall since the ID phase doesn't do anything that can't be undone. However with aggressive forwarding there is a fair amount of logic that would be wastefully exercised, a consideration for low-energy design. In this case, the branch is mostly not *taken*, i.e., the branch condition is false every time until we exit the loop. So eliminating the stall would be a win. Alternatively if a branch is mostly taken, starting to load the target instruction immediately that the branch target is known (in our microarchitecture, at the end of ID) rather than wait for the outcome to be known (at the end of EX in our design), would be a win.

	clock number										
instruction	1	2	3	4	5	6	7	8	9	10	11
`add R10, R10, R9`	F	D	X	M	W						
`sw R7,0(R5)`		F	D	X	M	W					
`addi R4,R4,4`			F	D	X	M	W				
`blt R8,R4,body`				F	D	X	M	W			
next instruction					F	-	D	X	M	W	
if not taken						-	F	D	X	M	W

Figure 4.7: Branch-induced stalls. *After fetching the next instruction outside the loop, we know the previous instruction is a branch and should stall until the outcome is known. If we predict* not taken *and keep fetching until the outcome is known and find we are wrong, we can eliminate the need for a stall at the cost of flushing more state if the branch is taken.*

Clearly, the loop control branch instruction will most often go the same way. We know here that the `breq` instruction controls a loop, but that's because we have the source code. How do we know in general when a branch is less or more likely to be taken? Many recent designs have hardware *branch predictors*. We can see from this example that a branch predictor will not be a huge win. If we predict the branch as not taken, that eliminates 1 stall (the only stall in the example), provided the prediction is correct. If the prediction is incorrect, we lose the opportunity to load the target instruction as early as possible and lose 1 cycle.

The simplest approach to branch prediction is *static prediction*, based on the observation that loops repeat by branching backwards. If you predict all forward branches as not taken and all backward branches as taken, you capture a large fraction of easily-predicted branch behaviour [Piguet 2006]. Our example may not be typical of machine code – it is more usual to put the test at the end of the loop and jump over the body the first time. Would this branch predictor work in this case?

A simple approach to *dynamic* branch prediction is to use 1 bit to record whether a branch is taken or not. In a case like a loop, 1 bit of prediction is potentially useful; in a case where prediction depends on the outcome of other branches a more complex strategy may be better. A simple way of storing state is in a *branch history table*, indexed by low-order bits of the instruction address. The more address bits used, the less chance two branches' predictions are confused with each other. A table of 4Ki entries suffices for smaller programs; current architectures may use bigger tables and more sophisticated schemes. There was a lot of research into branch prediction in the 1990s, when aggressive ILP was a major design goal [Yeh and Patt 1992, 1993; Kaeli and Emma 1997; Young and Smith 1999; Skadron et al. 1999]. If a branch is taken, the bit is set to 1, otherwise 0, and whatever was previously set is used to predict the branch outcome. A 1-bit scheme has the drawback that, since it changes every time the direction of the branch changes, if a branch mostly goes the same way, it mispredicts not only on the rare occasion when it goes the other way, but the next time when the direction reverts to the usual way (taken or not). A simple solution is to use 2 bits, in a scheme that requires the branch go twice in a different direction before the prediction changes.

Figure 4.8 illustrates state transitions of a 2-bit predictor. Each state is identified by two bits. If both bits are zeroes, that represents a prediction that the branch is not taken, which requires two successive instances of the branch being taken to flip the prediction. Both bits being ones means it takes two successive

bits	prediction	event	new bits
00	not taken	not taken	00
00		taken	01
01		not taken	00
01		taken	11
10	taken	not taken	00
10		taken	11
11		not taken	10
11		taken	11

Figure 4.8: Two-bit branch predictor state transitions. *Each predictor is stored in a table indexed by low address bits of branch instructions.*

instances of the branch to not be taken to flip the prediction. The other two states each require only 1 disagreeing branch direction to change the next prediction and can be pushed back to the "00" or "11" state in a single step. The scheme used here is a *2-bit saturating up and down counter*. It is "saturating" because it stops when it hits an end point and "up and down" because it has two end points, one for counting up, the other for counting down. A branch predictor, like many simple hardware constructs, can be described as a *finite state machine* (*FSM*), which I represent here as a table. You can also represent an FSM as a diagram with one node per state and arrows labeled with events indicating state transitions.

I illustrate how a branch instruction's address is used to look up a prediction in Figure 4.9. In this case the branch table only has 8 entries, not big enough to be useful, but we can see the whole example in one picture.

Branch prediction becomes a much more significant issue with superscalar

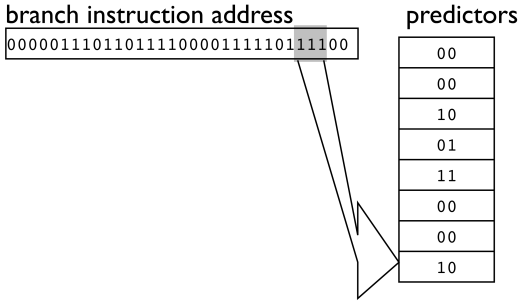


Figure 4.9: Finding a branch prediction. *Low bits of a branch instruction address are used to index a single global pattern table. In this toy example (table size only 8), 3 bits are needed. Instructions are word-aligned, so the low 2 bits of the instruction address are always 0 and not used in the index.*

pipelines, where deciding early to flush the pipeline and go on a new path makes a big difference, if you mostly choose the right option.

When we consider more exotic pipelines the benefits of branch prediction and other approaches to reduce branch latency become clearer.

4.2 More Exotic Pipelines

Three variations on pipelines add complication (assuming we are starting with a simple, regular instruction set: the IA32 is pretty complicated even in a simple implementation, for example):

- *deeper pipelines* – hazards have a higher cost the deeper the pipeline because there are more instructions in flight
- *multi-cycle instructions* – even in a simple RISC architecture, floating-point instructions cannot all be implemented in one execute cycle (particularly divide)
- *multiple instructions per clock* – a superscalar pipeline multiplies the opportunities for hazards

Aside from the standard kinds of hazards, handling interrupts becomes more complex the more complicated the pipeline. Ideally, you want your architecture to maintain *precise exceptions*: any instruction that logically entered the pipeline before that causing the exception should finish and any instruction that logically enters the pipeline later should not finish – and should not have any effect on the machine state. In other words, interrupts should not behave differently than if the instructions execute in order.

There is considerable complexity in handling floating point because some instructions take multiple cycles and hence make it hard to maintain precise exceptions (e.g. a divide overflow exception may take a few cycles to become apparent, implying that other logically later instructions that completed before the long-running instruction should be rolled back). Long instruction completions also make it possible even with our simple pipeline model to have WAR hazards.

Timing of deeper pipelines depends exactly how the stages are split.

Here I only consider multiple instructions per clock. Since this technique is in competition with multiple cores, it is useful to understand the basic concepts and how far they can go. I also examine tactics that can reduce dependences or pull them further apart. These techniques include instruction reordering, register renaming and loop unrolling. You can reduce stalls either by *static* or *dynamic scheduling*:

- *static scheduling* – the compiler (or a fanatical human who goes down to the machine code layer) can optimize ordering of instructions for a given pipeline
- *dynamic scheduling* (also *dynamic dispatch*) – the hardware determines the order of instructions at run time

A pipeline that can *issue* – start the EX stage – more than one instruction per clock is called *superscalar*. In the simplest scheme, the next k instructions are fetched and if there are no dependences between them limiting parallel execution, all are dispatched or issued simultaneously. A limitation of this scheme is that it's not necessarily a given that adjacent instructions have no dependences but other instructions further apart may be free to go. Another limitation of a simple scheme is that branches limit simple ILP. In a typical MIPS integer workload, between 15 and 25% of instructions (counted dynamically, i.e., as fraction of instructions executed) are branches, meaning you can typically expect 3–6 instructions between branches [Hennessy and Patterson 2012, p 149]. While floating point code often has longer sequences of instructions between branches, working around branches is a key aspect of achieving significant ILP.

In some schemes, dispatch and issue are treated separately[6]:

- *dispatch* – queue the instruction for execution
- *issue* – allocate a functional unit to the instruction and start its execute step

It is useful to treat dispatch and issue as separate concerns in out of order machines; in machines that start instructions strictly in order, there is no need to treat these steps individually. Note the usage above of "dynamic dispatch": this is a generic description that does not necessarily imply an instruction enters the EX stage at the same time as others dispatched with it.

A superscalar pipeline requires duplicated resources for any operations that could occur in parallel. Typically, the ALU is divided into *functional units*, a major grouping of related instructions, such as integer or floating point and the number of each type of functional unit limits the number of that type of instruction that can simultaneously be issued.

Before we go on, we need a little more terminology. We already know about data, control and structural hazards. Another type is a *name* hazard, a situation where instructions share the same data *resource*, usually a register, but do not actually interchange data. A *name dependence* usually arises because a machine does not have a limitless register set, so registers have to be recycled. Another

[6]Mark Smotherman has a nice summary of the terminology here: `http://www.cs.clemson.edu/~mark/464/dynsched.txt`

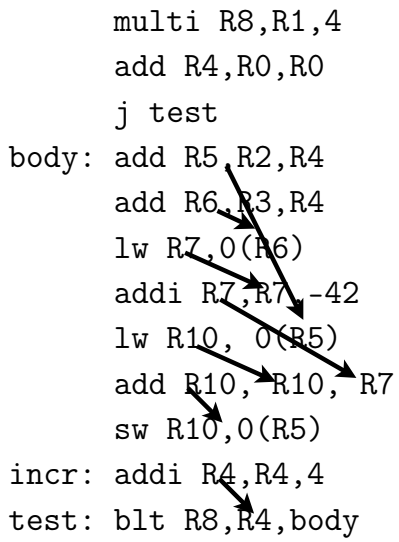


Figure 4.10: Dependences in one iteration of the loop. *To reduce clutter I omit dependences between initialization and the loop body.* `R4` *in the loop body depends on the initialization step in the second instruction and the loop test also depends on the value of* `R8`.

example is the call stack, which is recycled between calls, and limits any hardware attempt to convert function or method calls into threads [Postiff et al. 1998].

Static scheduling

Let us now return to our simple example and see what happens if we attempt to execute two instructions per clock. To start with, I look at reordering instructions and other changes that could be done at compile time.

In Figure 4.10, I illustrate data dependences using an arrow from the place the data is updated to the place it's used. To avoid cluttering the picture, I leave out dependences between the loop initialization and the body; of more interest is what happens when we repeat the loop. A question we need to ask is if these are true dependences, or name dependences. In one iteration of the loop, they are true dependences, limiting ILP. In a two-instruction per clock pipeline, we cannot issue two successive instructions if the second depends on the first. In the body of the loop, the only cases where pairs of instructions do not have a dependence on their immediate predecessor are the three adds. The first two adds can proceed in parallel; the second one can run in parallel with anything other than the other two adds, both of which use the value in `R4`, modified in the final add.

Using the same subset of the program as in Figure 4.5, let us see how much

parallelism we can extract in a simple scheme that fetches two instructions at a time and if there is no dependence, issues both at once. If there is a dependence, the second waits until the dependence is cleared. Figure 4.11 illustrates the outcome.

If we compare the result against eliminating stalls using forwarding but in a scalar pipeline in Figure 4.5, we've reduced total cycles from 11 to 9, not a huge win for significantly greater hardware resources.

Can we do better? So far, we have fudged the issue of multiple iterations of the loop. If we return to the original C-style code:

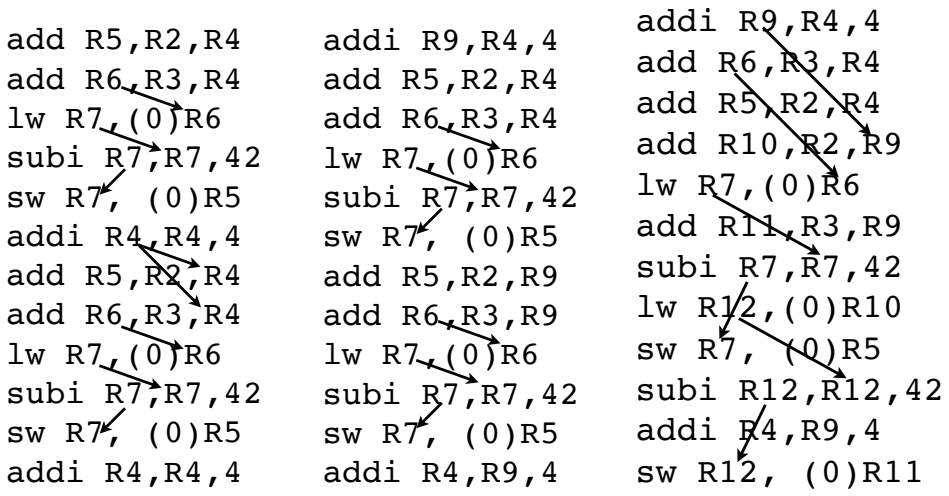
```
for (int i = 0; i < N; i++) {
    a[i] += b[i] - 42;
}
```

a simple observation is that the calculation for each value of `i` is independent so this code has more natural parallelism than is at first apparent. There's no reason if our hardware isn't clever enough that we shouldn't be able to run as many loop bodies as we have hardware resources for in parallel. The reuse of the variable `i` is an example of a name dependence, which we can break by systematically renaming `i` each iteration of the loop. The only real dependence is that we need to compute each new value of `i` based on the last one, but that's only one dependence rather than a long chain that imposes a strict ordering on our code.

So back to the machine code version: if we write out two iterations of the loop, leaving out the condition and branch, we have dependences between `R4`, the index variable, across iterations, but are these true data dependences? Not really, because we can replace `R4` by a different register. In Figure 4.12, I illustrate

	clock number								
instruction	1	2	3	4	5	6	7	8	9
`add R5,R2,R4`	F	D	X	M	W				
`add R6,R3,R4`	F	D	X	M	W				
`lw R7,0(R6)`		F	D	X	M	W			
`addi R7,R7,-42`		F	D	–	–	X	M	W	
`sw R7, 0(R5)`			F	–	–	D	X	M	W
`addi R4,R4,4`			F	–	–	D	X	M	W

Figure 4.11: Simple two-instruction-issue schedule. *If two instructions cannot execute on the same cycle, the second stalls. We fetch two instructions every cycle where there isn't a pending stall. Forwarding makes it possible to use a result at the end of the stage when it is created.*



(a) Two unrolls. **(b)** Rename R4. **(c)** More renaming.

Figure 4.12: Dependences in two instances of the loop. *To reduce clutter I omit dependences more than 3 instructions apart. I rename* R4 *as* R9*, then show with more aggressive renaming and reordering dependences can be moved further apart. Note that the final update of* R4 *is necessary so we can start the next iteration with the right value in* R4*. Check also for changes in instruction order facilitated by renaming.*

how two instances of the loop have minimal dependences between them – though the new register, R9, has many dependences to successor instructions (as does R4 in the original code, had I shown them). I then go on to show that I can increase the gap between dependences by increasingly aggressive renaming and taking advantage of that to reorder instructions.

What's the win here? The dependence between R4 and successor instructions need not be as close to them as in Figure 4.12a, if we rename R4 as R9 for the second instance of the loop body. We can move the initialization of R9 to the top as in Figure 4.12b, and we can get further gains by interleaving the code for the two instances of the loop, with further renaming of registers (Figure 4.12c). We now have the potential if we generalise to more than one instance of the loop to achieve a respectable level of ILP.

So far, I've assumed that we can have two instances of a loop. That is not in general true: if the loop executes an even number of times, we can do this, and adjust the stopping condition. What I've presented here is an example of *loop unrolling*. A compiler can generate code using the principles I illustrate here, but only for a loop where the stopping condition is a limit on a counter as in a typcial `for` loop. In that case, the compiler can generate two instances of the loop: one

that runs for $N\%k$ times, the other for $N \div k$ times. In this case, where $k = 2$, the compiler would generate code equivalent to

```
int i = 0;
int k = 2;
// a N%2 == 1 or 0; could use an if statement
// but the following generalises to k > 2
for (int j = 0; j < N%k; j++) {
    a[i] += b[i] - 42;
    i++;
}
for (int j = 0; j < N/k; j++) {
    a[i] += b[i] - 42;
    a[i+1] += b[i+1] - 42;
    i+=2;
}
```

We can potentially improve our unrolled code even further by using two registers for the different instances of the loop index from the start and incrementing each separately. However we have enough detail at this point to see how unrolling works in general and how it can be extended to multiple instances of the loop body. What we do not have is a way to do loop unrolling when the stopping condition is more complicated, i.e., we don't know even at run time (by the first iteration of the loop).

Dynamic scheduling and better branch prediction

There are three big downsides to static scheduling:

- an ideal schedule for one pipeline may not be ideal for another – recompiling code may be an option for software created in-house or on frequent release cycles, but maintaining versions of code for multiple pipelines is impractical for most software in common use
- some limits on parallelism may only be possible to resolve at run time
- you can run out of registers – even in a RISC design with 32 integer registers, some may be reserved in practice for specific purposes or e.g. you may be required to preserve and restore them; the example of Figure 4.12 uses 10 registers in its fully unrolled renamed form, 4 more than in its original form

The Control Data CDC 6600 was the first machine to tackle out of order execution in hardware. It had 10 functional units [Thornton 1963] and had a hardware structure called a *scoreboard* that kept track of dependences and identified which instructions could issue [Thornton 1980].

To justify out of order execution, you need to sacrifice about as much chip space as a functional unit. Once it became possible to add the equivalent in logic to another functional unit to implement a scoreboard-like feature on a single chip, commodity processors started to appear with out of order execution.

Another part of the picture, for a change not originally designed by Seymour Cray, is hardware loop unrolling. Robert Tomasulo, an IBM engineer, developed a hardware algorithm [Tomasulo 1967] bearing his name that was the first example of register renaming. The keys to the algorithm is *reservation stations* that hold an instruction until all its operands are available and internal register renaming. In an example like our unrolled loop, it would not be necessary to find new registers for the second (or subsequent) instances of the loop; the hardware would allocate virtual registers to the successive instances of the loop.

The important thing about both a scoreboard and Tomasulo's algorithm is that they make it possible to issue out of order, even though aggressively superscalar architectures were not feasible in the 1960s. A scoreboard makes it possible to issue instructions when data dependences are met; Tomasulo goes one step further and makes it possible to eliminate name dependences (at least between registers: name dependences as relate to memory addresses are another whole problem).

The major thing that these innovations add is that the sort of scheduling exercise illustrated in Figure 4.12 can work as well as the hardware available (including virtual registers, not visible to the programmer): provided there is a sufficiently large hardware instruction window, dependences can be limited to real dependences and as many functional units as are available can be kept busy, up to the limit imposed by true dependences. That leaves us with one major cause of stalls we need to reduce: branches.

So far the best we have is a 2-bit branch predictor, that can capture up to the order of 93% of branch behaviour. The remaining 7% is significant if the penalties are high. If for example we have a very aggressive design capable of issuing up to 8 instructions per clock and we mispredict a branch, we not only have to flush up to 7 instructions from the pipeline, but we have lost the opportunity to start up to 7 instructions in the correct path of the branch.

Before going to a more sophisticated branch prediction strategy, I introduce one more improvement in branching: a *branch target buffer* (or *BTB*). The win

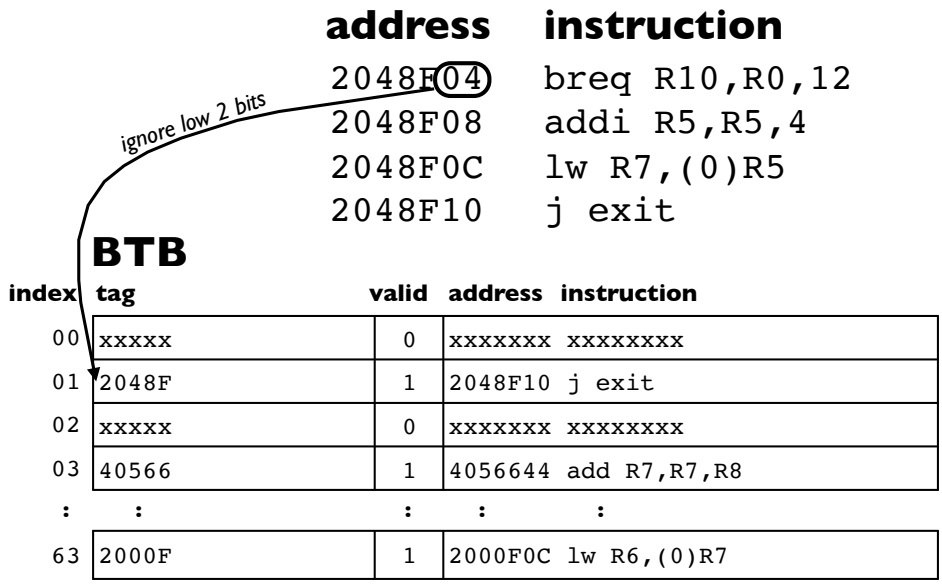


Figure 4.13: Possible branch table buffer organization. *The instruction would of course be represented in binary rather than a human-readable form. Some BTBs only represent the target address and others also include the branch prediction. Note that in a word-aligned machine with 32-bit instructions, it isn't necessary to store the low 2 bits of the instruction or use them in the index. In this example, the index uses 6 bits, leading to a 64-entry table.*

from a BTB, which stores the target instruction of the branch, is that as soon as the branch is resolved as taken, the target instruction can be inserted into the pipeline. In some schemes, a BTB may include branch prediction [Perleberg and Smith 1993], but I prefer to keep the terms separate. A BTB may store varying degrees of information from a prediction of the branch target address through to the target address and the actual instruction. Unlike a branch predictor, a BTB needs an accurate representation of the target instruction since it would be useless to start executing a completely wrong instruction, so a BTB typically includes a tag that allows the address of the *source branch* to be reconstructed. In other words, similarly to a cache, a BTB is indexed by part of the address of the branch being predicted and the rest of the branch address is used in a tag to check that the right target has been found. This scheme can obviously only work if branch instructions cannot vary the target address, i.e., it's always based on an immediate operand, not a register. Figure 4.13 illustrates a possible BTB organization. If the top branch instruction in the illustrated snippet of code is about to be executed, the BTB logic looks up the target instruction. If the branch is predicted as taken, it can be fetched immediately. In an aggressive scheme, the BTB can be looked

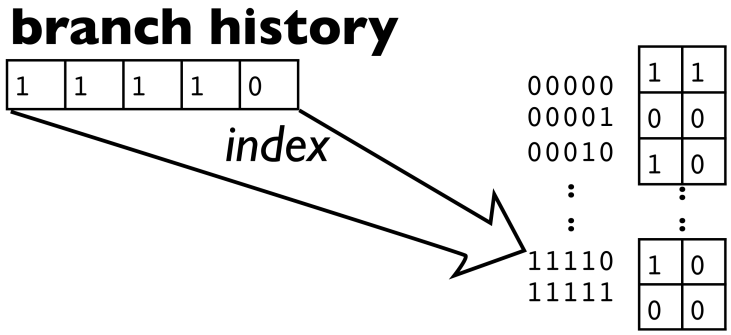


Figure 4.14: Two-level predictive branch. *After each branch, the pattern history for that branch is shifted left and the latest actual outcome becomes the low-order bit. The individual branch histories may be stored in various ways including a structure indexed and tagged like a cache.*

up before the instruction is decoded, since the tag ensures that a lookup will miss if the current instruction turns out not to be a branch.

