What every programmer should know about memory

September 21, 2007

This article was contributed by Ulrich Drepper

[*Editor's introduction: Ulrich Drepper recently approached us asking if we would be interested in publishing a lengthy document he had written on how memory and software interact. We did not have to look at the text for long to realize that it would be of interest to many LWN readers. Memory usage is often the determining factor in how software performs, but good information on how to avoid memory bottlenecks is hard to find. This series of articles should change that situation.*

The original document prints out at over 100 pages. We will be splitting it into about seven segments, each run 1-2 weeks after its predecessor. Once the entire series is out, Ulrich will be releasing the full text.

Reformatting the text from the original LaTeX has been a bit of a challenge, but the results, hopefully, will be good. For ease of online reading, Ulrich's footnotes have been placed {inline in the text}. Hyperlinked cross-references (and [bibliography references]) will not be possible until the full series is published.

Many thanks to Ulrich for allowing LWN to publish this material; we hope that it will lead to more memory-efficient software across our systems in the near future.]

目录

1 Introduction

In the early days computers were much simpler. The various components of a system, such as the CPU, memory, mass storage, and network interfaces, were developed together and, as a result, were quite balanced in their performance. For example, the memory and network interfaces were not (much) faster than the CPU at providing data.

This situation changed once the basic structure of computers stabilized and hardware developers concentrated on optimizing individual subsystems. Suddenly the performance of some components of the computer fell significantly behind and bottlenecks developed. This was especially true for mass storage and memory subsystems which, for cost reasons, improved more slowly relative to other components.

The slowness of mass storage has mostly been dealt with using software techniques: operating systems keep most often used (and most likely to be used) data in main memory, which can be accessed at a rate orders of magnitude faster than the hard disk. Cache storage was added to the storage devices themselves, which requires no changes in the operating system to increase performance. {*Changes are needed, however, to guarantee data integrity when using storage device caches.*} For the purposes of this paper, we will not go into more details of software optimizations for the mass storage access.

Unlike storage subsystems, removing the main memory as a bottleneck has proven much more difficult and almost all solutions require changes to the hardware. Today these changes mainly come in the following forms:

- RAM hardware design (speed and parallelism).
- Memory controller designs.
- CPU caches.
- Direct memory access (DMA) for devices.

For the most part, this document will deal with CPU caches and some effects of memory controller design. In the process of exploring these topics, we will explore DMA and bring it into the larger picture. However, we will start with an overview of the design for today's commodity hardware. This is a prerequisite to understanding the problems and the limitations of efficiently using memory subsystems. We will also learn about, in some detail, the different types of RAM and illustrate why these differences still exist.

This document is in no way all inclusive and final. It is limited to commodity hardware and further limited to a subset of that hardware. Also, many topics will be discussed in just enough detail for the goals of this paper. For such topics, readers are recommended to find more detailed documentation.

When it comes to operating-system-specific details and solutions, the text exclusively describes Linux. At no time will it contain any information about other OSes. The author has no interest in discussing the implications for other OSes. If the reader thinks s/he has to use a different OS they have to go to their vendors and demand they write documents similar to this one.

One last comment before the start. The text contains a number of occurrences of the term “usually” and other, similar qualifiers. The technology discussed here exists in many, many variations in the real world and this paper only addresses the most common, mainstream versions. It is rare that absolute statements can be made about this technology, thus the qualifiers.

1.1 Document Structure

This document is mostly for software developers. It does not go into enough technical details of the hardware to be useful for hardware-oriented readers. But before we can go into the practical information for developers a lot of groundwork must be laid.

To that end, the second section describes random-access memory (RAM) in technical detail. This section's content is nice to know but not absolutely critical to be able to understand the later sections. Appropriate back references to the section are added in places where the content is required so that the anxious reader could skip most of this section at first.

The third section goes into a lot of details of CPU cache behavior. Graphs have been used to keep the text from being as dry as it would otherwise be. This content is essential for an understanding of the rest of the document. Section 4 describes briefly how virtual memory is implemented. This is also required groundwork for the rest.

Section 5 goes into a lot of detail about Non Uniform Memory Access (NUMA) systems.

Section 6 is the central section of this paper. It brings together all the previous sections' information and gives programmers advice on how to write code which performs well in the various situations. The very impatient

reader could start with this section and, if necessary, go back to the earlier sections to freshen up the knowledge of the underlying technology.

Section 7 introduces tools which can help the programmer do a better job. Even with a complete understanding of the technology it is far from obvious where in a non-trivial software project the problems are. Some tools are necessary.

In section 8 we finally give an outlook of technology which can be expected in the near future or which might just simply be good to have.

1.2 Reporting Problems

The author intends to update this document for some time. This includes updates made necessary by advances in technology but also to correct mistakes. Readers willing to report problems are encouraged to send email.

1.3 Thanks

I would like to thank Johnray Fuller and especially Jonathan Corbet for taking on part of the daunting task of transforming the author's form of English into something more traditional. Markus Armbruster provided a lot of valuable input on problems and omissions in the text.

1.4 About this Document

The title of this paper is an homage to David Goldberg's classic paper “What Every Computer Scientist Should Know About Floating-Point Arithmetic” [goldberg]. Goldberg's paper is still not widely known, although it should be a prerequisite for anybody daring to touch a keyboard for serious programming.

2 Commodity Hardware Today

Understanding commodity hardware is important because specialized hardware is in retreat. Scaling these days is most often achieved horizontally instead of vertically, meaning today it is more cost-effective to use many smaller, connected commodity computers instead of a few really large and exceptionally fast (and expensive) systems. This is the case because fast and inexpensive network hardware is widely available. There are still situations where the large specialized systems have their place and these systems still provide a business opportunity, but the overall market is dwarfed by the commodity hardware market. Red Hat, as of 2007, expects that for future products, the “standard building blocks” for most data centers will be a

computer with up to four sockets, each filled with a quad core CPU that, in the case of Intel CPUs, will be hyper-threaded. {*Hyper-threading enables a single processor core to be used for two or more concurrent executions with just a little extra hardware.*} This means the standard system in the data center will have up to 64 virtual processors. Bigger machines will be supported, but the quad socket, quad CPU core case is currently thought to be the sweet spot and most optimizations are targeted for such machines.

Large differences exist in the structure of commodity computers. That said, we will cover more than 90% of such hardware by concentrating on the most important differences. Note that these technical details tend to change rapidly, so the reader is advised to take the date of this writing into account.

Over the years the personal computers and smaller servers standardized on a chipset with two parts: the Northbridge and Southbridge. Figure 2.1 shows this structure.

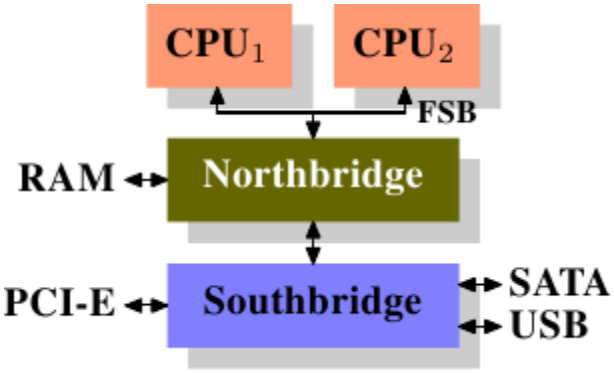


Figure 2.1: Structure with Northbridge and Southbridge

All CPUs (two in the previous example, but there can be more) are connected via a common bus (the Front Side Bus, FSB) to the Northbridge. The Northbridge contains, among other things, the memory controller, and its implementation determines the type of RAM chips used for the computer. Different types of RAM, such as DRAM, Rambus, and SDRAM, require different memory controllers.

To reach all other system devices, the Northbridge must communicate with the Southbridge. The Southbridge, often referred to as the I/O bridge, handles communication with devices through a variety of different buses. Today the PCI, PCI Express, SATA, and USB buses are of most importance, but PATA, IEEE 1394, serial, and parallel ports are also supported by the Southbridge. Older systems had AGP slots which were attached to the Northbridge. This was done for performance reasons related to insufficiently fast connections between the Northbridge and Southbridge. However, today the PCI-E slots are all connected to the Southbridge.

Such a system structure has a number of noteworthy consequences:

- All data communication from one CPU to another must travel over the same bus used to communicate with the Northbridge.
- All communication with RAM must pass through the Northbridge.
- The RAM has only a single port. {*We will not discuss multi-port RAM in this document as this type of RAM is not found in commodity hardware, at least not in places where the programmer has access to it. It can be found in specialized hardware such as network routers which depend on utmost speed.*}
- Communication between a CPU and a device attached to the Southbridge is routed through the Northbridge.

A couple of bottlenecks are immediately apparent in this design. One such bottleneck involves access to RAM for devices. In the earliest days of the PC, all communication with devices on either bridge had to pass through the CPU, negatively impacting overall system performance. To work around this problem some devices became capable of direct memory access (DMA). DMA allows devices, with the help of the Northbridge, to store and receive data in RAM directly without the intervention of the CPU (and its inherent performance cost). Today all high-performance devices attached to any of the buses can utilize DMA. While this greatly reduces the workload on the CPU, it also creates contention for the bandwidth of the Northbridge as DMA requests compete with RAM access from the CPUs. This problem, therefore, must to be taken into account.

A second bottleneck involves the bus from the Northbridge to the RAM. The exact details of the bus depend on the memory types deployed. On older systems there is only one bus to all the RAM chips, so parallel access is not possible. Recent RAM types require two separate buses (or channels as they are called for DDR2, see Figure 2.8) which doubles the available bandwidth. The Northbridge interleaves memory access across the channels. More recent memory technologies (FB-DRAM, for instance) add more channels.

With limited bandwidth available, it is important to schedule memory access in ways that minimize delays. As we will see, processors are much faster and must wait to access memory, despite the use of CPU caches. If multiple hyper-threads, cores, or processors access memory at the same time, the wait times for memory access are even longer. This is also true for DMA operations.

There is more to accessing memory than concurrency, however. Access patterns themselves also greatly influence the performance of the memory subsystem, especially with multiple memory channels. Refer to Section 2.2 for more details of RAM access patterns.

On some more expensive systems, the Northbridge does not actually contain the memory controller. Instead the Northbridge can be connected to a number of external memory controllers (in the following example, four of them).

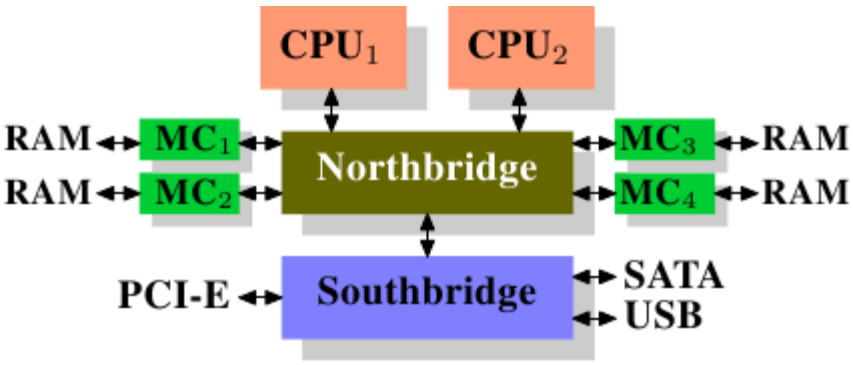


Figure 2.2: Northbridge with External Controllers

The advantage of this architecture is that more than one memory bus exists and therefore total bandwidth increases. This design also supports more memory. Concurrent memory access patterns reduce delays by simultaneously accessing different memory banks. This is especially true when multiple processors are directly connected to the Northbridge, as in Figure 2.2. For such a design, the primary limitation is the internal bandwidth of the Northbridge, which is phenomenal for this architecture (from Intel). {*For completeness it should be mentioned that such a memory controller arrangement can be used for other purposes such as "memory RAID" which is useful in combination with hotplug memory.*}

Using multiple external memory controllers is not the only way to increase memory bandwidth. One other increasingly popular way is to integrate memory controllers into the CPUs and attach memory to each CPU. This architecture is made popular by SMP systems based on AMD's Opteron processor. Figure 2.3 shows such a system. Intel will have support for the Common System Interface (CSI) starting with the Nehalem processors; this is basically the same approach: an integrated memory controller with the possibility of local memory for each processor.

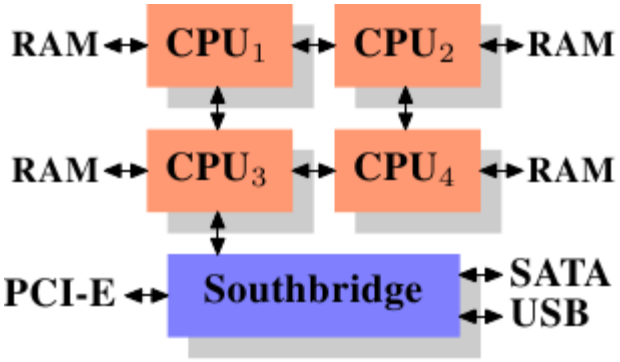


Figure 2.3: Integrated Memory Controller

With an architecture like this there are as many memory banks available as there are processors. On a quad-CPU machine the memory bandwidth is quadrupled without the need for a complicated Northbridge with enormous bandwidth. Having a memory controller integrated into the CPU has some additional advantages; we will not dig deeper into this technology here.

There are disadvantages to this architecture, too. First of all, because the machine still has to make all the memory of the system accessible to all processors, the memory is not uniform anymore (hence the name NUMA - Non-Uniform Memory Architecture - for such an architecture). Local memory (memory attached to a processor) can be accessed with the usual speed. The situation is different when memory attached to another processor is accessed. In this case the interconnects between the processors have to be used. To access memory attached to CPU_2 from CPU_1requires communication across one interconnect. When the same CPU accesses memory attached to CPU_4 two interconnects have to be crossed.

Each such communication has an associated cost. We talk about “NUMA factors” when we describe the extra time needed to access remote memory. The example architecture in Figure 2.3 has two levels for each CPU: immediately adjacent CPUs and one CPU which is two interconnects away. With more complicated machines the number of levels can grow significantly. There are also machine architectures (for instance IBM's x445 and SGI's Altix series) where there is more than one type of connection. CPUs are organized into nodes; within a node the time to access the memory might be uniform or have only small NUMA factors. The connection between nodes can be very expensive, though, and the NUMA factor can be quite high.

Commodity NUMA machines exist today and will likely play an even greater role in the future. It is expected that, from late 2008 on, every SMP machine will use NUMA. The costs associated with NUMA make it important to recognize when a program is running on a NUMA machine. In

Section 5 we will discuss more machine architectures and some technologies the Linux kernel provides for these programs.

Beyond the technical details described in the remainder of this section, there are several additional factors which influence the performance of RAM. They are not controllable by software, which is why they are not covered in this section. The interested reader can learn about some of these factors in Section 2.1. They are really only needed to get a more complete picture of RAM technology and possibly to make better decisions when purchasing computers.

The following two sections discuss hardware details at the gate level and the access protocol between the memory controller and the DRAM chips. Programmers will likely find this information enlightening since these details explain why RAM access works the way it does. It is optional knowledge, though, and the reader anxious to get to topics with more immediate relevance for everyday life can jump ahead to Section 2.2.5.

2.1 RAM Types

There have been many types of RAM over the years and each type varies, sometimes significantly, from the other. The older types are today really only interesting to the historians. We will not explore the details of those. Instead we will concentrate on modern RAM types; we will only scrape the surface, exploring some details which are visible to the kernel or application developer through their performance characteristics.

The first interesting details are centered around the question why there are different types of RAM in the same machine. More specifically, why there are both static RAM (SRAM {*In other contexts SRAM might mean "synchronous RAM".*}) and dynamic RAM (DRAM). The former is much faster and provides the same functionality. Why is not all RAM in a machine SRAM? The answer is, as one might expect, cost. SRAM is much more expensive to produce and to use than DRAM. Both these cost factors are important, the second one increasing in importance more and more. To understand these difference we look at the implementation of a bit of storage for both SRAM and DRAM.

In the remainder of this section we will discuss some low-level details of the implementation of RAM. We will keep the level of detail as low as possible. To that end, we will discuss the signals at a "logic level" and not at a level a hardware designer would have to use. That level of detail is unnecessary for our purpose here.

2.1.1 Static RAM

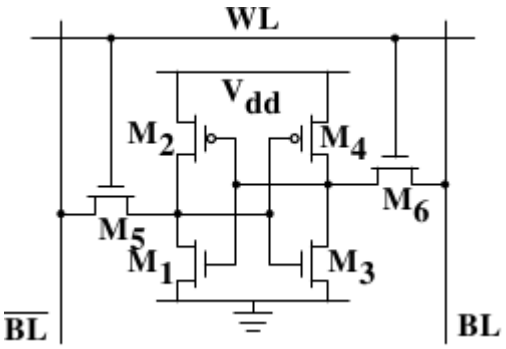


Figure 2.4: 6-T Static RAM

Figure 2.4 shows the structure of a 6 transistor SRAM cell. The core of this cell is formed by the four transistors $\mathbf{M_1}$ to $\mathbf{M_4}$ which form two cross-coupled inverters. They have two stable states, representing 0 and 1 respectively. The state is stable as long as power on $\mathbf{V_{dd}}$ is available.

If access to the state of the cell is needed the word access line **WL** is raised. This makes the state of the cell immediately available for reading on **BL** and **BL**. If the cell state must be overwritten the **BL** and **BL** lines are first set to the desired values and then **WL** is raised. Since the outside drivers are stronger than the four transistors ($\mathbf{M_1}$ through $\mathbf{M_4}$) this allows the old state to be overwritten.

See [sramwiki] for a more detailed description of the way the cell works. For the following discussion it is important to note that

- one cell requires six transistors. There are variants with four transistors but they have disadvantages.
- maintaining the state of the cell requires constant power.
- the cell state is available for reading almost immediately once the word access line **WL** is raised. The signal is as rectangular (changing quickly between the two binary states) as other transistor-controlled signals.
- the cell state is stable, no refresh cycles are needed.

There are other, slower and less power-hungry, SRAM forms available, but those are not of interest here since we are looking at fast RAM. These slow variants are mainly interesting because they can be more easily used in a system than dynamic RAM because of their simpler interface.

2.1.2 Dynamic RAM

Dynamic RAM is, in its structure, much simpler than static RAM. Figure 2.5 shows the structure of a usual DRAM cell design. All it consists of is

one transistor and one capacitor. This huge difference in complexity of course means that it functions very differently than static RAM.

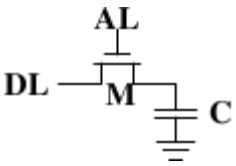


Figure 2.5: 1-T Dynamic RAM

A dynamic RAM cell keeps its state in the capacitor **C**. The transistor **M** is used to guard the access to the state. To read the state of the cell the access line **AL** is raised; this either causes a current to flow on the data line **DL** or not, depending on the charge in the capacitor. To write to the cell the data line **DL** is appropriately set and then **AL** is raised for a time long enough to charge or drain the capacitor.

There are a number of complications with the design of dynamic RAM. The use of a capacitor means that reading the cell discharges the capacitor. The procedure cannot be repeated indefinitely, the capacitor must be recharged at some point. Even worse, to accommodate the huge number of cells (chips with 10^9 or more cells are now common) the capacity to the capacitor must be low (in the femto-farad range or lower). A fully charged capacitor holds a few 10's of thousands of electrons. Even though the resistance of the capacitor is high (a couple of tera-ohms) it only takes a short time for the capacity to dissipate. This problem is called “leakage”.

This leakage is why a DRAM cell must be constantly refreshed. For most DRAM chips these days this refresh must happen every 64ms. During the refresh cycle no access to the memory is possible. For some workloads this overhead might stall up to 50% of the memory accesses
(see [highperfdram]).

A second problem resulting from the tiny charge is that the information read from the cell is not directly usable. The data line must be connected to a sense amplifier which can distinguish between a stored 0 or 1 over the whole range of charges which still have to count as 1.

A third problem is that charging and draining a capacitor is not instantaneous. The signals received by the sense amplifier are not rectangular, so a conservative estimate as to when the output of the cell is usable has to be used. The formulas for charging and discharging a capacitor are

$$
\begin{aligned}
Q_{\text{Charge}}(t) &= Q_0(1 - e^{-\frac{t}{RC}}) \\
Q_{\text{Discharge}}(t) &= Q_0 e^{-\frac{t}{RC}}
\end{aligned}
$$

This means it takes some time (determined by the capacity C and resistance R) for the capacitor to be charged and discharged. It also means that the current which can be detected by the sense amplifiers is not immediately available. Figure 2.6 shows the charge and discharge curves. The X—axis is measured in units of RC (resistance multiplied by capacitance) which is a unit of time.

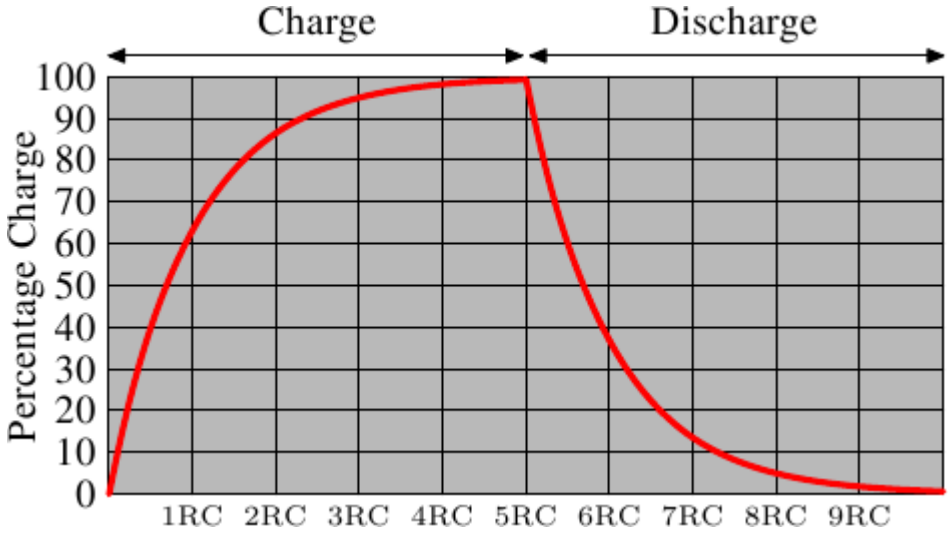


Figure 2.6: Capacitor Charge and Discharge Timing

Unlike the static RAM case where the output is immediately available when the word access line is raised, it will always take a bit of time until the capacitor discharges sufficiently. This delay severely limits how fast DRAM can be.

The simple approach has its advantages, too. The main advantage is size. The chip real estate needed for one DRAM cell is many times smaller than that of an SRAM cell. The SRAM cells also need individual power for the transistors maintaining the state. The structure of the DRAM cell is also simpler and more regular which means packing many of them close together on a die is simpler.

Overall, the (quite dramatic) difference in cost wins. Except in specialized hardware — network routers, for example — we have to live with main memory which is based on DRAM. This has huge implications on the programmer which we will discuss in the remainder of this paper. But first we need to look into a few more details of the actual use of DRAM cells.

2.1.3 DRAM Access

A program selects a memory location using a virtual address. The processor translates this into a physical address and finally the memory controller selects the RAM chip corresponding to that address. To select the individual memory cell on the RAM chip, parts of the physical address are passed on in the form of a number of address lines.

It would be completely impractical to address memory locations individually from the memory controller: 4GB of RAM would require 2^{32} address lines. Instead the address is passed encoded as a binary number using a smaller set of address lines. The address passed to the DRAM chip this way must be demultiplexed first. A demultiplexer with *N* address lines will have 2^N output lines. These output lines can be used to select the memory cell. Using this direct approach is no big problem for chips with small capacities.

But if the number of cells grows this approach is not suitable anymore. A chip with 1Gbit {*I hate those SI prefixes. For me a giga-bit will always be* 2^{30} *and not* 10^9 *bits.*} capacity would need 30 address lines and 2^{30} select lines. The size of a demultiplexer increases exponentially with the number of input lines when speed is not to be sacrificed. A demultiplexer for 30 address lines needs a whole lot of chip real estate in addition to the complexity (size and time) of the demultiplexer. Even more importantly, transmitting 30 impulses on the address lines synchronously is much harder than transmitting "only" 15 impulses. Fewer lines have to be laid out at exactly the same length or timed appropriately. {*Modern DRAM types like DDR3 can automatically adjust the timing but there is a limit as to what can be tolerated.*}

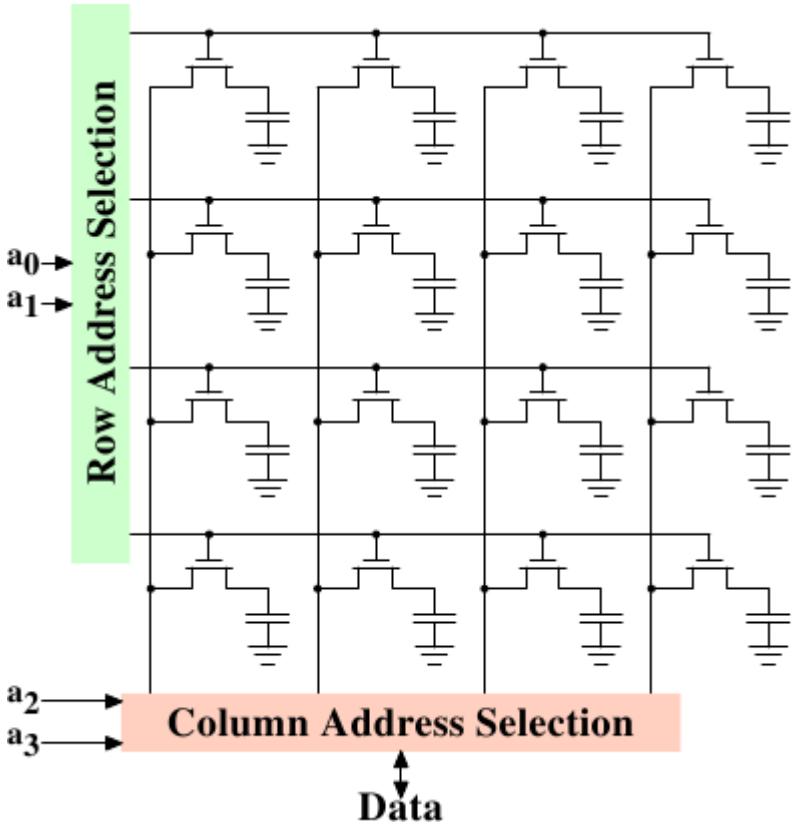


Figure 2.7: Dynamic RAM Schematic

Figure 2.7 shows a DRAM chip at a very high level. The DRAM cells are organized in rows and columns. They could all be aligned in one row but then the DRAM chip would need a huge demultiplexer. With the array approach the design can get by with one demultiplexer and one multiplexer of half the size. {*Multiplexers and demultiplexers are equivalent and the multiplexer here needs to work as a demultiplexer when writing. So we will drop the differentiation from now on.*} This is a huge saving on all fronts. In the example the address lines $\mathbf{a_0}$ and $\mathbf{a_1}$ through the *row address selection* (**RAS**) demultiplexer select the address lines of a whole row of cells. When reading, the content of all cells is thusly made available to the *column address selection* (**CAS**) {*The line over the name indicates that the signal is negated*} multiplexer. Based on the address lines $\mathbf{a_2}$ and $\mathbf{a_3}$ the content of one column is then made available to the data pin of the DRAM chip. This happens many times in parallel on a number of DRAM chips to produce a total number of bits corresponding to the width of the data bus.

For writing, the new cell value is put on the data bus and, when the cell is selected using the **RAS** and **CAS**, it is stored in the cell. A pretty straightforward design. There are in reality — obviously — many more complications. There need to be specifications for how much delay there is after the signal before the data will be available on the data bus for reading. The capacitors do not unload instantaneously, as described in the previous

section. The signal from the cells is so weak that it needs to be amplified. For writing it must be specified how long the data must be available on the bus after the **RAS** and **CAS** is done to successfully store the new value in the cell (again, capacitors do not fill or drain instantaneously). These timing constants are crucial for the performance of the DRAM chip. We will talk about this in the next section.

A secondary scalability problem is that having 30 address lines connected to every RAM chip is not feasible either. Pins of a chip are a precious resources. It is "bad" enough that the data must be transferred as much as possible in parallel (e.g., in 64 bit batches). The memory controller must be able to address each RAM module (collection of RAM chips). If parallel access to multiple RAM modules is required for performance reasons and each RAM module requires its own set of 30 or more address lines, then the memory controller needs to have, for 8 RAM modules, a whopping 240+ pins only for the address handling.

To counter these secondary scalability problems DRAM chips have, for a long time, multiplexed the address itself. That means the address is transferred in two parts. The first part consisting of address bits $\mathbf{a_0}$ and $\mathbf{a_1}$ in the example in Figure 2.7) select the row. This selection remains active until revoked. Then the second part, address bits $\mathbf{a_2}$ and $\mathbf{a_3}$, select the column. The crucial difference is that only two external address lines are needed. A few more lines are needed to indicate when the **RAS** and **CAS** signals are available but this is a small price to pay for cutting the number of address lines in half. This address multiplexing brings its own set of problems, though. We will discuss them in Section 2.2.

2.1.4 Conclusions

Do not worry if the details in this section are a bit overwhelming. The important things to take away from this section are:

- there are reasons why not all memory is SRAM
- memory cells need to be individually selected to be used
- the number of address lines is directly responsible for the cost of the memory controller, motherboards, DRAM module, and DRAM chip
- it takes a while before the results of the read or write operation are available

The following section will go into more details about the actual process of accessing DRAM memory. We are not going into more details of accessing SRAM, which is usually directly addressed. This happens for speed and because the SRAM memory is limited in size. SRAM is currently used in CPU caches and on-die where the connections are small and fully under

control of the CPU designer. CPU caches are a topic which we discuss later but all we need to know is that SRAM cells have a certain maximum speed which depends on the effort spent on the SRAM. The speed can vary from only slightly slower than the CPU core to one or two orders of magnitude slower.

2.2 DRAM Access Technical Details

In the section introducing DRAM we saw that DRAM chips multiplex the addresses in order to save resources. We also saw that accessing DRAM cells takes time since the capacitors in those cells do not discharge instantaneously to produce a stable signal; we also saw that DRAM cells must be refreshed. Now it is time to put this all together and see how all these factors determine how the DRAM access has to happen.

We will concentrate on current technology; we will not discuss asynchronous DRAM and its variants as they are simply not relevant anymore. Readers interested in this topic are referred to [highperfdram] and [arstechtwo]. We will also not talk about Rambus DRAM (RDRAM) even though the technology is not obsolete. It is just not widely used for system memory. We will concentrate exclusively on Synchronous DRAM (SDRAM) and its successors Double Data Rate DRAM (DDR).

Synchronous DRAM, as the name suggests, works relative to a time source. The memory controller provides a clock, the frequency of which determines the speed of the Front Side Bus (FSB) — the memory controller interface used by the DRAM chips. As of this writing, frequencies of 800MHz, 1,066MHz, or even 1,333MHz are available with higher frequencies (1,600MHz) being announced for the next generation. This does not mean the frequency used on the bus is actually this high. Instead, today's buses are double- or quad-pumped, meaning that data is transported two or four times per cycle. Higher numbers sell so the manufacturers like to advertise a quad-pumped 200MHz bus as an “effective” 800MHz bus.