A BTB, much like a cache, can experience misses. In the event that the branch is predicted as taken, a miss requires waiting for the target instruction but the BTB is updated for next time. If the branch is predicted as not taken, the BTB can be ignored.

On now to more sophisticated branch schemes. There are many of these [Yeh and Patt 1993; Skadron et al. 1999; Tyson 1994; Kaeli and Emma 1997] and there was considerable research on these in the 1990s at the height of ILP research and I only consider one in depth, an adaptive two-level predictor [Yeh and Patt 1991]. With the shift to multicore designs, branch prediction is unlikely to need more sophisticated schemes in the near future than the schemes of the 1990s. In this scheme, there is a predictor for each branch. Each predictor maintains a k bits of history of a branch. These k bits are used as an index into a pattern table that predicts what the next branch should do. The entry for each pattern is a saturating counter, much as for our single-level 2-bit predictor. The main difference is that a previous pattern of branches like *taken, taken, taken, not taken, taken*, if it's the same as the current pattern (for $k = 5$ in this example) selects the prediction, rather than the address of the branch. We still use the branch address as an index into the branch history, but use a global pattern table. Figure 4.14 illustrates the basic idea of the scheme. This scheme with a 512-entry 4-way associative history table was shown to have 97% accuracy in predicting branches in the original Yeh and Patt [1991] study.

A final wrinkle on branch prediction is *speculative execution*. If a branch

outcome cannot be determined in time to keep the pipeline busy, in a speculative machine, instructions that may have to be discarded are executed, with results in shadow registers, that are copied to the real registers when the instructions are committed. If the branch prediction is incorrect, the speculated instructions are discarded [Lee et al. 1995; Hiraki et al. 1998; Krishnan and Torrellas 1999]. Speculative execution can include speculative loads [Rogers and Li 1992] and even threads [Martínez and Torrellas 2002; Ceze et al. 2006]. Any memory access that's speculative should not cause a page fault, as that's a huge overhead compared with a pipeline stall, so implementation of speculation is very complex, yet it made it to commodity designs like the Pentium 4. The Pentium 4 could issue 3 instructions per clock but could have up to 60 instructions pending issue at any one time [Sohi 2001].

With all of this out of the way it now becomes possible to explore a reasonable level of ILP in a superscalar architecture. Paper exercises similar to that of 4.12 are instructive, though real design studies showing the effects of cache misses, TLB misses and unavoidable stalls for dependences show that in practice, it is hard to achieve much more than two instructions per clock on average.

Compiler-Exposed ILP

One more idea is to have the compiler expose ILP. A pioneering approach to this is packing multiple instructions in one long machine word. This idea, called *very long instruction word* (*VLIW*) was used in the Multiflow machines of the late 1980s. The intial design had 256-bit instructions containing 7 operations, followed by a more ambitious design with double the instruction width and 14 operations per instruction word. The idea was that a compiler technique called trace scheduling would expose enough ILP to fill a high fraction of the operations with useful work (otherwise a null operation or `NOP` had to be inserted) [Colwell et al. 1990].

The Intel IA64 was designed based on similar principles. Despite considerable money being thrown at the project not only by Intel but partners like HP, performance was disappointing. One of the lead architects on the Multiflow project, Bob Colwell, on joining Intel, led the design of the Pentium Pro and successors [Colwell and Steck 1995], negating the possibility that he could share lessons from Multiflow[7].

[7] I ran into Colwell on an online forum and he informed me that former Multiflow employees were not permitted to work on IA64 at either Intel or their partner HP fear of IP leakage.

The supposed gain of VLIW is to remove the hardware complexity of dynamic scheduling by having a smart compiler that can expose ILP. So why did VLIW fail? You could argue that Multiflow failed because it was a startup, and that what startups do: so why couldn't Intel and HP get it right? Mainly because ILP is not only dependent on statically-determined dependences. Memory delays are also a factor and, in the area where the IA64 was competing, cache misses are a significant factor. If any instruction in your long word has to stall for any reason, all the rest must stall, unless you go back to where you started, hardware to support dynamic scheduling. Multiflow avoided that particular problem by not using caches, not a practical approach for a general-purpose architecture. It was also focused on the high-performance computing market and floating-point code of that type typically has more ILP than integer because of long sequences of computation without branches. Generally, IPC varies a lot given cache misses, once of the arguments for moving away from aggressive pipelines to single-chip multiprocessors [Olukotun et al. 1996] – now known as multicore.

The IA64 included a few other innovations like bits that the compiler could set as hints to the hardware on available parallelism and *predicated instructions* [Tyson 1994]. A predicated instruction is tagged with a condition that must be true otherwise the instruction is not executed. Predicated instructions are intended to avoid the overheads of branches in short sequences of conditional code.

Despite the innovations, the IA64 was a market failure and the time when Intel was focused on that approach allowed AMD to dictate the design of 64-bit extensions to the IA32 architecture [Keltcher et al. 2003].

4.3 Summary

Increasing ILP was the key focus of computer architecture in the 1990s. Much of what I describe here was implemented in increasingly aggressive forms. By 2000, it was starting to become apparent that aggressive ILP was hitting limits and any new attempts at increasing ILP would have limited gains and significant costs in energy and heat – and clocks become increasingly hard to scale as the total wiring on the chip increases [Agarwal et al. 2000]. IBM's Power5 CPU pretty much did everything: it had 8 execution units, with a peak issue rate of 8 instructions per cycle (one per unit), hardware multithreading and speculative out of order execution and two cores [Kalla et al. 2004]. Unlike its predecessor the Power4 [Tendler et al. 2002], Power5 did not lead to mass-market designs, as IBM and Motorola lost the Apple account, the one major market for PC-scale

CPUs outside the Intel camp. At that point, it seemed that RISC had lost to CISC, though a better explanation is that aggressive ILP had peaked.

The case made by Olukotun et al. [1996] – and repeated in a special issue of *IEEE Computer* [Nayfeh and Olukotun 1997] on what to do with a billion transputers – is that more ILP should give way to more cores. That argument over 20 years later still looks good.

Exercises

In all examples where code is required, use the following:

```
for (int i = 0; i < N; i++)
    b[i] += a[i];
```

Translated into:

```
            addi R1,R0,0  # i = 0
            addi R4,R2,0  # copy base address of a
            addi R6,R3,0  # copy base address of b
            j forTest001
forBody001: lw R5,0(R4)    # get value of a[i]
            lw R7,0(R6)    # get value of b[i]
            add R7,R7,R5   # calculate a[i]+b[i]
            sw R7,0(R6)    # update b[i]
forNext001: addi R4,R4,4   # increment address of element of a
            addi R6,R6,4   # increment address of element of b
            addi R1,R1,1   # increment loop count
forTest001: blt R1, R8, forBody001
```

1. Do a pipeline timing diagram of the given example under the following assumptions:
   (a) No stalls.
   (b) Maximum stalls, based on when values are available (as in Section 4.1) and without any forwarding.
   (c) Stalls without maximum forwarding.
   (d) Unroll the loop once (two copies) and show how to reduce stalls by reordering and (if necessary) register renaming.
   (e) Can this example benefit from Tomasulo's algorithm? Explain.

2. Rework the pipeline timing diagram of the given example under the following assumptions:
   (a) We can issue any pair of instructions, subject only to dependences; assume the most aggressive achievable model of forwarding.
   (b) Add now the ability to do register renaming; assume the hardware is smart enough to rename any register after a write to it and no limit to the number of virtual registers (do you run out of paper?).
3. Now go back to the simpler dual-issue pipeline without register renaming and evaluate the effect of a simple 2-bit branch predictor.
   (a) Will a more sophisticated scheme like a 2-level adaptive scheme make a difference here? Explain.
   (b) Now assume we have an `if` statement in the loop that only does the assignment on odd values of `i`. Write out the assembly language for this case (you may fudge some details as long as the branches are plausible). Will a more sophisticated branch predictor help in this case? Explain.
4. Assume we have a floating-point pipeline in which a multiply takes 2 cycles and a divide 4 cycles (both in the execution stage; other stages are the same as for integer instructions). Explain how these instructions introduce new types of hazard not present in the integer pipeline and why they present problems for interrupts.
5. Look up how precise exceptions or precise interrupts are handled in multiple-issue implementations.
6. VLIW was based on the premise that a compiler technique, *trace scheduling*, could expose significant ILP. Look up trace scheduling and analyse its strengths and weaknesses.
7. The Pentium Pro introduced a concept of cracking complex instructions into simpler micro-operations, or μ-ops[8]. These μ-ops could be pipelined much more easily than the native instruction set.
   (a) Digital Equipment Corporation tried a similar idea with the VAX [Clark 1987]. See if you can find out what happened to that attempt. Can you still buy a VAX machine today?
   (b) Explain how this feature helped to bridge the gap between RISC and CISC and why Intel could not have done this with earlier designs.

[8]Pronounced "mu-ops" in deference to the pronunciation of the Greek letter μ.

5 Multiprocessors

If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens? – Seymour Cray.

MULTIPROCESSOR SYSTEMS ARE NOT A NEW CONCEPT – what is comparatively new is multicore designs. Multicore systems are not fundamentally different from older multiprocessor systems. They have two major advantages: lower cost and lower-latency interprocessor communication. Otherwise they present many of the same performance and software challenges.

In the days of big-iron multiprocessor systems, many models of parallelism were explored and the winner was shared-memory multiprocessors. I review here are few of the other variations, then focus on shared-memory systems and relate the general field to current multicore designs. I save other models of parallelism currently in use, vector instruction sets and GPUs, for the next chapter, since they are significantly different in implementation and efficiency issues, and only briefly review them here. s

5.1 Multiprocessor Models

Models of multiprocessor classically have been defined by whether they have more than one instruction stream, more than one data stream, or both:

- *SISD* – single instruction single data stream: a uniprocessor
- *SIMD* – single instruction multiple data stream: vector architectures for example, but there are other types
- *MIMD* – multiple instruction multiple data stream: more general types of multiprocessor, which run multiple threads or processes each relatively independent of each other

It's not clear that MISD – multiple instruction single data stream – makes sense. But maybe in a universe of rapid real-tme data flows that have to be processed

immediately because they can't be stored, that could change[1].

Another classification that cuts across these to some extent is memory organization:

- *shared memory* – all processes can access a single global memory (limited by protection in the operating system)
- *distributed memory* – processes have local memories that cannot be directly accessed; there are two models for distributed memory programming:
  - *message passing* – all communication is by messages similar to those you'd send over a network
  - *distributed shared memory* – the effect of a single global memory is faked using software, often using a combination of the virtual memory system and networking

The shared-memory plus MIMD model proved to be most popular because it most easily adapts to a variety of workloads, including multitasking a large number of single-threaded processes. It's possible to program in a message-passing style on a shared-memory machine, while distributed shared memory needs operating systems support for efficient implementation. In that sense a shared-memory machine is more general than a distributed-memory machine. MPI, now in common use as OpenMPI [Gabriel et al. 2004][2], is a message-passing API that can work efficiently on a variety of architectures, including networked systems and shared-memory systems.

Examples of vector additions to standard instruction sets include

- MMX [Peleg et al. 1997], SSE (Streaming SIMD Extension) extensions to the IA32 instruction set and successors (SSE1, 2, etc.) and AVX [Firasta et al. 2008]
- AltiVec extensions to the PowerPC [Diefendorff et al. 2000]

In one of the more extreme examples that has made it to a commodity product, the Cell processor designed by IBM, Toshiba and Sony has 8 vector units, each with a local memory. The Cell seemed to be an attempt at recreating all the hardware design errors of the past. Vector instruction sets only work well on specialised workloads, local memories put a lot of load on the programmer to get the right data to the right place at the right time and combining vector units with another model of parallelism (multiple cores) is an untried programming model.

[1] The Square Kilometre Array radio telescope project, once fully implemented, will generate data on such a massive scale that it will be impractical to store if it can't be processed in near real time [Wang et al. 2020] – but is is one stream of data or many?

[2] See also `http://www.open-mpi.org`.

The Cell was designed with two purposes in mind: developing HDTV codecs and the Playstation 3. For the former, it had prospects of success because computation is highly regular. Despite exaggerated expectations [Macedonia 2004], a handful of games developed specifically for the Playstation 3 was available at launch and it was notoriously difficult to program.

SIMD systems take two forms: applying the same operations at multiple CPUs and applying the same operation to multiple registers grouped together as a vector register. Early large-scale supercomputers such as those made by Cray were vector machines and had refinements like applying the same memory operation to sequential addresses, or locations with a fixed distance (*stride*) apart. Vector registers are common in GPU and similar instruction sets, such as the vector extensions of the Intel and PowerPC instruction sets. Vector instruction sets save a lot of time in avoiding the need to process multiple instructions and take advantage of high bandwidth of sequential or other regular memory access patterns as well as the speed of registers. However, they rely on problems that are well suited to highly regular computation on sequences of data.

In the past, there was another class of SIMD machines that were described as "massively parallel", exemplified by the Thinking Machines CM-1 and CM-2 ("CM" for "connection machine") that had up to 64Ki relatively simple 1-bit processors that could work simultaneously on the same instruction on different data; the effect was of 2048 parallel 32-bit integer processors. In addition to 1-bit integer processors, the CM-2 had floating-point units and the last model made, CM-5, gave up on 1-bit processing and used Sparc CPUs (a RISC design – but still programmed in SIMD mode, with an external controller that streamed the same instructions to each CPU). These machines seldom came close to their peak throughput and were notoriously hard to program. The nodes were arranged in a *hypercube* [Womble et al. 1999], a structure designed to minimise distances between nodes while also minimising the total number of interconnections.

Since GPUs have taken on a new life as an alternative to conventional performance-oriented architectures, I consider them separately. SIMD and vector architectures feed into the design of GPUs, so I add a little more detail as applies to GPUs in the next chapter.

5.2 Shared Memory Principles

Shared-memory systems have significant performance advantages over distributed memory systems up to the point where they run into scalability issues (though

you can argue that distributed memory systems only appear more scalable because they are unsuited to problems with a large amount of interprocess communication, IPC – not to be confused with IPC for instruction-level parallelism). Nonetheless shared memory can cause significant performance penalties if it is not well understood. Those issues start with performance problems generic to memory hierarchies and extend to those specific to shared memory.

In what follows, I talk about a "CPU" as synonymous with a core, since there is no logical difference.

First, let's review some memory hierarchy basics. At the top level, registers are specific to a CPU and not an issue for sharing. The TLB too tends to be specific to a CPU and isn't specific to multiprocessing[3], though failure to understand the TLB can cause major performance problems. Once we get to caches, we start to run into significant performance problems. Even though the L1 cache may be local to a CPU, we need to take into account shared memory and ensure that the caches remain consistent.

Maintaining *cache coherence* is one of the bigger problems of shared-memory multiprocessors. In addition to the usual cache tag scheme where we need a sufficient portion of the address to determine what memory locations a block represents and status bits to indicate validity and whether the block is dirty, we also need to know if a block is shared. The simplest way to do this would be to add a shared bit. However, keeping track of whether the block is not shared (*exclusive*) is a useful addition, because a non-shared block can immediately change to modified (or dirty) without waiting for any other caches to report back. One of the most common cache protocols is called MESI for having 4 states, modified, exclusive, shared and invalid. MESI is specifically well suited to a *write back* cache, i.e., one where blocks can be dirty. If a block is written *through*, i.e., all modifications immediately go to the next level down, a different protocol is needed. However, write-through caches are not in wide use and have seldom been used in real systems [Archibald and Baer 1986]. Early designs with relatively slow CPUs used write-through caches (e.g. some early Sequent systems – a company with a brief period of success mainly in the database server market) but they do not scale to faster designs, as the number of writes can easily saturate the bus.

Here is some common terminology:

- *multilevel inclusion* – bigger low-level components of the memory hierarchy include everything in the smaller higher levels (especially caches):

[3]This is not strictly accurate since shared memory involves sharing a page table, but the performance issues of a TLB tend not to be significantly exacerbated but this effect.

this makes coherence a lot easier to manage as absence in a lower level automatically means absence in a higher level; caches without inclusion have the *subset* property

- *snooping* – each cache controller watches the shared bus for transactions that relate to its content; snooping doesn't scale to very large systems and various *directory* schemes have been developed for very large shared-memory systems.

There are several variations on how cache coherence is implemented in practice. Using *snooping*, each CPU's cache controller watches for activity on a shared bus and either intervenes in other caches or modifies the state of its own if necessary. The MESI protocol is designed to reduce the need for snooping, because once a block is marked exclusive in a cache, the owner need not broadcast any actions on that block. It must however react if any other cache broadcasts an action. Let's examine in detail how the MESI protocol works in a variety of scenarios (from the point of view of a specific block in one cache – see Figure 5.1). In each case, assume that a miss results in initiating a read from main memory and this is aborted if another cache has a copy. If 1 other cache has a copy, it puts it on the bus for the requester; if it's shared, the owners race to put a copy on the bus.

In examples here I do not go into a lot of detail of when snooping is required. It can become a very complex topic as architectures become more sophisticated [Wolff and Porter 2020].

- *read*
  - *hit* – no action

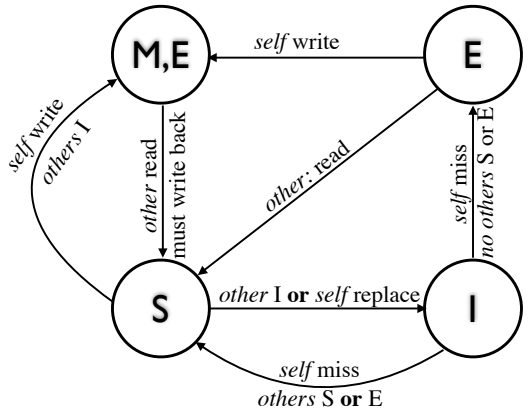


Figure 5.1: MESI state transitions. *Viewed from the perspective of one block in one cache: actions above the line trigger a transition, with consequences or explanations below the transition lines.*

  - *miss* – state currently I (if not, see replacement below)
    * *no copy (no other cache responds to snoop)* – state $\rightarrow E$
    * *another cache S* – state $\rightarrow S$
    * *another cache E* – state $\rightarrow S$; snoop makes owner set its state $\rightarrow S$
    * *another cache EM* – state $\rightarrow S$; snoop makes owner set its state $\rightarrow S$; write back to main memory
- *write*
  - *hit*
    * *state EM* – no action
    * *state E* – state $\rightarrow EM$
    * *state S* – *invalidate* signal sent on bus; state $\rightarrow EM$
  - *miss*
    * *no copy* – state $\rightarrow EM$
    * *another cache E or S* – state $\rightarrow EM$; *invalidate* signal sent on bus
    * *another cache EM* – state $\rightarrow EM$; snoop makes owner set its state $\rightarrow I$; write back to main memory
- *replacement* – what we do to a block we evict from the cache (on a read or write)
  - *state EM* – write to main memory, continue as for miss
  - *state E or S* – no action, continue as for miss; if state is S and only 1 other cache holds the block, it will still hold it in state S

The protocol doesn't have a way of turning a block that's shared back to exclusive if all other processors lose theirs. To do so, we would have to broadcast on the bus every time a shared block was evicted and keep a count of sharers. Note also that "main memory" really means the next level down and in current multicore designs is usually a shared L2 or L3 cache. This version minimises copying from main memory; more conservative designs copy to main memory whenever a copy is requested from another cache.

The Intel Nehalem architecture (launched with the Core i7, late 2008), with the major functions illustrated in Figure 5.2 and the die layout in Figure 5.3, is an example of a design with shared caches, in this case, L3. The version illustrated is from the first series with 8MiB of L3 cache; in more recent designs with 12MiB of L3 cache, caches would take up more than half the real eastate of the die. In the Nehalem design, the MESI protocol routes most requests via the L3 cache rather than having direct transactions between L2 caches, but is extended to something closer to the above, with an extra feature confusing called *forwarding*, making it a MESIF protocol, implementing the scheme I describe where caches forward a

value to another that requests it rather than going via main memory (or in this case the L3 cache). The forwarding feature is limited to processors outside a single multicore unit. In local core-core cache transactions, the L3 cache acts as a central repository for transactions with tag bits indicating the state of blocks in the individual cores, reducing the need for snooping [Molka et al. 2009].