For SDRAM today each data transfer consists of 64 bits — 8 bytes. The transfer rate of the FSB is therefore 8 bytes multiplied by the effective bus frequency (6.4GB/s for the quad-pumped 200MHz bus). That sounds a lot but it is the burst speed, the maximum speed which will never be surpassed. As we will see now the protocol for talking to the RAM modules has a lot of downtime when no data can be transmitted. It is exactly this downtime which we must understand and minimize to achieve the best performance.

2.2.1 Read Access Protocol

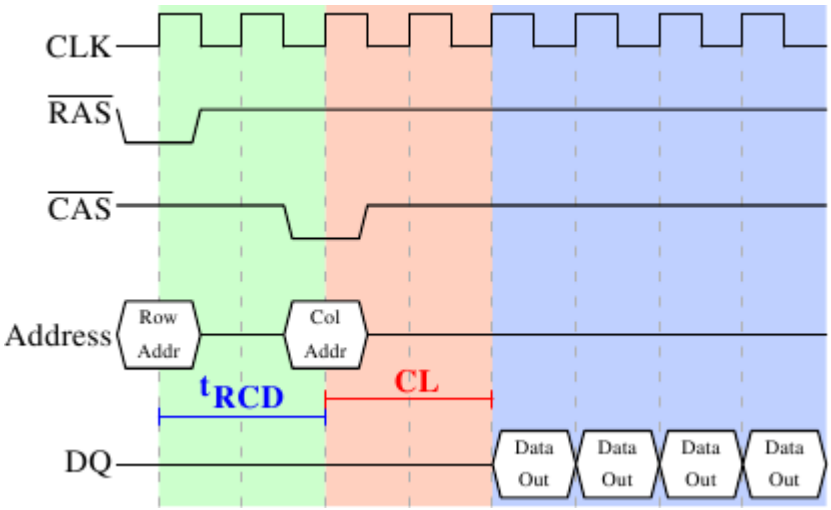


Figure 2.8: SDRAM Read Access Timing

Figure 2.8 shows the activity on some of the connectors of a DRAM module which happens in three differently colored phases. As usual, time flows from left to right. A lot of details are left out. Here we only talk about the bus clock, **RAS** and **CAS** signals, and the address and data buses. A read cycle begins with the memory controller making the row address available on the address bus and lowering the **RAS** signal. All signals are read on the rising edge of the clock (CLK) so it does not matter if the signal is not completely square as long as it is stable at the time it is read. Setting the row address causes the RAM chip to start latching the addressed row.

The **CAS** signal can be sent after $\mathbf{t_{RCD}}$ (**RAS**-to-**CAS** Delay) clock cycles. The column address is then transmitted by making it available on the address bus and lowering the **CAS** line. Here we can see how the two parts of the address (more or less halves, nothing else makes sense) can be transmitted over the same address bus.

Now the addressing is complete and the data can be transmitted. The RAM chip needs some time to prepare for this. The delay is usually called**CAS** Latency (CL). In Figure 2.8 the **CAS** latency is 2. It can be higher or lower, depending on the quality of the memory controller, motherboard, and DRAM module. The latency can also have half values. With CL=2.5 the first data would be available at the first *falling* flank in the blue area.

With all this preparation to get to the data it would be wasteful to only transfer one data word. This is why DRAM modules allow the memory controller to specify how much data is to be transmitted. Often the choice is between 2, 4, or 8 words. This allows filling entire lines in the caches without a new **RAS**/**CAS** sequence. It is also possible for the memory

controller to send a new **CAS** signal without resetting the row selection. In this way, consecutive memory addresses can be read from or written to significantly faster because the **RAS** signal does not have to be sent and the row does not have to be deactivated (see below). Keeping the row "open" is something the memory controller has to decide. Speculatively leaving it open all the time has disadvantages with real-world applications (see [highperfdram]). Sending new **CAS** signals is only subject to the Command Rate of the RAM module (usually specified as T*x*, where *x* is a value like 1 or 2; it will be 1 for high-performance DRAM modules which accept new commands every cycle).

In this example the SDRAM spits out one word per cycle. This is what the first generation does. DDR is able to transmit two words per cycle. This cuts down on the transfer time but does not change the latency. In principle, DDR2 works the same although in practice it looks different. There is no need to go into the details here. It is sufficient to note that DDR2 can be made faster, cheaper, more reliable, and is more energy efficient (see [ddrtwo] for more information).

2.2.2 Precharge and Activation

Figure 2.8 does not cover the whole cycle. It only shows parts of the full cycle of accessing DRAM. Before a new **RAS** signal can be sent the currently latched row must be deactivated and the new row must be precharged. We can concentrate here on the case where this is done with an explicit command. There are improvements to the protocol which, in some situations, allows this extra step to be avoided. The delays introduced by precharging still affect the operation, though.

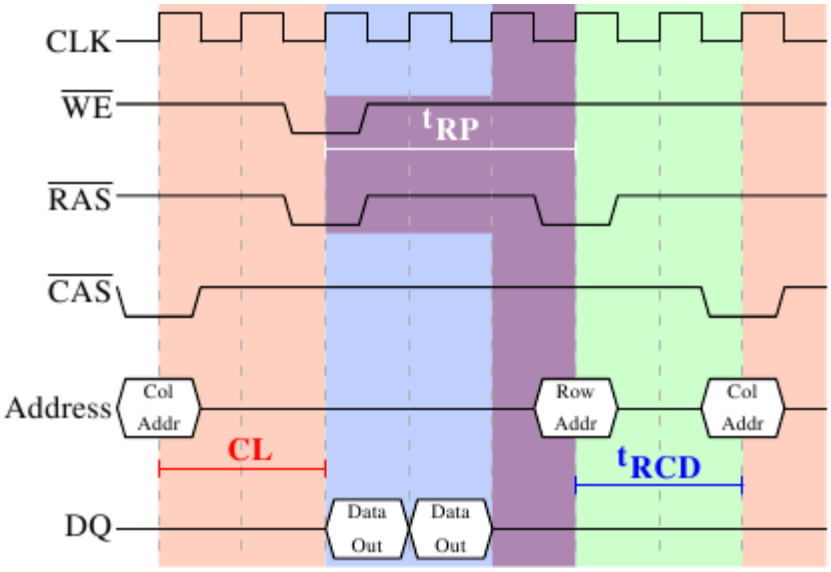


Figure 2.9: SDRAM Precharge and Activation

Figure 2.9 shows the activity starting from one **CAS** signal to the **CAS** signal for another row. The data requested with the first **CAS** signal is available as before, after CL cycles. In the example two words are requested which, on a simple SDRAM, takes two cycles to transmit. Alternatively, imagine four words on a DDR chip.

Even on DRAM modules with a command rate of one the precharge command cannot be issued right away. It is necessary to wait as long as it takes to transmit the data. In this case it takes two cycles. This happens to be the same as CL but that is just a coincidence. The precharge signal has no dedicated line; instead, some implementations issue it by lowering the Write Enable (**WE**) and **RAS** line simultaneously. This combination has no useful meaning by itself (see [micronddr] for encoding details).

Once the precharge command is issued it takes $\mathbf{t_{RP}}$ (Row Precharge time) cycles until the row can be selected. In Figure 2.9 much of the time (indicated by the purplish color) overlaps with the memory transfer (light blue). This is good! But $\mathbf{t_{RP}}$ is larger than the transfer time and so the next **RAS** signal is stalled for one cycle.

If we were to continue the timeline in the diagram we would find that the next data transfer happens 5 cycles after the previous one stops. This means the data bus is only in use two cycles out of seven. Multiply this with the FSB speed and the theoretical 6.4GB/s for a 800MHz bus become 1.8GB/s. That is bad and must be avoided. The techniques described in Section 6 help to raise this number. But the programmer usually has to do her share.

There is one more timing value for a SDRAM module which we have not discussed. In Figure 2.9 the precharge command was only limited by the data transfer time. Another constraint is that an SDRAM module needs time after a **RAS** signal before it can precharge another row (denoted as $\mathbf{t_{RAS}}$). This number is usually pretty high, in the order of two or three times the $\mathbf{t_{RP}}$ value. This is a problem if, after a **RAS** signal, only one **CAS**signal follows and the data transfer is finished in a few cycles. Assume that in Figure 2.9 the initial **CAS** signal was preceded directly by a**RAS** signal and that $\mathbf{t_{RAS}}$ is 8 cycles. Then the precharge command would have to be delayed by one additional cycle since the sum of $\mathbf{t_{RCD}}$, CL, and $\mathbf{t_{RP}}$ (since it is larger than the data transfer time) is only 7 cycles.

DDR modules are often described using a special notation: w-x-y-z-T. For instance: 2-3-2-8-T1. This means:

w 2 **CAS** Latency (CL)

x 3 **RAS**-to-**CAS** delay ($\mathbf{t_{RCD}}$)

y 2 RAS Precharge (t_{RP})

z 8 Active to Precharge delay (t_{RAS})

T T1 Command Rate

There are numerous other timing constants which affect the way commands can be issued and are handled. Those five constants are in practice sufficient to determine the performance of the module, though.

It is sometimes useful to know this information for the computers in use to be able to interpret certain measurements. It is definitely useful to know these details when buying computers since they, along with the FSB and SDRAM module speed, are among the most important factors determining a computer's speed.

The very adventurous reader could also try to tweak a system. Sometimes the BIOS allows changing some or all these values. SDRAM modules have programmable registers where these values can be set. Usually the BIOS picks the best default value. If the quality of the RAM module is high it might be possible to reduce the one or the other latency without affecting the stability of the computer. Numerous overclocking websites all around the Internet provide ample of documentation for doing this. Do it at your own risk, though and do not say you have not been warned.

2.2.3 Recharging

A mostly-overlooked topic when it comes to DRAM access is recharging. As explained in Section 2.1.2, DRAM cells must constantly be refreshed. This does not happen completely transparently for the rest of the system. At times when a row {*Rows are the granularity this happens with despite what [highperfdram] and other literature says (see [micronddr]).*} is recharged no access is possible. The study in [highperfdram] found that "*[s]urprisingly, DRAM refresh organization can affect performance dramatically*".

Each DRAM cell must be refreshed every 64ms according to the JEDEC specification. If a DRAM array has 8,192 rows this means the memory controller has to issue a refresh command on average every 7.8125µs (refresh commands can be queued so in practice the maximum interval between two requests can be higher). It is the memory controller's responsibility to schedule the refresh commands. The DRAM module keeps track of the address of the last refreshed row and automatically increases the address counter for each new request.

There is really not much the programmer can do about the refresh and the points in time when the commands are issued. But it is important to keep

this part to the DRAM life cycle in mind when interpreting measurements. If a critical word has to be retrieved from a row which currently is being refreshed the processor could be stalled for quite a long time. How long each refresh takes depends on the DRAM module.

2.2.4 Memory Types

It is worth spending some time on the current and soon-to-be current memory types in use. We will start with SDR (Single Data Rate) SDRAMs since they are the basis of the DDR (Double Data Rate) SDRAMs. SDRs were pretty simple. The memory cells and the data transfer rate were identical.

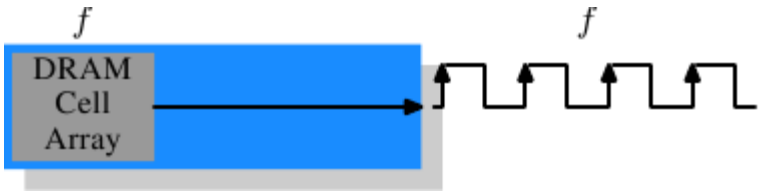


Figure 2.10: SDR SDRAM Operation

In Figure 2.10 the DRAM cell array can output the memory content at the same rate it can be transported over the memory bus. If the DRAM cell array can operate at 100MHz, the data transfer rate of the bus is thus 100Mb/s. The frequency *f* for all components is the same. Increasing the throughput of the DRAM chip is expensive since the energy consumption rises with the frequency. With a huge number of array cells this is prohibitively expensive. {*Power = Dynamic Capacity* × *Voltage*^{2} × *Frequency.*} In reality it is even more of a problem since increasing the frequency usually also requires increasing the voltage to maintain stability of the system. DDR SDRAM (called DDR1 retroactively) manages to improve the throughput without increasing any of the involved frequencies.

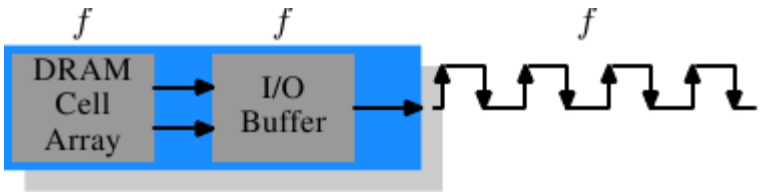


Figure 2.11: DDR1 SDRAM Operation

The difference between SDR and DDR1 is, as can be seen in Figure 2.11 and guessed from the name, that twice the amount of data is transported per cycle. I.e., the DDR1 chip transports data on the rising *and* falling edge.

This is sometimes called a “double-pumped” bus. To make this possible without increasing the frequency of the cell array a buffer has to be introduced. This buffer holds two bits per data line. This in turn requires that, in the cell array in Figure 2.7, the data bus consists of two lines. Implementing this is trivial: one only has the use the same column address for two DRAM cells and access them in parallel. The changes to the cell array to implement this are also minimal.

The SDR DRAMs were known simply by their frequency (e.g., PC100 for 100MHz SDR). To make DDR1 DRAM sound better the marketers had to come up with a new scheme since the frequency did not change. They came with a name which contains the transfer rate in bytes a DDR module (they have 64-bit busses) can sustain:

$$100\text{MHz} \times 64\text{bit} \times 2 = 1{,}600\text{MB/s}$$

Hence a DDR module with 100MHz frequency is called PC1600. With 1600 > 100 all marketing requirements are fulfilled; it sounds much better although the improvement is really *only* a factor of two. {*I will take the factor of two but I do not have to like the inflated numbers.*}

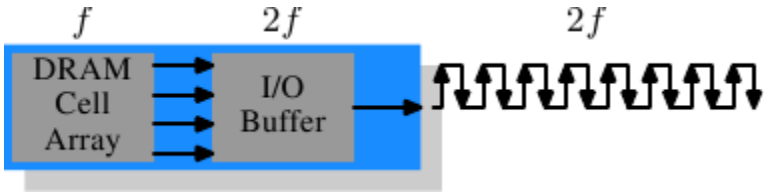


Figure 2.12: DDR2 SDRAM Operation

To get even more out of the memory technology DDR2 includes a bit more innovation. The most obvious change that can be seen in Figure 2.12 is the doubling of the frequency of the bus. Doubling the frequency means doubling the bandwidth. Since this doubling of the frequency is not economical for the cell array it is now required that the I/O buffer gets four bits in each clock cycle which it then can send on the bus. This means the changes to the DDR2 modules consist of making only the I/O buffer component of the DIMM capable of running at higher speeds. This is certainly possible and will not require measurably more energy, it is just one tiny component and not the whole module. The names the marketers came up with for DDR2 are similar to the DDR1 names only in the computation of the value the factor of two is replaced by four (we now have a quad-pumped bus). Figure 2.13 shows the names of the modules in use today.

Array	Bus	Data	Name	Name

Freq.	Freq.	Rate	(Rate)	(FSB)
133MHz	266MHz	4,256MB/s	PC2-4200	DDR2-533
166MHz	333MHz	5,312MB/s	PC2-5300	DDR2-667
200MHz	400MHz	6,400MB/s	PC2-6400	DDR2-800
250MHz	500MHz	8,000MB/s	PC2-8000	DDR2-1000
266MHz	533MHz	8,512MB/s	PC2-8500	DDR2-1066

Figure 2.13: DDR2 Module Names

There is one more twist to the naming. The FSB speed used by CPU, motherboard, and DRAM module is specified by using the *effective* frequency. I.e., it factors in the transmission on both flanks of the clock cycle and thereby inflates the number. So, a 133MHz module with a 266MHz bus has an FSB "frequency" of 533MHz.

The specification for DDR3 (the real one, not the fake GDDR3 used in graphics cards) calls for more changes along the lines of the transition to DDR2. The voltage will be reduced from 1.8V for DDR2 to 1.5V for DDR3. Since the power consumption equation is calculated using the square of the voltage this alone brings a 30% improvement. Add to this a reduction in die size plus other electrical advances and DDR3 can manage, at the same frequency, to get by with half the power consumption. Alternatively, with higher frequencies, the same power envelope can be hit. Or with double the capacity the same heat emission can be achieved.

The cell array of DDR3 modules will run at a quarter of the speed of the external bus which requires an 8 bit I/O buffer, up from 4 bits for DDR2. See Figure 2.14 for the schematics.

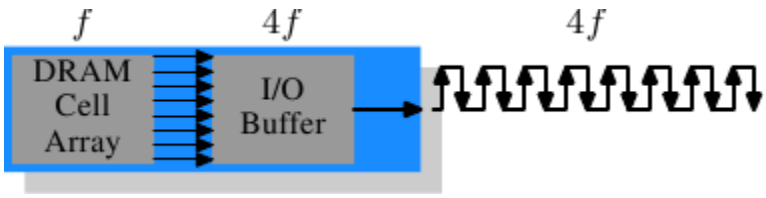


Figure 2.14: DDR3 SDRAM Operation

Initially DDR3 modules will likely have slightly higher **CAS** latencies just because the DDR2 technology is more mature. This would cause DDR3 to be useful only at frequencies which are higher than those which can be achieved with DDR2, and, even then, mostly when bandwidth is more important than latency. There is already talk about 1.3V modules which can achieve the same **CAS** latency as DDR2. In any case, the possibility of

achieving higher speeds because of faster buses will outweigh the increased latency.

One possible problem with DDR3 is that, for 1,600Mb/s transfer rate or higher, the number of modules per channel may be reduced to just one. In earlier versions this requirement held for all frequencies, so one can hope that the requirement will at some point be lifted for all frequencies. Otherwise the capacity of systems will be severely limited.

Figure 2.15 shows the names of the expected DDR3 modules. JEDEC agreed so far on the first four types. Given that Intel's 45nm processors have an FSB speed of 1,600Mb/s, the 1,866Mb/s is needed for the overclocking market. We will likely see more of this towards the end of the DDR3 lifecycle.

Array Freq.	Bus Freq.	Data Rate	Name (Rate)	Name (FSB)
100MHz	400MHz	6,400MB/s	PC3-6400	DDR3-800
133MHz	533MHz	8,512MB/s	PC3-8500	DDR3-1066
166MHz	667MHz	10,667MB/s	PC3-10667	DDR3-1333
200MHz	800MHz	12,800MB/s	PC3-12800	DDR3-1600
233MHz	933MHz	14,933MB/s	PC3-14900	DDR3-1866

Figure 2.15: DDR3 Module Names

All DDR memory has one problem: the increased bus frequency makes it hard to create parallel data busses. A DDR2 module has 240 pins. All connections to data and address pins must be routed so that they have approximately the same length. Even more of a problem is that, if more than one DDR module is to be daisy-chained on the same bus, the signals get more and more distorted for each additional module. The DDR2 specification allow only two modules per bus (aka channel), the DDR3 specification only one module for high frequencies. With 240 pins per channel a single Northbridge cannot reasonably drive more than two channels. The alternative is to have external memory controllers (as in Figure 2.2) but this is expensive.

What this means is that commodity motherboards are restricted to hold at most four DDR2 or DDR3 modules. This restriction severely limits the amount of memory a system can have. Even old 32-bit IA-32 processors can handle 64GB of RAM and memory demand even for home use is growing, so something has to be done.

One answer is to add memory controllers into each processor as explained in Section 2. AMD does it with the Opteron line and Intel will do it with their CSI technology. This will help as long as the reasonable amount of memory a processor is able to use can be connected to a single processor. In some situations this is not the case and this setup will introduce a NUMA architecture and its negative effects. For some situations another solution is needed.

Intel's answer to this problem for big server machines, at least for the next years, is called Fully Buffered DRAM (FB-DRAM). The FB-DRAM modules use the same components as today's DDR2 modules which makes them relatively cheap to produce. The difference is in the connection with the memory controller. Instead of a parallel data bus FB-DRAM utilizes a serial bus (Rambus DRAM had this back when, too, and SATA is the successor of PATA, as is PCI Express for PCI/AGP). The serial bus can be driven at a much higher frequency, reverting the negative impact of the serialization and even increasing the bandwidth. The main effects of using a serial bus are

1. more modules per channel can be used.
2. more channels per Northbridge/memory controller can be used.
3. the serial bus is designed to be fully-duplex (two lines).

An FB-DRAM module has only 69 pins, compared with the 240 for DDR2. Daisy chaining FB-DRAM modules is much easier since the electrical effects of the bus can be handled much better. The FB-DRAM specification allows up to 8 DRAM modules per channel.

Compared with the connectivity requirements of a dual-channel Northbridge it is now possible to drive 6 channels of FB-DRAM with fewer pins: 2×240 pins versus 6×69 pins. The routing for each channel is much simpler which could also help reducing the cost of the motherboards.

Fully duplex parallel busses are prohibitively expensive for the traditional DRAM modules, duplicating all those lines is too costly. With serial lines (even if they are differential, as FB-DRAM requires) this is not the case and so the serial bus is designed to be fully duplexed, which means, in some situations, that the bandwidth is theoretically doubled alone by this. But it is not the only place where parallelism is used for bandwidth increase. Since an FB-DRAM controller can run up to six channels at the same time the bandwidth can be increased even for systems with smaller amounts of RAM by using FB-DRAM. Where a DDR2 system with four modules has two channels, the same capacity can handled via four channels using an ordinary FB-DRAM controller. The actual bandwidth of the serial bus depends on the type of DDR2 (or DDR3) chips used on the FB-DRAM module.

We can summarize the advantages like this:

	DDR2	FB-DRAM
Pins	240	69
Channels	2	6
DIMMs/Channel	2	8
Max Memory	16GB	192GB
Throughput	~10GB/s	~40GB/s

There are a few drawbacks to FB-DRAMs if multiple DIMMs on one channel are used. The signal is delayed—albeit minimally—at each DIMM in the chain, which means the latency increases. But for the same amount of memory with the same frequency FB-DRAM can always be faster than DDR2 and DDR3 since only one DIMM per channel is needed; for large memory systems DDR simply has no answer using commodity components.

2.2.5 Conclusions

This section should have shown that accessing DRAM is not an arbitrarily fast process. At least not fast compared with the speed the processor is running and with which it can access registers and cache. It is important to keep in mind the differences between CPU and memory frequencies. An Intel Core 2 processor running at 2.933GHz and a 1.066GHz FSB have a clock ratio of 11:1 (note: the 1.066GHz bus is quad-pumped). Each stall of one cycle on the memory bus means a stall of 11 cycles for the processor. For most machines the actual DRAMs used are slower, thusly increasing the delay. Keep these numbers in mind when we are talking about stalls in the upcoming sections.

The timing charts for the read command have shown that DRAM modules are capable of high sustained data rates. Entire DRAM rows could be transported without a single stall. The data bus could be kept occupied 100%. For DDR modules this means two 64-bit words transferred each cycle. With DDR2-800 modules and two channels this means a rate of 12.8GB/s.

But, unless designed this way, DRAM access is not always sequential. Non-continuous memory regions are used which means precharging and new **RAS**signals are needed. This is when things slow down and when the DRAM modules need help. The sooner the precharging can happen and the **RAS** signal sent the smaller the penalty when the row is actually used.

Hardware and software prefetching (see Section 6.3) can be used to create more overlap in the timing and reduce the stall. Prefetching also helps shift

memory operations in time so that there is less contention at later times, right before the data is actually needed. This is a frequent problem when the data produced in one round has to be stored and the data required for the next round has to be read. By shifting the read in time, the write and read operations do not have to be issued at basically the same time.

2.3 Other Main Memory Users

Beside the CPUs there are other system components which can access the main memory. High-performance cards such as network and mass-storage controllers cannot afford to pipe all the data they need or provide through the CPU. Instead, they read or write the data directly from/to the main memory (Direct Memory Access, DMA). In Figure 2.1 we can see that the cards can talk through the South- and Northbridge directly with the memory. Other buses, like USB, also require FSB bandwidth—even though they do not use DMA—since the Southbridge is connected to the Northbridge through the FSB, too.

While DMA is certainly beneficial, it means that there is more competition for the FSB bandwidth. In times with high DMA traffic the CPU might stall more than usual while waiting for data from the main memory. There are ways around this given the right hardware. With an architecture as in Figure 2.3 one can make sure the computation uses memory on nodes which are not affected by DMA. It is also possible to attach a Southbridge to each node, equally distributing the load on the FSB of all the nodes. There are a myriad of possibilities. In Section 6 we will introduce techniques and programming interfaces which help achieving the improvements which are possible in software.

Finally it should be mentioned that some cheap systems have graphics systems without separate, dedicated video RAM. Those systems use parts of the main memory as video RAM. Since access to the video RAM is frequent (for a 1024x768 display with 16 bpp at 60Hz we are talking 94MB/s) and system memory, unlike RAM on graphics cards, does not have two ports this can substantially influence the systems performance and especially the latency. It is best to ignore such systems when performance is a priority. They are more trouble than they are worth. People buying those machines know they will not get the best performance.

3: CPU caches

October 1, 2007

This article was contributed by Ulrich Drepper

[*Editor's note: This is the second installment in Ulrich Drepper's "What*

every programmer should know about memory" document. Those who have not read the first part will likely want to start there. This is good stuff, and we once again thank Ulrich for allowing us to publish it.

One quick request: in a document of this length there are bound to be a few typographical errors remaining. If you find one, and wish to see it corrected, please let us know via mail to lwn@lwn.net rather than by posting a comment. That way we will be sure to incorporate the fix and get it back into Ulrich's copy of the document and other readers will not have to plow through uninteresting comments.]

CPUs are today much more sophisticated than they were only 25 years ago. In those days, the frequency of the CPU core was at a level equivalent to that of the memory bus. Memory access was only a bit slower than register access. But this changed dramatically in the early 90s, when CPU designers increased the frequency of the CPU core but the frequency of the memory bus and the performance of RAM chips did not increase proportionally. This is not due to the fact that faster RAM could not be built, as explained in the previous section. It is possible but it is not economical. RAM as fast as current CPU cores is orders of magnitude more expensive than any dynamic RAM.

If the choice is between a machine with very little, very fast RAM and a machine with a lot of relatively fast RAM, the second will always win given a working set size which exceeds the small RAM size and the cost of accessing secondary storage media such as hard drives. The problem here is the speed of secondary storage, usually hard disks, which must be used to hold the swapped out part of the working set. Accessing those disks is orders of magnitude slower than even DRAM access.

Fortunately it does not have to be an all-or-nothing decision. A computer can have a small amount of high-speed SRAM in addition to the large amount of DRAM. One possible implementation would be to dedicate a certain area of the address space of the processor as containing the SRAM and the rest the DRAM. The task of the operating system would then be to optimally distribute data to make use of the SRAM. Basically, the SRAM serves in this situation as an extension of the register set of the processor.

While this is a possible implementation, it is not viable. Ignoring the problem of mapping the physical resources of such SRAM-backed memory to the virtual address spaces of the processes (which by itself is terribly hard) this approach would require each process to administer in software the allocation of this memory region. The size of the memory region can vary from processor to processor (i.e., processors have different amounts of the expensive SRAM-backed memory). Each module which makes up part of a

program will claim its share of the fast memory, which introduces additional costs through synchronization requirements. In short, the gains of having fast memory would be eaten up completely by the overhead of administering the resources.

So, instead of putting the SRAM under the control of the OS or user, it becomes a resource which is transparently used and administered by the processors. In this mode, SRAM is used to make temporary copies of (to cache, in other words) data in main memory which is likely to be used soon by the processor. This is possible because program code and data has temporal and spatial locality. This means that, over short periods of time, there is a good chance that the same code or data gets reused. For code this means that there are most likely loops in the code so that the same code gets executed over and over again (the perfect case for *spatial locality*). Data accesses are also ideally limited to small regions. Even if the memory used over short time periods is not close together there is a high chance that the same data will be reused before long (*temporal locality*). For code this means, for instance, that in a loop a function call is made and that function is located elsewhere in the address space. The function may be distant in memory, but calls to that function will be close in time. For data it means that the total amount of memory used at one time (the working set size) is ideally limited but the memory used, as a result of the *random* access nature of RAM, is not close together. Realizing that locality exists is key to the concept of CPU caches as we use them today.

A simple computation can show how effective caches can theoretically be. Assume access to main memory takes 200 cycles and access to the cache memory take 15 cycles. Then code using 100 data elements 100 times each will spend 2,000,000 cycles on memory operations if there is no cache and only 168,500 cycles if all data can be cached. That is an improvement of 91.5%.

The size of the SRAM used for caches is many times smaller than the main memory. In the author's experience with workstations with CPU caches the cache size has always been around 1/1000th of the size of the main memory (today: 4MB cache and 4GB main memory). This alone does not constitute a problem. If the size of the working set (the set of data currently worked on) is smaller than the cache size it does not matter. But computers do not have large main memories for no reason. The working set is bound to be larger than the cache. This is especially true for systems running multiple processes where the size of the working set is the sum of the sizes of all the individual processes and the kernel.

What is needed to deal with the limited size of the cache is a set of good strategies to determine what should be cached at any given time. Since not

all data of the working set is used at *exactly* the same time we can use techniques to temporarily replace some data in the cache with other data. And maybe this can be done before the data is actually needed. This prefetching would remove some of the costs of accessing main memory since it happens asynchronously with respect to the execution of the program. All these techniques and more can be used to make the cache appear bigger than it actually is. We will discuss them in Section 3.3. Once all these techniques are exploited it is up to the programmer to help the processor. How this can be done will be discussed in Section 6.

3.1 CPU Caches in the Big Picture

Before diving into technical details of the implementation of CPU caches some readers might find it useful to first see in some more details how caches fit into the "big picture" of a modern computer system.

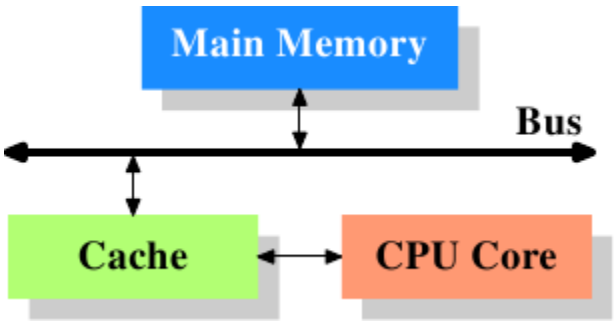


Figure 3.1: Minimum Cache Configuration

Figure 3.1 shows the minimum cache configuration. It corresponds to the architecture which could be found in early systems which deployed CPU caches. The CPU core is no longer directly connected to the main memory. {*In even earlier systems the cache was attached to the system bus just like the CPU and the main memory. This was more a hack than a real solution.*} All loads and stores have to go through the cache. The connection between the CPU core and the cache is a special, fast connection. In a simplified representation, the main memory and the cache are connected to the system bus which can also be used for communication with other components of the system. We introduced the system bus as "FSB" which is the name in use today; see Section 2.2. In this section we ignore the Northbridge; it is assumed to be present to facilitate the communication of the CPU(s) with the main memory.

Even though computers for the last several decades have used the von Neumann architecture, experience has shown that it is of advantage to separate the caches used for code and for data. Intel has used separate code and data caches since 1993 and never looked back. The memory regions

needed for code and data are pretty much independent of each other, which is why independent caches work better. In recent years another advantage emerged: the instruction decoding step for the most common processors is slow; caching decoded instructions can speed up the execution, especially when the pipeline is empty due to incorrectly predicted or impossible-to-predict branches.

Soon after the introduction of the cache, the system got more complicated. The speed difference between the cache and the main memory increased again, to a point that another level of cache was added, bigger and slower than the first-level cache. Only increasing the size of the first-level cache was not an option for economical reasons. Today, there are even machines with three levels of cache in regular use. A system with such a processor looks like Figure 3.2. With the increase on the number of cores in a single CPU the number of cache levels might increase in the future even more.

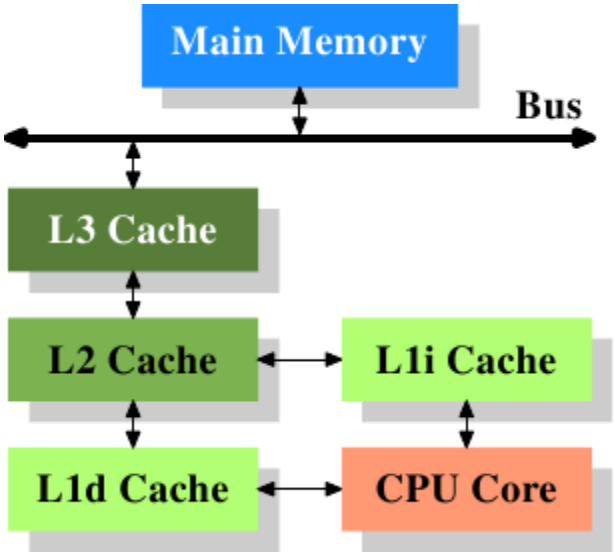


Figure 3.2: Processor with Level 3 Cache

Figure 3.2 shows three levels of cache and introduces the nomenclature we will use in the remainder of the document. L1d is the level 1 data cache, L1i the level 1 instruction cache, etc. Note that this is a schematic; the data flow in reality need not pass through any of the higher-level caches on the way from the core to the main memory. CPU designers have a lot of freedom designing the interfaces of the caches. For programmers these design choices are invisible.

In addition we have processors which have multiple cores and each core can have multiple “threads”. The difference between a core and a thread is that separate cores have separate copies of (almost {*Early multi-core processors even had separate 2nd level caches and no 3rd level cache.*}) all the hardware

resources. The cores can run completely independently unless they are using the same resources—e.g., the connections to the outside—at the same time. Threads, on the other hand, share almost all of the processor's resources. Intel's implementation of threads has only separate registers for the threads and even that is limited, some registers are shared. The complete picture for a modern CPU therefore looks like Figure 3.3.

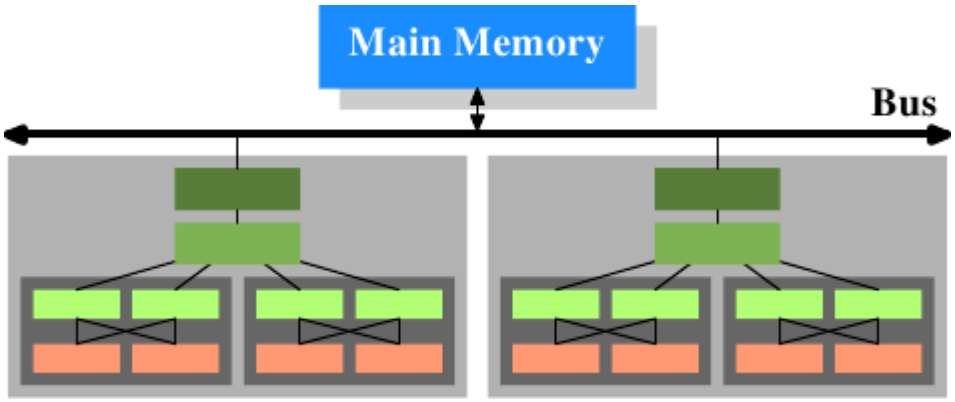


Figure 3.3: Multi processor, multi-core, multi-thread

In this figure we have two processors, each with two cores, each of which has two threads. The threads share the Level 1 caches. The cores (shaded in the darker gray) have individual Level 1 caches. All cores of the CPU share the higher-level caches. The two processors (the two big boxes shaded in the lighter gray) of course do not share any caches. All this will be important, especially when we are discussing the cache effects on multi-process and multi-thread applications.

3.2 Cache Operation at High Level

To understand the costs and savings of using a cache we have to combine the knowledge about the machine architecture and RAM technology from Section 2 with the structure of caches described in the previous section.

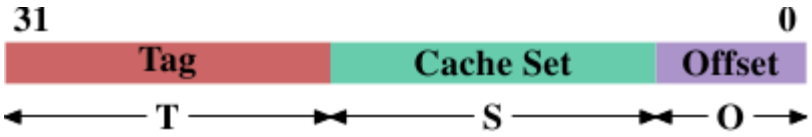
By default all data read or written by the CPU cores is stored in the cache. There are memory regions which cannot be cached but this is something only the OS implementers have to be concerned about; it is not visible to the application programmer. There are also instructions which allow the programmer to deliberately bypass certain caches. This will be discussed in Section 6.

If the CPU needs a data word the caches are searched first. Obviously, the cache cannot contain the content of the entire main memory (otherwise we would need no cache), but since all memory addresses are cacheable, each cache entry is *tagged* using the address of the data word in the main memory.

This way a request to read or write to an address can search the caches for a matching tag. The address in this context can be either the virtual or physical address, varying based on the cache implementation.

Since the tag requires space in addition to the actual memory, it is inefficient to chose a word as the granularity of the cache. For a 32-bit word on an x86 machine the tag itself might need 32 bits or more. Furthermore, since spatial locality is one of the principles on which caches are based, it would be bad to not take this into account. Since neighboring memory is likely to be used together it should also be loaded into the cache together. Remember also what we learned in Section 2.2.1: RAM modules are much more effective if they can transport many data words in a row without a new **CAS** or even **RAS** signal. So the entries stored in the caches are not single words but, instead, "lines" of several contiguous words. In early caches these lines were 32 bytes long; now the norm is 64 bytes. If the memory bus is 64 bits wide this means 8 transfers per cache line. DDR supports this transport mode efficiently.

When memory content is needed by the processor the entire cache line is loaded into the L1d. The memory address for each cache line is computed by masking the address value according to the cache line size. For a 64 byte cache line this means the low 6 bits are zeroed. The discarded bits are used as the offset into the cache line. The remaining bits are in some cases used to locate the line in the cache and as the tag. In practice an address value is split into three parts. For a 32-bit address it might look as follows:



With a cache line size of 2^{O} the low **O** bits are used as the offset into the cache line. The next **S** bits select the "cache set". We will go into more detail soon on why sets, and not single slots, are used for cache lines. For now it is sufficient to understand there are 2^{S} sets of cache lines. This leaves the top 32 - **S** - **O** = **T** bits which form the tag. These **T** bits are the value associated with each cache line to distinguish all the *aliases* {*All cache lines with the same* **S** *part of the address are known by the same alias.*} which are cached in the same cache set. The **S** bits used to address the cache set do not have to be stored since they are the same for all cache lines in the same set.

When an instruction modifies memory the processor still has to load a cache line first because no instruction modifies an entire cache line at once (exception to the rule: write-combining as explained in Section 6.1). The content of the cache line before the write operation therefore has to be

loaded. It is not possible for a cache to hold partial cache lines. A cache line which has been written to and which has not been written back to main memory is said to be “dirty”. Once it is written the dirty flag is cleared.

To be able to load new data in a cache it is almost always first necessary to make room in the cache. An eviction from L1d pushes the cache line down into L2 (which uses the same cache line size). This of course means room has to be made in L2. This in turn might push the content into L3 and ultimately into main memory. Each eviction is progressively more expensive. What is described here is the model for an *exclusive cache* as is preferred by modern AMD and VIA processors. Intel implements *inclusive caches* {*This generalization is not completely correct. A few caches are exclusive and some inclusive caches have exclusive cache properties.*} where each cache line in L1d is also present in L2. Therefore evicting from L1d is much faster. With enough L2 cache the disadvantage of wasting memory for content held in two places is minimal and it pays off when evicting. A possible advantage of an exclusive cache is that loading a new cache line only has to touch the L1d and not the L2, which could be faster.

The CPUs are allowed to manage the caches as they like as long as the memory model defined for the processor architecture is not changed. It is, for instance, perfectly fine for a processor to take advantage of little or no memory bus activity and proactively write dirty cache lines back to main memory. The wide variety of cache architectures among the processors for the x86 and x86-64, between manufacturers and even within the models of the same manufacturer, are testament to the power of the memory model abstraction.

In symmetric multi-processor (SMP) systems the caches of the CPUs cannot work independently from each other. All processors are supposed to see the same memory content at all times. The maintenance of this uniform view of memory is called “cache coherency”. If a processor were to look simply at its own caches and main memory it would not see the content of dirty cache lines in other processors. Providing direct access to the caches of one processor from another processor would be terribly expensive and a huge bottleneck. Instead, processors detect when another processor wants to read or write to a certain cache line.

If a write access is detected and the processor has a clean copy of the cache line in its cache, this cache line is marked invalid. Future references will require the cache line to be reloaded. Note that a read access on another CPU does not necessitate an invalidation, multiple clean copies can very well be kept around.

More sophisticated cache implementations allow another possibility to happen. If the cache line which another processor wants to read from or write to is currently marked dirty in the first processor's cache a different course of action is needed. In this case the main memory is out-of-date and the requesting processor must, instead, get the cache line content from the first processor. Through snooping, the first processor notices this situation and automatically sends the requesting processor the data. This action bypasses main memory, though in some implementations the memory controller is supposed to notice this direct transfer and store the updated cache line content in main memory. If the access is for writing the first processor then invalidates its copy of the local cache line.

Over time a number of cache coherency protocols have been developed. The most important is MESI, which we will introduce in Section 3.3.4. The outcome of all this can be summarized in a few simple rules:

- A dirty cache line is not present in any other processor's cache.
- Clean copies of the same cache line can reside in arbitrarily many caches.

If these rules can be maintained, processors can use their caches efficiently even in multi-processor systems. All the processors need to do is to monitor each others' write accesses and compare the addresses with those in their local caches. In the next section we will go into a few more details about the implementation and especially the costs.

Finally, we should at least give an impression of the costs associated with cache hits and misses. These are the numbers Intel lists for a Pentium M:

To Where	Cycles
Register	<= 1
L1d	~3
L2	~14
Main Memory	~240

These are the actual access times measured in CPU cycles. It is interesting to note that for the on-die L2 cache a large part (probably even the majority) of the access time is caused by wire delays. This is a physical limitation which can only get worse with increasing cache sizes. Only process shrinking (for instance, going from 60nm for Merom to 45nm for Penryn in Intel's lineup) can improve those numbers.

The numbers in the table look high but, fortunately, the entire cost does not have to be paid for each occurrence of the cache load and miss. Some parts

of the cost can be hidden. Today's processors all use internal pipelines of different lengths where the instructions are decoded and prepared for execution. Part of the preparation is loading values from memory (or cache) if they are transferred to a register. If the memory load operation can be started early enough in the pipeline, it may happen in parallel with other operations and the entire cost of the load might be hidden. This is often possible for L1d; for some processors with long pipelines for L2 as well.

There are many obstacles to starting the memory read early. It might be as simple as not having sufficient resources for the memory access or it might be that the final address of the load becomes available late as the result of another instruction. In these cases the load costs cannot be hidden (completely).

For write operations the CPU does not necessarily have to wait until the value is safely stored in memory. As long as the execution of the following instructions appears to have the same effect as if the value were stored in memory there is nothing which prevents the CPU from taking shortcuts. It can start executing the next instruction early. With the help of shadow registers which can hold values no longer available in a regular register it is even possible to change the value which is to be stored in the incomplete write operation.

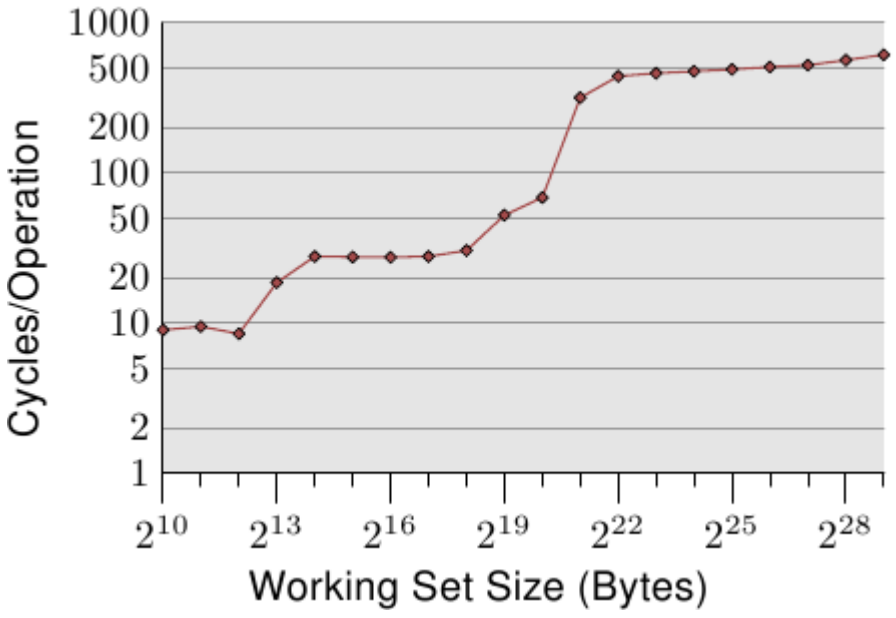


Figure 3.4: Access Times for Random Writes

For an illustration of the effects of cache behavior see Figure 3.4. We will talk about the program which generated the data later; it is a simple simulation of a program which accesses a configurable amount of memory repeatedly in a random fashion. Each data item has a fixed size. The number

of elements depends on the selected working set size. The Y–axis shows the average number of CPU cycles it takes to process one element; note that the scale for the Y–axis is logarithmic. The same applies in all the diagrams of this kind to the X–axis. The size of the working set is always shown in powers of two.

The graph shows three distinct plateaus. This is not surprising: the specific processor has L1d and L2 caches, but no L3. With some experience we can deduce that the L1d is 2^{13} bytes in size and that the L2 is 2^{20} bytes in size. If the entire working set fits into the L1d the cycles per operation on each element is below 10. Once the L1d size is exceeded the processor has to load data from L2 and the average time springs up to around 28. Once the L2 is not sufficient anymore the times jump to 480 cycles and more. This is when many or most operations have to load data from main memory. And worse: since data is being modified dirty cache lines have to be written back, too.

This graph should give sufficient motivation to look into coding improvements which help improve cache usage. We are not talking about a few measly percent here; we are talking about orders-of-magnitude improvements which are sometimes possible. In Section 6 we will discuss techniques which allow writing more efficient code. The next section goes into more details of CPU cache designs. The knowledge is good to have but not necessary for the rest of the paper. So this section could be skipped.

3.3 CPU Cache Implementation Details

Cache implementers have the problem that each cell in the huge main memory potentially has to be cached. If the working set of a program is large enough this means there are many main memory locations which fight for each place in the cache. Previously it was noted that a ratio of 1-to-1000 for cache versus main memory size is not uncommon.

3.3.1 Associativity

It would be possible to implement a cache where each cache line can hold a copy of any memory location. This is called a *fully associative cache*. To access a cache line the processor core would have to compare the tags of each and every cache line with the tag for the requested address. The tag would be comprised of the entire part of the address which is not the offset into the cache line (that means, **S** in the figure on Section 3.2 is zero).

There are caches which are implemented like this but, by looking at the numbers for an L2 in use today, will show that this is impractical. Given a 4MB cache with 64B cache lines the cache would have 65,536 entries. To

achieve adequate performance the cache logic would have to be able to pick from all these entries the one matching a given tag in just a few cycles. The effort to implement this would be enormous.

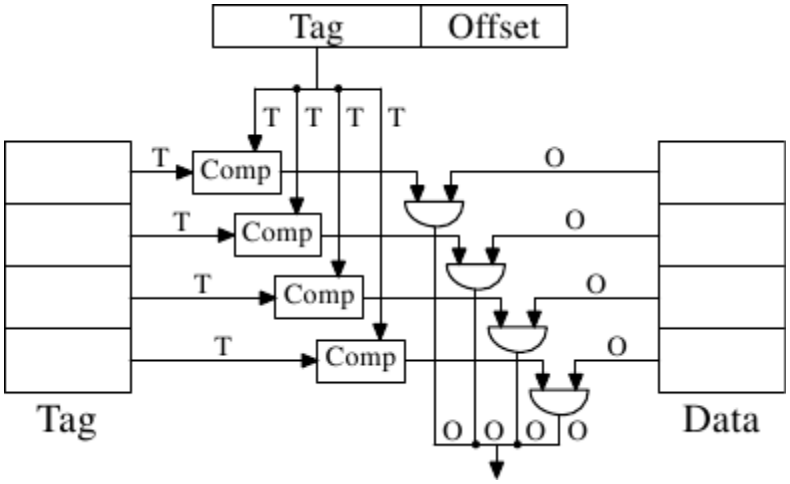


Figure 3.5: Fully Associative Cache Schematics

For each cache line a comparator is needed to compare the large tag (note, **S** is zero). The letter next to each connection indicates the width in bits. If none is given it is a single bit line. Each comparator has to compare two **T**-bit-wide values. Then, based on the result, the appropriate cache line content is selected and made available. This requires merging as many sets of **O** data lines as there are cache buckets. The number of transistors needed to implement a single comparator is large especially since it must work very fast. No iterative comparator is usable. The only way to save on the number of comparators is to reduce the number of them by iteratively comparing the tags. This is not suitable for the same reason that iterative comparators are not: it takes too long.

Fully associative caches are practical for small caches (for instance, the TLB caches on some Intel processors are fully associative) but those caches are small, really small. We are talking about a few dozen entries at most.

For L1i, L1d, and higher level caches a different approach is needed. What can be done is to restrict the search. In the most extreme restriction each tag maps to exactly one cache entry. The computation is simple: given the 4MB/64B cache with 65,536 entries we can directly address each entry by using bits 6 to 21 of the address (16 bits). The low 6 bits are the index into the cache line.

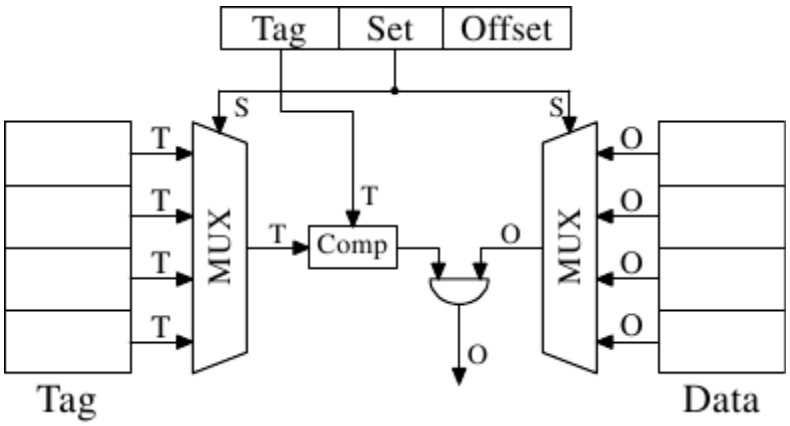


Figure 3.6: Direct-Mapped Cache Schematics

Such a *direct-mapped cache* is fast and relatively easy to implement as can be seen in Figure 3.6. It requires exactly one comparator, one multiplexer (two in this diagram where tag and data are separated, but this is not a hard requirement on the design), and some logic to select only valid cache line content. The comparator is complex due to the speed requirements but there is only one of them now; as a result more effort can be spent on making it fast. The real complexity in this approach lies in the multiplexers. The number of transistors in a simple multiplexer grows with O(log N), where N is the number of cache lines. This is tolerable but might get slow, in which case speed can be increased by spending more real estate on transistors in the multiplexers to parallelize some of the work and to increase the speed. The total number of transistors can grow slowly with a growing cache size which makes this solution very attractive. But it has a drawback: it only works well if the addresses used by the program are evenly distributed with respect to the bits used for the direct mapping. If they are not, and this is usually the case, some cache entries are heavily used and therefore repeatedly evicted while others are hardly used at all or remain empty.

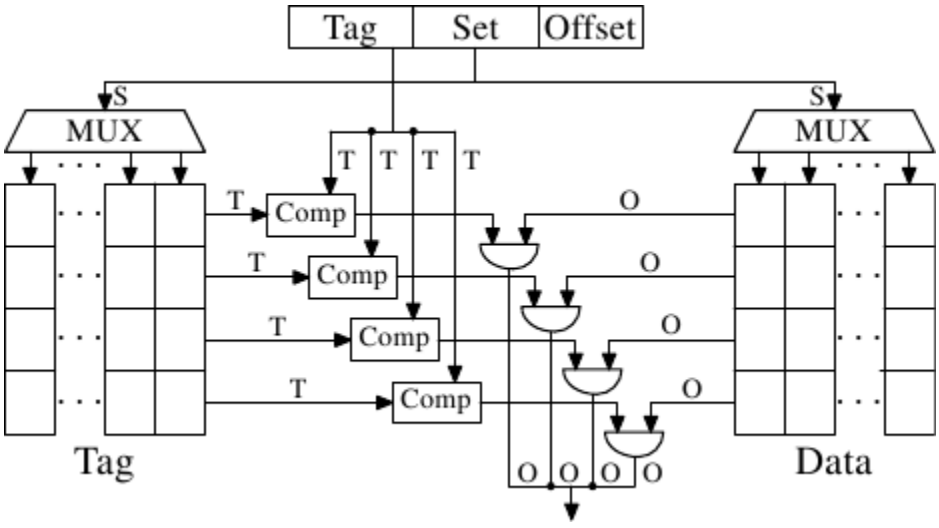


Figure 3.7: Set-Associative Cache Schematics

This problem can be solved by making the cache *set associative*. A set-associative cache combines the features of the full associative and direct-mapped caches to largely avoid the weaknesses of those designs. Figure 3.7 shows the design of a set-associative cache. The tag and data storage are divided into sets which are selected by the address. This is similar to the direct-mapped cache. But instead of only having one element for each set value in the cache a small number of values is cached for the same set value. The tags for all the set members are compared in parallel, which is similar to the functioning of the fully associative cache.

The result is a cache which is not easily defeated by unfortunate or deliberate selection of addresses with the same set numbers and at the same time the size of the cache is not limited by the number of comparators which can be implemented in parallel. If the cache grows it is (in this figure) only the number of columns which increases, not the number of rows. The number of rows only increases if the associativity of the cache is increased. Today processors are using associativity levels of up to 16 for L2 caches or higher. L1 caches usually get by with 8.

L2 Cache Size	Associativity							
	Direct		2		4		8	
	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64	CL=32	CL=64
512k	27,794,595	20,422,527	25,222,611	18,303,581	24,096,510	17,356,121	23,666,929	17,029,334
1M	19,007,315	13,903,854	16,566,738	12,127,174	15,537,500	11,436,705	15,162,895	11,233,896
2M	12,230,962	8,801,403	9,081,881	6,491,011	7,878,601	5,675,181	7,391,389	5,382,064
4M	7,749,986	5,427,836	4,736,187	3,159,507	3,788,122	2,418,898	3,430,713	2,125,103
8M	4,731,904	3,209,693	2,690,498	1,602,957	2,207,655	1,228,190	2,111,075	1,155,847
16M	2,620,587	1,528,592	1,958,293	1,089,580	1,704,878	883,530	1,671,541	862,324

Table 3.1: Effects of Cache Size, Associativity, and Line Size

Given our 4MB/64B cache and 8-way set associativity the cache we are left with has 8,192 sets and only 13 bits of the tag are used in addressing the cache set. To determine which (if any) of the entries in the cache set contains the addressed cache line 8 tags have to be compared. That is feasible to do in very short time. With an experiment we can see that this makes sense.

Table 3.1 shows the number of L2 cache misses for a program (gcc in this case, the most important benchmark of them all, according to the Linux kernel people) for changing cache size, cache line size, and associativity set size. In Section 7.2 we will introduce the tool to simulate the caches as required for this test.

Just in case this is not yet obvious, the relationship of all these values is that the cache size is

cache line size × associativity × number of sets

The addresses are mapped into the cache by using

O = $\log_2$ cache line size
S = $\log_2$ number of sets

in the way the figure in Section 3.2 shows.

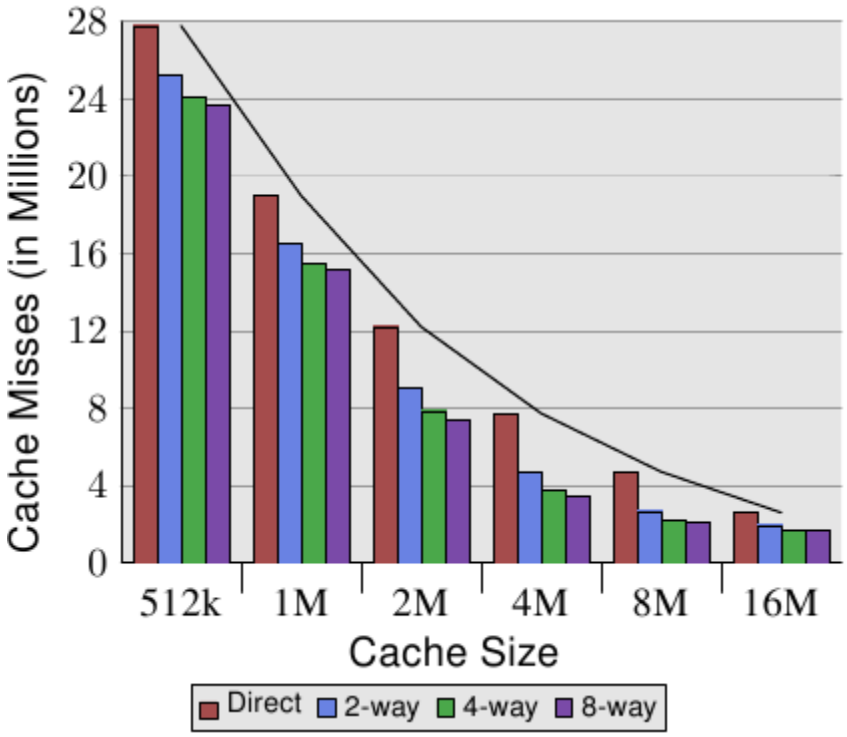


Figure 3.8: Cache Size vs Associativity (CL=32)

Figure 3.8 makes the data of the table more comprehensible. It shows the data for a fixed cache line size of 32 bytes. Looking at the numbers for a given cache size we can see that associativity can indeed help to reduce the number of cache misses significantly. For an 8MB cache going from direct mapping to 2-way set associative cache saves almost 44% of the cache

misses. The processor can keep more of the working set in the cache with a set associative cache compared with a direct mapped cache.

In the literature one can occasionally read that introducing associativity has the same effect as doubling cache size. This is true in some extreme cases as can be seen in the jump from the 4MB to the 8MB cache. But it certainly is not true for further doubling of the associativity. As we can see in the data, the successive gains are much smaller. We should not completely discount the effects, though. In the example program the peak memory use is 5.6M. So with a 8MB cache there are unlikely to be many (more than two) uses for the same cache set. With a larger working set the savings can be higher as we can see from the larger benefits of associativity for the smaller cache sizes.

In general, increasing the associativity of a cache above 8 seems to have little effects for a single-thread workload. With the introduction of multi-core processors which use a shared L2 the situation changes. Now you basically have two programs hitting on the same cache which causes the associativity in practice to be halved (or quartered for quad-core processors). So it can be expected that, with increasing numbers of cores, the associativity of the shared caches should grow. Once this is not possible anymore (16-way set associativity is already hard) processor designers have to start using shared L3 caches and beyond, while L2 caches are potentially shared by a subset of the cores.

Another effect we can study in Figure 3.8 is how the increase in cache size helps with performance. This data cannot be interpreted without knowing about the working set size. Obviously, a cache as large as the main memory would lead to better results than a smaller cache, so there is in general no limit to the largest cache size with measurable benefits.

As already mentioned above, the size of the working set at its peak is 5.6M. This does not give us any absolute number of the maximum beneficial cache size but it allows us to estimate the number. The problem is that not all the memory used is contiguous and, therefore, we have, even with a 16M cache and a 5.6M working set, conflicts (see the benefit of the 2-way set associative 16MB cache over the direct mapped version). But it is a safe bet that with the same workload the benefits of a 32MB cache would be negligible. But who says the working set has to stay the same? Workloads are growing over time and so should the cache size. When buying machines, and one has to choose the cache size one is willing to pay for, it is worthwhile to measure the working set size. Why this is important can be seen in the figures on Figure 3.10.

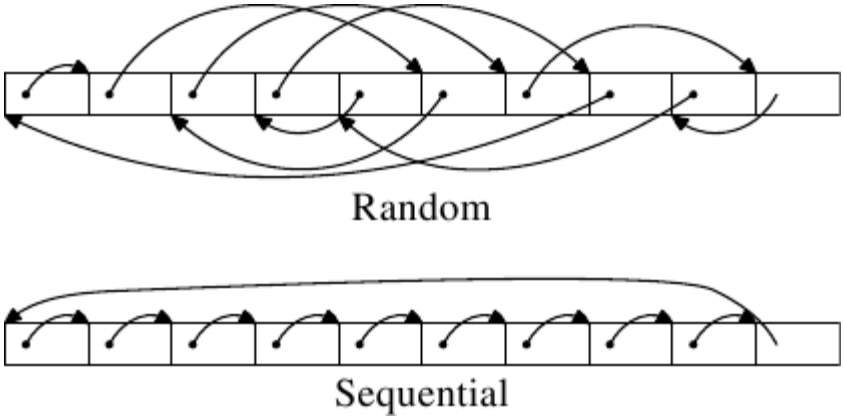


Figure 3.9: Test Memory Layouts

Two types of tests are run. In the first test the elements are processed sequentially. The test program follows the pointer `n` but the array elements are chained so that they are traversed in the order in which they are found in memory. This can be seen in the lower part of Figure 3.9. There is one back reference from the last element. In the second test (upper part of the figure) the array elements are traversed in a random order. In both cases the array elements form a circular single-linked list.

3.3.2 Measurements of Cache Effects

All the figures are created by measuring a program which can simulate working sets of arbitrary size, read and write access, and sequential or random access. We have already seen some results in Figure 3.4. The program creates an array corresponding to the working set size of elements of this type:

```
struct l {
  struct l *n;
  long int pad[NPAD];
};
```

All entries are chained in a circular list using the `n` element, either in sequential or random order. Advancing from one entry to the next always uses the pointer, even if the elements are laid out sequentially.

The `pad` element is the payload and it can grow arbitrarily large. In some tests the data is modified, in others the program only performs read operations.

In the performance measurements we are talking about working set sizes. The working set is made up of an array of `struct l` elements. A working set of 2^N bytes contains

2^N/`sizeof(struct l)`

elements. Obviously `sizeof(struct l)` depends on the value of `NPAD`. For 32-bit systems, `NPAD`=7 means the size of each array element is 32 bytes, for 64-bit systems the size is 64 bytes.

Single Threaded Sequential Access

The simplest case is a simple walk over all the entries in the list. The list elements are laid out sequentially, densely packed. Whether the order of processing is forward or backward does not matter, the processor can deal with both directions equally well. What we measure here—and in all the following tests—is how long it takes to handle a single list element. The time unit is a processor cycle. Figure 3.10 shows the result. Unless otherwise specified, all measurements are made on a Pentium 4 machine in 64-bit mode which means the structure `l` with `NPAD=0` is eight bytes in size.