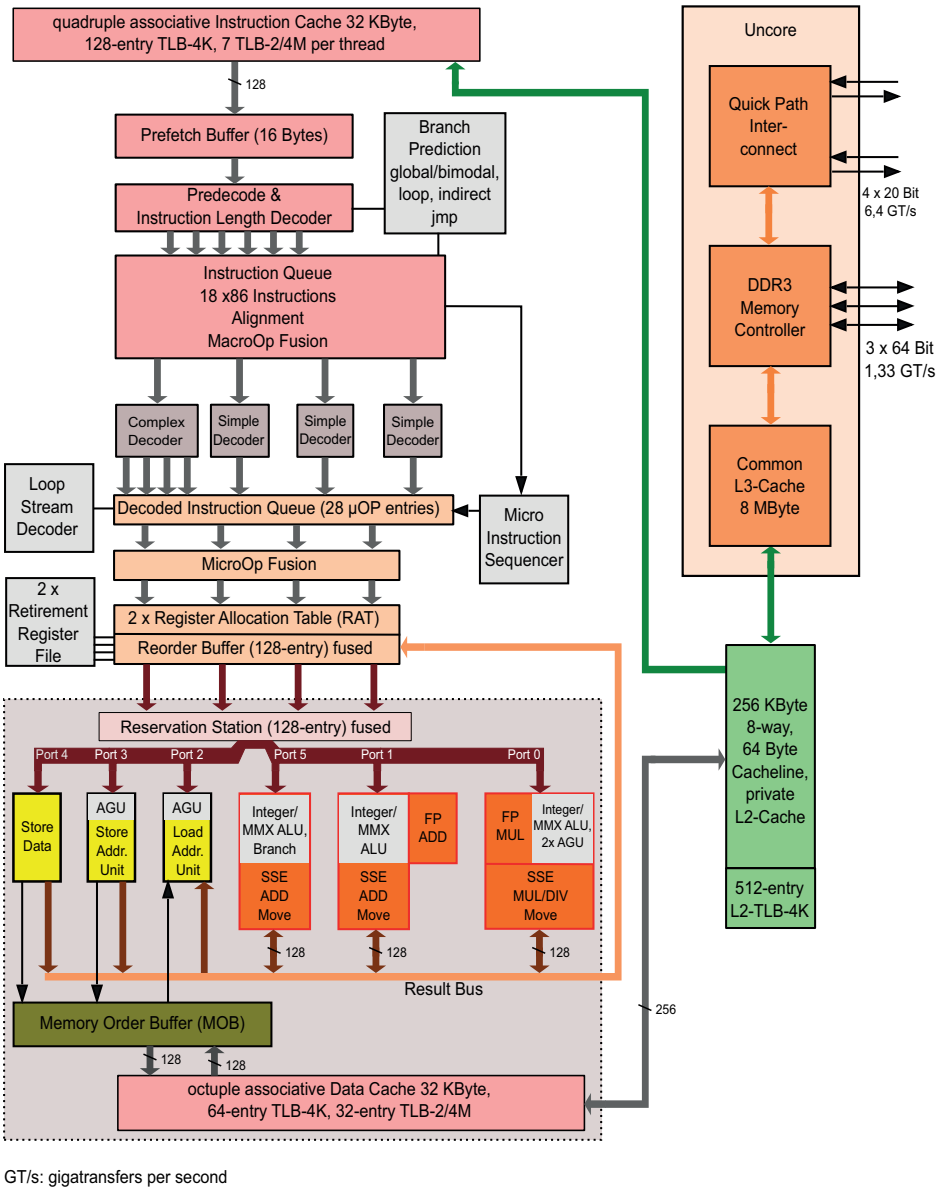


Figure 5.2: The Intel Nehalem archictecture. *Source: http://en.wikipedia.org/wiki/Nehalem_(microarchitecture).*

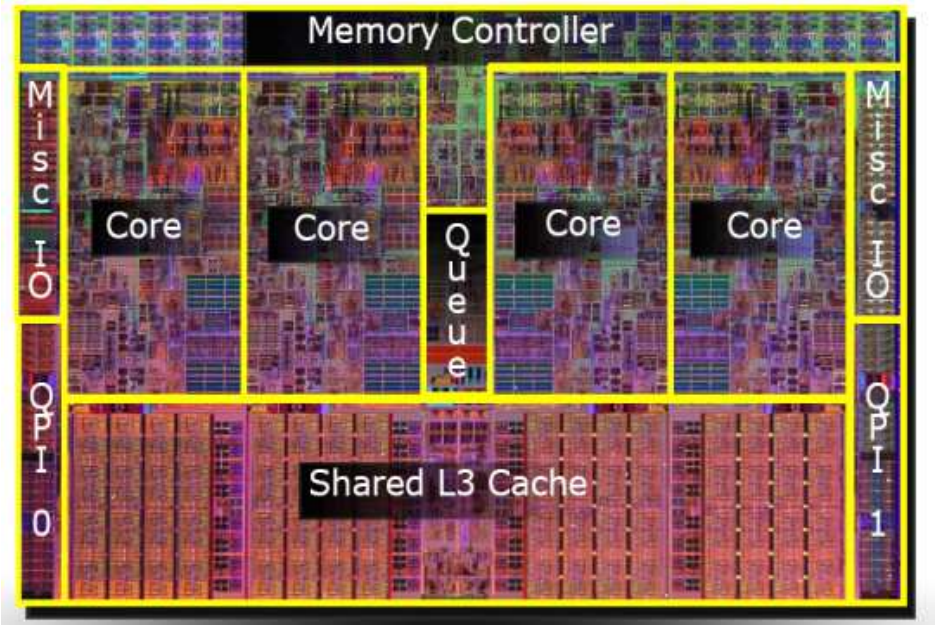


Figure 5.3: The Intel Nehalem die showing major components. *Source: http://arstechnica.com/uncategorized/2008/11/intels-3-2ghz-monster-nehalem-roars-onto-the-scene/.*

5.3 Shared Memory Performance

There are many performance factors in a shared-memory system. The less sharing there is, the fewer problems there are with scaling. Some problems are avoidable with careful programming, but any workload with a high rate of communication between components will not achieve a good speedup on any multiprocessor system.

Here are a few key factors in performance of shared-memory multiprocessors, that apply (almost) equally to multicore systems:

- *high rate of sharing* – as you should be able to see from the MESI protocol, modifying a variable in one CPU (or core) then reading it in another creates significant bus traffic. Even if you don't need to wait for the write to the lower level to complete, you need to wait for the other cache to broadcast on the bus. That, in some systems, may not be a huge penalty compared with waiting for the lower level of memory. Still, if it happens a lot, the bus can saturate.
- *false sharing* – if two or more variables that are actually not shared are in the same cache block, the coherence protocol doesn't know that: it only

sees whole cache blocks; in this scenario a lot of unnecessary delay and bus traffic can result

- *contention for locks* – if a lock is implemented as a simple spinlock relying on the cache coherence scheme to ensure that updates are propagated, the amount of bus traffic when a lock is released and set by one of several contending processes can be very high

These factors are in addition to the usual problems of scaling up a multiprocessor workload: load balance (ensuring the work is evenly split) and ensuring program correctness.

False Sharing

Let's use numbers from a real system, an Intel Nehalem design. I list some key numbers in Table 5.1. A few things need explanation: *snoop latency* is the extra time L3 must take before responding to a miss if a block is exclusive in a higher-level cache, since it must also check if the block has been modified. If a block is shared, L3 can immediately provide the block to the missing cache with the *shared access* latency. In this scheme, misses in L1 or L2 are handled out of L3, rather than the more aggressive scheme suggested in my definition of MESI. The reason for this is to relieve the high-level caches of the need to service requests from other caches. Since L3 is relatively fast compared with DRAM, this is a reasonable trade-off to avoid either the complication of another port on the L2 caches or forcing them to stall if they have a competing request from another core as well as servicing their own L1 misses. The Nehalem architecture includes other features we do not explore here including a fast interconnect for building multi-chip multiprocessor systems.

Given the Nehalem numbers, let's consider costs of cache misses and performance bugs such as false sharing. Suppose we have two sequences of code on two cores that each modify a separate variable that's in the same cache block. Let's take a short sequence of code in a loop as our example (using our simple RISC instruction set but with the Nehalem latencies):

#	**core 0**	**core 1**
1	`lw R6,0(R2)`	`lw R6,4(R2)`
2	`add R6,R6,R5`	`add R6,R6,R5`
3	`sw R6,0(R2)`	`sw R6,4(R2)`
4	`addi R4,R4,4`	`addi R4,R4,4`
5	`cmplt R8,R4,R7`	`cmplt R8,R4,R7`
6	`brne R8,R0,-20`	`brne R8,R0,-20`

level	latency	
	cycles	*ns*
L1	4	1.4
L2	10	3.4
L3	38	13
DRAM	191	65
multiprocessor overheads		
shared access	38	13
snoop latency	27	9.2

Table 5.1: Intel Nehalem latencies. *These are for a specific model, an Intel Xeon X5570 with core frerquency 2.933 GHz (cycle time hence 0.34ns), as determined by Molka et al. [2009]; at a different clock speed latencies will vary.*

These two examples pretty much do the same thing, except one has a variable at offset 0 from the address pointed to by R2 and the other a variable at offset 4 from the address in R2. The two cores' registers are independent, though I assume here that R2 has the same value in both cores. Unless we are extremely lucky and R2 is pointing at the last 4 bytes of a cache block, we will get false sharing here (assuming blocks are more than 4 bytes).

At the start of the loop, to keep things simple, neither core has a copy of the cache block and it is in the shared L3 cache and that they are running in lock-step until one has a cache miss. At instruction number 1, both processors have a

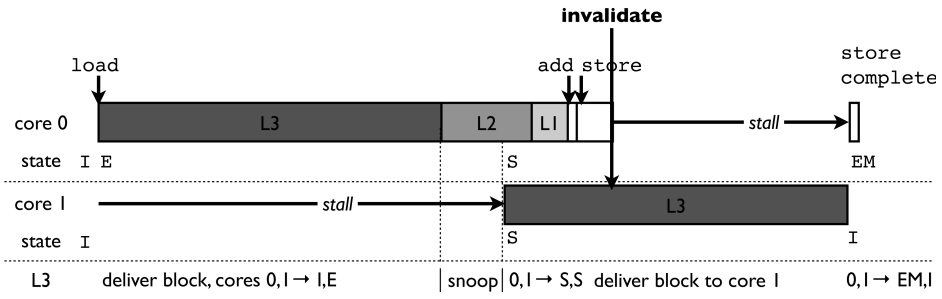


Figure 5.4: False sharing example. *Core 0 wins the race to acquire the block and reaches instruction #3 before core 1 completes handling its miss to L2. L3 maintains global state of shared blocks and issues a snoop on the bus to inform core 0 that its copy of the block is changed to state S before if can supply the block to core 1. Core 0 is stalled on cycle 4 of its store (#3) because it can only invalidate a block once the bus is free, before it can modify it. Latencies are drawn to scale; shading representing the memory hierarchy (darker ≡ slower).*

miss and whichever acquires the bus first issues a miss to L3. We must add the latencies for L3, L2 and L1; once the block is copied from L3 to L2, the shared bus is released and the other core can access the block from L3. Since the outcome doesn't differ, let's assume core 0 wins the race. The sequence of events starts out as in Figure 5.4 and goes downhill from there. After the illustrated steps, the bus protocol should allow core 1 to acquire the block (forcing core 0 to write it back so the state can be *S* again) to complete the load and it can then do the add (#2), since that only involves registers, even if core 0 invalidates core 1's copy of the block again. The chances are core 1 will have another miss when it tries to write the block at which point it retaliates by forcing core 0 to give up the block (writing it back as well, as part of the write miss from core 1). It's a useful exercise to calculate in this scenario how long it takes for both loops to complete as few as 10 iterations.

So how can we avoid this scenario?

One approach is to use processes rather than threads, with explicitly allocated shared memory and take care that you manage how data structures are used between cores or CPUs. While processes are heavier-weight entities than threads, it doesn't take a lot of inadvertent false sharing to cancel out the gains. Another approach is to pad all variables to a size that's a multiple of the cache block size, making sure that all variables that could be shared start on a cache block boundary. To do this, you may need your own memory allocator.

When I am not concerned with issues of differentiating processes and threads, I refer to *tasks*, the Unix name that encompasses both.

Locks

For multithreaded and multi-process programming, we need *mutual exclusion* to ensure predictable behaviour of updating shared variables. For example, if we have a global counter and two threads update it, we don't want a scenario where one thread loads it into a register, another thread also loads it into a register, then each increment the variable and store it back in memory, resulting in the counter only increasing by 1. A result that depends purely on the order competing tasks complete an action is called a *race condition* and is usually a programming error. Since synchronization is critical to parallel program correctness, we need efficient implementations so synchronization doesn't become a bottleneck.

To implement locks, atomic memory operations are required. A common one is *atomic swap* – in one uninterruptible instruction, the contents of a register and

a memory location are swapped.

An argument against using these primitives for a lock is that they can result in an *ABA* problem – if you look at the same memory location twice and see the same value, it may have changed twice (originally *A*, changed to *B* then back to *A*). If you really want to be sure that you are seeing the same value because it hasn't changed, another option in some instruction sets is the pair of instructions *Load Linked–Store Conditional* [Michael 2004]. The first of these, `ll`, is a load that marks the memory location as one that must be watched for changes; the second, `sc`, stores conditional on the memory location not having changed. A `sc` instruction has a destination register field that is set to 0 on success and a non-zero error code on failure. There is a limited time that the "reservation" holds, after which the `sc` automatically fails. There is also a restricted list of instructions that may appear between the two to minimize chances of breaking the design decisions behind the pair of instructions.

For purposes of understanding memory effects, our atomic swap is sufficient since the memory state changes are the same (excluding the need for "reserving" a memory location). Other atomic operations that may be useful include

- *test-and-set* (`tas`), in which a store only completes if memory contents matches a value provided to test against
- *compare-and-swap* (`cas`), like `tas` except a swap is performed, not a store
- *load-and-increment* (`lai` – named for consistency with RISC terminology; *fetch-and-add* is a common name for this operation), in which a value is retrieved from memory, an add is performed and it is stored back to memory; in RISC terms, it corresponds to an atomic sequence of a load, an add and a store

Simple Nonscalable Locks

A simple lock structure, a spinlock, is based on an atomic memory operation. There are several that can work. Earlier architectures used *test and set*, that tested a value for a specific condition and set it based on the outcome, and the instruction was guaranteed to complete without interruption (or to have that effect). From here on, I base examples on the more recent *atomic swap*, which allows swapping contents of a memory location with a register. Again, the operation is guaranteed to complete without being interrupted. A spinlock using an atomic swap operation could look like this (assuming there is a location in memory that's initially 0, with its address in register `R5` and `R6` is initialised with 1):

```
lock: swap  R6,0(R5)
      bne R6,R0,lock
```

If our swap operation gets back a 0, that means the lock wasn't previously held and we've now set it. If it's not a 0, we have to try again. For either outcome, we set the value of the lock to 1. If someone else held it already, we don't effectively change it, since it would be 1 already. This tight little loop continues until whoever else got in first releases the lock, which is done like this:

```
      sw  R0, 0(R5)
```

We rely on cache coherence to ensure that locking an unlocking is serialised and updated to each processor or core when the lock variable's value changes.

The spinlock strategy looks nice and simple. The bad stuff happens when a lot of tasks are trying to acquire the lock, particularly where every attempt at modifying it starts with the lock variable in the modified state in another cache. By contrast, in the best case, there is no contention at all on the lock: if you modify a block you already have as exclusive, all writes are local.

In a high-contention scenario, each time a task tries to acquire the lock, since the swap instruction is a memory write, the modified block must be invalidated out of any cache that holds it. The first core or CPU that tries to issue a swap instruction will cause an invalidate, unless it already has the block in the exclusive state. An invalidate of a modified block forces a write back, then the winning CPU gets an exclusive copy in its cache, where it will immediately write to it. Even if the effect of the write is to overwrite the lock variable with the exact same value as it had before, it's still a write. Every other process trying to acquire the lock will experience a cache miss and queue up on the bus. If the process that holds the lock meanwhile has a cache miss for any other purpose, it will also have to wait its turn for the bus, further slowing things down. When finally the holder of the lock releases it, it will also have a miss since it will now be exclusive and modified in another cache. The releasing task must invalidate the lock variable from any other cache that has a copy and modify it, causing a flurry of invalidates as each contender races to be the next one to acquire the lock. To make things worse, a spinlock does not ensure *fairness* – acquiring the lock depends on the race to swap the lock value, not the order the tasks arrived at the lock. This can result in *starvation*.

If you think this sounds pretty bad, you'd be right. That's why there has been considerable research into more scalable locking strategies. Before we get there, I consider some relatively simple improvements on a spinlock.

A *test-and-test-and-set* (TTS – named after the atomic operation common at the time it was implemented – my example uses atomic swap) lock spins on the lock value without trying to set it; it then tries to set it and goes back to the spin loop if it fails:

```
lock: lw  R4,0(R5)
      bne R4,R0,lock
      swap  R6,0(R5)
      bne R6,R0,lock
```

This variant saves a lot of write traffic but can still caused a flurry of invalidations when the lock is released if a number of tasks are spinning and all fall through to the swap instruction. For a very small number of tasks, it may even be slower than a spinlock because the code is longer [Machanick 2018a]. Also it does not fix the problem of starvation and it does not scale much better than a plain spinlock.

A *ticket lock* is based on the idea of taking a number when you arrive at an organization like a bank that establishes your place in the queue. In its simplest form, it improves on a spinlock by establishing a first-in-first-out (FIFO) ordering. To acquire a unique number, an atomic load and increment operation is required: you obtain a copy of the next value of the ticket while incrementing it. The task then spins on a global value that contains the value of the lock that is currently held. If you are first in, that value corresponds to the value of the ticket you just picked up. FIFO ordering is an improvement because it results in fairness (i.e., no starvation). Memory traffic is lower as well. While each waiter is spinning on the same shared variable, they are not trying to update it while spinning – only the lock holder resets the global lock value. It is also possible to implement *proportionate back-off*: rather than a tight spin loop, it is possible to wait a time interval proportional to the difference between your ticket value and the value of the lock holder.

A spinlock can also use *exponential backoff* – each time lock acquisition fails, it waits a longer time (e.g., doubling the delay each time, an exponentially-increasing delay).

None of these variants, however, scale well for large numbers of tasks because the number of shared accesses scales linearly with number of tasks.

More scalable Locks

The key issue then is reducing shared accesses. If you add in a requirement of FIFO ordering, the key insight is that a task needs only really check the status

of its predecessor in the queue. That reduces the need for shared accesses to determining your place in the order of tasks that try to acquire the lock. Once past that, a task needs only spin on a variable owned by its predecessor. Generically, this class of locks is called *queue locks* – even if the queue is sometimes not explicitly implemented, as long as the effect is FIFO ordering and only sharing information between a predecessor and successor, the label applies.

Why does this approach work? Once a task knows its predecessor, it can spin on the predecessor's instance of the lock and that variable will be cached until the predecessor releases the lock, so the waiting task does not cause any memory traffic while it is spinning, except a read miss on the first iteration of the spin loop and another miss when its predecessor releases the lock.

I start by creating such an algorithm from first principles, attempting to minimize shared accesses, then compare with published approaches. To minimize shared-memory accesses and atomic memory operations I need to work out which ones are actually needed. At minimum it is necessary for each contending task to:

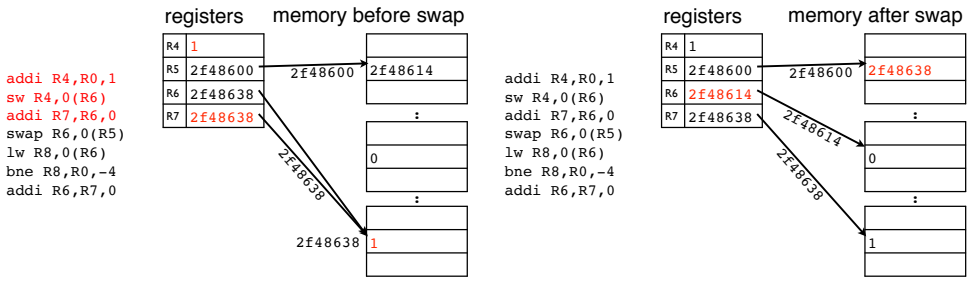
1. set its own lock variable to stake its claim
2. identify the predecessor to this thread (if any) in FIFO order and claim the next position
3. wait on the lock variable belonging to the thread that beat it to the critical section
4. enter the critical section
5. release its own lock variable

A simple generic implementation of a queued lock based on this list of requirements is as follows; since I aim to minimize contention, I call it *M-lock* (for minimal-contention lock).

Each contending task has to have a unique variable in shared memory, the address of which we store in register `R6`. We start out with the address of an initial instance of the lock in register `R5` pointing to a memory location containing the value 0, so the first contender to win will find the lock not set. There is an extra variable used to initialize the lock so for N contending tasks, there are $N+1$ lock variables representing a single lock. The extra variable is global to all tasks and acquiring a lock starts with swapping the current *address* of the local lock variable with the address in the global lock variable.

To lock:

```
addi R4,R0,1
sw  R4,0(R6)  # R6: address of local lock variable
addi R7,R6,0      # save for next time
```



(a) Just before swap. **(b)** After swap.

Figure 5.5: M-lock: a simple queue lock. *Contention for updates is reduced to grabbing the address of the last instance to hold the lock – either our predecessor or an extra variable set to unlocked (0) if we are the first thread to try to acquire the lock.*

```
      swap R6,0(R5)     # find out who beat us
test: lw R8,0(R6)
      bne R8,R0,test    # wait for the winner if any
      addi R6,R7,0 # restore the address of our lock
```

We release the lock as follows:

```
      sw  R0,0(R6) # release our lock
```

What we have done here is to set up our own lock variable with the value 1 and swap its address with whatever address is already stored in the global lock pointer. We now spin on the value at the address *previously* pointed at by the global lock pointer. The global lock pointer has to be initialised to point to a memory location with the lock value set at 0 so the first contender can get in. The effect of this approach is that each contender is spinning on a *different* lock variable that will be released only when the process or task that was immediately ahead of it in attempting to acquire the lock releases it. While there will be contention for swapping the address of our local lock with that stored in the global lock pointer location, that only happens once for each attempt at acquiring the lock. Subsequent spinning is on a memory location only known by the current stalled task and the owner of that location. Of course we need to ensure there is no false sharing though even this is not that important because we are spinning on a memory read rather than a write; if more than one core experiences an invalidation there will not be consequential series of follow-on invalidations. I illustrate the two crucial steps of the lock: initializing and the swap instruction in Figure 5.5. In Figure 5.5a, I illustrate the way state is set up in a scenario where the initial

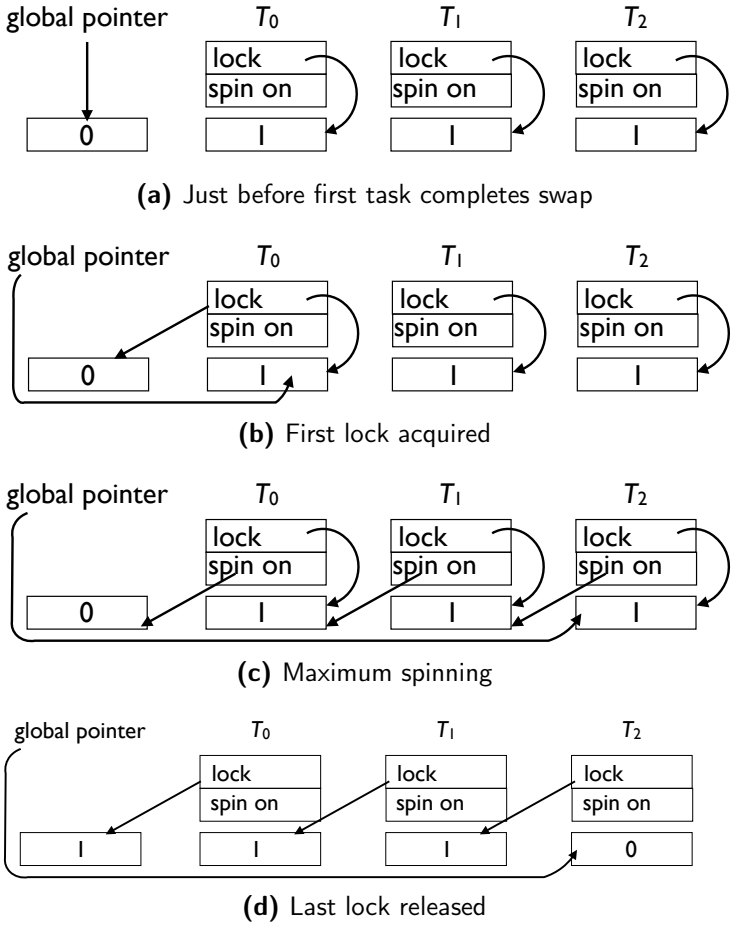


Figure 5.6: State of lock variables at the start and finish of a critical section. *Tasks are numbered in the oder that they attempt to acquire the lock. Each task swaps the global pointer with the pointer to the task's local lock. For T_0, this atomic swap is shown in two stages. When T_2 releases the lock, provided there is no other attempt at acquiring the lock, the global pointer points to what was previously the lock of T_2.*

lock is still zero, representing the case where the lock is not held. In Figure 5.5b, I show how the state changes when the current contender has acquired the lock.

Figure 5.6 illustrates the state of lock variables at various stages of lock acquisition, contention and release.