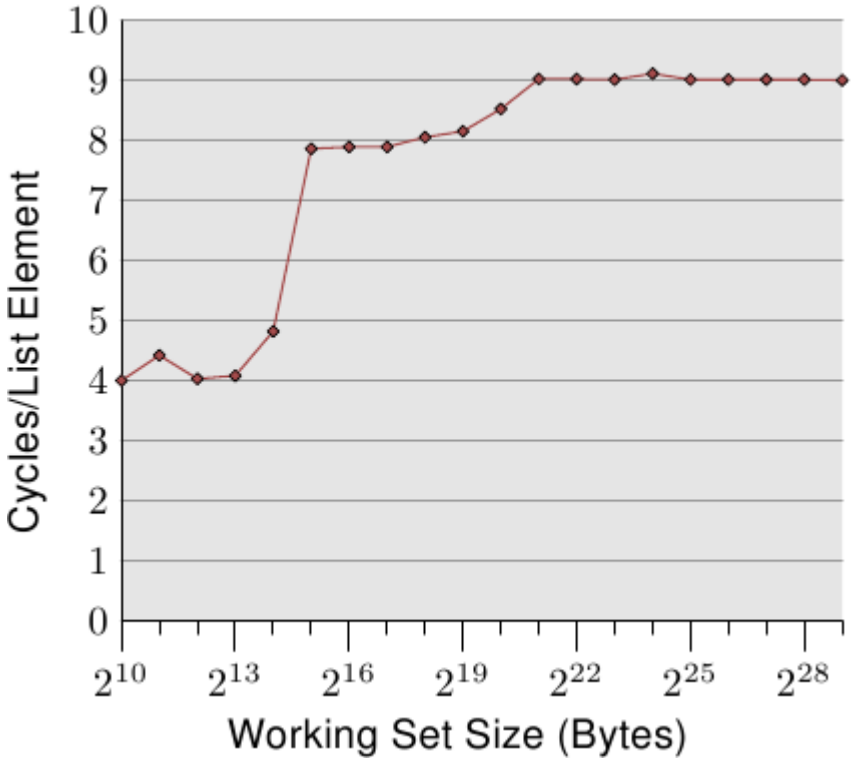


Figure 3.10: Sequential Read Access, NPAD=0

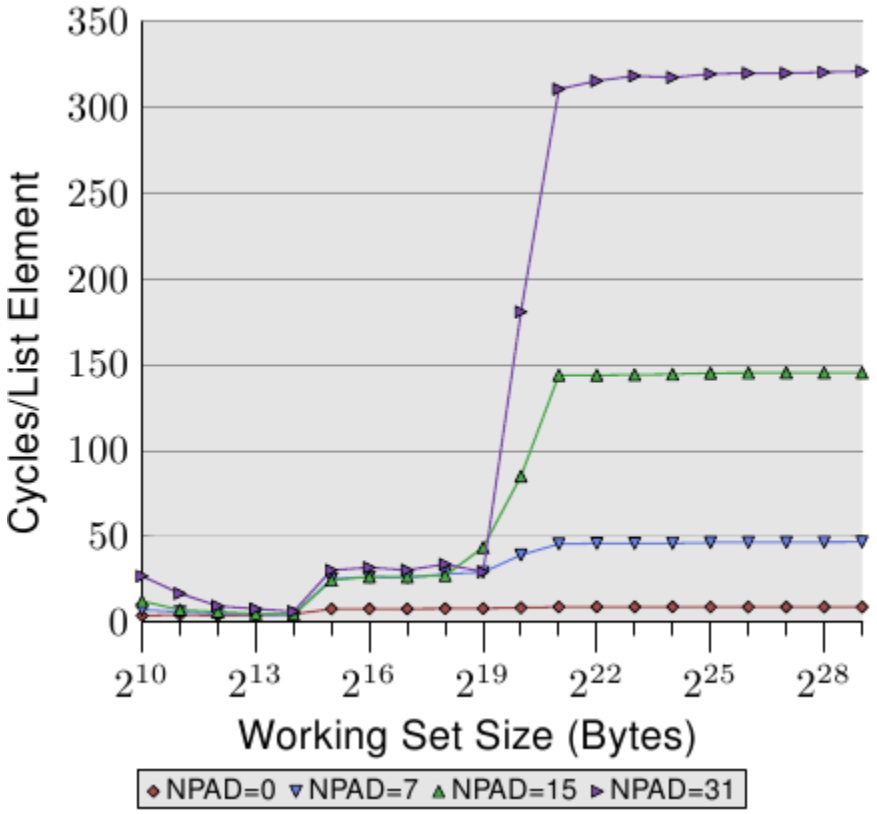


Figure 3.11: Sequential Read for Several Sizes

The first two measurements are polluted by noise. The measured workload is simply too small to filter the effects of the rest of the system out. We can safely assume that the values are all at the 4 cycles level. With this in mind we can see three distinct levels:

- Up to a working set size of 2^{14} bytes.
- From 2^{15} bytes to 2^{20} bytes.
- From 2^{21} bytes and up.

These steps can be easily explained: the processor has a 16kB L1d and 1MB L2. We do not see sharp edges in the transition from one level to the other because the caches are used by other parts of the system as well and so the cache is not exclusively available for the program data. Specifically the L2 cache is a unified cache and also used for the instructions (NB: Intel uses inclusive caches).

What is perhaps not quite expected are the actual times for the different working set sizes. The times for the L1d hits are expected: load times after an L1d hit are around 4 cycles on the P4. But what about the L2 accesses? Once the L1d is not sufficient to hold the data one might expect it would take 14 cycles or more per element since this is the access time for the L2. But the results show that only about 9 cycles are required. This discrepancy can be explained by the advanced logic in the processors. In anticipation of using consecutive memory regions, the processor *prefetches* the next cache line. This means that when the next line is actually used it is already halfway loaded. The delay required to wait for the next cache line to be loaded is therefore much less than the L2 access time.

The effect of prefetching is even more visible once the working set size grows beyond the L2 size. Before we said that a main memory access takes 200+ cycles. Only with effective prefetching is it possible for the processor to keep the access times as low as 9 cycles. As we can see from the difference between 200 and 9, this works out nicely.

We can observe the processor while prefetching, at least indirectly. In Figure 3.11 we see the times for the same working set sizes but this time we see the graphs for different sizes of the structure `l`. This means we have fewer but larger elements in the list. The different sizes have the effect that the distance between the `n` elements in the (still consecutive) list grows. In the four cases of the graph the distance is 0, 56, 120, and 248 bytes respectively.

At the bottom we can see the line from the previous graph, but this time it appears more or less as a flat line. The times for the other cases are simply so much worse. We can see in this graph, too, the three different levels and we see the large errors in the tests with the small working set sizes (ignore them again). The lines more or less all match each other as long as only the L1d is involved. There is no prefetching necessary so all element sizes just hit the L1d for each access.

For the L2 cache hits we see that the three new lines all pretty much match each other but that they are at a higher level (about 28). This is the level of the access time for the L2. This means prefetching from L2 into L1d is basically disabled. Even with NPAD=7 we need a new cache line for each iteration of the loop; for NPAD=0, instead, the loop has to iterate eight times before the next cache line is needed. The prefetch logic cannot load a new cache line every cycle. Therefore we see a stall to load from L2 in every iteration.

It gets even more interesting once the working set size exceeds the L2 capacity. Now all four lines vary widely. The different element sizes play obviously a big role in the difference in performance. The processor should recognize the size of the strides and not fetch unnecessary cache lines for NPAD=15 and 31 since the element size is smaller than the prefetch window (see Section 6.3.1). Where the element size is hampering the prefetching efforts is a result of a limitation of hardware prefetching: it cannot cross page boundaries. We are reducing the effectiveness of the hardware scheduler by 50% for each size increase. If the hardware prefetcher were allowed to cross page boundaries and the next page is not resident or valid the OS would have to get involved in locating the page. That means the program would experience a page fault it did not initiate itself. This is completely unacceptable since the processor does not know whether a page is not present or does not exist. In the latter case the OS would have to abort the process. In any case, given that, for NPAD=7 and higher, we need one cache line per list element the hardware prefetcher cannot do much. There simply is no time to load the data from memory since all the processor does is read one word and then load the next element.

Another big reason for the slowdown are the misses of the TLB cache. This is a cache where the results of the translation of a virtual address to a physical address are stored, as is explained in more detail in Section 4. The TLB cache is quite small since it has to be extremely fast. If more pages are accessed repeatedly than the TLB cache has entries for the translation from

virtual to physical address has to be constantly repeated. This is a very costly operation. With larger element sizes the cost of a TLB lookup is amortized over fewer elements. That means the total number of TLB entries which have to be computed per list element is higher.

To observe the TLB effects we can run a different test. For one measurement we lay out the elements sequentially as usual. We use `NPAD=7` for elements which occupy one entire cache line. For the second measurement we place each list element on a separate page. The rest of each page is left untouched and we do not count it in the total for the working set size. {*Yes, this is a bit inconsistent because in the other tests we count the unused part of the struct in the element size and we could define `NPAD` so that each element fills a page. In that case the working set sizes would be very different. This is not the point of this test, though, and since prefetching is ineffective anyway this makes little difference.*} The consequence is that, for the first measurement, each list iteration requires a new cache line and, for every 64 elements, a new page. For the second measurement each iteration requires loading a new cache line which is on a new page.

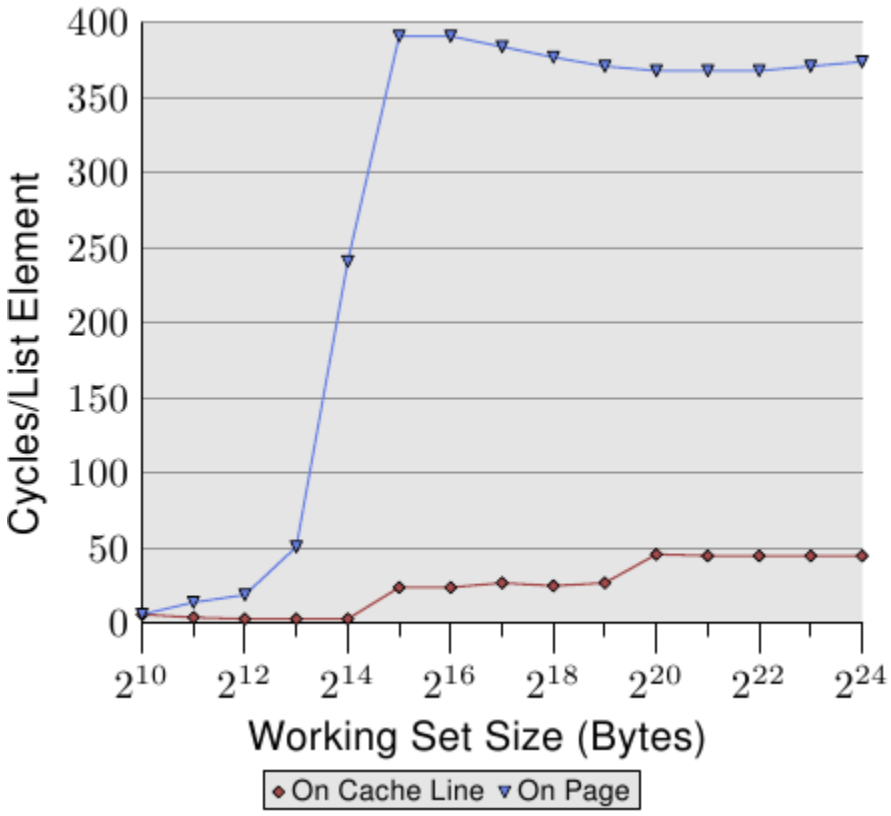


Figure 3.12: TLB Influence for Sequential Read

The result can be seen in Figure 3.12. The measurements were performed on the same machine as Figure 3.11. Due to limitations of the available RAM the working set size had to be restricted to 2^{24} bytes which requires 1GB to place the objects on separate pages. The lower, red curve corresponds exactly to the `NPAD`=7 curve in Figure 3.11. We see the distinct steps showing the sizes of the L1d and L2 caches. The second curve looks radically different. The important feature is the huge spike starting when the working set size reaches 2^{13} bytes. This is when the TLB cache overflows. With an element size of 64 bytes we can compute that the TLB cache has 64 entries. There are no page faults affecting the cost since the program locks the memory to prevent it from being swapped out.

As can be seen the number of cycles it takes to compute the physical address and store it in the TLB is very high. The graph in Figure 3.12 shows the extreme case, but it should now be clear that a significant factor in the slowdown for larger `NPAD` values is the reduced efficiency of the TLB cache.

Since the physical address has to be computed before a cache line can be read for either L2 or main memory the address translation penalties are additive to the memory access times. This in part explains why the total cost per list element for `NPAD`=31 is higher than the theoretical access time for the RAM.

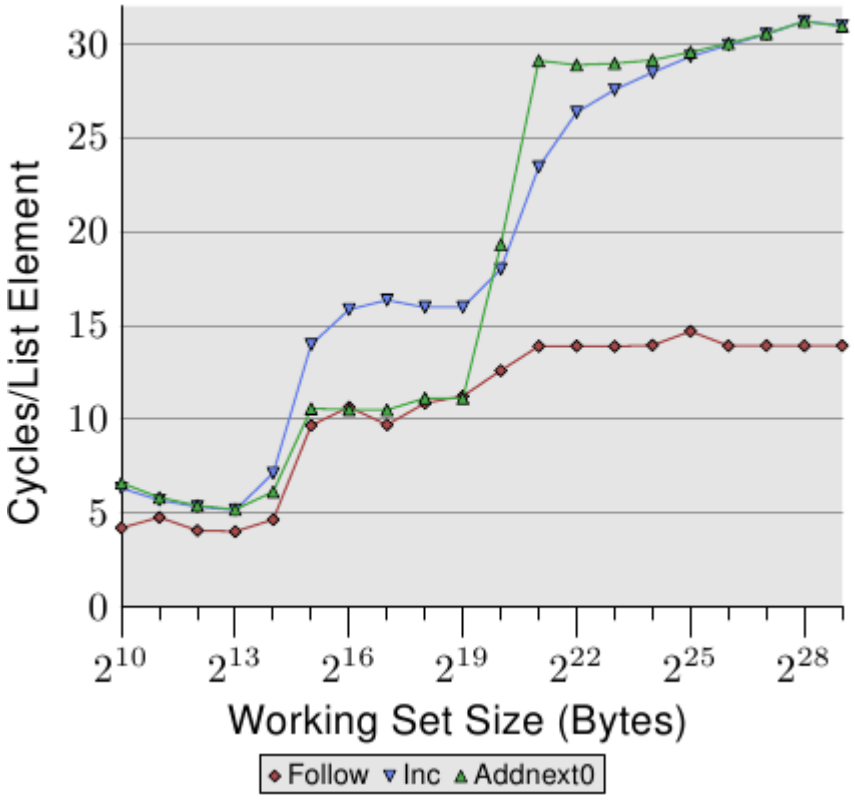


Figure 3.13: Sequential Read and Write, NPAD=1

We can glimpse a few more details of the prefetch implementation by looking at the data of test runs where the list elements are modified. Figure 3.13 shows three lines. The element width is in all cases 16 bytes. The first line is the now familiar list walk which serves as a baseline. The second line, labeled "Inc", simply increments the `pad[0]` member of the current element before going on to the next. The third line, labeled "Addnext0", takes the `pad[0]` list element of the *next* element and adds it to the `pad[0]` member of the current list element.

The naïve assumption would be that the "Addnext0" test runs slower because it has more work to do. Before advancing to the next list element a value from that element has to be loaded. This is why it is surprising to see that this test actually runs, for some working set sizes, faster than the "Inc" test. The explanation for this is that the load from the next list element is basically a forced prefetch. Whenever the program advances to the next list element we know for sure that element is already in the L1d cache. As a result we see that the "Addnext0" performs as well as the simple "Follow" test as long as the working set size fits into the L2 cache.

The "Addnext0" test runs out of L2 faster than the "Inc" test, though. It needs more data loaded from main memory. This is why the "Addnext0" test reaches the 28 cycles level for a working set size of 2^{21} bytes. The 28 cycles level is twice as high as the 14 cycles level the "Follow" test reaches. This is easy to explain, too. Since the other two tests modify memory an L2 cache eviction to make room for new cache lines cannot simply discard the data. Instead it has to be written to memory. This means the available bandwidth on the FSB is cut in half, hence doubling the time it takes to transfer the data from main memory to L2.

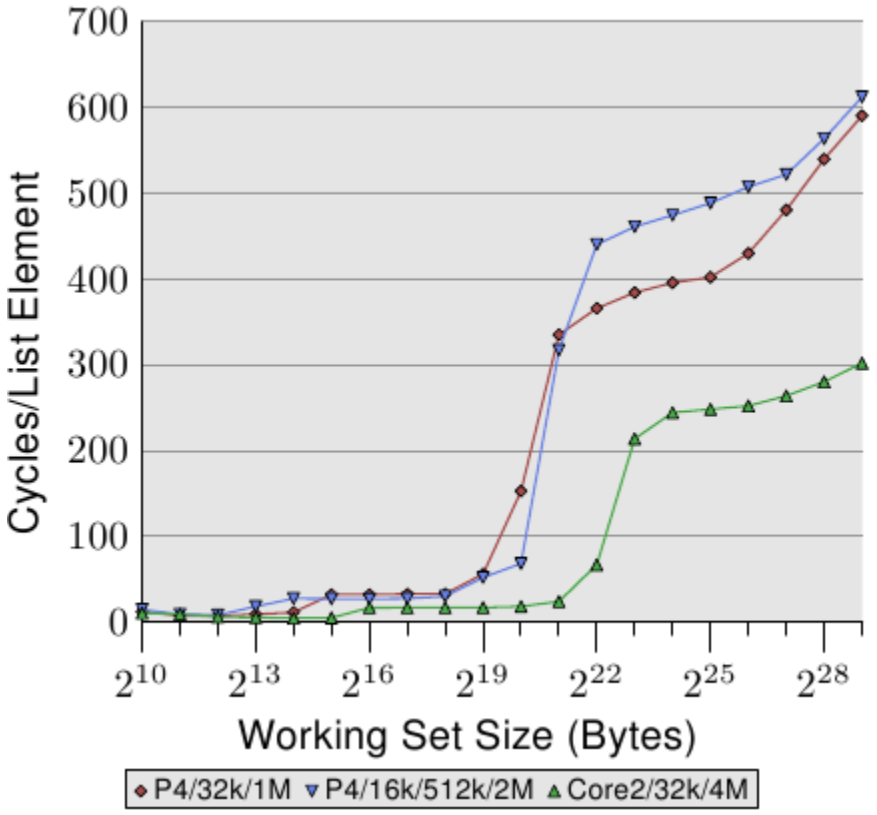


Figure 3.14: Advantage of Larger L2/L3 Caches

One last aspect of the sequential, efficient cache handling is the size of the cache. This should be obvious but it still should be pointed out. Figure 3.14 shows the timing for the Increment benchmark with 128-byte elements

(`NPAD`=15 on 64-bit machines). This time we see the measurement from three

different machines. The first two machines are P4s, the last one a Core2 processor. The first two differentiate themselves by having different cache sizes. The first processor has a 32k L1d and an 1M L2. The second one has 16k L1d, 512k L2, and 2M L3. The Core2 processor has 32k L1d and 4M L2.

The interesting part of the graph is not necessarily how well the Core2 processor performs relative to the other two (although it is impressive). The main point of interest here is the region where the working set size is too large for the respective last level cache and the main memory gets heavily involved.

Set	Sequential					Random				
Size	L2 Hit	L2 Miss	#Iter	Ratio Miss/Hit	L2 Accesses Per Iter	L2 Hit	L2 Miss	#Iter	Ratio Miss/Hit	L2 Accesses Per Iter

2^{20}	88,636	843	16,384	0.94%	5.5	30,462	4721	1,024	13.42%	34.4
2^{21}	88,105	1,584	8,192	1.77%	10.9	21,817	15,151	512	40.98%	72.2
2^{22}	88,106	1,600	4,096	1.78%	21.9	22,258	22,285	256	50.03%	174.0
2^{23}	88,104	1,614	2,048	1.80%	43.8	27,521	26,274	128	48.84%	420.3
2^{24}	88,114	1,655	1,024	1.84%	87.7	33,166	29,115	64	46.75%	973.1
2^{25}	88,112	1,730	512	1.93%	175.5	39,858	32,360	32	44.81%	2,256.8
2^{26}	88,112	1,906	256	2.12%	351.6	48,539	38,151	16	44.01%	5,418.1
2^{27}	88,114	2,244	128	2.48%	705.9	62,423	52,049	8	45.47%	14,309.0
2^{28}	88,120	2,939	64	3.23%	1,422.8	81,906	87,167	4	51.56%	42,268.3
2^{29}	88,137	4,318	32	4.67%	2,889.2	119,079	163,398	2	57.84%	141,238.5

Table 3.2: L2 Hits and Misses for Sequential and Random Walks, NPAD=0

As expected, the larger the last level cache is the longer the curve stays at the low level corresponding to the L2 access costs. The important part to notice is the performance advantage this provides. The second processor (which is slightly older) can perform the work on the working set of 2^{20} bytes twice as fast as the first processor. All thanks to the increased last level cache size. The Core2 processor with its 4M L2 performs even better.

For a random workload this might not mean that much. But if the workload can be tailored to the size of the last level cache the program performance can be increased quite dramatically. This is why it sometimes is worthwhile to spend the extra money for a processor with a larger cache.

Single Threaded Random Access Measurements

We have seen that the processor is able to hide most of the main memory and even L2 access latency by prefetching cache lines into L2 and L1d. This can work well only when the memory access is predictable, though.

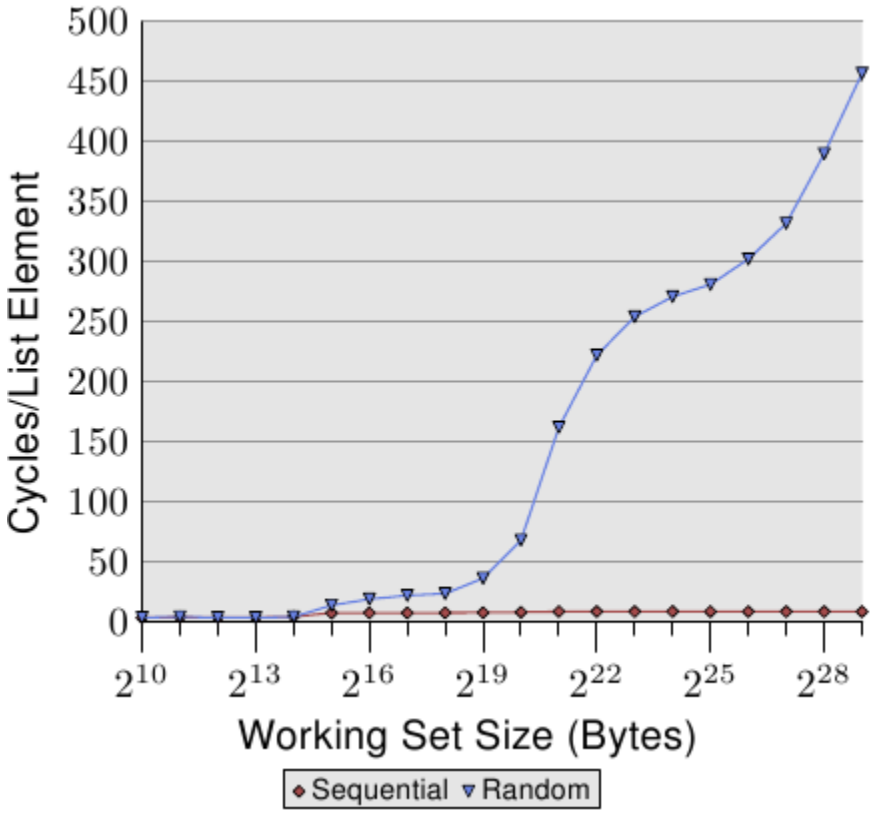


Figure 3.15: Sequential vs Random Read, NPAD=0

If the access is unpredictable or random the situation is quite different. Figure 3.15 compares the per-list-element times for the sequential access (same as in Figure 3.10) with the times when the list elements are randomly distributed in the working set. The order is determined by the linked list which is randomized. There is no way for the processor to reliably prefetch data. This can only work by chance if elements which are used shortly after one another are also close to each other in memory.

There are two important points to note in Figure 3.15. First, the large number is cycles needed for growing working set sizes. The machine makes it possible to access the main memory in 200-300 cycles but here we reach 450 cycles and more. We have seen this phenomenon before (compare Figure 3.11). The automatic prefetching is actually working to a disadvantage here.

The second interesting point is that the curve is not flattening at various plateaus as it has been for the sequential access cases. The curve keeps on rising. To explain this we can measure the L2 access of the program for the various working set sizes. The result can be seen in Figure 3.16 and Table 3.2.

The figure shows that, when the working set size is larger than the L2 size, the cache miss ratio (L2 misses / L2 access) starts to grow. The curve has a similar form to the one in Figure 3.15: it rises quickly, declines slightly, and starts to rise again. There is a strong correlation with the cycles per list element graph. The L2 miss rate will grow until it eventually reaches close to 100%. Given a large enough working set (and RAM) the probability that any of the randomly picked cache lines is in L2 or is in the process of being loaded can be reduced arbitrarily.

The increasing cache miss rate alone explains some of the costs. But there is another factor. Looking at Table 3.2 we can see in the L2/#Iter columns that the total number of L2 uses per iteration of the program is growing. Each working set is twice as large as the one before. So, without caching we would expect double the main memory accesses. With caches and (almost) perfect predictability we see the modest increase in the L2 use shown in the data for sequential access. The increase is due to the increase of the working set size and nothing else.

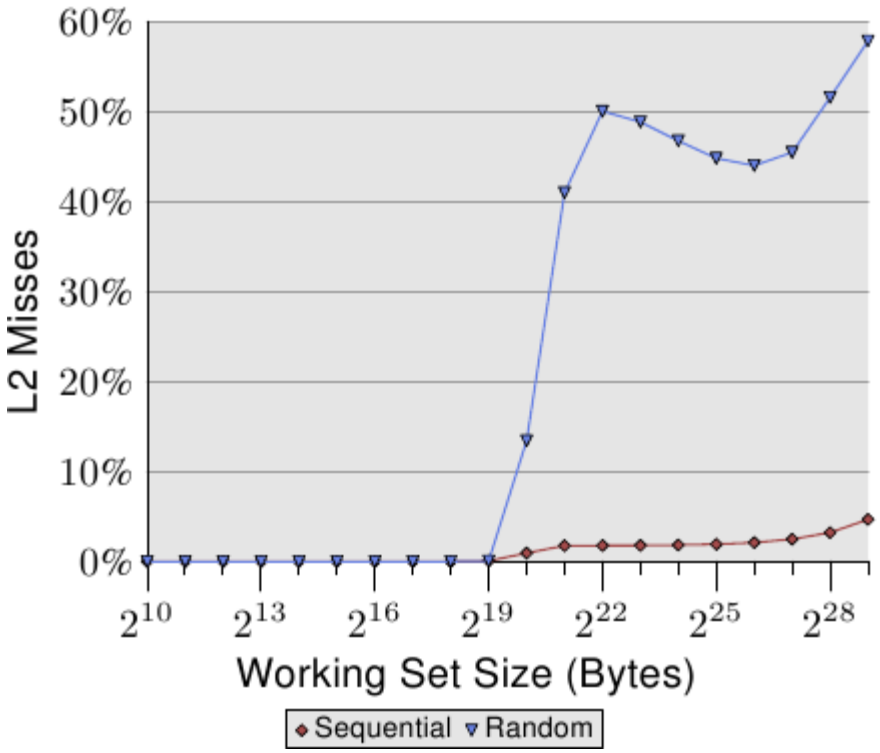


Figure 3.16: L2d Miss Ratio

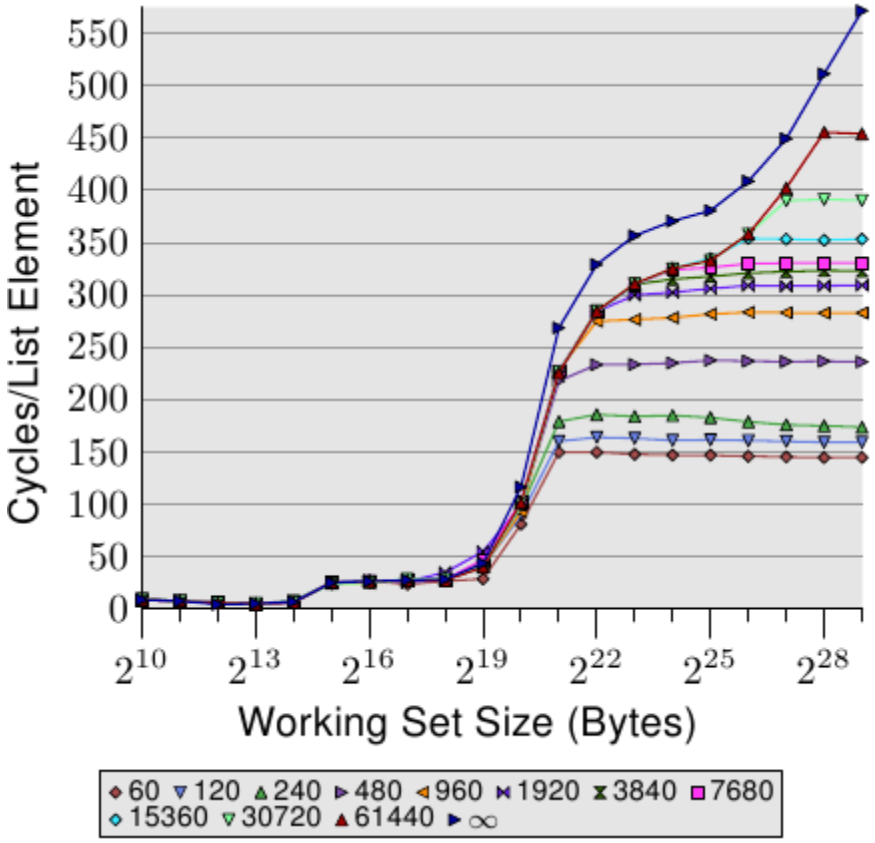


Figure 3.17: Page-Wise Randomization, NPAD=7

For random access the per-element time increases by more than 100% for each doubling of the working set size. This means the average access time per list element increases since the working set size only doubles. The reason behind this is a rising rate of TLB misses. In Figure 3.17 we see the cost for random accesses for `NPAD`=7. Only this time the randomization is modified. While in the normal case the entire list of randomized as one block (indicated by the label ∞) the other 11 curves show randomizations which are performed in smaller blocks. For the curve labeled '60' each set of 60 pages (245.760 bytes) is randomized individually. That means all list elements in the block are traversed before going over to an element in the next block. This has the effect that number of TLB entries which are used at any one time is limited.

The element size for `NPAD`=7 is 64 bytes, which corresponds to the cache line size. Due to the randomized order of the list elements it is unlikely that the hardware prefetcher has any effect, most certainly not for more than a handful of elements. This means the L2 cache miss rate does not differ significantly from the randomization of the entire list in one block. The performance of the test with increasing block size approaches

asymptotically the curve for the one-block randomization. This means the performance of this latter test case is significantly influenced by the TLB misses. If the TLB misses can be lowered the performance increases significantly (in one test we will see later up to 38%).