Queue locks are one of many approaches to scalable synchronization [Mellor-Crummey and Scott 1991]. Parallel programs use a range of primitives including *barriers*: a barrier with parameter N causes $N-1$ tasks to stall and when the Nth arrives at the barrier, all proceed. Implementing a barrier using spinlocks is very inefficient, even on a small number of processors and threads packages

```
typdef struct {
  LockT * next;
  bool locked;
} LockT;

void acquire (LockT *lock, LockT * I) {
  I->next = NULL;
  // atomically:
  // lock = I; return old value of lock
  LockT * pred = swap_ptr (lock, I);
  if (pred) { // queue not empty
    I->locked = true;
    pred->next = I;
    while (I->locked) ; spin
  }
}

void release (LockT *lock, LockT * I) {
  if (!I->next)
    // compare_and_swap true if swapped
    if (compare_and_swap (L, I, NULL))
      return;
    while (!I->next); // spin
  I->next->locked = false;
}
```

Figure 5.7: MCS algorithm. *It requires a* `compare_and_swap` *atomic memory operation to ensure FIFO ordering, as well as* `swap_ptr` *(described in the comment above its use) to implement atomic swap. Without this primitive, the release algorithm is more complicated and does not guarantee FIFO ordering.*

such as Pthreads implement barriers by queuing waiting threads and putting them to sleep. Even so the underlying implementation will be unscalable if it's based on spinlocks as a basic building block. One approach to implementing scalable barriers uses a tree structure [Markatos et al. 1991]; another is to synchronize with nearest neighbours, limiting global communication [Machanick 1996].

A widely used queue lock algorithm is that created by Mellor-Crummey and Scott [1991], called MCS after their initials. Study the code in Figure 5.7 – is it easier to understand than my simplified queue lock, M-lock? To make the comparison fairer, I present my M-lock algorithm in C code in Figure 5.8. The data structure, converted to C, looks more complex because it needs some double-indirection but the individual steps are simpler.

```
typedef struct {
    volatile uint8_t lockval;
} m_lock_t;

typedef struct {
    m_lock_t
      ** global, // points to global lock
      * mylock,  // points to lock owned by me
      * spinon;  // points to lock I spin on
} m_lock_node_t;

void m_acquire (m_lock_node_t * lock) {
    lock->spinon = swap_ptr(lock->global,lock->mylock);
    while (lock->spinon->lockval != 0) ; // spin
}

void m_release (m_lock_node_t * lock) {
  lock->mylock->lockval = 0;
  lock->mylock = lock->spinon;
  lock->mylock->lockval = 1; // ready for next time
}
```

Figure 5.8: M-lock in C. *Each local lock's value is initially 1; the global lock value is initially 0. Each local lock structure contains a pointer to the global pointer;* `swap_ptr` *is as in Figure 5.7.*

Does MCS gain speed in exchange for complexity? Unlikely [Machanick 2018a], as it requires a spin on release and uses two atomic swap primitives – one for acquire, one for release. This example illustrates that starting from machine code can be the best abstraction as trying to improve on the MCS lock based on its high-level language code does not make it obvious where to start in looking for improvements. MCS is a widely-studied algorithm, possibly obscuring other good ideas. M-lock is not original (the same algorithm has been independently discovered at least twice [Craig 1993; Magnusson et al. 1994]) – my point with this example is to illustrate how to get there from first principles.

5.4 Summary

Although there are many multiprocessor architectures, I focus here on shared memory MIMD architectures since they represent the mainstream of conventional computing. SIMD architectures in various forms have come and gone and remain persistent mainly because GPUs are such a large share of the market.

Shared-memory architectures will likely be with us for some time given that they are a natural organization for multi-core devices, because they accommodate so many different types of workloads, including parallel applications and multitasking. The performance issues covered here, before the multicore era, were primarily the concern of relatively high-end systems. Given that multiple cores are commonplace, understanding the performance problems and avoiding them is increasingly important both in user-level vode and system code.

There are times when a machine-code view points to performance enhancements i.e. a high-level language view is *not* always the best level of abstraction.

Exercises

1. Use the latencies in Table 5.1, and the timing illustrated in Figure 5.4. You can assume every instruction can complete in one clock if fully pipelined, a store modifies memory in the MEM stage (4th stage of the pipeline), and an invalidate requires the latency of a snoop for the invalidating core if the bus is not occupied.
   (a) Calculate the total time it takes before core 0 manages to complete the illustrated store instruction.
   (b) Now assume that core 1 acquires the block as soon as possible after core 0's store completes. Calculate the total time from the start of Figure 5.4 until core 1 completes its first store, assuming that core 0 continues with the loop with minimal stalls.
   (c) Calculate the total time it takes for 10 iterations of the loop for both cores, stating assumptions about timing of competing events.
   (d) Why is it a reasonable assumption that an invalidate requires a relatively short stall (here, 2 cycles), not a longer delay, e.g., the 27-cycle delay required for a snoop?
2. Will the M-lock as described on page 98 work as you expect if N tasks enter it and leave it, then try to re-enter at a later stage? Hint: what will the global

pointer have as its value, and what will be stored at that location?

3. Spinlocks are often used as a core primitive to implement more complex, scalable synchronization techniques like semaphores that put a process or thread to sleep and wake it when it reaches the head of a queue. Are spinlocks a reasonable choice in that scenario, or would you still look for a more scalable option like a ticket lock?
4. Is a test and set instruction superior or inferior to an atomic swap? Explain, considering the design philosophy of a RISC ISA.

6 GPUs

GPUs HAVE BEEN AROUND FOR A LONG TIME and represent an untidy mix of architectural ideas – so why are they worth considering separately? First, because they are a mix of architectural styles, they represent a case study in comparing the benefits and weaknesses of various models of parallelism. Secondly, because they are so widely available, there is more chance that, despite any difficulties in programming them, they may become established as an alternative platform for high-speed computation. It is that market, rather than the obvious one (given that the name means "graphics processing unit"), that hold some interest, because there are limits to how much further graphics performance needs to be taken. Once you can do realistic three-dimensional imaging in real time, where else can you go?

The fact that GPUs have become commodities left Silicon Graphics Incorporated (SGI), the leading player in high-end graphics, without a clear direction. For a while they were a big player in supercomputers and bought Cray for $740-million in 1996[1] and sold it off in 2000 for less than $100-million[2]. SGI finally went broke in 2009 and sold their remaining assets for $25-million[3].

General-purpose use of GPUs is abbreviated to GPGPU – a GPGPU application usually involves no graphics rendering, but exploits the high level of parallelism available on a GPU for computation. An interesting thought: if a GPU is made purely for computation, should we actually all it a GPU?

Seymour Cray died as a result of a car crash in 1996[4] so there goes the chance of a pithy quote from him on whether GPGPU is a good idea.

The idea of adapting a part designed for graphics processing to general

[1] `https://fcw.com/1996/03/sgi-to-acquire-cray-in-740m-buyout/246366/`
[2] `https://www.itweb.co.za/content/JBwEr7n56jRv6Db2`
[3] `https://www.infoworld.com/article/2634354/rackable-buying-sgi-for--25-million.html`
[4] `https://www.cgl.ucsf.edu/home/tef/cray/obit.html`

purpose computation is not new; as I describe on page 18, the Intel i860 featured as a component in large-scale supercomputers in the 1990s. I also note there that it was not a great success. Will GPUs be more successful as high-performance computation engines? If only because they are deployed on a vast scale whether used in that mode or not, there is a lot of ongoing investment in pursuing this question. Since the primary market for GPUs remains graphics, design compromises will tend to favour that application. Early attempts at using GPUs as compute engines ran into the problem that design compromises favouring speed in graphics rendering meant that CPUs could not in general be expected to implement the IEEE floating point standard as strictly as a general-purpose floating-point unit [Chinchilla et al. 2004; Meredith et al. 2009] (the odd wrong pixel is less noticeable than failing to render the next frame in time). Given the growth in the market for general-purpose use of GPGPU, manufacturers have started to pay attention to quality of their floating-point implementation [Krakiwsky et al. 2004; Whitehead and Fit-Florea 2011].

In this chapter, I briefly survey some of the architectures that contribute to the design of GPUs, adding to the discussion of Chapter 5.

6.1 Vector Processing

Vector processors fall into two broad categories: vector register architectures, and memory-based vector machines. The latter generally require vector registers to perform at a reasonable level, so I start with vector registers. Vector machines of the class of the designs created by Seymour Cray generally have very aggressive memory architectures. I briefly describe how these work; in the heyday of this class of machine, there was considerable research into designing memories for them [Cheung and Smith 1986; Weiss 1989; Valero et al. 1992; Seznec and Lenfant 1992], possibly a pointer to difficulties with vector register machines with no special memory architecture.

A vector register machine has registers that replicate a specific data type, and in some cases can be reconfigured as more lower-precision (narrower) or fewer higher-precision (wider) registers. For simplicity, I assume a vector register is comprised of a fixed number of scalar registers of a fixed size. To make the discussion concrete, I assume a vector register has length 64 (i.e., it can contain 64 distinct values). Vector architectures can generally either operate on a pair of vectors producing a vector result, or a vector and a scalar producing a vector result. In the latter case, the same scalar is an operand for each operation on the

source vector. For example, you may want to multiple each vector element by a constant.

In a case where the available vector length (here 64) is sufficient, you can use a single instruction to do the main work (e.g. multiplying all 64 elements by a constant, or adding all 4 elements to the equivalent entries of another vector). This single instruction has a latency dependent on how long the hardware takes to perform 64 operations. By contrast, if you use a scalar architecture, you will do the same 64 operations but need to wrap a loop around them, and add ancillary code for array indexing. The saving in the number of instructions executed in this case should be about a factor of 100. That is not as big a saving as it sounds, since the vector unit still has to do 64 operations, and those cannot happen instantaneously. However, contrast the requirements of speeding up the vector operations by adding more parallel hardware with doing the same for the scalar code. The scalar code has a loop, so you will need to unroll the loop, either by a coding technique (compiler optimisation or hand-unrolling it), or hardware support such as Tomasulo's algorithm. If you go the first route, you need to know in advance how many times it's worth unrolling the loop; in the latter case, you add significant hardware complexity including register renaming. On the other hand, to speed up the vector register code in hardware is relatively simple. You can add more functional units and provided there are no dependences between pipelined vector instructions, as many calculations as there are available functional units can start at once.

You can apply the same trick as with a scalar pipeline to reduce inter-calculation delays, forwarding a result to the input of a functional unit rather than going via the register file. In a vector architecture, forwarding is called *chaining*.

In a simple implementation of a vector machine, each ALU operation takes as long as in a single-issue scalar machine. The major saving is in fewer instruction issues and removing branches for loops. In the case of a simple loop with two ALU operations, in a typical RISC instruction set the loop body and condition would add up to about 10 instructions. For a 64-long register vector machine (without for now going into how the operands find their way to registers), the equivalent code would be about half the number of instructions and also would not require repetition. The scalar code would therefore require about 128 times the number of instructions, though the number of ALU operations would be the same, meaning that the practical speed gain would be relatively modest, especially if the ALU operations are multi-cycle floating point operations. The big gain from vector instructions comes from the extremely regular nature of the parallelism,

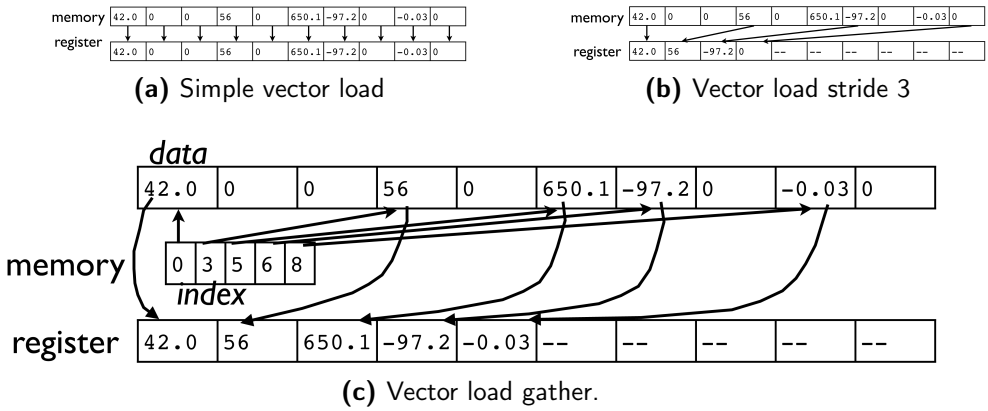


Figure 6.1: Variations on vector loads. *Stores have similar variations; the store version of a gather is a scatter. In stride mode, the load fetches data stride S apart. In gather mode, an index vector is used to find the offset from the start of the main array in memory.*

which makes it possible to split the work across multiple functional units without complications such as data and control hazards.

In big-iron vector machines such as those designed by Cray, vector ALU operations are accompanied by vector memory operations, which is where things start to get more interesting. In a simple example where the data size exactly matches the register length, a vector load instruction fetches the next N (64 in our example) elements of an array, sequentially from a given base address. Design for this case is straightforward and can exploit the fact that DRAM has efficient streaming modes[5]. There is a range of scenarios that cover cases where the data size is not an exact fit:

- *data shorter than the vector size* – one approach is to have a *vector length register* (*VLR*) that can be any value up to the hardware vector length; vector instructions' length is controlled by the VLR's value. Some machines also have a *maximum vector length* set in the hardware. The MVL can change in new hardware, avoiding the need to change the instruction set when the vector length changes
- *data longer than the vector size* – use the MVL value to create an outer loop that splits the problem of size N into the portion that is an exact multiple of MVL, repeated $\lfloor \frac{N}{MVL} \rfloor$ times, and the remainder (done once). This technique is called *strip mining*
- *elements not adjacent in memory* – we need a way to specify a *stride*, a

[5]Cray's early designs used SRAM for main memory so streaming was less of a time-saving.

distance apart of memory locations. Big-iron vector machines have this capability (e.g., a stride register could set the distance apart of successive elements for vector load or store)

- *sparse vectors* – in a case where a data structure has a lot of zeros (or other elements not of interest), it may be stored using indirect indices, e.g., element `i` is found at `A[index[i]]`. To handle this scenario, vector architectures may use *gather-scatter*:
  - *gather* – use an index vector to add to a base address to do loads
  - *scatter* – use an index vector to add to a base address to do stores

Gather-scatter can be used to save memory, if the index array is a smaller data size than the actual data and also as a way of accessing memory in different orders without having to sort the original data each time a different ordering is needed.

To implement all of these operations with an aggressive vector CPU that can do multiple ALU operations per cycle requires high memory bandwidth. To get some idea how much, if we consider a clock cycle time of the order of 2ns (500MHz: fast at the time of the later Cray vector machines), with main memory SRAMs with a cycle time of 15ns (available at the time), one vector load would saturate the memory system. If we start going more aggressive and allow more than one load or store per cycle, or add processors, the system is going to be memory-bound despite the fast SRAM main memory. The solution is to *multi-bank* the main memory. A bank is a division of memory that can be separately addressed. While one bank can't supply results as fast as the clock speed, pipelining access to multiple *banks* can. Supercomputers of that era could have of the order of 1024 banks. A similar effect can be achieve with multi-banked DRAM, though the startup delay is much higher. Modern architectures tend to reduce the need for memory banks by using very fast caches that can keep up with the CPU.

In Figure 6.2 I illustrate the general principle of multiple banks. In this case, there are four, and the total latency of a memory operation is 8 cycles. By starting bank requests on successive cycles, the total latency for 4 accesses is 11 cycles, rather than 32 cycles, as would be the case if each access had to be strictly sequential. In the illustrated scenario (where each bus transaction takes only 1 cycle), there is no contention for the interconnect (a hexagon in a timing diagram as seen here represents a state where values could be 1 or 0, and we don't care which, only that a transition occurred), so data and addresses could use a common bus, even though I illustrate the two separately. In high-end systems of the Cray era, a more complex interconnect was required.

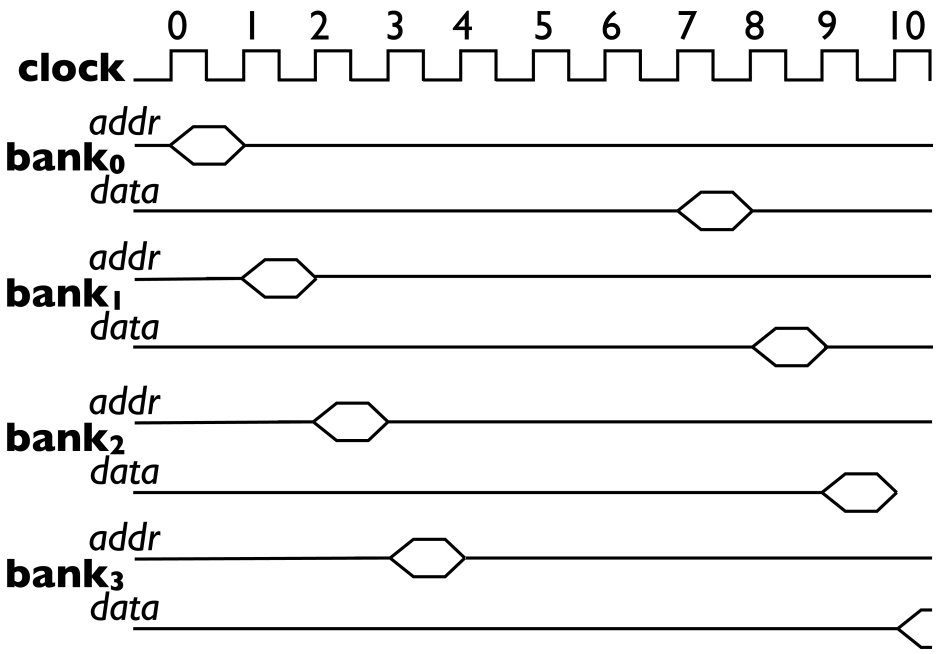


Figure 6.2: The principle of multiple memory banks. *An access is started on each bank in each successive cycle. Although there may be a sizeable delay before each bank responds, each bank will deliver its request in successive cycles after that delay.*

Another inhibition on vector mode is conditional code. If you have a program that for example should only do an ALU operation if a vector operand is non-zero, you want your vector code to apply the operation to every element except those that are zero. A common approach is to use a *vector mask*, a special register with 1 bit per element of a vector register. If the bit is 1, the operation is wanted. If not, it isn't. To implement vector masks, you need an instruction that resets the mask to all 1s (the default, meaning all operations are wanted), and vector compare instructions that set the corresponding bit of a vector mask based on whether the compare is true (`1`) or false (`0`). The mask then applies to whether the outcome of the ALU operation is stored on not; the time for the ALU operation is unchanged. The mask in effect only says whether it's written to the destination vector register element or not. While time and resources are wasted for results that aren't needed, the overall effect in most cases is still faster than executing the ALU operations in scalar mode with a loop.

Why aren't vector machines mainstream? Cray's machines peaked in the 1990s. Seymour Cray split from his company Cray Research in 1990 when the Cray 3 project was put on the back burner, and his new company, Cray Computers,

failed to sell more than one of its first model, the Cray 3, and folded before it could deliver its successor, the Cray 4. Cray Research continued for a while but with the end of the cold war, generous funding for high-speed computers of limited applicability faded and at around that time, RISC architectures started to deliver a significant fraction of the performance of specialist architectures at a fraction of the price. In the mass market, large SRAM memories with a thousand or more banks are not practical (not until someone finds a way to package them cheaply anyway).

The big question that all of this leads to is whether vector architectures embedded in GPUs and multimedia media extensions to instruction sets make any sense without the massive memory bandwidth available to a machine of the old Cray type. The Cray X1, for example, a successor to the T90, has 16 memory controllers per node supplying 204 Gibytes/s to each 4-core vector unit [Dunigan et al. 2005] – and this is an architecture with caches, unlike earlier Cray designs.

6.2 SIMD Extensions to Instruction Sets

Many media applications need relatively short data types, e.g. 8 bits per colour, and it's relatively easy to partition an arithmetic unit so that instead of 32 or 64-bit arithmetic, it can do multiple instances of a narrower unit like 8 bits. In the Intel IA32, multimedia extensions were added by a relatively modest extension of the existing ALU based on this principle. Where Cray vector registers were in the range of 64-128 long supporting full-size floating point, the data types arising from such modest extensions are limited to what can fit into a double-precision (64-bit) register. Intel's original MMX additions were based on that simple model. Later extensions, Streaming SIMD Extensions (SSE) double the register width to 128, and the next iteration, Advanced Vector Extensions (AVX), increased register width to 258, allowing up to 32 8-bit operations per register.

Because these are ad-hoc extensions with big jumps from each design, and without the advantage of the older vector architectures of hardware support for varying the vector length, the number of new instructions is large, several hundred counted across all Intel's variations [Firasta et al. 2008]. There are about 90 AVX instructions, if you do not count all variants of the same basic instruction separately, and the AVX reference runs to 750 pages [Intel 2009].

While compiler support for these instructions is improving, it is not nearly as easy for a compiler to spot opportunities to use them automatically as with a traditional vector architecture. They tend to be used more commonly in hand-

coded drivers or plug-ins for programs with intensive graphics requirements. Despite these problems, this form of limited SIMD does have some advantages. Unlike vector machines, a page fault across a load or store boundary can't happen – or at least it couldn't until Intel allowed loads to be explicitly unaligned with SSE [Lomont 2011], and relaxed the requirement for loads to be aligned in most cases with the AVX design [Intel 2009]. Also, the limited vector size is a better match to commodity memory systems that would battle to keep up with the demands of a full vector instruction set, with 64 or more double-precision floating point numbers per vector register.

SIMD extension instruction sets initially set out to be simple, avoiding the complications that attend traditional vector designs. Hundreds of instructions requiring a good fraction of 1,000 pages to document suggests that something is not quite right, given the original goals.

6.3 GPUs

Graphics processing units are increasingly migrating to the general purpose space (hence GPGPU: general-purpose use of GPUs). As with SIMD extensions to instruction sets, they suffer ad hoc design and repeated changes. That style of change has a venerable history. Silicon Graphics, the pioneer of high-speed 3D graphics, went through architecture iterations that reprised a good fraction of the major models of parallelism:

- *pipeline* – early versions of the SGI Geometry Engine were deeply pipelined, with some SIMD aspects [Harrell and Fouladi 1993]
- *heteorgenous architecture* – the Reality Engine used a small number of relatively non-exotic Intel i860 processors with hundreds of specialised cores [Akeley 1993]
- *SIMD* – the InfiniteReality system of the late 1990s uses a SIMD architecture [Montrym et al. 1997]

SGI early on realised the need for a high-level programming interface that hid the hardware, and developed GL, the basis for OpenGL, as an abstraction layer. That approach made it possible to change the underlying implementation radically as design trade-offs changed.

However, SGI did not ever envisage their graphics hardware being used for high-speed computation: they had a different department covering that, and they had very competitive machines in the 1990s, that were part of the reason that traditional supercomputer makers like Cray ran into trouble.

In more recent times, the underlying reason for rapid change in graphics hardware has not changed. As hardware becomes cheaper, approaches to graphics processing that previously were impractical become viable. Unlike with the history of SGI though, those changes are accompanied by an increasing demand to make it possible to run non-graphics applications on GPUs.

NVIDIA has addressed the problem of rapid hardware change providing C and C++ extensions called *CUDA* (Compute Unified Device Architecture) that allow programming that divides code between the host CPU and the graphics system. OpenCL (Open Computing Language) is a more generic alternative (extending C) that aims to be portable across a wider range of hardware, not only GPUs [Stone et al. 2010]. Aside from the usual portability concern (ideally, a recompile should be sufficient to run on a different CPU), *performance portability* is a hard problem [Du et al. 2012] even within one manufacturer's line: assumptions underlying your coding strategy may not apply on a different model.

A few basics apply to current designs. First, streaming access to memory hides latency. As with multiple banks in older designs, in current DRAM designs, every access after the first has no additional delay (up to the limit of a column access), over and above streaming time.

I examine briefly some of the features of a typical GPU from NVIDIA, and the performance portability problems that can arise.

First, the memory hierarchy of a GPU is complicated. In recent designs that support multiple SIMD threads to hide memory latency, there is a small cache for local variables that don't fit the streaming model. There may also be a local memory that is used for synchronization between threads (e.g. NVIDIA has a hardware barrier instruction [NVIDIA 2011]). Then there is a global memory that is separate from the main CPU's main memory. Second, NVIDIA hides frequent changes in the hardware by using an abstraction layer in the form of the PTX (Parallel Thread Execution) instruction set, that has to be translated at load time to the actual underlying machine instruction set.

PTX has about 40 basic instructions that Hennessy and Patterson [2012] use in examples. There are many other specialist instructions and when you add in all the available variations, the number blows out to hundreds, though the reference manual only runs to about 200 pages [NVIDIA 2011], potentially an improvement on Intel's AVX design at least in that respect. PTX hides some of the complexity of identifying threads and branching, allowing these to vary from implementation to implementation.

Here is a contrast between traditional vector and PTX code, implementing the

following function (DAXPY stands for double precision *a* times *x* *p*lus *y*, and is part of the popular Linpack benchmark suite):

```
void daxpy (int n, double a, double *x, double *y) {
      for (int i = 0; i < n; i++)
            y[i] = a * x[i] + y[i];
}
```

First, let's look at how a typical old-school vector instruction set would implement this. In pseudocode, it would be something like this for the body of the loop:

```
Vload Rx, x[i]    # get VL items starting at x[i]
Vload Ry, y[i]    # get VL items starting at y[i]
VSmuld Rx, a, Rx  # do vector*scalar multiply
VVaddd Ry, Ry, Rx  # do vector add
Vstore y[i],Ry    # vector store result
```

In a strip-mining solution, we need to take care of details like how often to repeat the loop and a fragment where the full vector length isn't needed.

A PTX version is significantly more complicated though superficially it looks similar. Part of the reason for that is that memory access always uses gather-scatter. Also, in creating SIMD code, you create a large number of threads, as part of the strategy of hiding memory latency by using threads. Rather than use vector registers, you allocate a block of threads, then do a calculation simultaneously with each thread doing a different part of the calculation. This would appear to throw away the performance advantage of sequential memory access, but if a program is written so adjacent threads access adjacent memory, the memory subsystem coalesces memory references. The basic steps in pseudocode are:

```
use thread id to create offset in array
load x[i+offset]
load y[i+offset]
do x*a multiplication
add to y
store y[i+offset]
```

This code is replicated across threads, each with a different `id` and hence offset in the array.

An important difference between GPU threads in the NVIDIA world and threads in a general CPU is that all threads are either executing the same

instruction or are idle. A combination of mechanisms makes this possible, including masks similar to those in vector machines and predicates, similar to those in VLIW machines. Branches allow threads to *diverge*, with hardware support to handle managing this. The unwary programmer can create code where most threads are idle.

Although CUDA and OpenCL provide an even higher-level abstraction than PTX, some basic understanding of the underlying hardware is necessary to program efficiently.

6.4 Review

Let's compare GPUs and multimedia extensions with what we generally know about instruction set design. Here are some core principles derived from the RISC movement and experience with supercomputers:

- *Amdahl's Law* – speedup depends on the whole workload, not only the subset that can be improved
- *make the common case fast* – a large instruction set with rarely-used instructions makes it harder to achieve overall speed improvement
- *minimise instruction format variation* – keep fetch and decode simple to make aggressive pipelines easier to implement
- *optimize for average throughput not peak throughput* – as Cray demonstrated in the days of big iron vector machines, a high peak throughput is meaningless if the average case isn't close to the peak
- *simple memory model for programming* – even if there's a complex memory hierarchy that varies from generation to generation of the hardware, a simple uniform model for programmers ensures code longevity and performance portability over time

Why then do multimedia extensions (Intel is not the only guilty party: the AltiVec extensions to PowerPC are also large and complicated, with a reference manual running to over 300 pages[6], if more regular in design than Intel's efforts [Freescale 2006]) and GPU instruction sets violate these principles?

A key consideration is the *real time argument*. In *hard real time*, if a deadline is not met, the system is broken; in *soft real time*, failing to meet a deadline is a performance bug but tolerable (e.g., if the picture pixelates but no so often as to be annoying, you keep watching your digital TV). While graphics rendering

[6] `https://www.nxp.com/docs/en/reference-manual/ALTIVECPEM.pdf`

is not strictly a hard real time application, the faster the graphics system, the better the quality of the picture you can realistically render. In graphics-intensive applications like a photo editor, implementing a filter fast enough to be usable adds value, even if the careful hand-coding necessary doesn't speed up the overall application, a very different consideration than applying Amdahl's Law. If the system is fast enough, expectations expand, but there is a limit to human perception. At some point, perception saturates and there is no point making graphics any faster. Once we approach that point, selling faster GPUs requires another market, hence the interest of GPU makers in selling to a broader base.

Once we exceed the limit of human perception, a model of GPU that has lower peak throughput but a much simpler instruction set that can be used effectively by compilers has a lot to recommend it [Machanick 2018b]. If such an instruction set had 80% of the peak throughput of a much more complex design, either it would be sufficient when the more complex design was sufficiently ahead of human perception, or it could be implemented as two independent cores with at least the same performance as the more complex design, with the option to use one of the other cores for non-graphics tasks. If the ISA were general enough to apply to ordinary workloads, instead of a separate GPU, a multicore design could have some cores used exclusively for graphics and others for computation, with the option to choose dynamically which to use. Another design challenge is how to organize memory so that both graphics and ordinary usage would be satisfied; high-end graphics systems generally avoid this problem with dedicated memory. A cost of dedicated memory in graphics systems is a memory hierarchy that's difficult for programmers.

A detailed design is necessary to evaluate these ideas, as was the case with the original argument for multiple cores (then called a chip multiprocessor [Olukotun et al. 1996]). A useful starting point would be a minimal RISC instruction set, with design studies to determine extra styles of instruction that add the maximum value for parallel execution modes. We can safely avoid ideas that failed in the past like VLIW, be cautious about adopting ideas that are hard to program like SIMD, give careful consideration to ideas that work well in limited cases like vector instructions, and shun ideas that make life for programmers hard, like local scratch memories under programmer control.

Exercises

1. You can find some specifications of the Cray T-90 here: `http://www.netlib.org/utk/papers/advanced-computers.0/crayv.html`. Based on numbers you find:
   (a) What is the maximum number of loads and stores possible in one clock cycle?
   (b) In an 8-processor configuration, with the maximum possible numbers of loads and stores, how many banks of 15ns RAM are required to keep up with demand? Assume each load or store can be divided into as many banks as are needed.
   (c) The top model of this range, the T932, had up to 32 processors and a slightly faster clock speed than that in the above reference (2.167ns). It had 1024 banks of RAM, and the RAM was upgraded from a 15ns cycle time to twice as fast. Was this upgrade necessary?
2. Look up details of the AltiVec instruction set. How does it compare with the other architecture styles we've examined? Is it a reasonable fit to the RISC philosophy?
3. Find a detailed example of NVIDIA PTX code. Explain how parallelism is achieved in the example.
4. If you were designing the Intel AVX instructions from scratch, rather than as an extension of previous designs, how different would your approach be?

7 Warehouse-Scale Computing

MASSIVE-SCALE COMPUTING in the 1990s was the province of high-performance computing (HPC), mainly a concern for computational sciences and large-scale engineering projects (e.g. simulating wind tunnels). Much of that market has disappeared into various models of scaling up commodity parts, e.g. clusters. In some cases, these designs use extra-fast interconnects, but many use commodity networks. A big change since the 1990s is the emergence of massive-scale computing mainly targeting ordinary consumers, not large commercial or research enterprises.

A key difference in the new kind of large-scale computing is economy of scale. Large service providers like Google and Amazon deal in customer bases in many millions, and achieve economy of scale on three fronts:

- *mass-market commodity hardware* – whereas supercomputer makers like Cray and Thinking Machines designed their own parts for a very limited market, this new category of computing draws on the low cost inherent in massive markets
- *purchasing at scale* – even given that these operations use commodity hardware, they score by being able to buy in massive quantities, and hence achieving a much lower price point per unit of work than even a home PC; this large scale also makes it possible for them to design custom configurations of commodity hardware and still arrive at a reasonable cost [Barroso et al. 2003]
- *massive user base* – unlike past HPC-oriented large-scale computing, the new services spread their costs over an enormous user base

All this being the case, some of the complexities of scaling up to extremely large systems remain. Google, Amazon, *et al.* to some extent have the luxury of choosing the services they offer, since many are offered at no charge, and as a way of building advertising revenue or directing users to for-money services (like buying books or apps).

As a generic term for such large-scale services, we use the term *warehouse-scale computer* (*WSC*). Google famously uses relatively entry-level computers, and lots of them. In an operation on this scale with over 50,000 computers, managing individual computers is not possible. A WSC operation has to have considerable support for automated managing of configurations, detecting errors and moving calculations when a computer fails. The range of applications run on these systems is highly variable, and that variation to some extent makes them viable. For example, a large part of Google's operation is web crawling to build search indexing. That kind of workload is both highly parallel and not interactive, and can soak up any available computational resources and network capacity. Multiplexing that kind of workload with requirements for more rapid response time is a good mix, as temporary demand for interactive response time can easily be accommodated by reducing resources for the other type of workload. Contrast this with an electricity grid where instant responsiveness requires not only a lot of spare capacity, but generators capable of rapid cycling up. In one such system for example (the Australian state of Queensland where I used to live), the last 1% of demand costs 100 times base load per kilowatt hour. Power utilities could learn a thing or too about load and demand balancing from WSC operators.

All of this is not however without significant challenges. Users of interactive services, especially those where they care about losing data and want access when they need it, expect a highly dependable service. Downtime of 1 day a year requires 99.7% availability and downtime of at most an hour requires 99.99% availability.

7.1 Fault tolerance and dependability

A key aspect of large-scale systems built out of reasonably reliable components is that the probability of failure increases as you scale up, because there are more parts to fail. First I start with some terminology in Table 7.1 and the following definition:

$$Availability = \frac{MTBF}{MTBF + MTTR} \tag{7.1}$$

or alternatively,

$$Availability = \frac{t_{total} - t_{down}}{t_{total}} \tag{7.2}$$

A key thing to understand is the difference between *dependability* and *reliability*.

term	definition
$MTBF$	mean time between failures: expected time before a module fails
$MTTR$	mean time to repair: expected time to fix a faulty module
reliability	measure of probability of no failures
dependability	measure of likelihood of being useful
fault tolerance	ability to work despite failures
availability	fraction of time a system is able to do work
durability	total time a system is useful
nines	availability of 99.9% is 3 nines for example

Table 7.1: Dependability terminology.

probability / year	failure type
0.02	disk failure
0.01	uncorrectable DRAM failure
0.3	bad software configuration

Table 7.2: Dependability example. *There are many other sources of failure like software crashes and uncorrected power glitches (assuming use of backup power).*

Something that's reliable has a low chance of failing. Something that's dependable has a low chance of not being usable despite failures. A way of ensuring that dependability is higher than reliability is by *fault tolerance*. Fault tolerance is often achieved using *redundancy* along with error checking and correction. For example, a RAID disk system may be configured so that if one drive fails, its content may be recreated. Although the disk subsystem has had a failure, it still works and is therefore dependable, even if it's not reliable.

Similar considerations apply to WSC with tens of thousands of computers. Not only the computers themselves, but networking, building power supplies and software can all fail. To make this concrete, let's take a centre with 2,500 computers and apply the failure rates in Table 7.3. Assume a hardware fault takes 1 hour to repair, and reboot takes $60s$. With the figures in Table 7.3, in an average year with 2,500 computers we get the expected number of failures in Table 7.3. Optimistically assume we can fix a bad software configuration with a reboot, and the others require a hardware repair taking an hour. Then the total time systems

expected failures / year	**failure type**
50	disk failure
25	uncorrectable DRAM failure
750	bad software configuration

Table 7.3: Expected number of failures for 2,500 computers.

are out of action is

$$\begin{aligned} hours_{outage} &= (50+25)\times 1 + 750 \times \frac{1}{60} \\ &= 87.5 \end{aligned}$$

Applying Equation 7.2, and noting there are 8766 hours in an average 365.25 day year:

$$\begin{aligned} availability &= \frac{8766-87.5}{8766} \\ &= 0.99 \end{aligned}$$

So any service requiring continuous use of all 2,500 computers would experience two nines of availability. A real system would have more modes of failure than those listed here, so availability without error correcting and fault tolerance would be considerably lower in practice.

A system like Google's relies on a combination of features to ensure dependability. First, there is considerable checking for potential faults. Second, when a highly distributed computation has a few outstanding results, rather than wait for them, they are farmed out again to the network. Third, there is a high degree of replication of data, to ensure that a hard failure can be recovered. This replication is also required for performance, so fault tolerance falls out of the basic design, rather than being an expensive add-on [Barroso et al. 2003]. In general ensuring high availability in such a large-scale system is a complex task, and the ability of large operations like Google and Amazon to maintain services with high dependability, especially as Google has history of rapid evolution of their user-level software offerings, is a considerable achievement.

Fault tolerance is a large and complex topic; whole courses are given on the subject. I leave it here with a few of the key concepts, rather than an in-depth coverage.

7.2 Programming model

WSC provides parallelism on an unprecedented scale. Given that ordinary-scale parallelism can be hard to use, as we've seen in preceding chapters, does WSC provide a model for large-scale parallelism, or is it only good for thousands of uniprocessor workloads (in itself a useful feature)?

A lot hinges on the programming model and the nature of parallel workloads. Google uses an approach derived from two LISP programming constructs, `map` and `reduce`. In LISP (a predecessor of modern functional languages), `map` is a family of functions that apply another function to each element of a data structure, producing another data structure usually of similar size. The LISP `reduce` function applies a function pairwise to elements of a data structure to produce a single value. An example of application of a LISP-style `map` operation would be to take a list of words and return a list of the length of each. An example of a LISP-style `reduce` operation would be to take a list of numbers and return their sum (here, the applied function would be "+").

Google's MapReduce[1] and the free equivalent, Hadoop MapReduce (Hadoop is an Apache project, including a distributed file system with related tools and services[2]), are based on the LISP `map` and `reduce` concepts. In Common LISP, a map operation looks like this:

```
(map 'list 'length '("fred" "jim" "james"))
=> (4 3 5)
```

and a reduce operation looks like this:

```
(reduce '+ '(4 3 5))
=> 12
```

with a lot of variations possible[3]. The single-quote symbol in LISP forces the next item to be passed to the calling function without evaluation. In the call of `map`, the first argument says what kind of structure to return (here, a list) and the second the operation (here, `length`) to apply to the given list (the strings in parentheses),

[1] You can find a MapReduce tutorial here: `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`.

[2] `https://hadoop.apache.org`

[3] More on `map` here: `http://www.lispworks.com/documentation/HyperSpec/Body/f_map.htm` and on `reduce` here: `http://www.lispworks.com/documentation/HyperSpec/Body/f_reduce.htm`.

In a MapReduce implementation, a map operation takes as input a function and a list of values. The function produces an intermediate value in the form of a list of keys and values, and a reduce operation applies another function to the result of map. In a typical application, the items in the list of values would be large enough to schedule as work units on separate machines, so the map and reduce stages provide a model of parallelism. Part of the fault tolerance in the design is periodic checks on whether the sub-tasks have completed. If they don't after a timeout, the master process restarts them on another node. Coordination and synchronization occurs in effect by a combination of the reduce tasks waiting for map outputs, and the master process waiting for the reduce tasks to complete. Scalability depends on reasonably large chunks of work in each of the map and reduce list elements, and on a reasonable load balance. The overall approach appears to be very successful, given the scale of Google's operation. Within 5 years of the development of the initial implementation in 2003, Google had more than 10,000 internal MapReduce programs, and each day 100,000 MapReduce programs processed about twenty petabytes (PB = $10^{15}B$) of data [Dean and Ghemawat 2008].

MapReduce has been replaced by Google as their programming model by Cloud Dataflow, which is more complex to describe but more scalable [Akidau et al. 2015]. A comprehensive study of the options is worthy of a whole course. Whatever approach is used has to observe a few key principles to achieve scale:

- *minimum communication* – each scheduled unit of work should be reasonably large and able to complete without sharing data with other work units
- *coordination* – there should be a strategy to ensure that outstanding work is completed and there is a reasonable balance between waiting for uncomplete work and scheduling new work; coordination decouples communication and cooperation from computation [Gelernter and Carriero 1992; Tanenbaum and van Steen 2002, p 700]
- *load balance* – work should be spread reasonably evenly over available resources; while rebalancing load by migrating workloads is theoretically possible, the costs in time lost to communication seldom make the move worthwhile
- *fault tolerance* – there should be a fallback strategy to cope with parts of the workload failing to complete

The programming model is interesting to the computer architect because there has to be one for an architecture to be usable (hence the drive to find usable models for GPUs that can use a reasonable fraction of their theoretical throughput, rather than

hand-tuning assembly language). MapReduce is successful up to petabyte-sized data sets and therefore validates the broad concept of WSC; that doesn't mean a better model wasn't needed when data became even larger.

MapReduce has another aspect of interest to a computer architect: it is similar in some ways to a dataflow architecture, in which operations are fired by availability of operands, rather than being driven by order of the code. Dataflow was a style of architecture that attracted some research interest in the 1990s [Ghosal and Bhuyan 1990; Arvind and Nikhil 1990; Lieverse et al. 1999]. Despite some attempts to revive the concept [Swanson et al. 2006; Petersen et al. 2006], dataflow has not been widely adopted because it's too hard to build hardware that fully exploits theoretically available parallelism in the model without changes to programming languages. Though dataflow languages were also an area of active research for two decades [Traub 1986; Johnston et al. 2004], in practice it is hard to sell a new architecture without the option of (mostly) running existing code. Some versions of Intel's IA32 pipelines use dataflow [Papworth 1996], though the parallelism in that case is relatively local (instruction or micro-op reordering). In the case of MapReduce, dataflow is more a coordination (large-scale parallelism) concept than a highly local form of parallelism, and seems to work well at that scale; use of dataflow languages for coordination is an idea developed independently of MapReduce [Lombide Carreton and D'Hondt 2010]. Dataflow architectures today survive in specialist domains [Vo 2011] and in FPGA-based designs, where the programming model is nonstandard anyway [Silva and Lopes 2010; Voigt et al. 2010; Ferlin et al. 2011].

I wrote the first version of this book before being aware that Google had announced Cloud Dataflow. Given the above analysis I was not surprised by that development but I can't claim to have had the idea first.

7.3 Hardware Design

One of the most important considerations of a system on this scale is cost per unit of work. In the early 1990s, when the RISC revolution was at its height, I made the observation that a high-end box was seldom worth the extra cost because the maximum performance was had from a machine a step or two down from that with as much RAM and disk as you could afford. Any machine that you could not afford to populate to the maximum with RAM would no longer be worth the cost of upgrades in a year or two. Google has made a similar discovery: they generally use components typical of a mid-range rather than top

operation	**latency**
network switch	$10\mu s$
local RAM access	$100ns = 0.1\mu s$
disk latency	12ms

Table 7.4: Performance parameters for scalability. *Disk latency is based on half a rotation for a 7,200rpm disk (4ms) plus a conservative 8ms average seek time, assuming a cheaper design than a fast enterprise drive. Local RAM access assumes a miss to DRAM.*

of the line PC. An important consideration in their design is overall cost, including power consumption and heat. Another important consideration in design for scale is network bandwidth and latency. If the network within a building has high bandwidth and low latency, workloads that require some communication can be accommodated within a building or if the requirements for communication are higher, within a single rack with a single fast ethernet switch.

To get some idea of how things scale, let's take some numbers. Actual latency of ethernet depends on how loaded the network is as well as how many switches there are between the nodes sharing information and an accurate model of performance should be based on real workloads [Jin and Caesar 2010]; network latency in Table 7.4 is a little on the optimistic side. On the other hand, disk latencies and memory are on the high side: I assume that as with the Google philosophy, we are aiming for a midrange PC configuration, rather than enterprise-grade drives, and that all memory access are misses to DRAM. This combination of assumptions reduces the penalties for remote access, and puts an upper bound on scalability estimates.

Taking all this into account, let's estimate the fraction of memory accesses that can be remote without doubling average memory access time. That is a break-even point of a fashion: with that amount of overhead, it would be better if we could make the work go to the remote node rather than access its data. Let's go back to our relative execution time formula (Equation 3.1), remembering that we are not really calculating execution time. In this case, we are not even calculating relative execution time as before, just comparing local and remote memory accesses. Assume that the basic latency numbers are a close enough approximation to overall transaction time, which is true of relatively small accesses, and we have a workload where we only have local and remote memory accesses, and no disk accesses. Then our average memory access time is:

$$t_{MEM} = t_{local} + t_{remote} \tag{7.3}$$

I define local access time t_{local} using the fraction of memory references that are local mem_{local} and time to access RAM t_{RAM}

$$t_{local} = mem_{local} \times t_{RAM} \tag{7.4}$$

and remote access time t_{local} using the fraction of memory references that are remote mem_{local} and time to access the network t_{NW} (using the above assumptions, as defined in Table 7.4):

$$t_{remote} = mem_{remote} \times t_{NW} \tag{7.5}$$

We want to find the point where $t_{MEM} = 2 \times t_{local}$, which leads to

$$\begin{aligned}
2 \times t_{local} &= t_{local} + t_{remote} \\
t_{local} &= t_{remote} \\
mem_{local} \times t_{RAM} &= mem_{remote} \times t_{NW} \\
0.1 mem_{local} &= 10 mem_{remote} \\
\frac{mem_{local}}{mem_{remote}} &= 100
\end{aligned} \tag{7.6}$$

In other words, if more than 1% of memory references are remote, we are going to see a slowdown of at least 2 versus purely local computation, and we need to rethink our programming strategy.

The calculation I present here is very optimistic: in a real machine in which most memory references are cached, going over the network is a much larger performance hit, even if we don't add all the components of network latency I've missed here. What I have not gone into is how memory is accessed over a network. In most cases, there will be more to it than putting a packet on a network: a process at the other node will have to interpret the packet and send a response.