3.3.3 Write Behavior

Before we start looking at the cache behavior when multiple execution contexts (threads or processes) use the same memory we have to explore a detail of cache implementations. Caches are supposed to be coherent and this coherency is supposed to be completely transparent for the userlevel code. Kernel code is a different story; it occasionally requires cache flushes.

This specifically means that, if a cache line is modified, the result for the system after this point in time is the same as if there were no cache at all and the main memory location itself had been modified. This can be implemented in two ways or policies:

- write-through cache implementation;
- write-back cache implementation.

The write-through cache is the simplest way to implement cache coherency. If the cache line is written to, the processor immediately also writes the cache line into main memory. This ensures that, at all times, the main memory and cache are in sync. The cache content could simply be discarded whenever a cache line is replaced. This cache policy is simple but not very fast. A program which, for instance, modifies a local variable over and over again would create a lot of traffic on the FSB even though the data is likely not used anywhere else and might be short-lived.

The write-back policy is more sophisticated. Here the processor does not immediately write the modified cache line back to main memory. Instead, the cache line is only marked as dirty. When the cache line is dropped from the cache at some point in the future the dirty bit will instruct the processor to write the data back at that time instead of just discarding the content.

Write-back caches have the chance to be significantly better performing, which is why most memory in a system with a decent processor is cached this way. The processor can even take advantage of free capacity on the FSB to store the content of a cache line before the line has to be evacuated. This allows the dirty bit to be cleared and the processor can just drop the cache line when the room in the cache is needed.

But there is a significant problem with the write-back implementation. When more than one processor (or core or hyper-thread) is available and

accessing the same memory it must still be assured that both processors see the same memory content at all times. If a cache line is dirty on one processor (i.e., it has not been written back yet) and a second processor tries to read the same memory location, the read operation cannot just go out to the main memory. Instead the content of the first processor's cache line is needed. In the next section we will see how this is currently implemented.

Before we get to this there are two more cache policies to mention:

- write-combining; and
- uncacheable.

Both these policies are used for special regions of the address space which are not backed by real RAM. The kernel sets up these policies for the address ranges (on x86 processors using the Memory Type Range Registers, MTRRs) and the rest happens automatically. The MTRRs are also usable to select between write-through and write-back policies.

Write-combining is a limited caching optimization more often used for RAM on devices such as graphics cards. Since the transfer costs to the devices are much higher than the local RAM access it is even more important to avoid doing too many transfers. Transferring an entire cache line just because a word in the line has been written is wasteful if the next operation modifies the next word. One can easily imagine that this is a common occurrence, the memory for horizontal neighboring pixels on a screen are in most cases neighbors, too. As the name suggests, write-combining combines multiple write accesses before the cache line is written out. In ideal cases the entire cache line is modified word by word and, only after the last word is written, the cache line is written to the device. This can speed up access to RAM on devices significantly.

Finally there is uncacheable memory. This usually means the memory location is not backed by RAM at all. It might be a special address which is hardcoded to have some functionality outside the CPU. For commodity hardware this most often is the case for memory mapped address ranges which translate to accesses to cards and devices attached to a bus (PCIe etc). On embedded boards one sometimes finds such a memory address which can be used to turn an LED on and off. Caching such an address would obviously be a bad idea. LEDs in this context are used for debugging or status reports and one wants to see this as soon as possible. The memory on PCIe cards can change without the CPU's interaction, so this memory should not be cached.

3.3.4 Multi-Processor Support

In the previous section we have already pointed out the problem we have when multiple processors come into play. Even multi-core processors have the problem for those cache levels which are not shared (at least the L1d).

It is completely impractical to provide direct access from one processor to the cache of another processor. The connection is simply not fast enough, for a start. The practical alternative is to transfer the cache content over to the other processor in case it is needed. Note that this also applies to caches which are not shared on the same processor.

The question now is when does this cache line transfer have to happen? This question is pretty easy to answer: when one processor needs a cache line which is dirty in another processor's cache for reading or writing. But how can a processor determine whether a cache line is dirty in another processor's cache? Assuming it is just because a cache line is loaded by another processor would be suboptimal (at best). Usually the majority of memory accesses are read accesses and the resulting cache lines are not dirty. Processor operations on cache lines are frequent (of course, why else would we have this paper?) which means broadcasting information about changed cache lines after each write access would be impractical.

What developed over the years is the MESI cache coherency protocol (Modified, Exclusive, Shared, Invalid). The protocol is named after the four states a cache line can be in when using the MESI protocol:

- **Modified**: The local processor has modified the cache line. This also implies it is the only copy in any cache.
- **Exclusive**: The cache line is not modified but known to not be loaded into any other processor's cache.
- **Shared**: The cache line is not modified and might exist in another processor's cache.
- **Invalid**: The cache line is invalid, i.e., unused.

This protocol developed over the years from simpler versions which were less complicated but also less efficient. With these four states it is possible to efficiently implement write-back caches while also supporting concurrent use of read-only data on different processors.

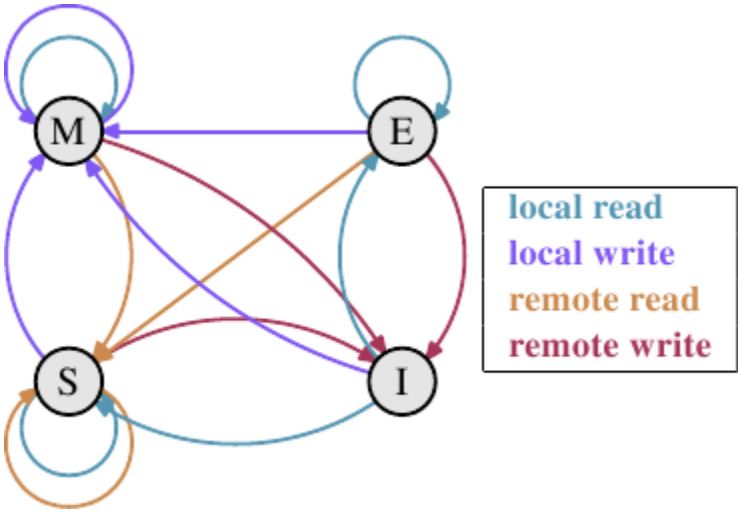


Figure 3.18: MESI Protocol Transitions

The state changes are accomplished without too much effort by the processors listening, or snooping, on the other processors' work. Certain operations a processor performs are announced on external pins and thus make the processor's cache handling visible to the outside. The address of the cache line in question is visible on the address bus. In the following description of the states and their transitions (shown in Figure 3.18) we will point out when the bus is involved.

Initially all cache lines are empty and hence also Invalid. If data is loaded into the cache for writing the cache changes to Modified. If the data is loaded for reading the new state depends on whether another processor has the cache line loaded as well. If this is the case then the new state is Shared, otherwise Exclusive.

If a Modified cache line is read from or written to on the local processor, the instruction can use the current cache content and the state does not change. If a second processor wants to read from the cache line the first processor has to send the content of its cache to the second processor and then it can change the state to Shared. The data sent to the second processor is also received and processed by the memory controller which stores the content in memory. If this did not happen the cache line could not be marked as Shared. If the second processor wants to write to the cache line the first processor sends the cache line content and marks the cache line locally as Invalid. This is the infamous “Request For Ownership” (RFO) operation. Performing this operation in the last level cache, just like the I→M transition is comparatively expensive. For write-through caches we also have to add the time it takes to write the new cache line content to the next higher-level cache or the main memory, further increasing the cost.

If a cache line is in the Shared state and the local processor reads from it no state change is necessary and the read request can be fulfilled from the cache. If the cache line is locally written to the cache line can be used as well but the state changes to Modified. It also requires that all other possible copies of the cache line in other processors are marked as Invalid. Therefore the write operation has to be announced to the other processors via an RFO message. If the cache line is requested for reading by a second processor nothing has to happen. The main memory contains the current data and the local state is already Shared. In case a second processor wants to write to the cache line (RFO) the cache line is simply marked Invalid. No bus operation is needed.

The Exclusive state is mostly identical to the Shared state with one crucial difference: a local write operation does *not* have to be announced on the bus. The local cache copy is known to be the only one. This can be a huge advantage so the processor will try to keep as many cache lines as possible in the Exclusive state instead of the Shared state. The latter is the fallback in case the information is not available at that moment. The Exclusive state can also be left out completely without causing functional problems. It is only the performance that will suffer since the E→M transition is much faster than the S→M transition.

From this description of the state transitions it should be clear where the costs specific to multi-processor operations are. Yes, filling caches is still expensive but now we also have to look out for RFO messages. Whenever such a message has to be sent things are going to be slow.

There are two situations when RFO messages are necessary:

- A thread is migrated from one processor to another and all the cache lines have to be moved over to the new processor once.
- A cache line is truly needed in two different processors. {*At a smaller level the same is true for two cores on the same processor. The costs are just a bit smaller. The RFO message is likely to be sent many times.*}

In multi-thread or multi-process programs there is always some need for synchronization; this synchronization is implemented using memory. So there are some valid RFO messages. They still have to be kept as infrequent as possible. There are other sources of RFO messages, though. In Section 6 we will explain these scenarios. The Cache coherency protocol messages must be distributed among the processors of the system. A MESI transition cannot happen until it is clear that all the processors in the system have had a chance to reply to the message. That means that the longest possible time a reply can take determines the speed of the coherency protocol. {*Which is*

why we see nowadays, for instance, AMD Opteron systems with three sockets. Each processor is exactly one hop away given that the processors only have three hyperlinks and one is needed for the Southbridge connection.} Collisions on the bus are possible, latency can be high in NUMA systems, and of course sheer traffic volume can slow things down. All good reasons to focus on avoiding unnecessary traffic.

There is one more problem related to having more than one processor in play. The effects are highly machine specific but in principle the problem always exists: the FSB is a shared resource. In most machines all processors are connected via one single bus to the memory controller (see Figure 2.1). If a single processor can saturate the bus (as is usually the case) then two or four processors sharing the same bus will restrict the bandwidth available to each processor even more.

Even if each processor has its own bus to the memory controller as in Figure 2.2 there is still the bus to the memory modules. Usually this is one bus but, even in the extended model in Figure 2.2, concurrent accesses to the same memory module will limit the bandwidth.

The same is true with the AMD model where each processor can have local memory. Yes, all processors can concurrently access their local memory quickly. But multi-thread and multi-process programs--at least from time to time--have to access the same memory regions to synchronize.

Concurrency is severely limited by the finite bandwidth available for the implementation of the necessary synchronization. Programs need to be carefully designed to minimize accesses from different processors and cores to the same memory locations. The following measurements will show this and the other cache effects related to multi-threaded code.

Multi Threaded Measurements

To ensure that the gravity of the problems introduced by concurrently using the same cache lines on different processors is understood, we will look here at some more performance graphs for the same program we used before. This time, though, more than one thread is running at the same time. What is measured is the fastest runtime of any of the threads. This means the time for a complete run when all threads are done is even higher. The machine used has four processors; the tests use up to four threads. All processors share one bus to the memory controller and there is only one bus to the memory modules.

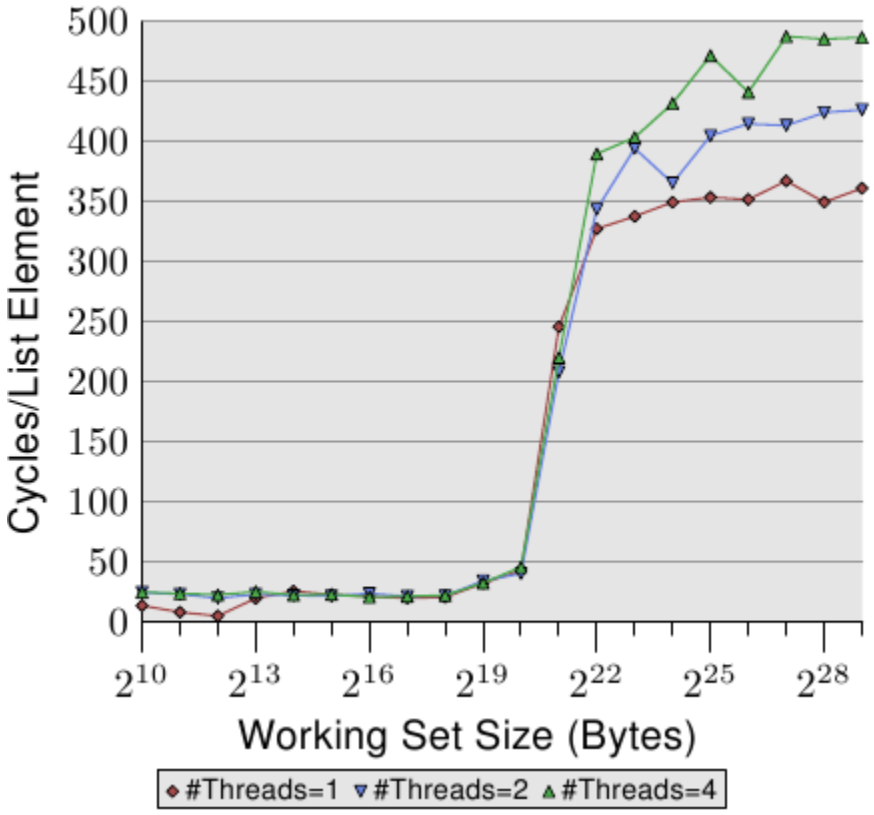


Figure 3.19: Sequential Read Access, Multiple Threads

Figure 3.19 shows the performance for sequential read-only access for 128 bytes entries (`NPAD`=15 on 64-bit machines). For the curve for one thread we can expect a curve similar to Figure 3.11. The measurements are for a different machine so the actual numbers vary.

The important part in this figure is of course the behavior when running multiple threads. Note that no memory is modified and no attempts are made to keep the threads in sync when walking the linked list. Even though no RFO messages are necessary and all the cache lines can be shared, we see up to an 18% performance decrease for the fastest thread when two threads are used and up to 34% when four threads are used. Since no cache lines have to be transported between the processors this slowdown is solely caused by one or both of the two bottlenecks: the shared bus from the processor to the memory controller and bus from the memory controller to the memory modules. Once the working set size is larger than the L3 cache in this machine all three threads will be prefetching new list elements. Even with two threads the available bandwidth is not sufficient to scale linearly (i.e., have no penalty from running multiple threads).

When we modify memory things get even uglier. Figure 3.20 shows the results for the sequential Increment test.

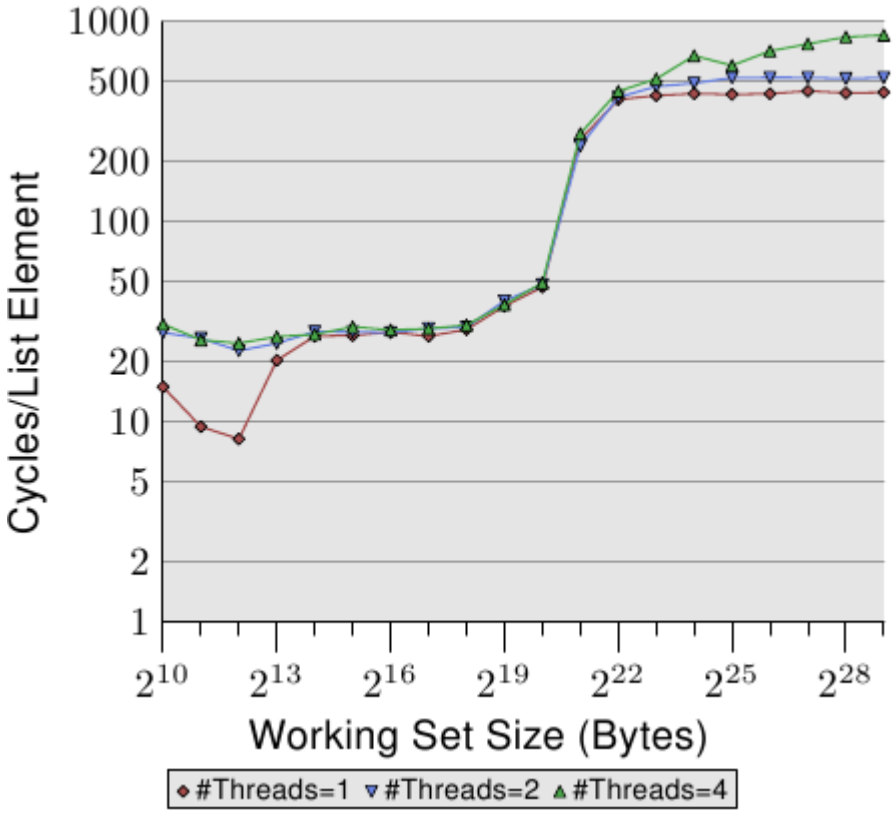


Figure 3.20: Sequential Increment, Multiple Threads

This graph is using a logarithmic scale for the Y axis. So, do not be fooled by the apparently small differences. We still have about a 18% penalty for running two threads and now an amazing 93% penalty for running four threads. This means the prefetch traffic together with the write-back traffic is pretty much saturating the bus when four threads are used.

We use the logarithmic scale to show the results for the L1d range. What can be seen is that, as soon as more than one thread is running, the L1d is basically ineffective. The single-thread access times exceed 20 cycles only when the L1d is not sufficient to hold the working set. When multiple threads are running, those access times are hit immediately, even with the smallest working set sizes.

One aspect of the problem is not shown here. It is hard to measure with this specific test program. Even though the test modifies memory and we therefore must expect RFO messages we do not see higher costs for the L2 range when more than one thread is used. The program would have to use a large amount of memory and all threads must access the same memory in parallel. This is hard to achieve without a lot of synchronization which would then dominate the execution time.

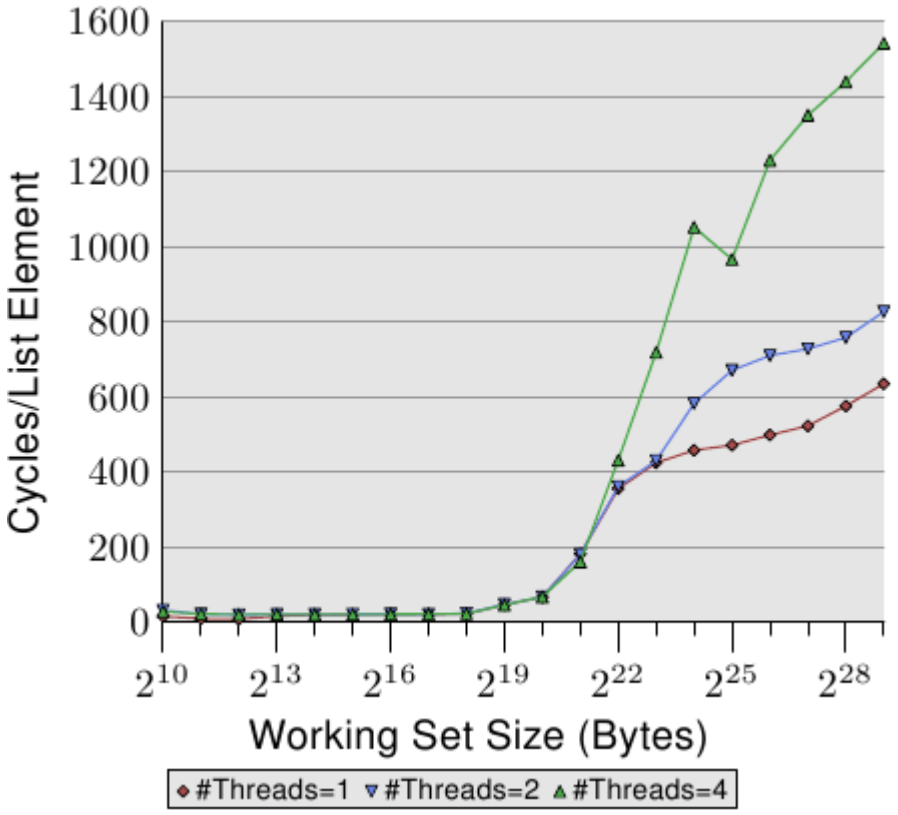


Figure 3.21: Random Addnextlast, Multiple Threads

Finally in Figure 3.21 we have the numbers for the Addnextlast test with random access of memory. This figure is provided mainly to show to the appallingly high numbers. It now take around 1,500 cycles to process a single list element in the extreme case. The use of more threads is even more questionable. We can summarize the efficiency of multiple thread use in a table.

#Threads	Seq Read	Seq Inc	Rand Add
2	1.69	1.69	1.54
4	2.98	2.07	1.65

Table 3.3: Efficiency for Multiple Threads

The table shows the efficiency for the multi-thread run with the largest working set size in the three figures on Figure 3.21. The number shows the best possible speed-up the test program incurs for the largest working set size by using two or four threads. For two threads the theoretical limits for the speed-up are 2 and, for four threads, 4. The numbers for two threads are not that bad. But for four threads the numbers for the last test show that it is almost not worth it to scale beyond two threads. The additional benefit is

minuscule. We can see this more easily if we represent the data in Figure 3.21 a bit differently.

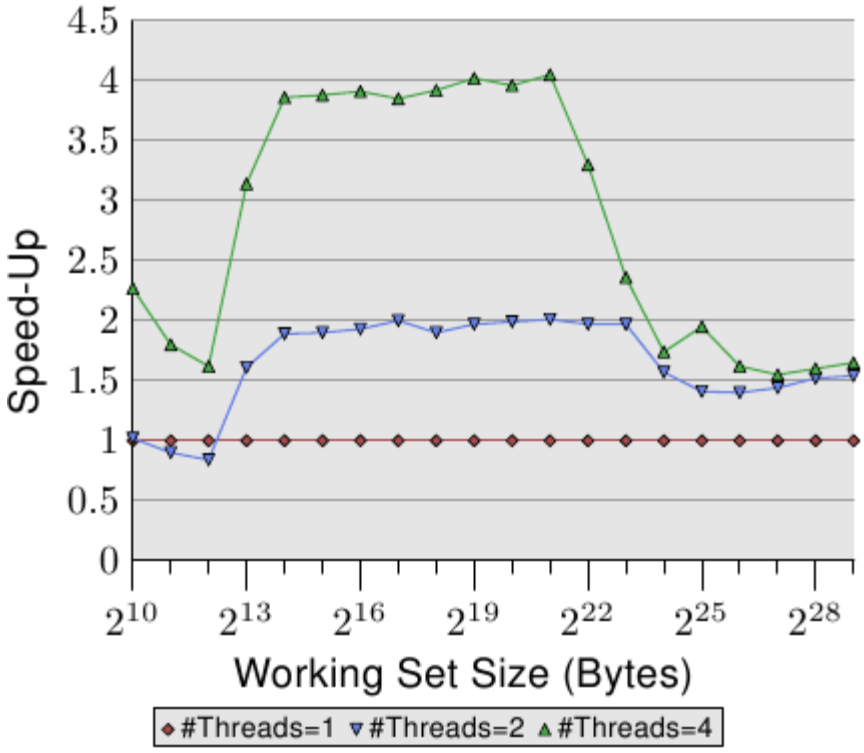


Figure 3.22: Speed-Up Through Parallelism

The curves in Figure 3.22 show the speed-up factors, i.e., relative performance compared to the code executed by a single thread. We have to ignore the smallest sizes, the measurements are not accurate enough. For the range of the L2 and L3 cache we can see that we indeed achieve almost linear acceleration. We almost reach factors of 2 and 4 respectively. But as soon as the L3 cache is not sufficient to hold the working set the numbers crash. They crash to the point that the speed-up of two and four threads is identical (see the fourth column in Table 3.3). This is one of the reasons why one can hardly find motherboard with sockets for more than four CPUs all using the same memory controller. Machines with more processors have to be built differently (see Section 5).

These numbers are not universal. In some cases even working sets which fit into the last level cache will not allow linear speed-ups. In fact, this is the norm since threads are usually not as decoupled as is the case in this test program. On the other hand it is possible to work with large working sets and still take advantage of more than two threads. Doing this requires thought, though. We will talk about some approaches in Section 6.

Special Case: Hyper-Threads

Hyper-Threads (sometimes called Symmetric Multi-Threading, SMT) are implemented by the CPU and are a special case since the individual threads cannot really run concurrently. They all share almost all the processing resources except for the register set. Individual cores and CPUs still work in parallel but the threads implemented on each core are limited by this restriction. In theory there can be many threads per core but, so far, Intel's CPUs at most have two threads per core. The CPU is responsible for time-multiplexing the threads. This alone would not make much sense, though. The real advantage is that the CPU can schedule another hyper-thread when the currently running hyper-thread is delayed. In most cases this is a delay caused by memory accesses.

If two threads are running on one hyper-threaded core the program is only more efficient than the single-threaded code if the *combined* runtime of both threads is lower than the runtime of the single-threaded code. This is possible by overlapping the wait times for different memory accesses which usually would happen sequentially. A simple calculation shows the minimum requirement on the cache hit rate to achieve a certain speed-up.

The execution time for a program can be approximated with a simple model with only one level of cache as follows (see [htimpact]):

$$T_{exe} = N[(1-F_{mem})T_{proc} + F_{mem}(G_{hit}T_{cache} + (1-G_{hit})T_{miss})]$$

The meaning of the variables is as follows:

N = Number of instructions.

F_{mem} = Fraction of N that access memory.

G_{hit} = Fraction of loads that hit the cache.

T_{proc} = Number of cycles per instruction.

T_{cache} = Number of cycles for cache hit.

T_{miss} = Number of cycles for cache miss.

T_{exe} = Execution time for program.

For it to make any sense to use two threads the execution time of each of the two threads must be at most half of that of the single-threaded code. The only variable on either side is the number of cache hits. If we solve the equation for the minimum cache hit rate required to not slow down the thread execution by 50% or more we get the graph in Figure 3.23.

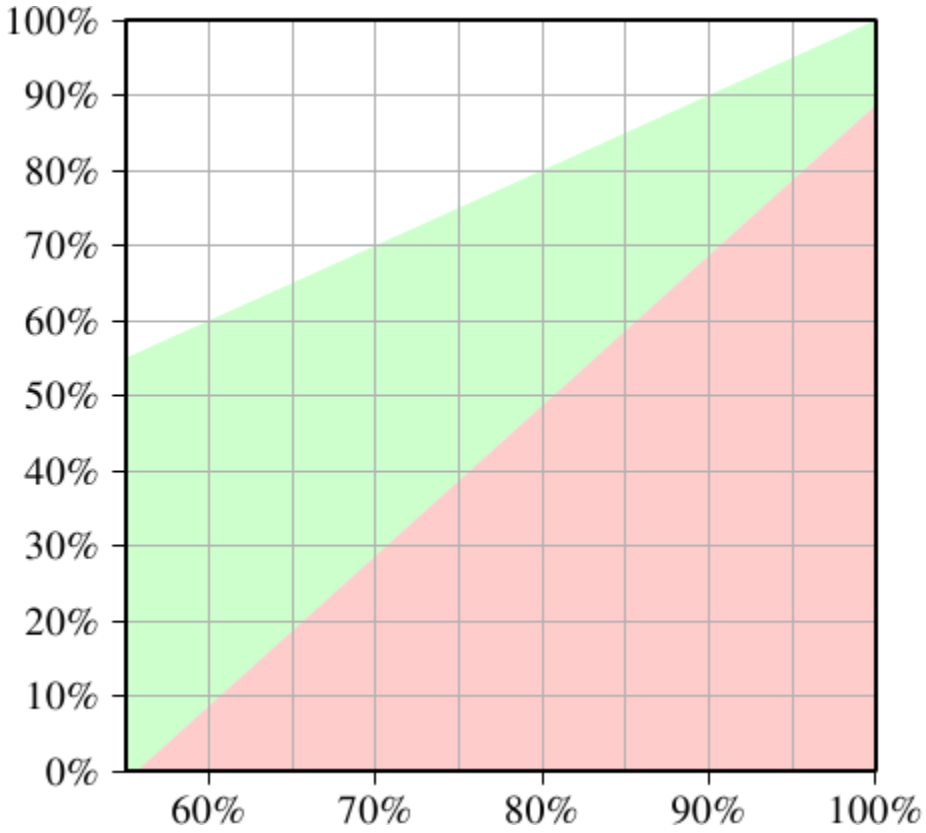


Figure 3.23: Minimum Cache Hit Rate For Speed-Up

The X–axis represents the cache hit rate G_{hit} of the single-thread code. The Y–axis shows the required cache hit rate for the multi-threaded code. This value can never be higher than the single-threaded hit rate since, otherwise, the single-threaded code would use that improved code, too. For single-threaded hit rates below 55% the program can in all cases benefit from using threads. The CPU is more or less idle enough due to cache misses to enable running a second hyper-thread.

The green area is the target. If the slowdown for the thread is less than 50% and the workload of each thread is halved the combined runtime might be less than the single-thread runtime. For the modeled system here (numbers for a P4 with hyper-threads were used) a program with a hit rate of 60% for the single-threaded code requires a hit rate of at least 10% for the dual-threaded program. That is usually doable. But if the single-threaded code has a hit rate of 95% then the multi-threaded code needs a hit rate of at least 80%. That is harder. Especially, and this is the problem with hyper-threads, because now the effective cache size (L1d here, in practice also L2 and so on) available to each hyper-thread is cut in half. Both hyper-threads use the same cache to load their data. If the working set of the two threads is non-overlapping the original 95% hit rate could also be cut in half and is therefore much lower than the required 80%.

Hyper-Threads are therefore only useful in a limited range of situations. The cache hit rate of the single-threaded code must be low enough that given the equations above and reduced cache size the new hit rate still meets the goal. Then and only then can it make any sense at all to use hyper-threads. Whether the result is faster in practice depends on whether the processor is sufficiently able to overlap the wait times in one thread with execution times in the other threads. The overhead of parallelizing the code must be added to the new total runtime and this additional cost often cannot be neglected.

In Section 6.3.4 we will see a technique where threads collaborate closely and the tight coupling through the common cache is actually an advantage. This technique can be applicable to many situations if only the programmers are willing to put in the time and energy to extend their code.

What should be clear is that if the two hyper-threads execute completely different code (i.e., the two threads are treated like separate processors by the OS to execute separate processes) the cache size is indeed cut in half which means a significant increase in cache misses. Such OS scheduling practices are questionable unless the caches are sufficiently large. Unless the workload for the machine consists of processes which, through their design, can indeed benefit from hyper-threads it might be best to turn off hyper-threads in the computer's BIOS. {*Another reason to keep hyper-threads enabled is debugging. SMT is astonishingly good at finding some sets of problems in parallel code.*}

3.3.5 Other Details

So far we talked about the address as consisting of three parts, tag, set index, and cache line offset. But what address is actually used? All relevant processors today provide virtual address spaces to processes, which means that there are two different kinds of addresses: virtual and physical.

The problem with virtual addresses is that they are not unique. A virtual address can, over time, refer to different physical memory addresses. The same address in different process also likely refers to different physical addresses. So it is always better to use the physical memory address, right?

The problem here is that instructions use virtual addresses and these have to be translated with the help of the Memory Management Unit (MMU) into physical addresses. This is a non-trivial operation. In the pipeline to execute an instruction the physical address might only be available at a later stage. This means that the cache logic has to be very quick in determining whether the memory location is cached. If virtual addresses could be used the cache lookup can happen much earlier in the pipeline and in case of a cache hit the

memory content can be made available. The result is that more of the memory access costs could be hidden by the pipeline.