Let's now consider a simple memory hierarchy in which latency for a cache hit is hidden by the pipeline and so is effectively the same as the clock speed. Let's set the clock speed to 2GHz, or $0.5ns$. Let's conservatively allow 10% of memory references to miss to DRAM (a high miss rate in most practical systems, e.g. recent Intel designs with 8-12MiB of L3 cache; here I only consider 1 level of cache for simplicity). Then applying Equation 3.1, the average local memory access time is

$$\begin{aligned}
t_e &= t_{h_1} + \sum_{i=1}^{n} p_{m_i} \times r_{m_i} \\
&= 0.5ns + 100ns \times 0.1 \\
&= 10.5ns
\end{aligned}$$

In the units used to derive Equation 7.6, $10.5ns = 0.015\mu s$. So rewriting the local memory term:

$$\begin{aligned} 0.015mem_{local} &= 10mem_{remote} \\ \frac{mem_{local}}{mem_{remote}} &= 666.7 \end{aligned} \tag{7.7}$$

These numbers should give some indication, without working through in full detail, that a model like MapReduce has to distribute relatively large chunks of work that can be computed independently, only communicating results after reasonably long computation.

To make things worse, the minimal network latency only applies if you stay within one network switch. Typically a network switch will cover one rack; there may be several switches covering a full warehouse, and once you go out to the wider Internet, latency quickly mount up. 4000 km, about the distance across continental United States, is about 0.01 light seconds, so the shortest time (unless you can find a way to work around relativity) that you can access information over that distance is about $20ms$, 2,000 times our extremely optimistic network access time, though to be fair, this time I'm counting the round trip, so let's call it at least a factor of 1,000.

Note in all this I didn't mention disks. Clearly, a delay of the order of 1000 times the minimum delay on a network is also a big factor in performance, but that's a factor without highly distributed systems. Disk latency can to some extent be hidden by accessing large units, and by cacheing disk contents in RAM. Accessing a disk over a network doesn't significantly increase the latency, but disk bandwidth tends to be higher than network bandwidth, and less subject to contention. In that sense there is a mismatch between the two technologies. A disk works best streaming large quantities of data, but a network works best with smallish packets, not bigger than a few thousand bytes. Using flash-based SSDs reduces this gap somewhat but they also stream efficiently, favouring larger units of access.

7.4 Warehouse Design

Although WSC uses commodity parts, these will generally be packaged into rack-mount systems for ease of maintenance. A rack can be design to use a single network switch, and packaging can be optimized to fit requirements like

minimising network cabling, even distribution of power, quick identification of faulty systems and selectively replacing obsolete models.

A critical aspect of the overall design is heat dissipation. Even if the Google approach of using mid-range systems is followed, a few thousand PCs in a warehouse adds up to significant heat to extract. A midrange CPU is likely to generate about 100W of heat. To allow for all components, let's take a ballpark figure of 300W (Google reports CPU use as about a third of the total energy budget [Barroso and Hölzle 2009, p 10]). If we have a warehouse of 2,500 computers, that adds up to 750kW of heat (to a good approximation; some of the electricity actually does get used for useful work). Large computer installations may use water as a heat exchange medium, potentially a significant factor in their environmental footprint. As WSC becomes an increasing component of computer services, environmental footprint will become an increasingly important issue, including energy consumption and lifecycle costs [Chang et al. 2012]. By contrast, if you have a single PC in an office or in your home, the impact of its heat dissipation on heating and air conditioning is negligible.

A typical warehouse-scale system will include a large diesel backup power supply, as well as more instantaneous backup UPS power [Barroso and Hölzle 2009].

The overall design of cooling, power supply and component positioning is very complex, and can make a big difference to life cycle costs.

Exercises

Note that the standard abbreviation for byte is "B" and for bit is "b". Recall that binary prefixes have an "i" added to differentiate them from decimal multiples (e.g., Ki means 2^{10}, whereas G means 10^9).

1. With an average year of 8766 hours, how many hours of downtime does four nines of availability represent?
2. You would like to offer four nines of availability on a 2,500 server configuration. Which of the following gets you closest to this goal (starting from the base of the figures in Table 7.3, which gave us 2 nines of availability):
   (a) replacing the hard drives by solid-state drives (SSDs), reducing the expected number of failures to 10 a year
   (b) replacing the RAM with DRAMs with error checking and correction (ECC), reducing the number of uncorrectable failures to 20 per year

(c) running a more robust operating system that reduces failure to 250 per year

Given the above, comment on Google's actual approach, which is to tolerate failures.

3. Google is a significant investor in clean energy technologies, and Apple has reportedly commissioned one of the largest solar energy facilities not owned by a power utility. Discuss why this may be the case.
4. Use the Intel Nehalem latencies from Table 5.1, with the network latency in this chapter (Table 7.4):
   (a) Assume a uniprocessor task running on a local CPU, and redo the calculation for the fraction of remote accesses that double the average memory access time, assuming global miss rates from L1 or 10%, from L2 of 1% and from L3 of 0.1%.
   (b) Now redo the calculation assuming 10% of L2 misses incur snoop latency (implying a multiprocessor task). How does this change your answer?
   (c) Adjust your answers by doubling network latency to allow for the round trip, and adding 10% to allow for network congestion. How much of a difference does this adjustment make?
   (d) Assume the network latency adds for each switch. How much difference will it make if you have to go through 3 switches to obtain a data item?
   (e) In general terms, discuss the performance hit going to a remote machine rather than local accesses, even with multiprocessor overheads.
5. Gbit ethernet switches are commodity technology. Let's consider whether 10Gbit switches are worth considering. Assume switching latency is the same, and the only change is the transfer rate.
   (a) Ignoring switching latency and packet overheads, how long does it take to move a packet of 4KiB at 1Gb/s?
   (b) Ignoring switching latency and packet overheads, how long does it take to move a packet of 4KiB at 10Gb/s?
   (c) How big a difference is there in these two numbers if we add $10\mu s$ switching latency?
   (d) MapReduce operates on relatively large chunks of data. Relate switching latency to transfer time in this example, and explain why MapReduce is generally used that way.
   (e) If you were designing a new WSC facility would you consider 10Gb

ethernet switches? Explain.

6. It's been a while since I updated references. Choose a part of this chapter that you find interesting and look up more recent material.
   (a) Have the fundamentals shifted or are the issues the same, just on a bigger scale?
   (b) How much different is Cloud Dataflow from MapReduce? Do these differences have much of an effect on the ideal hardware architecture?

8 New Developments

UP TO NOW my focus has been on classic issues that have not changed much in twenty years or more – some of the best ideas go back to the 1960s. I change focus now to recent developments – possible new breakthroughs and some new ideas that may or may not work out.

Two of these relate to a revival of specialist modes of processing – special-purpose processors and FPGAs (field-programmable gate arrays). These are not new ideas but specialist niches with high computational demands are giving them a new lease of life.

A particular application of FPGAs is in astronomy, where the Square Kilometre Array, which will be by far the world's largest radio telescope when complete [Dewdney et al. 2009; Arshakian et al. 2009], has massive data requirements. Deep learning – an approach to neural networks – has led to a revival of specialist architectures, driven in part by another big data need – the kind of massive data sets generated by companies like Google.

Another area that is likely to see growing interest is three-dimensional (3D) packaging, either of multiple dies or as a way of structuring a die to include more components. Finally, I review new trends in nonvolatile RAM (NVRAM). Flash for a long time has been the mainstay of nonvolatile RAM and is increasing seen as a replacement for disks in solid-state drives (SSDs). However flash has serious limitations and new generations of nonvolatile RAM are attempting to address those. NVRAM has some connection to 3D packaging as at least one design has a 3D internal structure.

The remainder of this chapter is structured as follows. In §8.1 I review some approaches to going 3D, which sets the scene for NVRAM in §8.2. I follow this with a review of deep learning in §8.3 and end with §8.4 that contains an outline of use of FPGAs in the SKA project. I conclude by summarizing where these trends are likely to take us (§8.5).

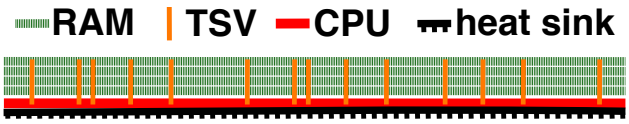


Figure 8.1: 3D die stacking with CPU and RAM in one package.

8.1 Three Dimensions

One of the key problems of scaling Moore's Law is wiring delays and total wiring length. Another is the limits to scaling down – if components become small enough, effects like quantum tunnelling break classical electronics [Cavin et al. 2012]. A way of working around this is to add more layers to a die (chip). That means more parts are reachable through short paths. An intermediate strategy is to layer parts in three dimenstions – more like 2.5D than 3D since the parts are still in essence designed in a plane.

One approach to going 3-dimensional is 3D die stacking, which has the advantage that dissimilar technologies can relatively easily be combined. For example, PicoServer was a design study that combined a multicore CPU layer with DRAM layers, using through-silicon vias (TSVs) to create a package that eliminated off-chip latencies. Figure 8.1 illustrates the general idea. The advantage of this form of die stacking is that it can utilize standard technologies to build a very compact system and the reduced latencies mean that the system can run at lower clock speed for a given level of performance, hence saving energy [Kgil et al. 2008]. The drawback of this approach is that heat dissipation limits the number of layers and potential overall speed. Possibly for this reason, more recent work has focused on building fast RAM by including a logic layer in the 3D stack as in Hybrid Memory Cube (HMC), which stacks DRAM on top of a logic layer designed to give faster access to DRAM – rather than including a CPU layer in the stack [Courtland 2014].

What of a truly 3D design, one that implements logic using all three dimensions, rather than stacking 2D layers?

3D Xpoint (pronounced "cross-point") nonvolatile RAM (NVRAM) is one example where the 3D structure plays a role in the memory design [Bourzac 2017] and there are moves to implement logic structures in 3 dimensions [Cartwright 2011]. Moving into three dimensions is in its infancy – expect more in this space. In particular, 3D Xpoint really exploits the extra dimension in its internal electronics; that is a radical idea that could make real innovations in other areas

possible in future.

8.2 Nonvolatile RAM

3D Xpoint is a good opening to the next topic – NVRAM. The current dominant NVRAM technology is flash, available in a variety of formats and form factors from entry-level cards and USB flash drives to high-end disk replacements.

Flash, broadly speaking, has two problems:

- *speed* – while of the order of 1000 times faster than disk, it is still about 1000 times slower than DRAM and writes are particularly slow
- *endurance* – flash is written by erasing existing contents then writing and there is a limit to the number of erase-write cycles before flash starts to wear out

Most flash on sale today is based on NAND gates to store bits and cannot be addressed at a fine-grained level like DRAM. Instead, it is block-addressed. This is usually not a problem when using it as a disk replacement as disks work much the same way. NOR flash can be byte-addressed but is more expensive as it requires 2–4 times the chip space for the same storage [Kgil and Mudge 2006]. Flash dates back to the 1980s [Masuoka et al. 1987] and many improvements have since been made to the basic technology. Products on sale have advertised endurance ranging from under 10,000 erase cycles to over 100,000. Even the higher end is far lower than the number of writes a disk or DRAM can perform without wearing out. An approach to minimizing damage is *wear levelling* [Chang 2007], where frequent written blocks are moved. However, for wear levelling to be effective, a significant number of free blocks is needed to reduce performance impacts of copying.

The holy grail of NVRAM is a RAM with the speed of DRAM and the endurance of disk – as well as of course a cost closer to disk than DRAM. The closer we can get to that the easier it would be to retire disk entirely and do away with DRAM as the main memory. Why would we want to do that? For long-running systems the ability to checkpoint and restart requires complex software interventions. If the main memory were nonvolatile, all you would need to checkpoint is enough state to recover to the last known state of the caches and registers, a much smaller problem. Also, flash, while fast enough to be a good replacement for disk, is inconvenient because of the endurance problem.

If we cannot achieve the ideal case, an NVRAM at least as fast as flash but a lot more durable would still be a win. Intel and Micron announced 3D Xpoint

in 2015 to much fanfare [Farrow 2015]; actual product has been slow to appear. At time of writing the only announcement has been an SSD trademarked Optane [Bourzac 2017]. If 3D Xpoint does turn out to be much faster and more durable than flash, the remaining challenge is to make it affordable. Intel and Micron appear to have given up on the product after trying to reposition it as a fast cache for disks and flash SSDs, indicating that they were struggling to get pricing to an affordable level[1].

NVRAM is a highly active research area with different device physics being explored; examples include [Eshita et al. 2014]:

- ferroelectric RAM
- magnetoresistive RAM (MRAM)
- phase change RAM (PCRAM)
- resistive RAM (ReRAM)

From the point of view of the computer architect, it does not matter which of these wins (or indeed something completely different) – the key question is how do we change the memory hierarchy if we get a new technology closer to DRAM in speed but nonvolatile – particularly if it has the endurance to use as a main memory or disk replacement?

The current multilevel cache-DRAM-swap hierarchy is designed around the existing range of speed gaps; though flash radically altered the speed gap for swap, it did not alter it enough to be a real game changer. NVRAM research could be if it delivers as promised, though there is a huge gap between the research lab and a mass-market afforable product.

8.3 Deep Learning Architectures

Neural nets go back a long way to the idea of a *perceptron* – a simple model of a neuron with a number of inputs, a function and a possibly different number of outputs [Rosenblatt 1958]. Early work on simulating neural nets on computers took the form of n inputs, k function nodes and m outputs. While the area became a hot topic in AI [Minsky and Papert 1969], a theoretical limitation on the capability of the original model reduced that enthusiasm considerably.

A neural net is a *classifier*: it splits the search space into categories. It does so by learning weights in its decision function until the classification is correct

[1] `https://www.pcgamer.com/intels-optane-business-is-hurtling-towards-an-unglorified-end/`

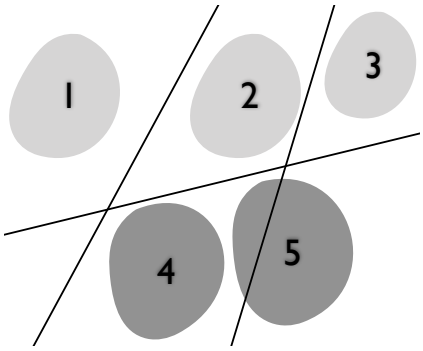


Figure 8.2: Limits of perceptrons. *Different categories that can be separated with a straight line when represented on a plane are* linearly separable. *As illustrated, some are and some are not. A simple single-layer neural net cannot correctly linearly separate in all cases, limiting its usefulness.*

(the *training* phase). The original model was shown to fail to *linearly separate* the search space in even some trivial cases. Visualize this as the search space represented as points on a plane. A correct classifier finds a line on the plane that splits the points of the space into two categories if this is possible; there are cases where a simple single-layer neural net fails when this should be possible. Figure 8.2 illustrates linear separability. If 1, 2 and 3 are one category and 4 and 5 another, they are linearly separable. If however 3 and 5 form a category, they can only approximately be separated as you cannot draw a line between them and other categories that completely misses both. The particular difficulty raised was the XOR problem, which cannot be linearly separated as illustrated in Figure 8.3. No linear function can split the space into $\{0,0\}$ and $\{1,1\}$.

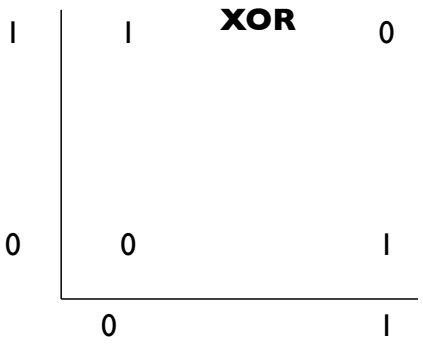


Figure 8.3: XOR and linear separation. *A simple neural net cannot separate the space into outputs that are all 1 or all 0 because you cannot create a linear function that divides the space that way.*

While a book by Minsky and Papert [1969] is widely held to have torpedoed neural net research, it was more complex than that [Olazaran 1996] – in essence, the AI world had a philosophy war that the symbolic processing camp won, pushing neural nets to the periphery.

In the late 1980s, there was a big revival in neural net research with the growing acceptance that one or more *hidden laters* – in effect additional "neurons" between the initial layer and output – solved the lack of generality of the original design. The dominant approach became learning using *back propagation* – the hidden-layer weights were adjusted to correct for the error in the classification and this adjustment was used to adjust the weights of the input layer [Hecht-Nielsen 1992; Svozil et al. 1997].

More recently, *deep learning* – machine learning models with multiple layers, including multi-layer neural nets [LeCun et al. 2015] – have become increasingly popular as a solution to processing the vast amounts of data generated for example by social media. In the hardware world, there is a growing number of projects to implement special-purpose hardware and software architectures using GPUs [Collobert and Weston 2008; Ahmed et al. 2015; Arel et al. 2009; Gupta et al. 2015] to accelerate particularly the learning phase, which takes the most time.

This kind of hardware acceleration is relatively simple because the learning algorithm is relatively regular and can for example use limited-precision arithmetic.

Google internally uses what they call Tensor Flow Architecture; little detail has been released but a software platform [Abadi 2016] is available for general use; the main idea is using data flow graphs to direct numeric computation[2]. Google's Tensor Processing Units draw on ideas from GPUs but are more specialised to this workload[3].

Does any of this make sense? Over past decades there was a lot of research into custom architectures. These generally died on one or more of these issues:

- too much overhead transferring data and control between the general and specialist CPU (particularly if there was a specialist local memory)
- by the time the custom hardware was designed and programmed, Moore's Law had advanced conventional CPUs so far, there was little point
- the cost could not be justified *vs.* a room full of conventional computers
- they were too specialist to attract a big enough community developing an ecosystem (toolchains, support chips, etc.)

In this case, a small number of big adopters like Google with a big enough need

[2]More at `https://www.tensorflow.org`

[3]`https://cloud.google.com/tpu/docs/system-architecture-tpu-vm`

could overcome the traditional objections to this sort of architecture.

8.4 FPGAs and the SKA

An FPGA consists of two core components: lookup tables (LUTs) that you can think of as a way of implementing a logic function in a truth table and routing logic that ties logic functions together. Detail varies a lot in particular designs – some have a large on-chip memory, others have logic functionality built in even to the level of complexity of a whole CPU. A CPU on an FPGA can be configured as a *hard core* [Güneysu 2011], meaning it is some form of CPU with its logic implemented hardware, or a *soft core*, meaning it is implemented in LUTs [Lysecky and Vahid 2005] possibly with the aid of some dedicated hardware.

The SKA is the biggest of the big data science projects, with total data generated once complete estimated to be many times the total data flow of the entire worldwide Internet[4]. Even though the Internet will likely scale way bigger than its current size, SKA remains an impressively large project and its Science Data Processor[5] sub-project is a key component as it will generally not be possible to store that volume of data for later processing.

The SKA data architecture includes FPGA platforms such as SKARAB [Madisa et al. 2018]. Because of the massive data requirement, the main problems with FPGAs are less relevant to SKA than is generally the case. These problems include:

- too much overhead transferring data and control between the general and specialist CPU (particularly if there was a specialist local memory)
- by the time the custom FPGA was designed and programmed, Moore's Law had advanced conventional CPUs so far, there was little point
- the cost could not be justified *vs.* a room full of conventional computers
- they were too specialist to attract a big enough community developing an ecosystem (toolchains, support chips, etc.)

This list may look a tad familiar. In this case, the FPGA engine can sit between the data sources and other software so latency to access specialist memory is not an issue: data enters the FPGA board and a processed version of it exits. While FPGA programming is harder than regular coding, the win is big enough here to be worth the effort. While more conventional computers could do the same thing,

[4] `https://www.skatelescope.org/signal-processing`
[5] `https://www.skatelescope.org/sdp`

reducing a rack full of equipment to one board is a significant gain in a project of this scale.

8.5 Summary

Of all these developments, those most likely to be of general applicability are going 3D and advances in NVRAM. Going 3D could open up new areas of the design space not previously explored and better NVRAM could radically shake up the memory hierarchy.

FPGAs and deep learning engines are just a new iteration of an old idea – custom hardware. All that has changed is that there are some really big projects now that can justify the cost; that does not mean the ideas necessarily generalize to more niches. They could but, if history is a guide, they won't.

A good understanding of the fundamentals and what did or did not work in the past does not stop you from thinking out of the box or being innovative. But it can save you a lot of pain from reliving mistakes of the past. Even rewriting this chapter as new developments appear has revealed disappointments (like the market failure of 3D Xpoint). But not to worry: there will always be something exciting and new in the future.

Exercises

1. It is becoming increasingly useful to package dies (chips) in 3 dimensions as well as to build 3-dimensional logic structures within a die.
   (a) Discuss in general terms, making clear that you understand the difference between the two concepts.
   (b) Compare 3D Xpoint RAM and HMC RAM in terms of the way they are constructed.
   (c) Which category does picoserver fit into? Discuss whether this is a good idea.
2. If nonvolatile RAM (NVRAM) could be made almost as fast as DRAM
   (a) Explain how this could alter the memory hierarchy.
   (b) Explain potential benfits for long-running computations.
   (c) Would this be a win for mobile devices? Explain.
3. Are deep learning architectures likely to win wide acceptance? Consider the list of points that have worked against specialist processors on page 136.

4. Are FPGAs likely to win wide acceptance? Consider the list of points that have worked against specialist processors on page 137.
5. Examples in this chapter mainly focus on Google. Look up what other big players in WSC are up to, such as Amazon. Are the only working on software, or do they have their own hardware projects?

References

Aasaraai, K. and Moshovos, A. (2010). An efficient non-blocking data cache for soft processors. In *2010 International Conf. on Reconfigurable Computing and FPGAs (ReConFig)*, pages 19 –24.

Abadi, M. (2016). Tensorflow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 1–1.

Agarwal, V., Hrishikesh, M. S., Keckler, S. W., and Burger, D. (2000). Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. 27th annual int. symp. on Computer architecture (ISCA'00)*, ISCA '00, pages 248–259, New York, NY, USA. ACM.

Ahmed, E., Jones, M., and Marks, T. K. (2015). An improved deep learning architecture for person re-identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3908–3916.

Akeley, K. (1993). Reality Engine graphics. In *Proc. 20th annual conf. on Computer graphics and interactive techniques (SIGGRAPH'93)*, SIGGRAPH '93, pages 109–116, New York, NY, USA. ACM.

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al. (2015). The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12).

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. Spring joint computer conference*, AFIPS '67 (Spring), pages 483–485.

Amdahl, G. M., Blaauw, G. A., and Brooks, F. P. (1964). Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8(2):87 –101.

Anderson, T. E., Levy, H. M., Bershad, B. N., and Lazowska, E. D. (1991). The interaction of architecture and operating system design. *SIGARCH Comput. Archit. News*, 19(2):108–120.

Apple (2012). About the virtual memory system. Online: `https://developer.apple.com/library/mac/#documentation/performance/conceptual/managingmemory/articles/aboutmemory.html` last accessed 6 July 2012.

Archibald, J. and Baer, J.-L. (1986). Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298.

Arel, I., Rose, D. C., and Coop, R. (2009). DeSTIN: A scalable deep learning architecture with application to high-dimensional robust pattern recognition. In *AAAI Fall Symposium: Biologically Inspired Cognitive Architectures*.

Arshakian, T. G., Beck, R., Krause, M., and Sokoloff, D. (2009). Evolution of magnetic fields in galaxies and future observational tests with the square kilometre array. *Astronomy & Astrophysics*, 494(1):21–32.

Arvind, K. and Nikhil, R. S. (1990). Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318.

Asanović, K. and Patterson, D. A. (2014). Instruction sets should be free: The case for RISC-V. Technical Report UCB/EECS-2014-146, University of California, Berkeley. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html`.

Bach, M., Charney, M., Cohn, R., Demikhovsky, E., Devor, T., Hazelwood, K., Jaleel, A., Luk, C.-K., Lyons, G., Patil, H., et al. (2010). Analyzing parallel programs with Pin. *Computer*, 43(3):34–41.

Barroso, L., Dean, J., and Holzle, U. (2003). Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28.

Barroso, L. A. and Hölzle, U. (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool. online `http://www.morganclaypool.com/doi/pdf/10.2200/S00193ED1V01Y200905CAC006`.

Belayneh, S. and Kaeli, D. (1996). A discussion of non-blocking/lockup-free caches. *Computer Architecture News*, 24(3):18–25.

Bell, G. (2008). Bell's law for the birth and death of computer classes. *Commun. ACM*, 51(1):86–94.

Bennet, J., Carter, J., and Zwaenepoel, W. (1990). Adaptive software cache management for distributed shared memory architectures. In *Proc. 17th Int. Symp. on Computer Architecture (ISCA '90)*, pages 125–134, Seattle, WA.

Berrendorf, R., Burg, H. C., Detert, U., Esser, R., Gerndt, M., and Knecht, R. (1994). Intel Paragon XP/S – architecture, software environment, and performance. Technical Report KFA-ZAM-IB-9409, Jülich Supercomputing Centre, Jülich, Germany.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. (2011). The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7.

Binkert, N. L., Dreslinski, R. G., Hsu, L. R., Lim, K. T., Saidi, A. G., and Reinhardt, S. K. (2006). The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60.

Binkert, N. L., Hallnor, E. G., and Reinhardt, S. K. (2003). Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 36–43.

Bordawekar, R. R. (2000). Quantitative characterization and analysis of the I/O behavior of a commercial distributed-shared-memory machine. *IEEE Trans. on parallel and distributed systems*, 11(5):509–526.

Borg, A., Kessler, R., and Wall, D. (1990). Generation and analysis of very long address traces. In *Proc. 17th Int. Symp. on Computer Architecture (ISCA '90)*, pages 270–279.

Bourzac, K. (2017). Has intel created a universal memory technology?[news]. *IEEE Spectrum*, 54(5):9–10.

Bricklin, D. and Frankston, B. (1979). *VisiCalc Computer Software Program for the Apple II and II Plus*. Personal Software, Inc, Sunnyvale, CA.

Callahan, D., Cocke, J., and Kennedy, K. (1988). Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358.

Cartwright, J. (2011). Intel enters the third dimension. *Nature News*.

Cavin, R. K., Lugli, P., and Zhirnov, V. V. (2012). Science and engineering beyond moore's law. *Proceedings of the IEEE*, 100(Special Centennial Issue):1720–1749.

Ceze, L., Tuck, J., Torrellas, J., and Cascaval, C. (2006). Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proc. 33rd Int. Symp. on Computer Architecture*, pages 227–238, Boston.

Chang, J., Meza, J., Ranganathan, P., Shah, A., Shih, R., and Bash, C. (2012). Totally green: evaluating and designing servers for lifecycle environmental impact. In *Proc. 17th int. conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*, ASPLOS '12, pages 25–36, New York, NY, USA. ACM.

Chang, L.-P. (2007). On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 1126–1130, New York, NY, USA. ACM.

Chatterjee, D., DeOrio, A., and Bertacco, V. (2009). GCS: High-performance gate-level simulation with GPGPUs. In *Design, Automation Test in Europe Conference Exhibition DATE '09.*, pages 1332–1337.

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.

Cheriton, D., Goosen, H., and Boyle, P. (1989). Multi-level shared caching techniques for scalability VMP-MC. In *Proc. 16th Int. Symp. on Computer Architecture (ISCA '89)*, pages 16–24, Jerusalem.

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., Goosen, H., and Machanick, P. (1991). Restructuring a parallel simulation to improve cache behavior in a shared-memory multiprocessor: A first experience. In *Proc. Int. Symp. on Shared Memory Multiprocessing*, pages 109–118, Tokyo.

Cheriton, D., Gupta, A., Boyle, P., and Goosen, H. (1988). The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proc. 15th Int. Symp. on Computer Architecture (ISCA '88)*, pages 410–421, Honolulu.

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.

Cheung, T. and Smith, J. (1986). A simulation study of the CRAY X-MP memory system. *IEEE Transactions on computers*, C-35(7):613–622.

Chinchilla, F., Gamblin, T., Sommervoll, M., and Prins, J. F. (2004). Parallel N-body simulation using GPUs. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill. `http://wwwx.cs.unc.edu/~tgamblin/gpgpu/GPGPfinalReport.pdf`.

Clark, D. W. (1987). Pipelining and performance in the VAX 8800 processor. *ACM SIGARCH Computer Architecture News*, 15(5):173–177.

Cmelik, B. and Keppel, D. (1994). Shade: a fast instruction-set simulator for execution profiling. *SIGMETRICS Perform. Eval. Rev.*, 22(1):128–137.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM.

Colwell, R. and Steck, R. (1995). A 0.6μ BiCMOS processor with dynamic execution. In *Proc. 42nd IEEE Int. Conf. on Solid-State Circuits (ISSCC)*, pages 176–177, 361.

Colwell, R. P., Gehringer, E. F., and Jensen, E. D. (1988). Performance effects of architectural complexity in the Intel 432. *ACM Trans. Comput. Syst.*, 6(3):296–339.

Colwell, R. P., Hall, W. E., Joshi, C. S., Papworth, D. B., Rodman, P. K., and Tornes, J. E. (1990). Architecture and implementation of a VLIW supercomputer. In *Proc. 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 910–919, Los Alamitos, CA, USA. IEEE Computer Society Press.

Courtland, R. (2014). Memory in the third dimension. *IEEE Spectrum*, 51(1):60–61.

Craig, T. (1993). Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR 93-02-02, University of Washington. `https://dada.cs.washington.edu/research/tr/1993/02/UW-CSE-93-02-02.pdf`.

De Michell, G. and Gupta, R. K. (1997). Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365.

Dean, J. and Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Denning, P. J. (1968). The working set model for program behavior. *Commun. ACM*, 11(5):323–333.

Dewdney, P. E., Hall, P. J., Schilizzi, R. T., and Lazio, T. J. L. (2009). The square kilometre array. *Proceedings of the IEEE*, 97(8):1482–1496.

Diefendorff, K., Dubey, P., Hochsprung, R., and Scale, H. (2000). AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., and Dongarra, J. (2012). From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407.

Dunigan, T.H., J., Vetter, J., White, J.B., I., and Worley, P. (2005). Performance evaluation of the Cray X1 distributed shared-memory architecture. *IEEE Micro*, 25(1):30 – 40.

Dwarkadas, S., Keleher, P., Cox, A., and Zwaenepoel, W. (1993). Release consistent software distributed shared memory on emerging network technology. In *Proc. 20th Int. Symp. on Computer Architecture (ISCA '93)*, pages 144–155, San Diego, CA.

Engblom, J. and Ermedahl, A. (1999). Pipeline timing analysis using a trace-driven simulator. In *Proc. Sixth Int. Conf, on Real-Time Computing Systems and Applications (RTCSA)*, pages 88–95.

Eshita, T., Wang, W., Nakamura, K., Mihara, S., Saito, H., Hikosaka, Y., Inoue, K., Kawashima, S., Yamaguchi, H., and Nomura, K. (2014). Development of ferroelectric RAM (FRAM) for mass production. In *2014 Joint IEEE International Symposium on the Applications of Ferroelectrics, International Workshop on Acoustic Transduction Materials and Devices & Workshop on Piezoresponse Force Microscopy (ISAF/IWATMD/PFM)*, pages 1–3. IEEE.

Farrow, R. (2015). Interview with darrell long. *;login:*, 40(6):36–37.

Ferlin, E. P., Lopes, H. S., Lima, C. R. E., and Perretto, M. (2011). Prada; a high-performance reconfigurable parallel architecture based on the dataflow model. *Int. J. High Perform. Syst. Archit.*, 3(1):41–55.

Fick, D., Dreslinski, R., Giridhar, B., Kim, G., Seo, S., Fojtik, M., Satpathy, S., Lee, Y., Kim, D., Liu, N., Wieckowski, M., Chen, G., Mudge, T., Sylvester, D., and Blaauw, D. (2012). Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores. In *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, pages 190–192.

Firasta, N., Buxton, M., Jinbo, P., Nasri, K., and Kuo, S. (2008). Intel® AVX: New frontiers in performance improvements and energy efficiency. Technical report, Intel. `https://www.codeproject.com/Articles/27116/Intel-AVX-New-Frontiers-in-Performance-Improvement` accessed 17 June 2022.

Freescale (2006). *AltiVec Technology Programming Environments Manual*. Freescale Semiconductor. online `https://www.nxp.com/docs/en/reference-manual/ALTIVECPEM.pdf` accessed 17 June 2022.

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Commun. ACM*, 35(2):97–107.

Ghosal, D. and Bhuyan, L. N. (1990). Performance evaluation of a dataflow architecture. *IEEE Trans. Comput.*, 39(5):615–627.

Gifford, D. and Spector, A. (1987). Case study: IBM's system/360-370 architecture. *Comm. ACM*, 30(4):291–307.

Grimes, J., Kohn, L., and Bharadhwaj, R. (1989). The Intel i860 64-bit processor: A general-purpose CPU with 3D graphics capabilities. *IEEE Comput. Graph. Appl.*, 9(4):85–94.

Güneysu, T. (2011). Utilizing hard cores of modern FPGA devices for high-performance cryptography. *Journal of Cryptographic Engineering*, 1(1):37–55.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., Mudge, T., and Brown, R. (2001). MiBench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE Int. Workshop on Workload Characterization, 2001 (WWC-4)*, pages 3–14.

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

Harrell, C. B. and Fouladi, F. (1993). Graphics rendering architecture for a high performance desktop workstation. In *Proc. 20th Ann. Conf on Computer graphics and Interactive Techniques*, pages 93–100.

Harris, C., Beckett, G., Bording, C., Carey, D., Chew, A., Elwell, A., Deeptimahanti, D., Grimwood, D., Maxville, V., O'Shea, M., et al. (2015). HPC technology update. Technical report, Pawsey Supercomputing Centre.

Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In Wechsler, H., editor, *Neural networks for perception*, pages 65–93. Elsevier.

Hennessy, J. and Patterson, D. (1990). *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann, San Francisco, CA, 1st edition.

Hennessy, J. and Patterson, D. (2012). *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann, San Francisco, CA, 5th edition.

Hennessy, J. and Patterson, D. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kauffmann, San Francisco, CA, 6th edition.

Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

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.

Inouye, J., Konuru, R., Walpole, J., and Sears, B. (1992). The effects of virtually addressed caches on virtual memory design and performance. Technical Report CS/E 92-010, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Engineering.

Intel (2009). Intel® advanced vector extensions programming reference. Technical Report 319433-006, Inte.

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.

Jin, D. and Caesar, D. N. M. (2010). Efficient gigabit ethernet switch models for large-scale simulation. In *2010 IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 1–10.

Johnston, W. M., Hanna, J. R. P., and Millar, R. J. (2004). Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34.

Kaeli, D. and Emma, P. (1997). Improving the accuracy of history-based branch prediction. *IEEE Trans. on Computers*, 46(4):469–472.

Kalla, R., Sinharoy, B., and Tendler, J. (2004). IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47.

Kang, S. C., Nicopoulos, C., Lee, H., and Kim, J. (2011). A high-performance and energy-efficient virtually tagged stack cache architecture for multi-core environments. In *Proc. IEEE 13th Int. Conf. on High Performance Computing and Communications (HPCC)*, pages 58 –67.

Keltcher, C., McGrath, K., Ahmed, A., and Conway, P. (2003). The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66 – 76.

Kgil, T., D'Souza, S., Saidi, A., Binkert, N., Dreslinski, R., Reinhardt, S., Flautner, K., and Mudge, T. (2006). PicoServer: Using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, San Jose, CA.

Kgil, T. and Mudge, T. (2006). Flashcache: a NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112. ACM.

Kgil, T., Saidi, A., Binkert, N., Reinhardt, S., Flautner, K., and Mudge, T. (2008). PicoServer: Using 3D stacking technology to build energy efficient servers. *J. Emerg. Technol. Comput. Syst.*, 4(4):16:1–16:34.

Kim, N., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J., Irwin, M., Kandemir, M., and Narayanan, V. (2003). Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75.

Krakiwsky, S., Turner, L., and Okoniewski, M. (2004). Acceleration of finite-difference time-domain (FDTD) using graphics processor units (GPU). In *Proc. IEEE MTT-S Int. Microwave Symp.*, volume 2, pages 1033–1036 Vol.2.

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

Lam, M., Rothberg, E., and Wolf, M. (1991). The cache performance and optimizations of blocked algorithms. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-4)*, pages 63–74, Santa Clara, CA.

Lam, M. S. and Wilson, R. P. (1992). Limits of control flow on parallelism. In *Proc. 19th Ann. Int. Symp. on Computer Architecture*, pages 46–57, Queensland, Australia.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Larus, J. R. (1990). SPIM S20: A MIPS R2000 simulator. Technical Report 966, University of Wisconsin-Madison Department of Computer Sciences.

Lavington, S. H. (1978). The Manchester Mark I and Atlas: a historical perspective. *Commun. ACM*, 21(1):4–12.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436.

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

Lee, J., Kim, J., Jang, C., Kim, S., Egger, B., Kim, K., and Han, S. (2008). FaCSim: a fast and cycle-accurate architecture simulator for embedded systems. In *Proc. 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 89–100, New York, NY, USA. ACM.

Li, P. (2019). Reduce static code size and improve RISC-V compression. Technical Report UCB/EECS-2019-46, University of California, Berkeley.

Lieverse, P., Deprettere, E. F., Kienhuis, A. C. J., and De Kock, E. A. (1999). A clustering approach to explore grain-sizes in the definitionof processing elements in dataflow architectures. *J. VLSI Signal Process. Syst.*, 22(1):9–20.

Liu, H. and Wee, S. (2009). Web server farm in the cloud: Performance evaluation and dynamic architecture. In Jaatun, M., Zhao, G., and Rong, C., editors, *Cloud Computing*, volume 5931 of *Lecture Notes in Computer Science*, pages 369–380. Springer Berlin / Heidelberg.

Lombide Carreton, A. and D'Hondt, T. (2010). A hybrid visual dataflow language for coordination in mobile ad hoc networks. In *Proc. 12th int. conf. on Coordination Models and Languages (COORDINATION'10)*, COORDINATION'10, pages 76–91, Berlin, Heidelberg. Springer-Verlag.

Lomont, C. (2011). Introduction to Intel® Advanced Vector Extensions. Technical report, Intel.

Lowrey, T. (2002). Three-dimensional (3d) programmable device. US Patent 6,501,111.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, pages 190–200.

Lysecky, R. and Vahid, F. (2005). A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, pages 18–23. IEEE Computer Society.

Macedonia, M. (2004). The digital world's midlife crisis. *Computer*, 37(8):100 – 101.

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 J.*, (25):68–73.

Machanick, P. (2004). Initial Experiences with Dreamy Memory and the RAMpage Memory Hierarchy. In *Proc. Ninth Asia-Pacific Computer Systems Architecture Conf.*, pages 146–159, Beijing.

Machanick, P. (2018a). A preliminary study of minimal-contention locks. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 269–278. ACM.

Machanick, P. (2018b). Project CrayOn: Back to the future for a more general-purpose GPU? In *Proc. 2nd Workshop on Pioneering Processor Paradigms*, Vienna, Austria.

Machanick, P. and Salverda, P. (1998). Preliminary investigation of the RAMpage memory hierarchy. *South African Computer J.*, (21):16–25.

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.

Madisa, K., Marais, N., Ramaila, A., and den Heever, L. V. (2018). Integration of MeerKAT and SKA Telescopes using KATCP/Tango Translators. In *Proc. of International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS'17), Barcelona, Spain, 8-13 October 2017*, number 16 in International Conference on Accelerator and Large Experimental Control Systems, pages 1964–1968, Geneva, Switzerland. JACoW. https://doi.org/10.18429/JACoW-ICALEPCS2017-THSH201.

Magnusson, P., Landin, A., and Hagersten, E. (1994). Queue locks on cache coherent multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*, pages 165–171. IEEE.

Maqbool, J., Oh, S., and Fox, G. C. (2015). Evaluating ARM HPC clusters for scientific workloads. *Concurrency and Computation: Practice and Experience*, 27(17):5390–5410.

Markatos, E., Crovella, M., Das, P., Dubnicki, C., and LeBlanc, T. (1991). The effects of multiprogramming on barrier synchronization. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Processing*, pages 662–669.

Martínez, J. F. and Torrellas, J. (2002). Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X: Proc. 10th Int. Conf. on Architectural support for programming languages and operating systems*, pages 18–29, San Jose, CA.

Masuoka, F., Momodomi, M., Iwata, Y., and Shirota, R. (1987). New ultra high density EPROM and flash EEPROM with NAND structure cell. In *Electron Devices Meeting, 1987 International*, pages 552–555. IEEE.

Mattioli, M. (2022). Meet the fam1ly. *IEEE Micro*, 42(3):78–84.

Mayer, A. J. W. (1982). The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10.

Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems (TOCS)*, 9(1):21–65.

Meredith, J. S., Alvarez, G., Maier, T. A., Schulthess, T. C., and Vetter, J. S. (2009). Accuracy and performance of graphics processors: A quantum Monte Carlo application case study. *Parallel Computing*, 35(3):151–163.

Michael, M. M. (2004). ABA prevention using single-word instructions. Technical Report RC23089 (W0401-136), IBM Research Division.

Minsky, M. L. and Papert, S. A. (1969). *Perceptrons: an introduction to computational geometry*. MIT Press, Cambridge, MA.

Mironov, D., Ubar, R., Devadze, S., Raik, J., and Jutman, A. (2010). Structurally synthesized multiple input BDDs for speeding up logic-level simulation of digital circuits. In *13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 658 –663.

Molka, D., Hackenberg, D., Schone, R., and Muller, M. (2009). Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proc. 18th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 261–270.

Montrym, J. S., Baum, D. R., Dignam, D. L., and Migdal, C. J. (1997). InfiniteReality: a real-time graphics system. In *Proc. 24th annual conf. on Computer graphics and interactive techniques (SIGGRAPH '97)*, SIGGRAPH '97, pages 293–302, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

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

Moudgill, M., Wellman, J.-D., and Moreno, J. (1999). Environment for PowerPC microarchitecture exploration. *IEEE Micro*, 19(3):15 –25.

Nambiar, R., Wakou, N., Carman, F., and Majdalany, M. (2011). Transaction processing performance council (TPC): State of the council 2010. In Nambiar, R. and Poess, M., editors, *Performance Evaluation, Measurement and Characterization of Complex Systems*, volume 6417 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg.

Nayfeh, B. and Olukotun, K. (1997). A single-chip multiprocessor. *Computer*, 30(9):79–85.

Nayfeh, B. A. and Olukotun, K. (1994). Exploring the design space for a shared-cache multiprocessor. In *ISCA '94: Proc. 21st Ann. Int. Symp. on Computer Architecture*, pages 166–175, Chicago, Illinois, United States.

NVIDIA (2011). PTX: Parallel thread execution ISA version 2.3.

Olazaran, M. (1996). A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659.

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.

Organick, E. I. (1983). *A programmer's view of the Intel 432 system*. McGraw-Hill, Inc., New York, NY, USA.

Pagelkopf, D., Moe, R., Lincoln, N. R., Krueger, L., Krohn, H., Kort, R., Hutson, M., Hawley, C. L., Grinna, D., Bhend, B., et al. (1975). Reminiscences of computer architecture and computer design at Control Data Corporation. `http://conservancy.umn.edu/bitstream/handle/11299/104327/1/oh321cdc.pdf`.

Papworth, D. (1996). Tuning the Pentium Pro microarchitecture. *IEEE Micro*, 16(2):8–15.

Patterson, D. A. (1985). Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21.

Patterson, D. A. and Ditzel, D. R. (1980). The case for the reduced instruction set computer. *Computer Architecture News*, 8(6):25–33.

Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, New York, NY, USA. ACM.

Peleg, A., Wilkie, S., and Weiser, U. (1997). Intel MMX for multimedia pcs. *Commun. ACM*, 40(1):24–38.

Perleberg, C. and Smith, A. (1993). Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412.

Petersen, A., Putnam, A., Mercaldi, M., Schwerin, A., Eggers, S., Swanson, S., and Oskin, M. (2006). Reducing control overhead in dataflow architectures.

In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 182–191, New York, NY, USA. ACM.

Piguet, C. (2006). Ultra-low-power processor design. In Oklobdzija, V. G. and Krishnamurthy, R. K., editors, *High-Performance Energy-Efficient Microprocessor Design*, Integrated Circuits and Systems, pages 1–30. Springer US. DOI:10.1007/978-0-387-34047-0_1.

Poess, M., Nambiar, R. O., Vaid, K., Stephens, Jr., J. M., Huppler, K., and Haines, E. (2010). Energy benchmarks: a detailed analysis. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, pages 131–140, New York, NY, USA. ACM.

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

Rahman, N. and Raman, R. (2000). Analysing cache effects in distribution sorting. *J. of Experimental Algorithmics (JEA)*, 5:14.

Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). PIN: a binary instrumentation tool for computer architecture research and education. In *Proc. 2004 workshop on Computer architecture education: held in conjunction with the 31st Int. Symp. on Computer Architecture*, WCAE '04, New York, NY, USA. ACM.

Reinberg, A. R. and Zahorik, R. C. (2004). X-point memory cell. US Patent 6,777,705.

Rogers, A. and Li, K. (1992). Software support for speculative loads. In *Proc. 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-5)*, pages 38–50.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Rosenblum, M., Herrod, S., Witchel, E., and Gupta, A. (1995). Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43.

Russinovich, M. (2007). Inside the Windows Vista kernel: Part 2. *TechNet Magazine*.

Schoeberl, M. (2008). A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1–2):265 – 286.

Schoeberl, M., Preußer, T. B., and Uhrig, S. (2010). The embedded Java benchmark suite JemBench. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 120–127, New York, NY, USA. ACM.

Seznec, A. and Lenfant, J. (1992). Interleaved parallel schemes: improving memory throughput on supercomputers. In *Proc. 19th annual int. symp. on Computer architecture (ISCA '92)*, ISCA '92, pages 246–255, New York, NY, USA. ACM.

Silva, J. L. E. and Lopes, J. J. (2010). A dynamic dataflow architecture using partial reconfigurable hardware as an option for multiple cores. *W. Trans. on Comp.*, 9(5):429–444.

Simunic, T., Benini, L., and De Micheli, G. (1999). Cycle-accurate simulation of energy consumption in embedded systems. In *Proc. 36th Design Automation Conference*, pages 867–872.

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.

Sohi, G. (2001). Microprocessors – 10 years back, 10 years ahead. In Wilhelm, R., editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 209–218. Springer Berlin / Heidelberg.

Stone, J., Gohara, D., and Shi, G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73.