Processor designers are currently using virtual address tagging for the first level caches. These caches are rather small and can be cleared without too much pain. At least partial clearing the cache is necessary if the page table tree of a process changes. It might be possible to avoid a complete flush if the processor has an instruction which specifies the virtual address range which has changed. Given the low latency of L1i and L1d caches (~3 cycles) using virtual addresses is almost mandatory.

For larger caches including L2, L3, ... caches physical address tagging is needed. These caches have a higher latency and the virtual→physical address translation can finish in time. Because these caches are larger (i.e., a lot of information is lost when they are flushed) and refilling them takes a long time due to the main memory access latency, flushing them often would be costly.

It should, in general, not be necessary to know about the details of the address handling in those caches. They cannot be changed and all the factors which would influence the performance are normally something which should be avoided or is associated with high cost. Overflowing the cache capacity is bad and all caches run into problems early if the majority of the used cache lines fall into the same set. The latter can be avoided with virtually addressed caches but is impossible for user-level processes to avoid for caches addressed using physical addresses. The only detail one might want to keep in mind is to not map the same physical memory location to two or more virtual addresses in the same process, if at all possible.

Another detail of the caches which is rather uninteresting to programmers is the cache replacement strategy. Most caches evict the Least Recently Used (LRU) element first. This is always a good default strategy. With larger associativity (and associativity might indeed grow further in the coming years due to the addition of more cores) maintaining the LRU list becomes more and more expensive and we might see different strategies adopted.

As for the cache replacement there is not much a programmer can do. If the cache is using physical address tags there is no way to find out how the virtual addresses correlate with the cache sets. It might be that cache lines in all logical pages are mapped to the same cache sets, leaving much of the cache unused. If anything, it is the job of the OS to arrange that this does not happen too often.

With the advent of virtualization things get even more complicated. Now not even the OS has control over the assignment of physical memory. The

Virtual Machine Monitor (VMM, aka hypervisor) is responsible for the physical memory assignment.

The best a programmer can do is to a) use logical memory pages completely and b) use page sizes as large as meaningful to diversify the physical addresses as much as possible. Larger page sizes have other benefits, too, but this is another topic (see Section 4).

3.4 Instruction Cache

Not just the data used by the processor is cached; the instructions executed by the processor are also cached. However, this cache is much less problematic than the data cache. There are several reasons:

- The quantity of code which is executed depends on the size of the code that is needed. The size of the code in general depends on the complexity of the problem. The complexity of the problem is fixed.
- While the program's data handling is designed by the programmer the program's instructions are usually generated by a compiler. The compiler writers know about the rules for good code generation.
- Program flow is much more predictable than data access patterns. Today's CPUs are very good at detecting patterns. This helps with prefetching.
- Code always has quite good spatial and temporal locality.

There are a few rules programmers should follow but these mainly consist of rules on how to use the tools. We will discuss them in Section 6. Here we talk only about the technical details of the instruction cache.

Ever since the core clock of CPUs increased dramatically and the difference in speed between cache (even first level cache) and core grew, CPUs have been pipelined. That means the execution of an instruction happens in stages. First an instruction is decoded, then the parameters are prepared, and finally it is executed. Such a pipeline can be quite long (> 20 stages for Intel's Netburst architecture). A long pipeline means that if the pipeline stalls (i.e., the instruction flow through it is interrupted) it takes a while to get up to speed again. Pipeline stalls happen, for instance, if the location of the next instruction cannot be correctly predicted or if it takes too long to load the next instruction (e.g., when it has to be read from memory).

As a result CPU designers spend a lot of time and chip real estate on branch prediction so that pipeline stalls happen as infrequently as possible.

On CISC processors the decoding stage can also take some time. The x86 and x86-64 processors are especially affected. In recent years these

processors therefore do not cache the raw byte sequence of the instructions in L1i but instead they cache the decoded instructions. L1i in this case is called the “trace cache”. Trace caching allows the processor to skip over the first steps of the pipeline in case of a cache hit which is especially good if the pipeline stalled.

As said before, the caches from L2 on are unified caches which contain both code and data. Obviously here the code is cached in the byte sequence form and not decoded.

To achieve the best performance there are only a few rules related to the instruction cache:

1. Generate code which is as small as possible. There are exceptions when software pipelining for the sake of using pipelines requires creating more code or where the overhead of using small code is too high.
2. Whenever possible, help the processor to make good prefetching decisions. This can be done through code layout or with explicit prefetching.

These rules are usually enforced by the code generation of a compiler. There are a few things the programmer can do and we will talk about them in Section 6.

3.4.1 Self Modifying Code

In early computer days memory was a premium. People went to great lengths to reduce the size of the program to make more room for program data. One trick frequently deployed was to change the program itself over time. Such Self Modifying Code (SMC) is occasionally still found, these days mostly for performance reasons or in security exploits.

SMC should in general be avoided. Though it is generally executed correctly there are boundary cases in which it is not and it creates performance problems if not done correctly. Obviously, code which is changed cannot be kept in the trace cache which contains the decoded instructions. But even if the trace cache is not used because the code has not been executed at all (or for some time) the processor might have problems. If an upcoming instruction is changed while it already entered the pipeline the processor has to throw away a lot of work and start all over again. There are even situations where most of the state of the processor has to be tossed away.

Finally, since the processor assumes - for simplicity reasons and because it is true in 99.9999999% of all cases - that the code pages are immutable, the L1i implementation does not use the MESI protocol but instead a simplified SI protocol. This means if modifications are detected a lot of pessimistic assumptions have to be made.

It is highly advised to avoid SMC whenever possible. Memory is not such a scarce resource anymore. It is better to write separate functions instead of modifying one function according to specific needs. Maybe one day SMC support can be made optional and we can detect exploit code trying to modify code this way. If SMC absolutely has to be used, the write operations should bypass the cache as to not create problems with data in L1d needed in L1i. See Section 6.1 for more information on these instructions.

Normally on Linux it is easy to recognize programs which contain SMC. All program code is write-protected when built with the regular toolchain. The programmer has to perform significant magic at link time to create an executable where the code pages are writable. When this happens, modern Intel x86 and x86-64 processors have dedicated performance counters which count uses of self-modifying code. With the help of these counters it is quite easily possible to recognize programs with SMC even if the program will succeed due to relaxed permissions.

3.5 Cache Miss Factors

We have already seen that when memory accesses miss the caches, the costs skyrocket. Sometimes this is not avoidable and it is important to understand the actual costs and what can be done to mitigate the problem.

3.5.1 Cache and Memory Bandwidth

To get a better understanding of the capabilities of the processors we measure the bandwidth available in optimal circumstances. This measurement is especially interesting since different processor versions vary widely. This is why this section is filled with the data of several different machines. The program to measure performance uses the SSE instructions of the x86 and x86-64 processors to load or store 16 bytes at once. The working set is increased from 1kB to 512MB just as in our other tests and it is measured how many bytes per cycle can be loaded or stored.

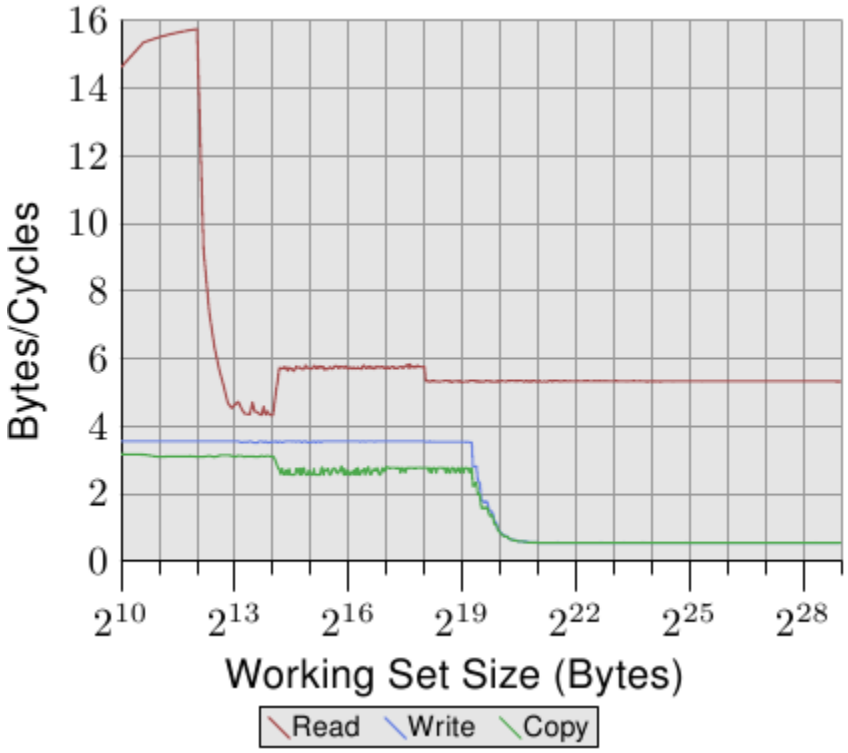


Figure 3.24: Pentium 4 Bandwidth

Figure 3.24 shows the performance on a 64-bit Intel Netburst processor. For working set sizes which fit into L1d the processor is able to read the full 16 bytes per cycle, i.e., one load instruction is performed per cycle

(the `movaps` instruction moves 16 bytes at once). The test does not do

anything with the read data, we test only the read instructions themselves. As soon as the L1d is not sufficient anymore the performance goes down dramatically to less than 6 bytes per cycle. The step at 2^{18} bytes is due to the exhaustion of the DTLB cache which means additional work for each new page. Since the reading is sequential prefetching can predict the accesses perfectly and the FSB can stream the memory content at about 5.3 bytes per cycle for all sizes of the working set. The prefetched data is not propagated into L1d, though. These are of course numbers which will never be achievable in a real program. Think of them as practical limits.

What is more astonishing than the read performance is the write and copy performance. The write performance, even for small working set sizes, does not ever rise above 4 bytes per cycle. This indicates that, in these Netburst processors, Intel elected to use a Write-Through mode for L1d where the performance is obviously limited by the L2 speed. This also means that the performance of the copy test, which copies from one memory region into a second, non-overlapping memory region, is not significantly worse. The necessary read operations are so much faster and can partially overlap with

the write operations. The most noteworthy detail of the write and copy measurements is the low performance once the L2 cache is not sufficient anymore. The performance drops to 0.5 bytes per cycle! That means write operations are by a factor of ten slower than the read operations. This means optimizing those operations is even more important for the performance of the program.

In Figure 3.25 we see the results on the same processor but with two threads running, one pinned to each of the two hyper-threads of the processor.

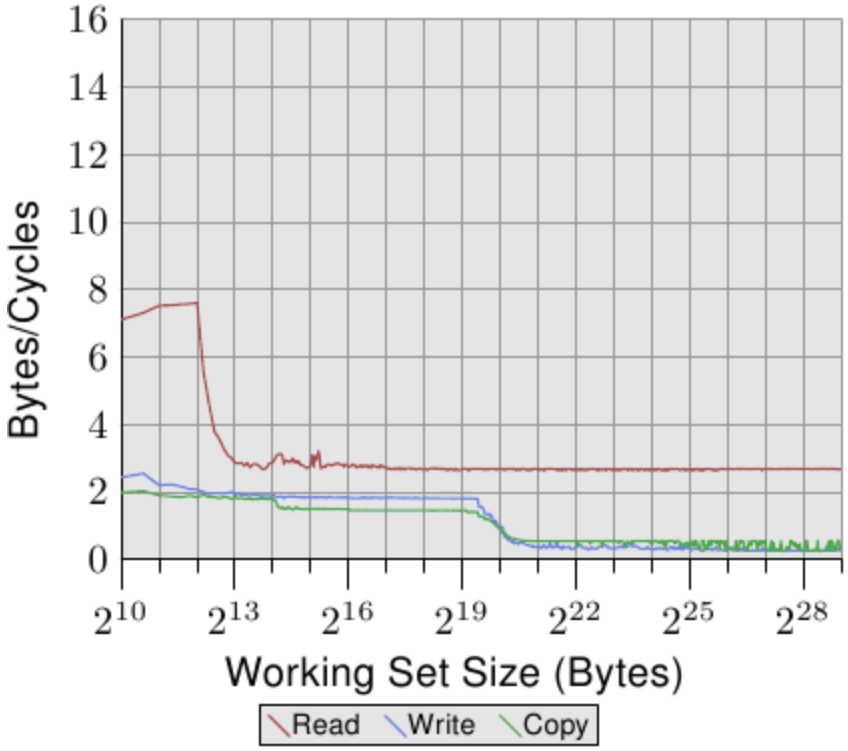


Figure 3.25: P4 Bandwidth with 2 Hyper-Threads

The graph is shown at the same scale as the previous one to illustrate the differences and the curves are a bit jittery simply because of the problem of measuring two concurrent threads. The results are as expected. Since the hyper-threads share all the resources except the registers each thread has only half the cache and bandwidth available. That means even though each thread has to wait a lot and could award the other thread with execution time this does not make any difference since the other thread also has to wait for the memory. This truly shows the worst possible use of hyper-threads.

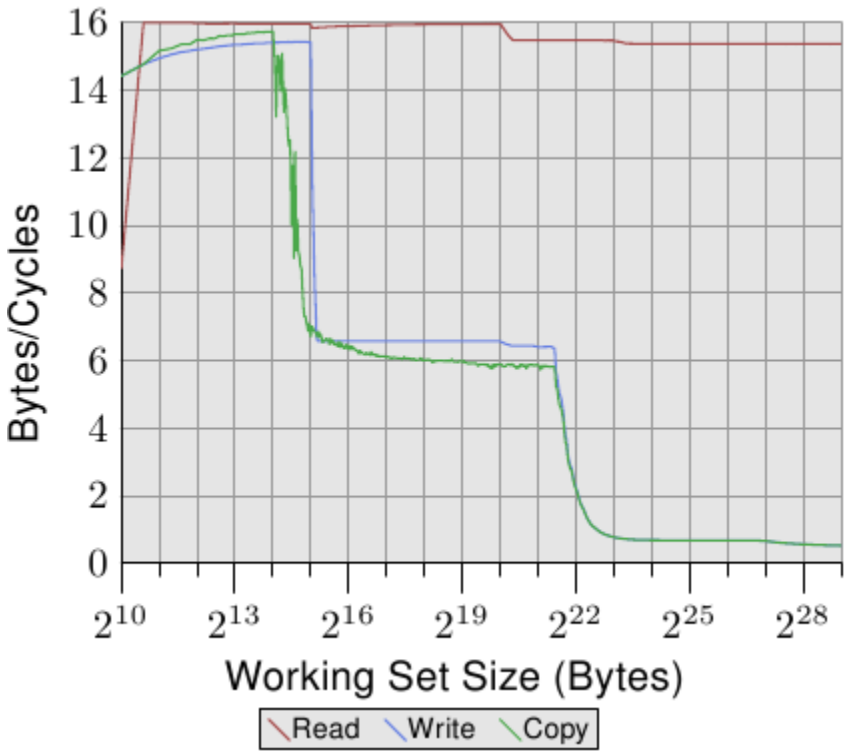


Figure 3.26: Core 2 Bandwidth

Compared to Figures 3.24 and 3.25 the results in Figures 3.26 and 3.27 look quite different for an Intel Core 2 processor. This is a dual-core processor with shared L2 which is four times as big as the L2 on the P4 machine. This only explains the delayed drop-off of the write and copy performance, though.

There are much bigger differences. The read performance throughout the working set range hovers around the optimal 16 bytes per cycle. The drop-off in the read performance after 2^{20} bytes is again due to the working set being too big for the DTLB. Achieving these high numbers means the processor is not only able to prefetch the data and transport the data in time. It also means the data is prefetched into L1d.

The write and copy performance is dramatically different, too. The processor does not have a Write-Through policy; written data is stored in L1d and only evicted when necessary. This allows for write speeds close to the optimal 16 bytes per cycle. Once L1d is not sufficient anymore the performance drops significantly. As with the Netburst processor, the write performance is significantly lower. Due to the high read performance the difference is even higher here. In fact, when even the L2 is not sufficient anymore the speed difference increases to a factor of 20! This does not mean the Core 2 processors perform poorly. To the contrary, their performance is always better than the Netburst core's.

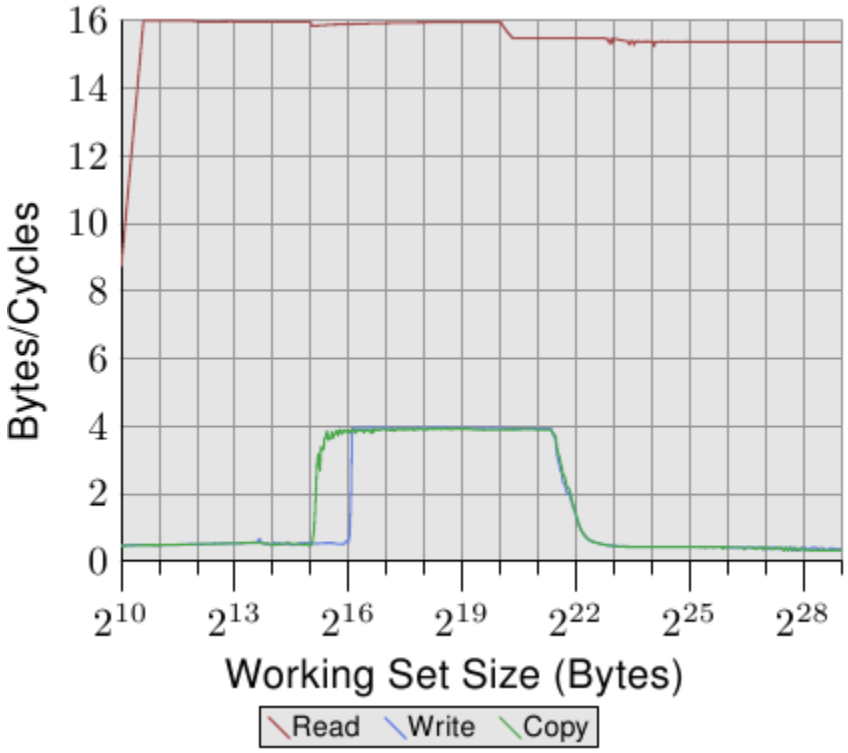


Figure 3.27: Core 2 Bandwidth with 2 Threads

In Figure 3.27, the test runs two threads, one on each of the two cores of the Core 2 processor. Both threads access the same memory, not necessarily perfectly in sync, though. The results for the read performance are not different from the single-threaded case. A few more jitters are visible which is to be expected in any multi-threaded test case.

The interesting point is the write and copy performance for working set sizes which would fit into L1d. As can be seen in the figure, the performance is the same as if the data had to be read from the main memory. Both threads compete for the same memory location and RFO messages for the cache lines have to be sent. The problematic point is that these requests are not handled at the speed of the L2 cache, even though both cores share the cache. Once the L1d cache is not sufficient anymore modified entries are flushed from each core's L1d into the shared L2. At that point the performance increases significantly since now the L1d misses are satisfied by the L2 cache and RFO messages are only needed when the data has not yet been flushed. This is why we see a 50% reduction in speed for these sizes of the working set. The asymptotic behavior is as expected: since both cores share the same FSB each core gets half the FSB bandwidth which means for large working sets each thread's performance is about half that of the single threaded case.

Because there are significant differences between the processor versions of one vendor it is certainly worthwhile looking at the performance of other vendors' processors, too. Figure 3.28 shows the performance of an AMD family 10h Opteron processor. This processor has 64kB L1d, 512kB L2, and 2MB of L3. The L3 cache is shared between all cores of the processor. The results of the performance test can be seen in Figure 3.28.

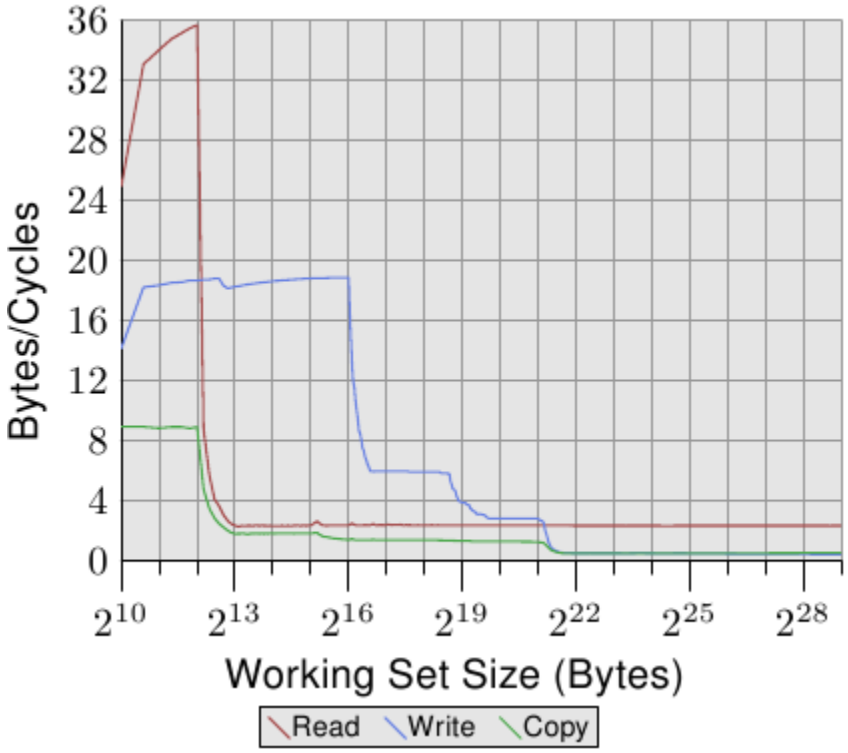


Figure 3.28: AMD Family 10h Opteron Bandwidth

The first detail one notices about the numbers is that the processor is capable of handling two instructions per cycle if the L1d cache is sufficient. The read performance exceeds 32 bytes per cycle and even the write performance is, with 18.7 bytes per cycle, high. The read curve flattens quickly, though, and is, with 2.3 bytes per cycle, pretty low. The processor for this test does not prefetch any data, at least not efficiently.

The write curve on the other hand performs according to the sizes of the various caches. The peak performance is achieved for the full size of the L1d, going down to 6 bytes per cycle for L2, to 2.8 bytes per cycle for L3, and finally .5 bytes per cycle if not even L3 can hold all the data. The performance for the L1d cache exceeds that of the (older) Core 2 processor, the L2 access is equally fast (with the Core 2 having a larger cache), and the L3 and main memory access is slower.

The copy performance cannot be better than either the read or write performance. This is why we see the curve initially dominated by the read performance and later by the write performance.

The multi-thread performance of the Opteron processor is shown in Figure 3.29.

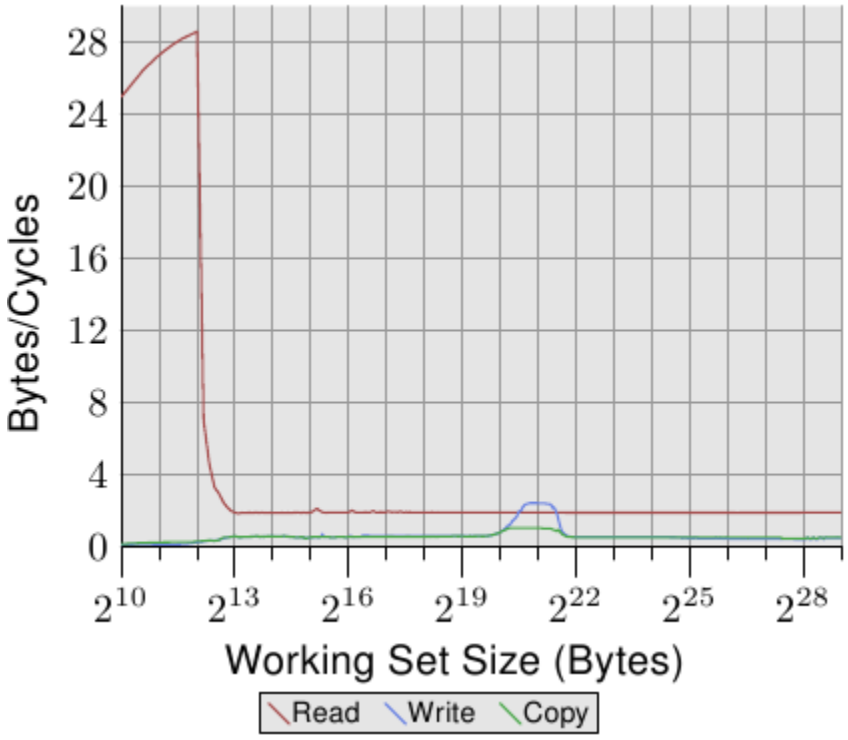


Figure 3.29: AMD Fam 10h Bandwidth with 2 Threads

The read performance is largely unaffected. Each thread's L1d and L2 works as before and the L3 cache is in this case not prefetched very well either. The two threads do not unduly stress the L3 for their purpose. The big problem in this test is the write performance. All data the threads share has to go through the L3 cache. This sharing seems to be quite inefficient since even if the L3 cache size is sufficient to hold the entire working set the cost is significantly higher than an L3 access. Comparing this graph with Figure 3.27 we see that the two threads of the Core 2 processor operate at the speed of the shared L2 cache for the appropriate range of working set sizes. This level of performance is achieved for the Opteron processor only for a very small range of the working set sizes and even here it approaches *only* the speed of the L3 which is slower than the Core 2's L2.

3.5.2 Critical Word Load

Memory is transferred from the main memory into the caches in blocks which are smaller than the cache line size. Today 64 *bits* are transferred at

once and the cache line size is 64 or 128 *bytes*. This means 8 or 16 transfers per cache line are needed.

The DRAM chips can transfer those 64-bit blocks in burst mode. This can fill the cache line without any further commands from the memory controller and the possibly associated delays. If the processor prefetches cache lines this is probably the best way to operate.

If a program's cache access of the data or instruction caches misses (that means, it is a compulsory cache miss, because the data is used for the first time, or a capacity cache miss, because the limited cache size requires eviction of the cache line) the situation is different. The word inside the cache line which is required for the program to continue might not be the first word in the cache line. Even in burst mode and with double data rate transfer the individual 64-bit blocks arrive at noticeably different times. Each block arrives 4 CPU cycles or more later than the previous one. If the word the program needs to continue is the eighth of the cache line, the program has to wait an additional 30 cycles or more after the first word arrives.

Things do not necessarily have to be like this. The memory controller is free to request the words of the cache line in a different order. The processor can communicate which word the program is waiting on, the *critical word*, and the memory controller can request this word first. Once the word arrives the program can continue while the rest of the cache line arrives and the cache is not yet in a consistent state. This technique is called Critical Word First & Early Restart.

Processors nowadays implement this technique but there are situations when that is not possible. If the processor prefetches data the critical word is not known. Should the processor request the cache line during the time the prefetch operation is in flight it will have to wait until the critical word arrives without being able to influence the order.

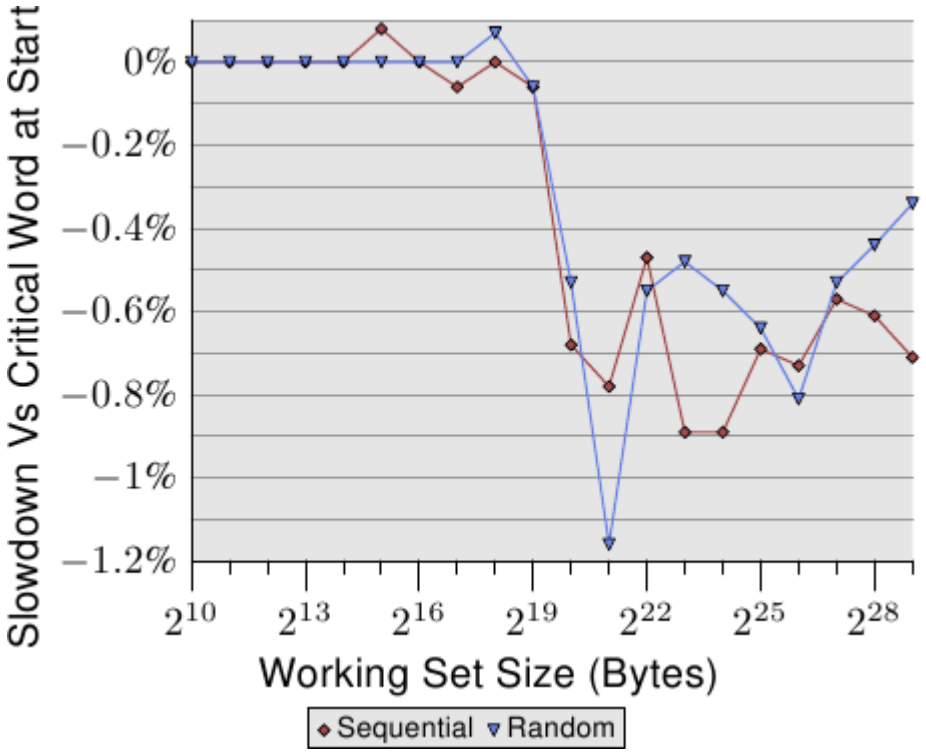


Figure 3.30: Critical Word at End of Cache Line

Even with these optimizations in place the position of the critical word on a cache line matters. Figure 3.30 shows the Follow test for sequential and random access. Shown is the slowdown of running the test with the pointer which is chased in the first word versus the case when the pointer is in the last word. The element size is 64 bytes, corresponding the cache line size. The numbers are quite noisy but it can be seen that, as soon as the L2 is not sufficient to hold the working set size, the performance of the case where the critical word is at the end is about 0.7% slower. The sequential access appears to be affected a bit more. This would be consistent with the aforementioned problem when prefetching the next cache line.

3.5.3 Cache Placement

Where the caches are placed in relationship to the hyper-threads, cores, and processors is not under control of the programmer. But programmers can determine where the threads are executed and then it becomes important how the caches relate to the used CPUs.

Here we will not go into details of when to select what cores to run the threads. We will only describe architecture details which the programmer has to take into account when setting the affinity of the threads.

Hyper-threads, by definition share everything but the register set. This includes the L1 caches. There is not much more to say here. The fun starts

with the individual cores of a processor. Each core has at least its own L1 caches. Aside from this there are today not many details in common:

- Early multi-core processors had separate L2 caches and no higher caches.
- Later Intel models had shared L2 caches for dual-core processors. For quad-core processors we have to deal with separate L2 caches for each pair of two cores. There are no higher level caches.
- AMD's family 10h processors have separate L2 caches and a unified L3 cache.

A lot has been written in the propaganda material of the processor vendors about the advantage of their respective models. Separate L2 caches have an advantage if the working sets handled by the cores do not overlap. This works well for single-threaded programs. Since this is still often the reality today this approach does not perform too badly. But there is always some overlap. The caches all contain the most actively used parts of the common runtime libraries which means some cache space is wasted.

Completely sharing all caches beside L1 as Intel's dual-core processors do can have a big advantage. If the working set of the threads working on the two cores overlaps significantly the total available cache memory is increased and working sets can be bigger without performance degradation. If the working sets do not overlap Intel's Advanced Smart Cache management is supposed to prevent any one core from monopolizing the entire cache.

If both cores use about half the cache for their respective working sets there is some friction, though. The cache constantly has to weigh the two cores' cache use and the evictions performed as part of this rebalancing might be chosen poorly. To see the problems we look at the results of yet another test program.

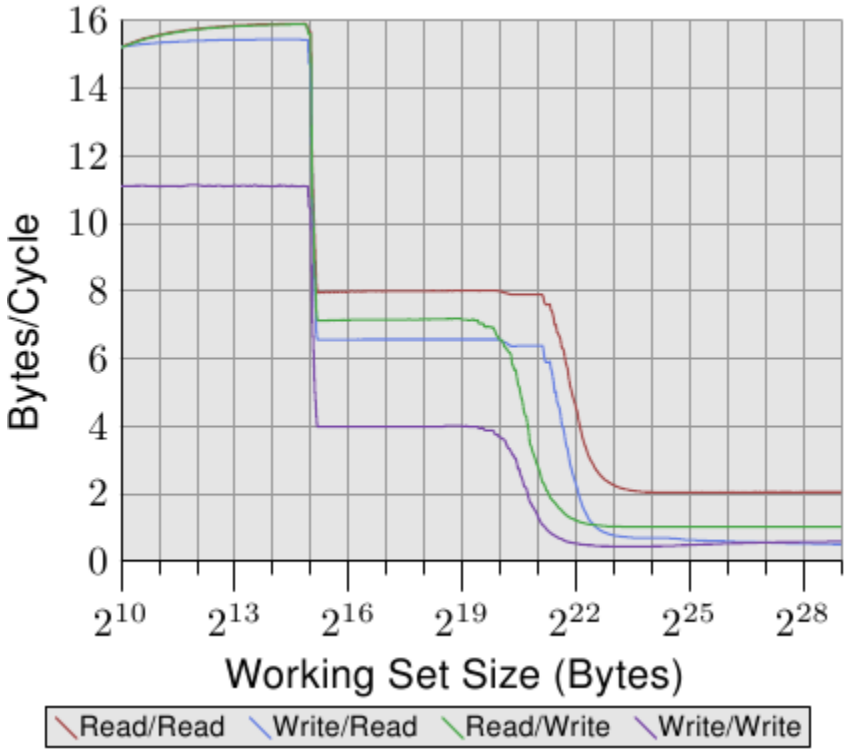


Figure 3.31: Bandwidth with two Processes

The test program has one process constantly reading or writing, using SSE instructions, a 2MB block of memory. 2MB was chosen because this is half the size of the L2 cache of this Core 2 processor. The process is pinned to one core while a second process is pinned to the other core. This second process reads and writes a memory region of variable size. The graph shows the number of bytes per cycle which are read or written. Four different graphs are shown, one for each combination of the processes reading and writing. The read/write graph is for the background process, which always uses a 2MB working set to write, and the measured process with variable working set to read.

The interesting part of the graph is the part between 2^{20} and 2^{23} bytes. If the L2 cache of the two cores were completely separate we could expect that the performance of all four tests would drop between 2^{21} and 2^{22} bytes, that means, once the L2 cache is exhausted. As we can see in Figure 3.31 this is not the case. For the cases where the background process is writing this is most visible. The performance starts to deteriorate before the working set size reaches 1MB. The two processes do not share memory and therefore the processes do not cause RFO messages to be generated. These are pure cache eviction problems. The smart cache handling has its problems with the effect that the experienced cache size per core is closer to 1MB than the 2MB per core which are available. One can only hope that, if caches shared

between cores remain a feature of upcoming processors, the algorithm used for the smart cache handling will be fixed.

Having a quad-core processor with two L2 caches was just a stop-gap solution before higher-level caches could be introduced. This design provides no significant performance advantage over separate sockets and dual-core processors. The two cores communicate via the same bus which is, at the outside, visible as the FSB. There is no special fast-track data exchange.

The future of cache design for multi-core processors will lie in more layers. AMD's 10h family of processors make the start. Whether we will continue to see lower level caches be shared by a subset of the cores of a processor remains to be seen. The extra levels of cache are necessary since the high-speed and frequently used caches cannot be shared among many cores. The performance would be impacted. It would also require very large caches with high associativity. Both numbers, the cache size and the associativity, must scale with the number of cores sharing the cache. Using a large L3 cache and reasonably-sized L2 caches is a reasonable trade-off. The L3 cache is slower but it is ideally not as frequently used as the L2 cache.

For programmers all these different designs mean complexity when making scheduling decisions. One has to know the workloads and the details of the machine architecture to achieve the best performance. Fortunately we have support to determine the machine architecture. The interfaces will be introduced in later sections.

3.5.4 FSB Influence

The FSB plays a central role in the performance of the machine. Cache content can only be stored and loaded as quickly as the connection to the memory allows. We can show how much so by running a program on two machines which only differ in the speed of their memory modules. Figure 3.32 shows the results of the Addnext0 test (adding the content of the next elements `pad[0]` element to the own `pad[0]` element) for `NPAD`=7 on a 64-bit machine. Both machines have Intel Core 2 processors, the first uses 667MHz DDR2 modules, the second 800MHz modules (a 20% increase).

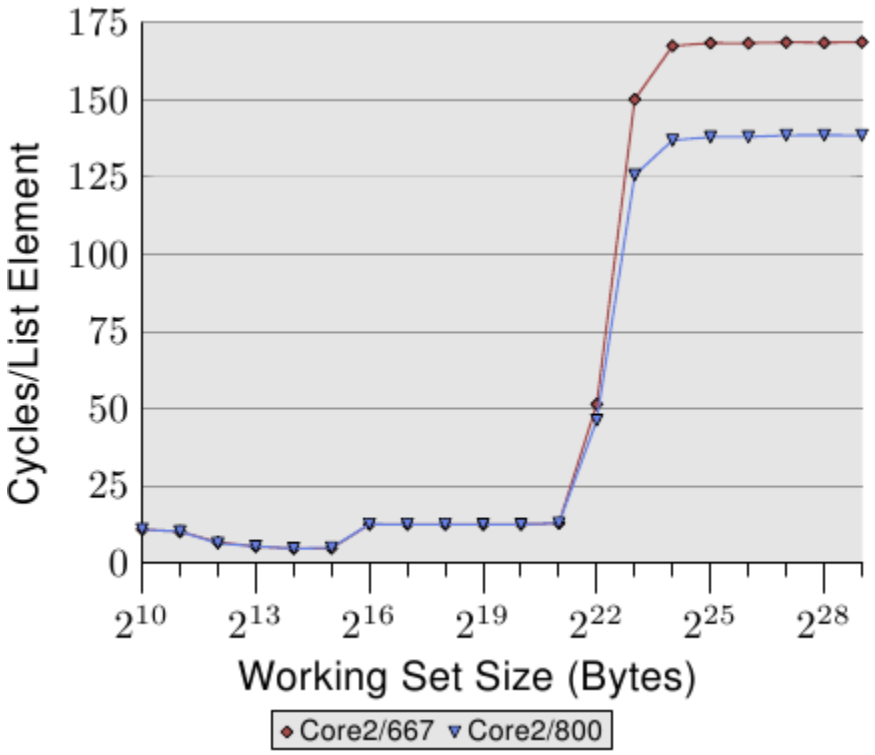


Figure 3.32: Influence of FSB Speed

The numbers show that, when the FSB is really stressed for large working set sizes, we indeed see a large benefit. The maximum performance increase measured in this test is 18.2%, close to the theoretical maximum. What this shows is that a faster FSB indeed can pay off big time. It is not critical when the working set fits into the caches (and these processors have a 4MB L2). It must be kept in mind that we are measuring one program here. The working set of a system comprises the memory needed by all concurrently running processes. This way it is easily possible to exceed 4MB memory or more with much smaller programs.

Today some of Intel's processors support FSB speeds up to 1,333MHz which would mean another 60% increase. The future is going to see even higher speeds. If speed is important and the working set sizes are larger, fast RAM and high FSB speeds are certainly worth the money. One has to be careful, though, since even though the processor might support higher FSB speeds the motherboard/Northbridge might not. It is critical to check the specifications.

[*Editor's note: this the third installment of Ulrich Drepper's "What every programmer should know about memory" document; this section talks about virtual memory, and TLB performance in particular. Those who have not read part 1 and part 2 may wish to do so now. As always, please send typo*

reports and the like to lwn@lwn.net rather than posting them as comments here.]

4 Virtual Memory

The virtual memory subsystem of a processor implements the virtual address spaces provided to each process. This makes each process think it is alone in the system. The list of advantages of virtual memory are described in detail elsewhere so they will not be repeated here. Instead this section concentrates on the actual implementation details of the virtual memory subsystem and the associated costs.

A virtual address space is implemented by the Memory Management Unit (MMU) of the CPU. The OS has to fill out the page table data structures, but most CPUs do the rest of the work themselves. This is actually a pretty complicated mechanism; the best way to understand it is to introduce the data structures used to describe the virtual address space.

The input to the address translation performed by the MMU is a virtual address. There are usually few—if any—restrictions on its value. Virtual addresses are 32-bit values on 32-bit systems, and 64-bit values on 64-bit systems. On some systems, for instance x86 and x86-64, the addresses used actually involve another level of indirection: these architectures use segments which simply cause an offset to be added to every logical address. We can ignore this part of address generation, it is trivial and not something that programmers have to care about with respect to performance of memory handling. {*Segment limits on x86 are performance-relevant but that is another story.*}

4.1 Simplest Address Translation

The interesting part is the translation of the virtual address to a physical address. The MMU can remap addresses on a page-by-page basis. Just as when addressing cache lines, the virtual address is split into distinct parts. These parts are used to index into various tables which are used in the construction of the final physical address. For the simplest model we have only one level of tables.

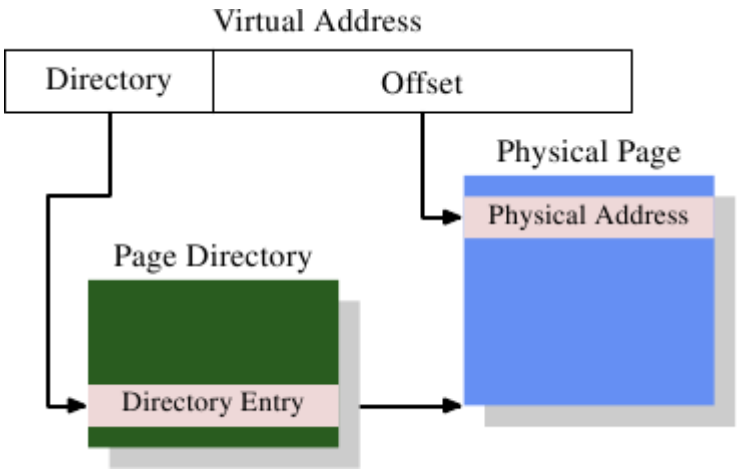


Figure 4.1: 1-Level Address Translation

Figure 4.1 shows how the different parts of the virtual address are used. A top part is used to select an entry in a Page Directory; each entry in that directory can be individually set by the OS. The page directory entry determines the address of a physical memory page; more than one entry in the page directory can point to the same physical address. The complete physical address of the memory cell is determined by combining the page address from the page directory with the low bits from the virtual address. The page directory entry also contains some additional information about the page such as access permissions.

The data structure for the page directory is stored in memory. The OS has to allocate contiguous physical memory and store the base address of this memory region in a special register. The appropriate bits of the virtual address are then used as an index into the page directory, which is actually an array of directory entries.

For a concrete example, this is the layout used for 4MB pages on x86 machines. The Offset part of the virtual address is 22 bits in size, enough to address every byte in a 4MB page. The remaining 10 bits of the virtual address select one of the 1024 entries in the page directory. Each entry contains a 10 bit base address of a 4MB page which is combined with the offset to form a complete 32 bit address.

4.2 Multi-Level Page Tables

4MB pages are not the norm, they would waste a lot of memory since many operations an OS has to perform require alignment to memory pages. With 4kB pages (the norm on 32-bit machines and, still, often on 64-bit machines), the Offset part of the virtual address is only 12 bits in size. This leaves 20 bits as the selector of the page directory. A table with 2^{20} entries is

not practical. Even if each entry would be only 4 bytes the table would be 4MB in size. With each process potentially having its own distinct page directory much of the physical memory of the system would be tied up for these page directories.

The solution is to use multiple levels of page tables. These can then represent a sparse huge page directory where regions which are not actually used do not require allocated memory. The representation is therefore much more compact, making it possible to have the page tables for many processes in memory without impacting performance too much.

Today the most complicated page table structures comprise four levels. Figure 4.2 shows the schematics of such an implementation.

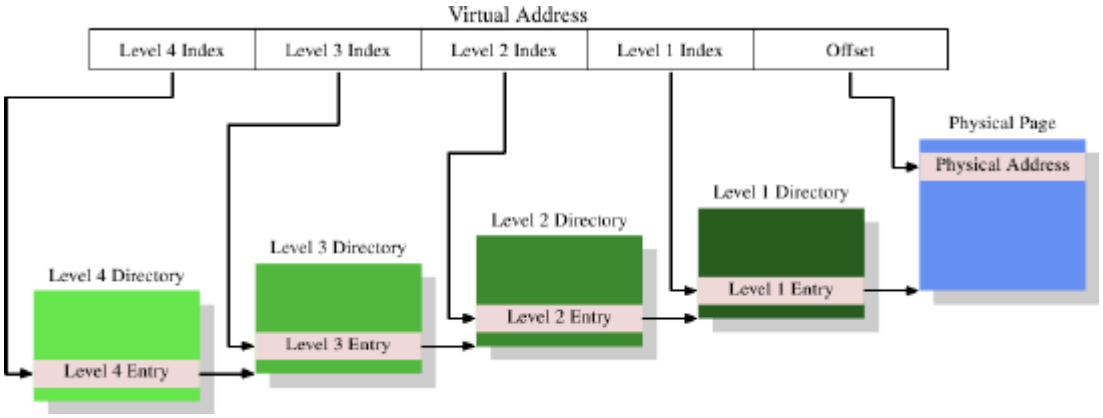


Figure 4.2: 4-Level Address Translation

The virtual address is, in this example, split into at least five parts. Four of these parts are indexes into the various directories. The level 4 directory is referenced using a special-purpose register in the CPU. The content of the level 4 to level 2 directories is a reference to next lower level directory. If a directory entry is marked empty it obviously need not point to any lower directory. This way the page table tree can be sparse and compact. The entries of the level 1 directory are, just like in Figure 4.1, partial physical addresses, plus auxiliary data like access permissions.

To determine the physical address corresponding to a virtual address the processor first determines the address of the highest level directory. This address is usually stored in a register. Then the CPU takes the index part of the virtual address corresponding to this directory and uses that index to pick the appropriate entry. This entry is the address of the next directory, which is indexed using the next part of the virtual address. This process continues until it reaches the level 1 directory, at which point the value of the directory entry is the high part of the physical address. The physical address is completed by adding the page offset bits from the virtual address.

This process is called page tree walking. Some processors (like x86 and x86-64) perform this operation in hardware, others need assistance from the OS.

Each process running on the system might need its own page table tree. It is possible to partially share trees but this is rather the exception. It is therefore good for performance and scalability if the memory needed by the page table trees is as small as possible. The ideal case for this is to place the used memory close together in the virtual address space; the actual physical addresses used do not matter. A small program might get by with using just one directory at each of levels 2, 3, and 4 and a few level 1 directories. On x86-64 with 4kB pages and 512 entries per directory this allows the addressing of 2MB with a total of 4 directories (one for each level). 1GB of contiguous memory can be addressed with one directory for levels 2 to 4 and 512 directories for level 1.

Assuming all memory can be allocated contiguously is too simplistic, though. For flexibility reasons the stack and the heap area of a process are, in most cases, allocated at pretty much opposite ends of the address space. This allows either area to grow as much as possible if needed. This means that there are most likely two level 2 directories needed and correspondingly more lower level directories.

But even this does not always match current practice. For security reasons the various parts of an executable (code, data, heap, stack, DSOs, aka shared libraries) are mapped at randomized addresses [nonselsec]. The randomization extends to the relative position of the various parts; that implies that the various memory regions in use in a process are widespread throughout the virtual address space. By applying some limits to the number of bits of the address which are randomized the range can be restricted, but it certainly, in most cases, will not allow a process to run with just one or two directories for levels 2 and 3.

If performance is really much more important than security, randomization can be turned off. The OS will then usually at least load all DSOs contiguously in virtual memory.

4.3 Optimizing Page Table Access

All the data structures for the page tables are kept in the main memory; this is where the OS constructs and updates the tables. Upon creation of a process or a change of a page table the CPU is notified. The page tables are used to resolve every virtual address into a physical address using the page table walk described above. More to the point: at least one directory for each level is used in the process of resolving a virtual address. This requires up to

four memory accesses (for a single access by the running process) which is slow. It is possible to treat these directory table entries as normal data and cache them in L1d, L2, etc., but this would still be far too slow.

From the earliest days of virtual memory, CPU designers have used a different optimization. A simple computation can show that only keeping the directory table entries in the L1d and higher cache would lead to horrible performance. Each absolute address computation would require a number of L1d accesses corresponding to the page table depth. These accesses cannot be parallelized since they depend on the previous lookup's result. This alone would, on a machine with four page table levels, require at the very least 12 cycles. Add to that the probability of an L1d miss and the result is nothing the instruction pipeline can hide. The additional L1d accesses also steal precious bandwidth to the cache.

So, instead of just caching the directory table entries, the complete computation of the address of the physical page is cached. For the same reason that code and data caches work, such a cached address computation is effective. Since the page offset part of the virtual address does not play any part in the computation of the physical page address, only the rest of the virtual address is used as the tag for the cache. Depending on the page size this means hundreds or thousands of instructions or data objects share the same tag and therefore same physical address prefix.

The cache into which the computed values are stored is called the Translation Look-Aside Buffer (TLB). It is usually a small cache since it has to be extremely fast. Modern CPUs provide multi-level TLB caches, just as for the other caches; the higher-level caches are larger and slower. The small size of the L1TLB is often made up for by making the cache fully associative, with an LRU eviction policy. Recently, this cache has been growing in size and, in the process, was changed to be set associative. As a result, it might not be the oldest entry which gets evicted and replaced whenever a new entry has to be added.

As noted above, the tag used to access the TLB is a part of the virtual address. If the tag has a match in the cache, the final physical address is computed by adding the page offset from the virtual address to the cached value. This is a very fast process; it has to be since the physical address must be available for every instruction using absolute addresses and, in some cases, for L2 look-ups which use the physical address as the key. If the TLB lookup misses the processor has to perform a page table walk; this can be quite costly.

Prefetching code or data through software or hardware could implicitly prefetch entries for the TLB if the address is on another page. This cannot be

allowed for hardware prefetching because the hardware could initiate page table walks that are invalid. Programmers therefore cannot rely on hardware prefetching to prefetch TLB entries. It has to be done explicitly using prefetch instructions. TLBs, just like data and instruction caches, can appear in multiple levels. Just as for the data cache, the TLB usually appears in two flavors: an instruction TLB (ITLB) and a data TLB (DTLB). Higher-level TLBs such as the L2TLB are usually unified, as is the case with the other caches.

4.3.1 Caveats Of Using A TLB

The TLB is a processor-core global resource. All threads and processes executed on the processor core use the same TLB. Since the translation of virtual to physical addresses depends on which page table tree is installed, the CPU cannot blindly reuse the cached entries if the page table is changed. Each process has a different page table tree (but not the threads in the same process) as does the kernel and the VMM (hypervisor) if present. It is also possible that the address space layout of a process changes. There are two ways to deal with this problem:

- The TLB is flushed whenever the page table tree is changed.
- The tags for the TLB entries are extended to additionally and uniquely identify the page table tree they refer to.

In the first case the TLB is flushed whenever a context switch is performed. Since, in most OSes, a switch from one thread/process to another one requires executing some kernel code, TLB flushes are restricted to entering and leaving the kernel address space. On virtualized systems it also happens when the kernel has to call the VMM and on the way back. If the kernel and/or VMM does not have to use virtual addresses, or can reuse the same virtual addresses as the process or kernel which made the system/VMM call, the TLB only has to be flushed if, upon leaving the kernel or VMM, the processor resumes execution of a different process or kernel.

Flushing the TLB is effective but expensive. When executing a system call, for instance, the kernel code might be restricted to a few thousand instructions which touch, perhaps, a handful of new pages (or one huge page, as is the case for Linux on some architectures). This work would replace only as many TLB entries as pages are touched. For Intel's Core2 architecture with its 128 ITLB and 256 DTLB entries, a full flush would mean that more than 100 and 200 entries (respectively) would be flushed unnecessarily. When the system call returns to the same process, all those flushed TLB entries could be used again, but they will be gone. The same is true for often-used code in the kernel or VMM. On each entry into the kernel the TLB has to be filled from scratch even though the page tables for

the kernel and VMM usually do not change and, therefore, TLB entries could, in theory, be preserved for a very long time. This also explains why the TLB caches in today's processors are not bigger: programs most likely will not run long enough to fill all these entries.

This fact, of course, did not escape the CPU architects. One possibility to optimize the cache flushes is to individually invalidate TLB entries. For instance, if the kernel code and data falls into a specific address range, only the pages falling into this address range have to evicted from the TLB. This only requires comparing tags and, therefore, is not very expensive. This method is also useful in case a part of the address space is changed, for instance, through a call to `munmap`.

A much better solution is to extend the tag used for the TLB access. If, in addition to the part of the virtual address, a unique identifier for each page table tree (i.e., a process's address space) is added, the TLB does not have to be completely flushed at all. The kernel, VMM, and the individual processes all can have unique identifiers. The only issue with this scheme is that the number of bits available for the TLB tag is severely limited, while the number of address spaces is not. This means some identifier reuse is necessary. When this happens the TLB has to be partially flushed (if this is possible). All entries with the reused identifier must be flushed but this is, hopefully, a much smaller set.

This extended TLB tagging is of general advantage when multiple processes are running on the system. If the memory use (and hence TLB entry use) of each of the runnable processes is limited, there is a good chance the most recently used TLB entries for a process are still in the TLB when it gets scheduled again. But there are two additional advantages:

1. Special address spaces, such as those used by the kernel and VMM, are often only entered for a short time; afterward control is often returned to the address space which initiated the call. Without tags, two TLB flushes are performed. With tags the calling address space's cached translations are preserved and, since the kernel and VMM address space do not often change TLB entries at all, the translations from previous system calls, etc. can still be used.
2. When switching between two threads of the same process no TLB flush is necessary at all. Without extended TLB tags the entry into the kernel destroys the first thread's TLB entries, though.

Some processors have, for some time, implemented these extended tags. AMD introduced a 1-bit tag extension with the Pacifica virtualization extensions. This 1-bit Address Space ID (ASID) is, in the context of

virtualization, used to distinguish the VMM's address space from that of the guest domains. This allows the OS to avoid flushing the guest's TLB entries every time the VMM is entered (for instance, to handle a page fault) or the VMM's TLB entries when control returns to the guest. The architecture will allow the use of more bits in the future. Other mainstream processors will likely follow suit and support this feature.

4.3.2 Influencing TLB Performance

There are a couple of factors which influence TLB performance. The first is the size of the pages. Obviously, the larger a page is, the more instructions or data objects will fit into it. So a larger page size reduces the overall number of address translations which are needed, meaning that fewer entries in the TLB cache are needed. Most architectures allow the use of multiple different page sizes; some sizes can be used concurrently. For instance, the x86/x86-64 processors have a normal page size of 4kB but they can also use 4MB and 2MB pages respectively. IA-64 and PowerPC allow sizes like 64kB as the base page size.

The use of large page sizes brings some problems with it, though. The memory regions used for the large pages must be contiguous in physical memory. If the unit size for the administration of physical memory is raised to the size of the virtual memory pages, the amount of wasted memory will grow. All kinds of memory operations (like loading executables) require alignment to page boundaries. This means, on average, that each mapping wastes half the page size in physical memory for each mapping. This waste can easily add up; it thus puts an upper limit on the reasonable unit size for physical memory allocation.

It is certainly not practical to increase the unit size to 2MB to accommodate large pages on x86-64. This is just too large a size. But this in turn means that each large page has to be comprised of many smaller pages. And these small pages have to be contiguous in *physical* memory. Allocating 2MB of contiguous physical memory with a unit page size of 4kB can be challenging. It requires finding a free area with 512 contiguous pages. This can be extremely difficult (or impossible) after the system runs for a while and physical memory becomes fragmented.

On Linux it is therefore necessary to pre-allocate these big pages at system start time using the special `hugetlbfs` filesystem. A fixed number of physical pages are reserved for exclusive use as big virtual pages. This ties down resources which might not always be used. It also is a limited pool; increasing it normally means restarting the system. Still, huge pages are the way to go in situations where performance is a premium, resources are

plenty, and cumbersome setup is not a big deterrent. Database servers are an example.

Increasing the minimum virtual page size (as opposed to optional big pages) has its problems, too. Memory mapping operations (loading applications, for example) must conform to these page sizes. No smaller mappings are possible. The location of the various parts of an executable have, for most architectures, a fixed relationship. If the page size is increased beyond what has been taken into account when the executable or DSO was built, the load operation cannot be performed. It is important to keep this limitation in mind. Figure 4.3 shows how the alignment requirements of an ELF binary can be determined. It is encoded in the ELF program header.

```
$ eu-readelf -l /bin/ls
Program Headers:
  Type   Offset   VirtAddr           PhysAddr           FileSiz
MemSiz   Flg Align
...
  LOAD   0x000000 0x0000000000400000 0x0000000000400000 0x0132ac
0x0132ac R E 0x200000
  LOAD   0x0132b0 0x00000000006132b0 0x00000000006132b0 0x001a71
0x001a71 RW  0x200000
...
```

Figure 4.3: ELF Program Header Indicating Alignment Requirements

In this example, an x86-64 binary, the value is
0x200000 = 2,097,152 = 2MB which corresponds to the maximum page size supported by the processor.

There is a second effect of using larger page sizes: the number of levels of the page table tree is reduced. Since the part of the virtual address corresponding to the page offset increases, there are not that many bits left which need to be handled through page directories. This means that, in case of a TLB miss, the amount of work which has to be done is reduced.

Beyond using large page sizes, it is possible to reduce the number of TLB entries needed by moving data which is used at the same time to fewer pages. This is similar to some optimizations for cache use we talked about above. Only now the alignment required is large. Given that the number of TLB entries is quite small this can be an important optimization.

4.4 Impact Of Virtualization

Virtualization of OS images will become more and more prevalent; this means another layer of memory handling is added to the picture. Virtualization of processes (basically jails) or OS containers do not fall into this category since only one OS is involved. Technologies like Xen or KVM enable—with or without help from the processor—the execution of independent OS images. In these situations there is one piece of software alone which directly controls access to the physical memory.

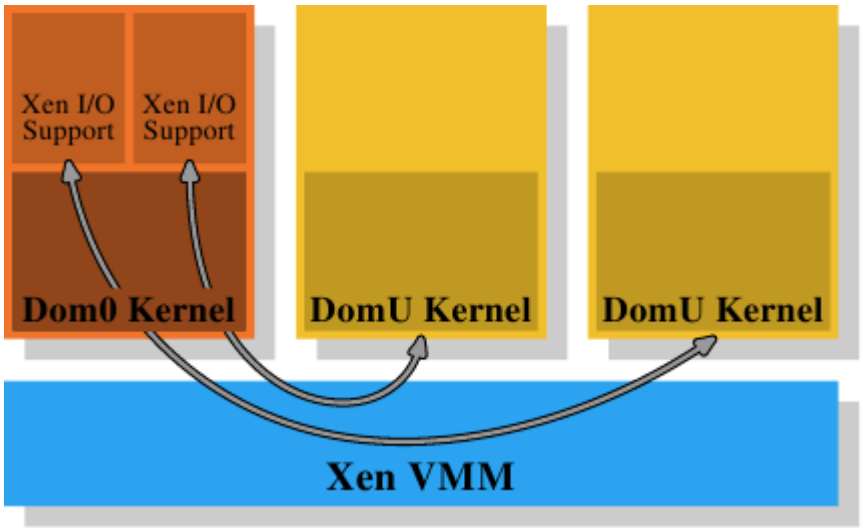


Figure 4.4: Xen Virtualization Model

In the case of Xen (see Figure 4.4) the Xen VMM is that piece of software. The VMM does not implement many of the other hardware controls itself, though. Unlike VMMs on other, earlier systems (and the first release of the Xen VMM) the hardware outside of memory and processors is controlled by the privileged Dom0 domain. Currently, this is basically the same kernel as the unprivileged DomU kernels and, as far as memory handling is concerned, they do not differ. Important here is that the VMM hands out physical memory to the Dom0 and DomU kernels which, themselves, then implement the usual memory handling as if they were running directly on a processor.

To implement the separation of the domains which is required for the virtualization to be complete, the memory handling in the Dom0 and DomU kernels does *not* have unrestricted access to physical memory. The VMM does not hand out memory by giving out individual physical pages and letting the guest OSes handle the addressing; this would not provide any protection against faulty or rogue guest domains. Instead, the VMM creates its own page table tree for each guest domain and hands out memory using these data structures. The good thing is that access to the administrative information of the page table tree can be controlled. If the code does not have appropriate privileges it cannot do anything.

This access control is exploited in the virtualization Xen provides, regardless of whether para- or hardware (aka full) virtualization is used. The guest domains construct their page table trees for each process in a way which is intentionally quite similar for para- and hardware virtualization. Whenever the guest OS modifies its page tables the VMM is invoked. The VMM then uses the updated information in the guest domain to update its own shadow page tables. These are the page tables which are actually used by the hardware. Obviously, this process is quite expensive: each modification of the page table tree requires an invocation of the VMM. While changes to the memory mapping are not cheap without virtualization they become even more expensive now.

The additional costs can be really large, considering that the changes from the guest OS to the VMM and back themselves are already quite expensive. This is why the processors are starting to have additional functionality to avoid the creation of shadow page tables. This is good not only because of speed concerns but it also reduces memory consumption by the VMM. Intel has Extended Page Tables (EPTs) and AMD calls it Nested Page Tables (NPTs). Basically both technologies have the page tables of the guest OSes produce virtual physical addresses. These addresses must then be further translated, using the per-domain EPT/NPT trees, into actual physical addresses. This will allow memory handling at almost the speed of the no-virtualization case since most VMM entries for memory handling are removed. It also reduces the memory use of the VMM since now only one page table tree for each domain (as opposed to process) has to be maintained.

The results of the additional address translation steps are also stored in the TLB. That means the TLB does not store the virtual physical address but, instead, the complete result of the lookup. It was already explained that AMD's Pacifica extension introduced the ASID to avoid TLB flushes on each entry. The number of bits for the ASID is one in the initial release of the processor extensions; this is just enough to differentiate VMM and guest OS. Intel has virtual processor IDs (VPIDs) which serve the same purpose, only there are more of them. But the VPID is fixed for each guest domain and therefore it cannot be used to mark separate processes and avoid TLB flushes at that level, too.

The amount of work needed for each address space modification is one problem with virtualized OSes. There is another problem inherent in VMM-based virtualization, though: there is no way around having two layers of memory handling. But memory handling is hard (especially when taking complications like NUMA into account, see Section 5). The Xen approach of using a separate VMM makes optimal (or even good) handling hard since all the complications of a memory management implementation,

including “trivial” things like discovery of memory regions, must be duplicated in the VMM. The OSes have fully-fledged and optimized implementations; one really wants to avoid duplicating them.