Svozil, D., Kvasnicka, V., and Pospichal, J. (1997). Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62.

Swanson, S., Putnam, A., Mercaldi, M., Michelson, K., Petersen, A., Schwerin, A., Oskin, M., and Eggers, S. J. (2006). Area-performance trade-offs in tiled dataflow architectures. In *ISCA'06: Proc. 33rd Int. Symp. on Computer Architecture*, pages 314–326, Boston.

Tanenbaum, A. S. and van Steen, M. (2002). *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ.

Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B. (2002). POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25.

Thornton, J. E. (1963). Considerations in computer design – leading to the Control Data 6600. Technical report, Control Data Corp. `http://archive.computerhistory.org/resources/text/CDC/CDC.6600.1963.102641207.pdf`.

Thornton, J. E. (1980). The CDC 6600 project. *Annals of the History of Computing*, 2(4):338 –348.

Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Research and Development*, 11(1):25–33.

Traub, K. R. (1986). A compiler for the MIT tagged-token dataflow architecture. Technical Report 10.5555/889962, MIT, Cambridge, MA, USA. `http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-370.pdf`.

Tyson, G. S. (1994). The effects of predicated execution on branch prediction. In *Proc. 27th Ann. Int. Symp. on Microarchitecture*, pages 196–206, San Jose, CA.

Uhlig, R. A. and Mudge, T. N. (1997). Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170.

Valero, M., Lang, T., and Ayguadé, E. (1992). Conflict-free access of vectors with power-of-two strides. In *Proc. 6th int. conf. on Supercomputing (ICS '92)*, ICS '92, pages 149–156, New York, NY, USA. ACM.

Vo, H. T. (2011). *Designing a parallel dataflow architecture for streaming large-scale visualization on heterogeneous platforms*. PhD thesis, University of Utah, Salt Lake City, UT, USA. AAI3454865.

Voigt, S., Baesler, M., and Teufel, T. (2010). Dynamically reconfigurable dataflow architecture for high-performance digital signal processing. *J. Syst. Archit.*, 56(11):561–576.

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.

Wang, R., Tobar, R., Dolensky, M., An, T., Wicenec, A., Wu, C., Dulwich, F., Podhorszki, N., Anantharaj, V., Suchyta, E., et al. (2020). Processing full-scale square kilometre array data on the summit supercomputer. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE.

Wang, S., Ananthanarayanan, G., Zeng, Y., Goel, N., Pathania, A., and Mitra, T. (2019). High-throughput cnn inference on embedded ARM Big.LITTLE multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267.

Weiss, S. (1989). An aperiodic storage scheme to reduce memory conflicts in vector processors. In *Proc. 16th annual int. symp. on Computer architecture (ISCA '89)*, ISCA '89, pages 380–386, New York, NY, USA. ACM.

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.

Whitehead, N. and Fit-Florea, A. (2011). Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA white paper, `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Floating_Point_on_NVIDIA_GPU_White_Paper.pdf`.

Wolff, W. and Porter, B. (2020). Performance optimization on big. little architectures: a memory-latency aware approach. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 51–61.

Womble, D. E., Dosanjh, S. S., Hendrickson, B., Heroux, M. A., Plimpton, S. J., Tomkins, J. L., and Greenberg, D. S. (1999). Massively parallel computing: A Sandia perspective. *Parallel Computing*, 25(13–14):1853–1876.

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

Wynters, E. (2011). Parallel processing on NVIDIA graphics processing units using CUDA. *J. Comput. Sci. Coll.*, 26(3):58–66.

Xiao, L., Zhang, X., and Kubricht, S. A. (2000). Improving memory performance of sorting algorithms. *J. of Experimental Algorithmics (JEA)*, 5:3.

Yeager, K. C. (1996). The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41.

Yeap, G. C.-F. (2002). Leakage current in low standby power and high performance devices: trends and challenges. In *Proc. 2002 Int. Symp. on Physical design*, pages 22–27, San Diego, CA.

Yeh, T.-Y. and Patt, Y. N. (1991). Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA. ACM.

Yeh, T.-Y. and Patt, Y. N. (1992). Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Ann. Int. Symp. on Computer Architecture*, pages 124–134.

Yeh, T.-Y. and Patt, Y. N. (1993). A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. 20th Ann. 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.

Zivkov, B., Ferguson, B., and Gupta, M. (1994). R4200: a high-performance MIPS microprocessor for portables. In *Compcon Spring '94, Digest of Papers.*, pages 18 –25.

Zukowski, M., Héman, S., Nes, N., and Boncz, P. (2006). Super-scalar RAM-CPU cache compression. In *Proc. 22nd Int. Conf. on Data Engineering (ICDE '06)*, page 59.

A Generating Traces using Pin

GENERATING A TRACE FILE is useful for evaluating your own workloads using a trace-driven simulator. My own simplified memory simulator uses traces, and I use Pin to generate them. Pin is a toolkit for instrumenting code [Luk et al. 2005]. It comes with an example to generate traces of instructions (the `itrace` tool) and an example to generate traces of data references (`pinatrace`) and these are easy to adapt to create a new tool that creates a trace file recording each memory accessed and whether it is an instruction reference, a data read or a data write.

Pin instruments Intel instruction sets, so it is not ideal for a RISC-oriented course but is free and relatively easy to use.

I describe here how to obtain Pin and the steps to create your own example.

A.1 Obtaining Pin

Pin is obtainable from Intel[1]. Although there are versions for Windows and macOS, the Linux version is relatively easy and straightforward to use. It installs easily on a Mac because the underlying system is a variant on Unix. However, Apple has protections that need to be undone to run code that could be a security hazard so it is easier if you can use Linux.

Find the instructions to download an install. Here are the steps I used; this will change with new versions:

```
$ mkdir Pin
$ cd Pin
$ wget https://software.intel.com/sites/landingpage/pintool/downloads/pin-...
$ tar fxz pin-3.23-98579-gb15ab7903-gcc-linux.tar.gz
```

[1] `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html`

```
$ cd pin-3.23-98579-gb15ab7903-gcc-linux
$ cd source/tools/ManualExamples
$ make all TARGET=intel64
$ ../../../pin -t obj-intel64/inscount0.so -- /bin/ls
$ cat inscount.out
```

These steps download it (you can use the web interface instead of `wget` and move the download to a suitable place), unarchive it, compile the sample examples and run one. To see the output, it is generally in *`toolname`*`.out` but you can change the output file name using the `-o` *`filename`* option in some tools. My simple trace-generating tool that I describe below doesn't allow that; if you would like to experiment with coding such options, check other examples like `inscount0.cpp`.

Note that using `cat` to display contents would not be so great for a trace file or longer output: you can use commands like `less` to view it a screen full at a time or `head` to see the first few lines or `tail` to view the last few.

With this done you are in a position to create a new example.

A.2 Trace example

To make an example that outputs traces in the format I want, I start with the two tools that each do part of it. In examples that follow, I run them as in the manual, using `/bin/ls` as the program to instrument (listing the contents of the current directory is a reasonably short run of code unless your Pin examples are in a different place with a lot of files or directories – in examples I run, less than 1-million instructions).

The output format I want is a letter indicating instruction address (`I`), data read (`R`) or data write (`W`) – followed by the hexadecimal representation of the address of the instruction or data.

One of the given tools, `itrace`, generates output consisting of a hexadecimal number representing the address of each instruction executed, e.g.:

```
0x7f3583a0c100
0x7f3583a0c103
0x7f3583a0cdf0
0x7f3583a0cdf4
0x7f3583a0cdf5
```

Another, `pinatrace`, outputs the machine instruction address followed by the address of a data reference for each instruction that accesses memory, e.g. (not

from the start, I choose a part with reads and writes):

```
0x7f84cd9bce03: W 0x7fff7fa5f778
0x7f84cd9bce18: W 0x7f84cd9e85e0
0x7f84cd9bce1f: R 0x7f84cd9e8e68
0x7f84cd9bce29: R 0x7f84cd9e9000
0x7f84cd9bce30: W 0x7f84cd9e99f8
```

The Intel instruction set can have more than one memory reference in an instruction. If so, `pinatrace` generates more than one line for a given instruction.

From these two examples, it is possible to synthesise a tool that provides the format I want. The closest to it is `pinatrace` so I modify that to create a new tool, `alltrace.cpp`. The modifications are simple enough to itemize:

- take out the output of the instruction address from both `RecordMemRead` and `RecordMemWrite`
- add a function
  `VOID printip(VOID* ip)  fprintf(trace, "I %p\n", ip);`
  to print an instruction address
- add in a line to invoke printing the instruction address before the loop that prints data reference addresses:
  ```
  INS_InsertPredicatedCall(ins, IPOINT_BEFORE,
   (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
  ```
- change the output file name from `pinatrace.out` to `alltrace.out`
- in `makefile.rules` add `alltrace` to the list of tools

With all this done, rerunning

```
$ make all TARGET=intel64
```

should create the new tool, which you can test by:

```
$ ../../../pin ot obj-intel64/alltrace.so -- /bin/ls
```

and the output should look like this (not from the start, I again choose a part with reads and writes):

```
I 0x7f87cb63fe03
W 0x7ffd2e62afa8
I 0x7f87cb63fe04
I 0x7f87cb63fe08
I 0x7f87cb63fe0a
```

```
I 0x7f87cb63fe0e
I 0x7f87cb63fe11
I 0x7f87cb63fe18
W 0x7f87cb66b5e0
I 0x7f87cb63fe1f
R 0x7f87cb66be68
I 0x7f87cb63fe26
I 0x7f87cb63fe29
R 0x7f87cb66c000
I 0x7f87cb63fe30
W 0x7f87cb66c9f8
```

Note: addresses will not match from one run to the next as code could load and run differently.

B Simplified Simulator

A TRACE-DRIVEN SIMULATOR can demonstrate memory effects based on a real workload though with less fidelity than a full-system simulator. However a full-system simulator is complex to implement and use, and generally runs much slower than a trace-driven simulator. A trace-driven simulator can also approximate memory effects without designing a full instruction set, at the cost of accuracy,

The simulator described here is of sufficient complexity to understand the main concepts, but simple enough to understand in its entirety.

The remainder of this appendix explains the basics of trace-driven simulation, documents how this simulator is implemented and gives an example of usage.

B.1 Basics

Simulation, with varying degrees of fidelity, allows a designer or performance tuner to quantify performance effects including altering design parameters, such as size, organization and speed of different parts of the memory hierarchy.

Ideally a simulator should capture all effects – how instructions behave, interactions with the operating system and the outside world, memory hierarchy and energy usage.

Comprehensive simulations that aim to capture all such effects need to model the complete system, not just the workload of interest. SimOS [Rosenblum et al. 1995] pioneered complete-system simulation in the 1990s. A later project, M5, included ability to model network interactions in a complete system simulation [Binkert et al. 2006]. M5 merged with the GEMS project to create gem5, which is highly configurable, including fine details of the memory hierarchy [Binkert et al. 2011]. This generality is at the cost of complexity: gem5 is a large project and its highly configurable design requires a significant learning curve to get up to speed.

Since general-purpose research-quality simulators are complex to set up, I rely instead for purposes of course material on my own simplified trace-driven simulator, SimpleCacheSim, which uses traces that are easy to generate using a Pin tool as described in Appendix A.

B.2 Implementation

A simulator needs to be configured. For a memory system, the minimum is parameter defining the layers of the hierarchy in simple terms: how big each is, how it is structured into blocks (or lines in cache terminology), access time at each layer and any other details of how a level is organized or overheads necessary for a bare-bones simulation.

In the remainder of this section, I describe the data used to configure a simulation and trace files. I got on to describe data structures used in the implementation, the main functions that drive the simulation and the source and header files.

B.2.1 Data

To configure a simulation, a file must be provided containing the following on each line:

<*bytes*> <*block size*> <*hit time*> <*tag check time*> <*associativity*> <*split*>

These are defined as:

- <*bytes*> – total size of the cache
- <*block size*> – block size in bytes
- <*hit time*> – time for a hit (0 for L1 data accesses – if you assume that data accesses are fully pipelined)
- <*tag check time*> – time to look up tags (not used in hits)
- <*associativity*> – 1 for direct-mapped (DM); a power of 2 $\leq$ <*bytes*>/<*block size*>
- <*split*> – 0 for unified data (D) and instruction (I), 1 for split; ignored except in L1

For the DRAM layer, all numbers are 0 except the hit time, used to cost lowest-level cache (LLC) misses. Infinite DRAM is modelled, i.e., no misses from DRAM. To model misses from DRAM (page faults) accurately would require

simulation of at least part of an operating system and that makes no sense with a single instruction stream.

In a real cache, associativity need not necessarily be a power of 2 though it is easier to implement that way. For example, if you have a 3MiB cache that is 3-way associative, the size of each way is still a power of 2.

On page 162 in Appendix A, I give an example of a trace file. Each line of a trace file is structured as follows:

<*type*> <*hex number*>

where <*type*> is one of:

- `I` – instruction fetch
- `R` – data read
- `W` – data write
- `X` – exception (ignored in this simulation)

Reading the trace file ends at end of file or if `#eof` is read. The `#eof` is added by Pin tools and I choose to leave it like that for consistency with Pin though reading to end of file is easy enough. `X` is an option to simulate a delay for handling an exception (the hex number is the delay) but is not used in examples I develop for architecture; I use this feature for operating system simulations.

B.2.2 Data Structures

In order to maintain abstraction, details of `struct` types is hidden in the C file that implements them; they can only be accessed in other files through a pointer or a function in the C file that defines the type. A `typedef` names `struct` types for convenience.

Defined in `cachesetup.c`

Cache parameters are stored in a `struct CacheSetup` (`CacheSetupT`), containing:

- `totalblocks`, `blocksize`, `hittime`, `lookupoverhead`, `associativity`, `split`

All are as described in the configuration file (except the `0` or `1` value of `split` is stored as type **bool**).

Defined in `cachetypes.c`

A cache is represented as an array of data structures, one per level – except if there is a split L1, in which case the first element of the array is the L1I (instruction) cache and the second the L1D (data) cache. The lowest level of the array represents the DRAM layer. Each level is represented as `struct Cache` (named as type `CacheT` for convenience, using a `typedef` – any name ending in `T` from here on is a type). Each level contains:

- an array of arrays of `RawCacheT`
  - each array at one level represents a way of an N-way associative cache
- latencies for hit time and lookup overhead
- associativity
- whether split
- `assocmask` used in associativity calculations
- array of stats (`AllStatsT`)

The `struct AllStats` (`AllStatsT`) contains `StatsT` values for

- `hitcount`, `misscount`, `replacecount`, `hitcost`, `misscost` (all as pointers since the type of `StatsT` is defined elsewhere)

Defined in `rawcachetypes.c`

`struct RawCache` (`RawCacheT`)

- `Nblocks` (number of blocks in this cache structure)
- `blocksize` in bytes
- array of blocks (each type `CacheBlockT`)
- `addressmask` and `indexmask` (for extracting component of an address)
- `offsetbits` and `indexbits` (number of bits for address components)

A raw cache is direct-mapped (DM) and can be used as a building block. To make an N-way associative cache, implement it as N raw caches, each $\frac{1}{N}$th of the required size. It does not implement any timing as it is a general-purpose building block and actual timing should be based on how it is used.

`struct Cacheblock` (`CacheblockT`) contains the tags and enough of the address to identify the block uniquely.

Defined in `stats.c`

`struct Stats` (`StatsT`) keeps track of statistics for instructions, data reads and data writes.

B.2.3 Main Functions

Files are listed in the order of first function call, starting from `main`.

`cachesim.c`

The simulation starts from the main program contained in this file:

- checks the command line (should give the configuration file name)
- checks that there is at least one usable file name in the workload file (read from `stdin`: redirect a file name on the command line if needed)
- creates a parameter data structure containing the configuration
- calls `simulateMultilevelAssoc` with the parameters to do the simulation
  - if there is more than one trace file in the workload, each is run to completion as a separate process and reported separately
- deallocates the parameters and workload data structures

`cachesetup.c`

Code in this file sets up parameters in a data structure suitable for intializing a multilevel associative cache. This is the glue between the main program and the actual simulation. It contains functions to get the command line, check the parameter file and turn the parameters into a format that can easily be used to initialize.

`simulateMultilevelAssoc.c`

- uses the supplied parameters to configure a multilevel cache with the levels given in the configuration file, with associativity and timing for each recorded in a single data structure
- for each trace file:
  - reads each trace line, discarding "`X`" for exception lines, and passes it to `handleReference`

`multilevelAssoc.c`

This file contains the main implementation of a multilevel associative cache.

- `handleReference` checks for a hit and if the reference is a hit, it updates the statistics; if not, it calls `handleMiss`

- `handleMiss` finds the level at which the block is found (if not in the LLC, it "finds" it in DRAM[1]), works out whether it must replace anything in layers above that and, in the event of any replacement, calls `maintaininclusion` to ensure that multilevel inclusion is maintained.

Maintaining multilevel inclusion is one of the more complicated details. Anything evicted from a given layer below L1 should result in eviction of a copy in any higher level cache. Multilevel inclusion is especially important for multiprocessor cache protocols as this property ensures that anything in a higher level cache is also in the LLC, so coherence protocols can start with the LLC tags.

`rawcache.c`

This file implements various utility functions for basic operations on a cache in addition to the following:

- `rawCacheHit` checks if there is a hit in a DM cache (possibly a way in an associative cache)
- `insert` adds a block by setting its tag as valid and storing the address bits
- `mustWriteback` returns **true** if block should be written back before replacement

B.2.4 Code files

Source files

- `IOutils.c` – open a file, find out its size
- `cachesetup.c` – create and access cache parameters
- `cachesim.c` – main program: sets up, launches,ends simulation
- `error.c` – reports and handles errors (option to exit)
- `get_args.c` – opens and reads configuration file named on command line
- `multilevelAssoc.c` – implements associative multilevel cache simulation
- `rawcache.c` – implements a single DM cache with no timing
- `readfile.c` – read file into buffer as a `'\0'`-terminated string
- `readtrace.c` – read next line from the trace file
- `simulateMultilevelAssoc.c` – pass non-exception trace records to simulator
- `stats.c` – keep track of fetch, read, write stats in a `struct`

[1] DRAM contents is not modelled; all references are treated as hits.

- `stringutils.c` – turn a buffer of lines into an array, one string per line
- `workload.c` – manage a list of trace files

Header files

All provide interfaces to the source files of similar name, except for `generaltypes.h`:

- `IOutils.h`
- `cachesetup.h`
- `error.h`
- `generaltypes.h` – names for widely-used types like sizes, counters
- `get_args.h`
- `multilevelAssoc.h`
- `rawcache.h`
- `readfile.h`
- `readtrace.h`
- `simulateMultilevelAssoc.h`
- `stats.h`
- `stringutils.h`
- `workload.h`

B.3 Usage

On a typical Linux machine (tested on Ubuntu and Mac), navigate to a suitable directory and download the repository from GitHub:

```
$ git clone https://github.com/philip-mach/SimpleCacheSim.git
$ cd SimpleCacheSim
```

You should now be in a directory `SimpleCacheSim` that contains everything: the source and header files, a `Data` directory and a `Makefile`. The data directory contains:

- `L3-unified-2way.conf` – a sample configuration file
- `test-small.trace` – a minimal trace file
- `test-small.workload` – a file using the minimal trace file
- `test.trace` – a longer trace file
- `test.workload` – a workload file using the longer trace file

There is more detail of how all this works in the `README.md` file. This displays with formatting on the GitHub site (it is in markdown format). The origins of

the trace files are lost in the mists of time but most likely are a truncated version of tracing a run of `/bin/ls` (the default example in the Pin documentation – see page 162).

To make a runnable version on any Unix platform (including Linux and Mac), type on the command line (note that the dollar sign at the beginning signifies the command prompt: don't type that):

```
$ make
```

Here is an example of a run of the simulator.

```
$ ./cachesim Data/L3-unified-2way.conf < Data/test.workload
blks blksize hitT lookupT assoc split? Total Bytes
L1: 1024 32 1 1 1 0 32768
L2: 8192 32 10 2 2 0 262144
L3: 65536 64 30 5 2 0 4194304
DRAM: 120
workoad [0], 3 levels
level Hits misses incl. hit t miss t
$[L1] 954019 14082 61 954019 1093575
$[L2] 5729 8353 0 0 313370
$[L3] 1958 6395 0 0 807449
Total elapsed time 3168413, total hits 961706, total misses 28830,
      evictions for inclusion 61; instructions: 598247
```

I split the last output line to fit on the page. Note that output lines are distinguished by not starting with the command prompt – though it may look confusing that I use the dollar sign as an abbreviation for "cache"[2] in the output summary.

B.3.1 Configuration file

Figure B.1 illustrates how a configuration file is structured. Each line contains a single layer of the memory hierarchy, except that a split L1 cache is represented in 2 lines (with a 1 as the last number on the first line to indicate this). This last position is ignored in all lines besides the first. The other exception is the DRAM configuration, which appears last and only contains a number for access latency since misses from DRAM are not modelled; other positions are filled with zeros (but are ignored in the simulator).

All numbers representing memory sizes including ways of a cache are required to be powers of 2 (except DRAM, which is not configured to have a specific size,

[2]Not an original architecture pun.

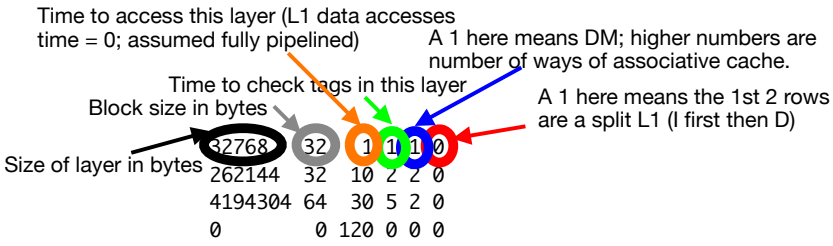


Figure B.1: Example of a configuration file. *Each line represents a separate layer, unless you have a split L1 cache, in which case the first line is L1I (instruction) and the second line L1D (data). DRAM only has one parameter that is used, access time (misses from DRAM are not modelled – zeros are not permitted for sizes in other layers but are ignored in the DRAM layer).*

since misses – aka page faults – aren't modelled). The size of each cache level is given in bytes: the size must be a multiple of a power of 2 times the block (line) size.

Time to access a layer is the time for a hit; for L1D, this is assumed to be zero since data accesses should be fully pipelined and the instruction latency is already accounted for in the hit time for L1I (instruction access). Time to check tags is used when a miss is handled: this is additional overhead for checking for replacements. The next number represents the number of ways and must also be a power of 2. Since total cache size is also a power of 2, this ensures that each way's size is also a power of 2.

B.3.2 Output

For each level, the following is reported:

- level – given from L1 down, not including DRAM
- hits – the number of accesses found in that level
- misses – the number of accesses *not* found in that level
- evictions to maintain inclusion – the number of times a block had to be removed because it is no longer represented at a lower level
- hit time – elapsed time spent on hits to that level
- miss time – elapsed time spent on misses from that level

Hit times are only calculated for L1 as hits in each lower layer are accounted for in miss time for the layer above. Accordingly, hit and miss times are not given for DRAM as there are no misses and hits in DRAM are taken into account in calculating miss time from the LLC.