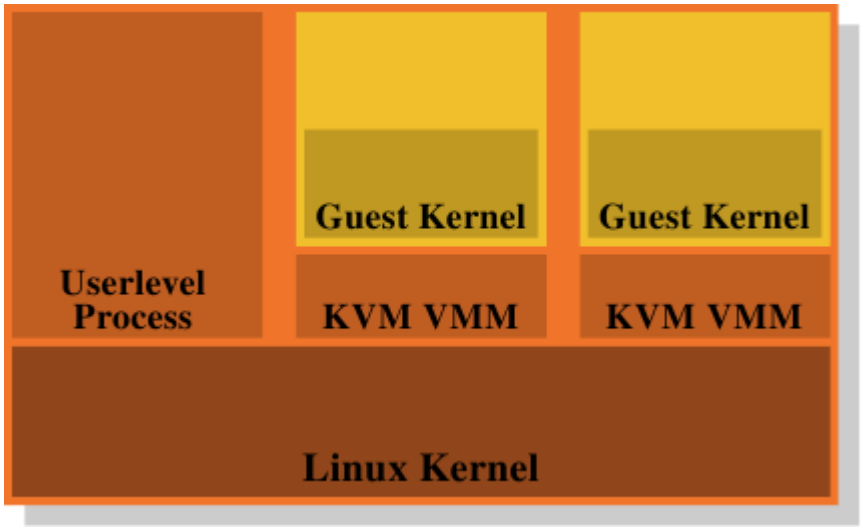


Figure 4.5: KVM Virtualization Model

This is why carrying the VMM/Dom0 model to its conclusion is such an attractive alternative. Figure 4.5 shows how the KVM Linux kernel extensions try to solve the problem. There is no separate VMM running directly on the hardware and controlling all the guests; instead, a normal Linux kernel takes over this functionality. This means the complete and sophisticated memory handling functionality in the Linux kernel is used to manage the memory of the system. Guest domains run alongside the normal user-level processes in what the creators call “guest mode”. The virtualization functionality, para- or full virtualization, is controlled by another user-level process, the KVM VMM. This is just another process which happens to control a guest domain using the special KVM device the kernel implements.

The benefit of this model over the separate VMM of the Xen model is that, even though there are still two memory handlers at work when guest OSes are used, there only needs to be one implementation, that in the Linux kernel. It is not necessary to duplicate the same functionality in another piece of code like the Xen VMM. This leads to less work, fewer bugs, and, perhaps, less friction where the two memory handlers touch since the memory handler in a Linux guest makes the same assumptions as the memory handler in the outer Linux kernel which runs on the bare hardware.

Overall, programmers must be aware that, with virtualization used, the cost of memory operations is even higher than without virtualization. Any optimization which reduces this work will pay off even more in virtualized

environments. Processor designers will, over time, reduce the difference more and more through technologies like EPT and NPT but it will never completely go away.

This article was contributed by Ulrich Drepper

[*Editor's note: welcome to part 5 of Ulrich Drepper's "What every programmer should know about memory". This part is the first half of section 6, which discusses what programmers can do to improve the memory performance of their code. Part 1, part 2, part 3, and part 4 remain online for those who have not yet read them.*]

[*Editor's note: welcome to part 4 of Ulrich Drepper's "What every programmer should know about memory"; this section discusses the particular challenges associated with non-uniform memory access (NUMA) systems. Those who have not read part 1, part 2, and part 3 may wish to do so now. As always, please send typo reports and the like to lwn@lwn.net rather than posting them as comments here.*]

5 NUMA Support

In Section 2 we saw that, on some machines, the cost of access to specific regions of physical memory differs depending on where the access originated. This type of hardware requires special care from the OS and the applications. We will start with a few details of NUMA hardware, then we will cover some of the support the Linux kernel provides for NUMA.

5.1 NUMA Hardware

Non-uniform memory architectures are becoming more and more common. In the simplest form of NUMA, a processor can have local memory (see Figure 2.3) which is cheaper to access than memory local to other processors. The difference in cost for this type of NUMA system is not high, i.e., the NUMA factor is low.

NUMA is also—and especially—used in big machines. We have described the problems of having many processors access the same memory. For commodity hardware all processors would share the same Northbridge (ignoring the AMD Opteron NUMA nodes for now, they have their own problems). This makes the Northbridge a severe bottleneck since *all* memory traffic is routed through it. Big machines can, of course, use custom hardware in place of the Northbridge but, unless the memory chips used have multiple ports—i.e. they can be used from multiple busses—there still is a bottleneck. Multiport RAM is complicated and expensive to build and support and, therefore, it is hardly ever used.

The next step up in complexity is the model AMD uses where an interconnect mechanism (Hypertransport in AMD's case, technology they licensed from Digital) provides access for processors which are not directly connected to the RAM. The size of the structures which can be formed this way is limited unless one wants to increase the diameter (i.e., the maximum distance between any two nodes) arbitrarily.

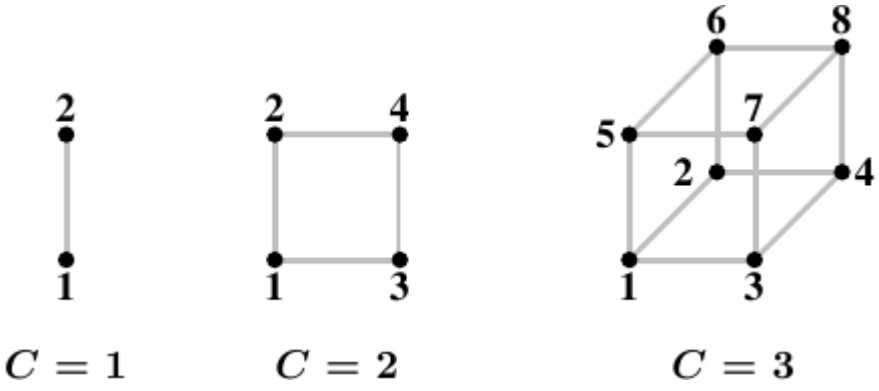


Figure 5.1: Hypercubes

An efficient topology for the nodes is the hypercube, which limits the number of nodes to $\mathbf{2^C}$ where **C** is the number of interconnect interfaces each node has. Hypercubes have the smallest diameter for all systems with $\mathbf{2^n}$ CPUs. Figure 5.1 shows the first three hypercubes. Each hypercube has a diameter of **C** which is the absolute minimum. AMD's first-generation Opteron processors have three hypertransport links per processor. At least one of the processors has to have a Southbridge attached to one link, meaning, currently, that a hypercube with **C=2** can be implemented directly and efficiently. The next generation is announced to have four links, at which point **C=3** hypercubes will be possible.

This does not mean, though, that larger accumulations of processors cannot be supported. There are companies which have developed crossbars allowing larger sets of processors to be used (e.g., Newisys's Horus). But these crossbars increase the NUMA factor and they stop being effective at a certain number of processors.

The next step up means connecting groups of CPUs and implementing a shared memory for all of them. All such systems need specialized hardware and are by no means commodity systems. Such designs exist at several levels of complexity. A system which is still quite close to a commodity machine is IBM x445 and similar machines. They can be bought as ordinary 4U, 8-way machines with x86 and x86-64 processors. Two (at some point up to four) of these machines can then be connected to work as a single machine with shared memory. The interconnect used introduces a

significant NUMA factor which the OS, as well as applications, must take into account.

At the other end of the spectrum, machines like SGI's Altix are designed specifically to be interconnected. SGI's NUMAlink interconnect fabric is very fast and has a low latency; both of these are requirements for high-performance computing (HPC), especially when Message Passing Interfaces (MPI) are used. The drawback is, of course, that such sophistication and specialization is very expensive. They make a reasonably low NUMA factor possible but with the number of CPUs these machines can have (several thousands) and the limited capacity of the interconnects, the NUMA factor is actually dynamic and can reach unacceptable levels depending on the workload.

More commonly used are solutions where clusters of commodity machines are connected using high-speed networking. But these are not NUMA machines; they do not implement a shared address space and therefore do not fall into any category which is discussed here.

5.2 OS Support for NUMA

To support NUMA machines, the OS has to take the distributed nature of the memory into account. For instance, if a process is run on a given processor, the physical RAM assigned to the process's address space should come from local memory. Otherwise each instruction has to access remote memory for code and data. There are special cases to be taken into account which are only present in NUMA machines. The text segment of DSOs is normally present exactly once in a machine's physical RAM. But if the DSO is used by processes and threads on all CPUs (for instance, the basic runtime libraries like `libc`) this means that all but a few processors have to have remote accesses. The OS ideally would “mirror” such DSOs into each processor's physical RAM and use local copies. This is an optimization, not a requirement, and generally hard to implement. It might not be supported or only in a limited fashion.

To avoid making the situation worse, the OS should not migrate a process or thread from one node to another. The OS should already try to avoid migrating processes on normal multi-processor machines because migrating from one processor to another means the cache content is lost. If load distribution requires migrating a process or thread off of a processor, the OS can usually pick an arbitrary new processor which has sufficient capacity left. In NUMA environments the selection of the new processor is a bit more limited. The newly selected processor should not have higher access costs to the memory the process is using than the old processor; this restricts the list

of targets. If there is no free processor matching that criteria available, the OS has no choice but to migrate to a processor where memory access is more expensive.

In this situation there are two possible ways forward. First, one can hope the situation is temporary and the process can be migrated back to a better-suited processor. Alternatively, the OS can also migrate the process's memory to physical pages which are closer to the newly-used processor. This is quite an expensive operation. Possibly huge amounts of memory have to be copied, albeit not necessarily in one step. While this is happening the process, at least briefly, has to be stopped so that modifications to the old pages are correctly migrated. There are a whole list of other requirements for page migration to be efficient and fast. In short, the OS should avoid it unless it is really necessary.

Generally, it cannot be assumed that all processes on a NUMA machine use the same amount of memory such that, with the distribution of processes across the processors, memory usage is also equally distributed. In fact, unless the applications running on the machines are very specific (common in the HPC world, but not outside) the memory use will be very unequal. Some applications will use vast amounts of memory, others hardly any. This will, sooner or later, lead to problems if memory is always allocated local to the processor where the request is originated. The system will eventually run out of memory local to nodes running large processes.

In response to these severe problems, memory is, by default, not allocated exclusively on the local node. To utilize all the system's memory the default strategy is to stripe the memory. This guarantees equal use of all the memory of the system. As a side effect, it becomes possible to freely migrate processes between processors since, on average, the access cost to all the memory used does not change. For small NUMA factors, striping is acceptable but still not optimal (see data in Section 5.4).

This is a pessimization which helps the system avoid severe problems and makes it more predictable under normal operation. But it does decrease overall system performance, in some situations significantly. This is why Linux allows the memory allocation rules to be selected by each process. A process can select a different strategy for itself and its children. We will introduce the interfaces which can be used for this in Section 6.

5.3 Published Information

The kernel publishes, through the `sys` pseudo file system (sysfs),

information about the processor caches below

```
/sys/devices/system/cpu/cpu*/cache
```

In Section 6.2.1 we will see interfaces which can be used to query the size of the various caches. What is important here is the topology of the caches. The directories above contain subdirectories (named `index*`) which list information about the various caches the CPU possesses. The files `type`, `level`, and `shared_cpu_map` are the important files in these directories as far as the topology is concerned. For an Intel Core 2 QX6700 the information looks as in Table 5.1.

		type	level	shared_cpu_map
cpu0	index0	Data	1	00000001
	index1	Instruction	1	00000001
	index2	Unified	2	00000003
cpu1	index0	Data	1	00000002
	index1	Instruction	1	00000002
	index2	Unified	2	00000003
cpu2	index0	Data	1	00000004
	index1	Instruction	1	00000004
	index2	Unified	2	0000000c
cpu3	index0	Data	1	00000008
	index1	Instruction	1	00000008
	index2	Unified	2	0000000c

Table 5.1: `sysfs` Information for Core 2 CPU Caches

What this data means is as follows:

- Each core {*The knowledge that `cpu0` to `cpu3` are cores comes from another place that will be explained shortly.*} has three caches: L1i, L1d, L2.
- The L1d and L1i caches are not shared with anybody—each core has its own set of caches. This is indicated by the bitmap in `shared_cpu_map`having only one set bit.

- The L2 cache on `cpu0` and `cpu1` is shared, as is the L2 on `cpu2` and `cpu3`.

If the CPU had more cache levels, there would be more `index*` directories.

For a four-socket, dual-core Opteron machine the cache information looks like Table 5.2:

		type	level	shared_cpu_map
cpu0	index0	Data	1	00000001
	index1	Instruction	1	00000001
	index2	Unified	2	00000001
cpu1	index0	Data	1	00000002
	index1	Instruction	1	00000002
	index2	Unified	2	00000002
cpu2	index0	Data	1	00000004
	index1	Instruction	1	00000004
	index2	Unified	2	00000004
cpu3	index0	Data	1	00000008
	index1	Instruction	1	00000008
	index2	Unified	2	00000008
cpu4	index0	Data	1	00000010
	index1	Instruction	1	00000010
	index2	Unified	2	00000010
cpu5	index0	Data	1	00000020
	index1	Instruction	1	00000020
	index2	Unified	2	00000020
cpu6	index0	Data	1	00000040
	index1	Instruction	1	00000040
	index2	Unified	2	00000040
cpu7	index0	Data	1	00000080
	index1	Instruction	1	00000080
	index2	Unified	2	00000080

Table 5.2: `sysfs` Information for Opteron CPU Caches

As can be seen these processors also have three caches: L1i, L1d, L2. None of the cores shares any level of cache. The interesting part for this system is the processor topology. Without this additional information one cannot make sense of the cache data. The `sys` file system exposes this information in the files below

```
/sys/devices/system/cpu/cpu*/topology
```

Table 5.3 shows the interesting files in this hierarchy for the SMP Opteron machine.

	physical_ package_id	core_id	core_ siblings	thread_ siblings
cpu0	0	0	00000003	00000001
cpu1		1	00000003	00000002
cpu2	1	0	0000000c	00000004
cpu3		1	0000000c	00000008
cpu4	2	0	00000030	00000010
cpu5		1	00000030	00000020
cpu6	3	0	000000c0	00000040
cpu7		1	000000c0	00000080

Table 5.3: `sysfs` Information for Opteron CPU Topology

Taking Table 5.2 and Table 5.3 together we can see that no CPU has hyper-threads (the `thread_siblings` bitmaps have one bit set), that the system in fact has four processors (`physical_package_id` 0 to 3), that each processor has two cores, and that none of the cores share any cache. This is exactly what corresponds to earlier Opterons.

What is completely missing in the data provided so far is information about the nature of NUMA on this machine. Any SMP Opteron machine is a NUMA machine. For this data we have to look at yet another part of the `sys` file system which exists on NUMA machines, namely in the hierarchy below

```
/sys/devices/system/node
```

This directory contains a subdirectory for every NUMA node on the system. In the node-specific directories there are a number of files. The important files and their content for the Opteron machine described in the previous two tables are shown in Table 5.4.

	cpumap	distance			
node0	00000003	10	20	20	20
node1	0000000c	20	10	20	20
node2	00000030	20	20	10	20
node3	000000c0	20	20	20	10

Table 5.4: `sysfs` Information for Opteron Nodes

This information ties all the rest together; now we have a complete picture of the architecture of the machine. We already know that the machine has four processors. Each processor constitutes its own node as can be seen by the bits set in the value in `cpumap` file in the `node*`directories.

The `distance` files in those directories contains a set of values, one for each node, which represent a cost of memory accesses at the respective nodes. In this example all local memory accesses have the cost 10, all remote access to any other node has the cost 20. {*This is, by the way, incorrect. The ACPI information is apparently wrong since, although the processors used have three coherent HyperTransport links, at least one processor must be connected to a Southbridge. At least one pair of nodes must therefore have a larger distance.*} This means that, even though the processors are organized as a two-dimensional hypercube (see Figure 5.1), accesses between processors which are not directly connected is not more expensive. The relative values of the costs should be usable as an estimate of the actual difference of the access times. The accuracy of all this information is another question.

5.4 Remote Access Costs

The distance is relevant, though. In [amdccnuma] AMD documents the NUMA cost of a four socket machine. For write operations the numbers are shown in Figure 5.3.

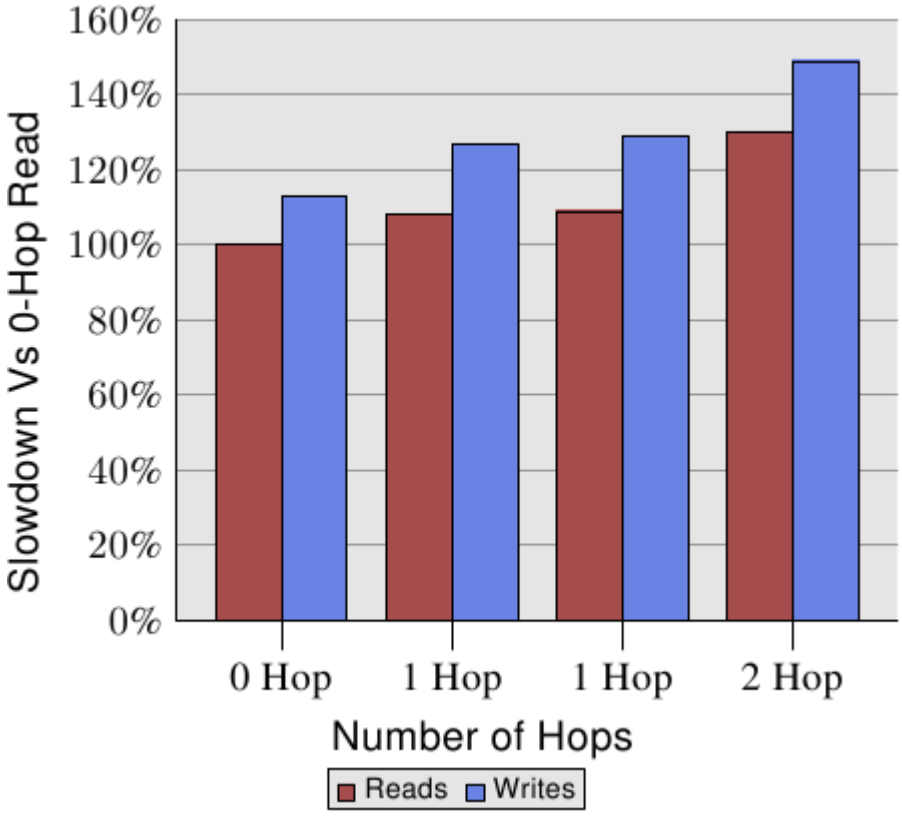


Figure 5.3: Read/Write Performance with Multiple Nodes

Writes are slower than reads, this is no surprise. The interesting parts are the costs of the 1- and 2-hop cases. The two 1-hop cases actually have slightly different costs. See [amdccnuma] for the details. The fact we need to remember from this chart is that 2-hop reads and writes are 30% and 49% (respectively) slower than 0-hop reads. 2-hop writes are 32% slower than 0-hop writes, and 17% slower than 1-hop writes. The relative position of processor and memory nodes can make a big difference. The next generation of processors from AMD will feature four coherent HyperTransport links per processor. In that case a four socket machine would have diameter of one. With eight sockets the same problem returns, with a vengeance, since the diameter of a hypercube with eight nodes is three.

All this information is available but it is cumbersome to use. In Section 6.5 we will see an interface which helps accessing and using this information easier.

The last piece of information the system provides is in the status of a process itself. It is possible to determine how the memory-mapped files, Copy-On-Write (COW) pages and anonymous memory are distributed over the nodes in the system. {*Copy-On-Write is a method often used in OS implementations when a memory page has one user at first and then has to*

be copied to allow independent users. In many situations the copying is unnecessary, at all or at first, in which case it makes sense to only copy when either user modifies the memory. The operating system intercepts the write operation, duplicates the memory page, and then allows the write instruction to proceed.} Each process has a file`/proc/PID/numa_maps`,

where `PID` is the ID of the process, as shown in Figure 5.2.

```
00400000 default file=/bin/cat mapped=3 N3=3
00504000 default file=/bin/cat anon=1 dirty=1 mapped=2 N3=2
00506000 default heap anon=3 dirty=3 active=0 N3=3
38a9000000 default file=/lib64/ld-2.4.so mapped=22 mapmax=47 N1=22
38a9119000 default file=/lib64/ld-2.4.so anon=1 dirty=1 N3=1
38a911a000 default file=/lib64/ld-2.4.so anon=1 dirty=1 N3=1
38a9200000 default file=/lib64/libc-2.4.so mapped=53 mapmax=52 N1=51
N2=2
38a933f000 default file=/lib64/libc-2.4.so
38a943f000 default file=/lib64/libc-2.4.so anon=1 dirty=1 mapped=3
mapmax=32 N1=2 N3=1
38a9443000 default file=/lib64/libc-2.4.so anon=1 dirty=1 N3=1
38a9444000 default anon=4 dirty=4 active=0 N3=4
2b2bbcdce000 default anon=1 dirty=1 N3=1
2b2bbcde4000 default anon=2 dirty=2 N3=2
2b2bbcde6000 default file=/usr/lib/locale/locale-archive mapped=11
mapmax=8 N0=11
7fffedcc7000 default stack anon=2 dirty=2 N3=2
```

Figure 5.2: Content of `/proc/`*`PID`*`/numa_maps`

The important information in the file is the values for `N0` to `N3`, which indicate the number of pages allocated for the memory area on nodes 0 to 3. It is a good guess that the program was executed on a core on node 3. The program itself and the dirtied pages are allocated on that node. Read-only mappings, such as the first mapping for `ld-2.4.so` and `libc-2.4.so` as well as the shared file `locale-archive` are allocated on other nodes.

As we have seen in Figure 5.3 the read performance across nodes falls by 9% and 30% respectively for 1- and 2-hop reads. For execution, such reads are needed and, if the L2 cache is missed, each cache line incurs these additional costs. All the costs measured for large workloads beyond the size

of the cache would have to be increased by 9%/30% if the memory is remote to the processor.

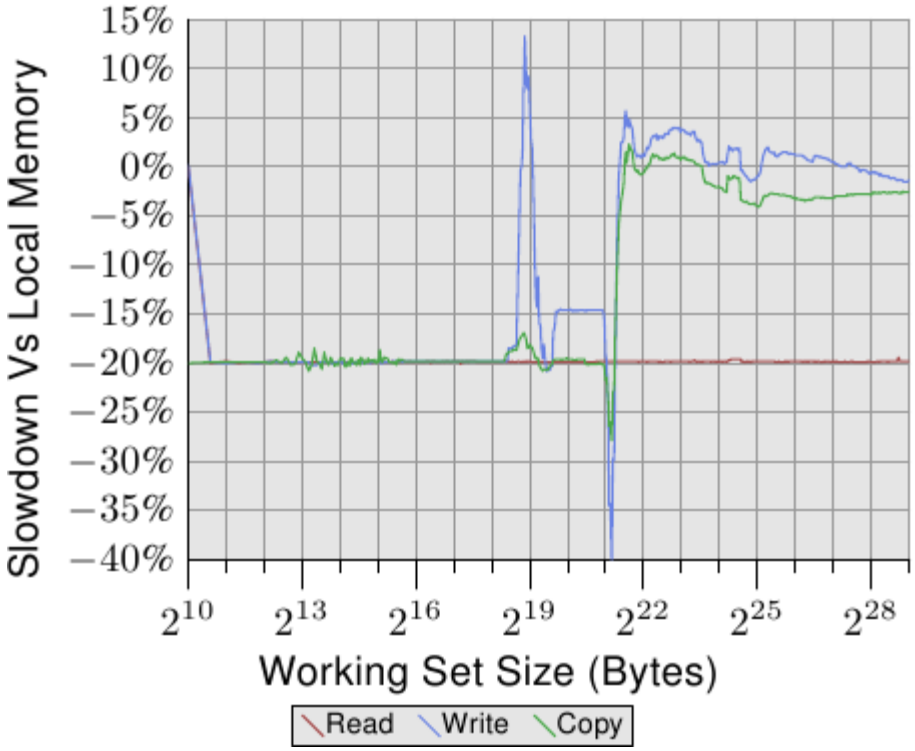


Figure 5.4: Operating on Remote Memory

To see the effects in the real world we can measure the bandwidth as in Section 3.5.1 but this time with the memory being on a remote node, one hop away. The result of this test when compared with the data for using local memory can be seen in Figure 5.4. The numbers have a few big spikes in both directions which are the result of a problem of measuring multi-threaded code and can be ignored. The important information in this graph is that read operations are always 20% slower. This is significantly slower than the 9% in Figure 5.3, which is, most likely, not a number for uninterrupted read/write operations and might refer to older processor revisions. Only AMD knows.

For working set sizes which fit into the caches, the performance of write and copy operations is also 20% slower. For working sets exceeding the size of the caches, the write performance is not measurably slower than the operation on the local node. The speed of the interconnect is fast enough to keep up with the memory. The dominating factor is the time spent waiting on the main memory.

6 What Programmers Can Do

After the descriptions in the previous sections it is clear that there are many, many opportunities for programmers to influence a program's performance, positively or negatively. And this is for memory-related operations only. We will proceed in covering the opportunities from the ground up, starting with the lowest levels of physical RAM access and L1 caches, up to and including OS functionality which influences memory handling.

6.1 Bypassing the Cache

When data is produced and not (immediately) consumed again, the fact that memory store operations read a full cache line first and then modify the cached data is detrimental to performance. This operation pushes data out of the caches which might be needed again in favor of data which will not be used soon. This is especially true for large data structures, like matrices, which are filled and then used later. Before the last element of the matrix is filled the sheer size evicts the first elements, making caching of the writes ineffective.

For this and similar situations, processors provide support for *non-temporal* write operations. Non-temporal in this context means the data will not be reused soon, so there is no reason to cache it. These non-temporal write operations do not read a cache line and then modify it; instead, the new content is directly written to memory.

This might sound expensive but it does not have to be. The processor will try to use write-combining (see Section 3.3.3) to fill entire cache lines. If this succeeds no memory read operation is needed at all. For the x86 and x86-64 architectures a number of intrinsics are provided by gcc:

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);

#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);

#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

These instructions are used most efficiently if they process large amounts of data in one go. Data is loaded from memory, processed in one or more steps, and then written back to memory. The data “streams” through the processor, hence the names of the intrinsics.

The memory address must be aligned to 8 or 16 bytes respectively. In code using the multimedia extensions it is possible to replace the normal `_mm_store_*` intrinsics with these non-temporal versions. In the matrix multiplication code in Section 9.1 we do not do this since the written values are reused in a short order of time. This is an example where using the stream instructions is not useful. More on this code in Section 6.2.1.

The processor's write-combining buffer can hold requests for partial writing to a cache line for only so long. It is generally necessary to issue all the instructions which modify a single cache line one after another so that the write-combining can actually take place. An example for how to do this is as follows:

```
#include <emmintrin.h>
void setbytes(char *p, int c)
{
  __m128i i = _mm_set_epi8(c, c, c, c,
                           c, c, c, c,
                           c, c, c, c,
                           c, c, c, c);
  _mm_stream_si128((__m128i *)&p[0], i);
  _mm_stream_si128((__m128i *)&p[16], i);
  _mm_stream_si128((__m128i *)&p[32], i);
  _mm_stream_si128((__m128i *)&p[48], i);
}
```

Assuming the pointer `p` is appropriately aligned, a call to this function will set all bytes of the addressed cache line to `c`. The write-combining logic will see the four generated `movntdq` instructions and only issue the write command for the memory once the last instruction has been executed. To summarize, this code sequence not only avoids reading the cache line before it is written, it also avoids polluting the cache with data which might not be needed soon. This can have huge benefits in certain situations. An example of everyday code using this technique is the `memset` function in the C runtime, which should use a code sequence like the above for large blocks.

Some architectures provide specialized solutions. The PowerPC architecture defines the `dcbz` instruction which can be used to clear an entire cache line.

The instruction does not really bypass the cache since a cache line is allocated for the result, but no data is read from memory. It is more limited than the non-temporal store instructions since a cache line can only be set to all-zeros and it pollutes the cache (in case the data is non-temporal), but no write-combining logic is needed to achieve the results.

To see the non-temporal instructions in action we will look at a new test which is used to measure writing to a matrix, organized as a two-dimensional array. The compiler lays out the matrix in memory so that the leftmost (first) index addresses the row which has all elements laid out sequentially in memory. The right (second) index addresses the elements in a row. The test program iterates over the matrix in two ways: first by increasing the column number in the inner loop and then by increasing the row index in the inner loop. This means we get the behavior shown in Figure 6.1.

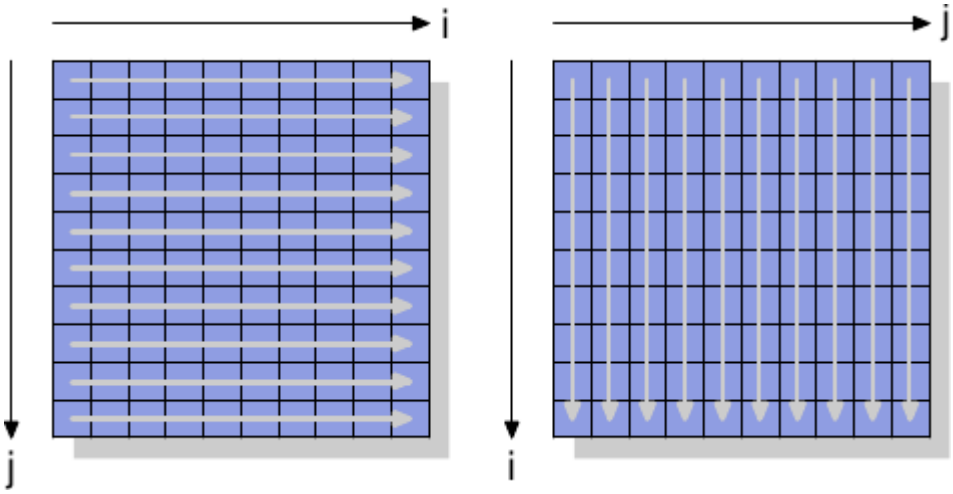


Figure 6.1: Matrix Access Pattern

We measure the time it takes to initialize a 3000×3000 matrix. To see how memory behaves, we use store instructions which do not use the cache. On IA-32 processors the “non-temporal hint” is used for this. For comparison we also measure ordinary store operations. The results can be seen in Table 6.1.

	Inner Loop Increment	
	Row	Column
Normal	0.048s	0.127s
Non-Temporal	0.048s	0.160s

Table 6.1: Timing Matrix Initialization

For the normal writes which do use the cache we see the expected result: if memory is used sequentially we get a much better result, 0.048s for the whole operation translating to about 750MB/s, compared to the more-or-less random access which takes 0.127s (about 280MB/s). The matrix is large enough that the caches are essentially ineffective.

The part we are mainly interested in here are the writes bypassing the cache. It might be surprising that the sequential access is just as fast here as in the case where the cache is used. The reason for this behavior is that the processor is performing write-combining as explained above. Additionally, the memory ordering rules for non-temporal writes are relaxed: the program needs to explicitly insert memory barriers (`sfence` instructions for the x86 and x86-64 processors). This means the processor has more freedom to write back the data and thereby using the available bandwidth as well as possible.

In the case of column-wise access in the inner loop the situation is different. The results are significantly slower than in the case of cached accesses (0.16s, about 225MB/s). Here we can see that no write combining is possible and each memory cell must be addressed individually. This requires constantly selecting new rows in the RAM chips with all the associated delays. The result is a 25% worse result than the cached run.

On the read side, processors, until recently, lacked support aside from weak hints using non-temporal access (NTA) prefetch instructions. There is no equivalent to write-combining for reads, which is especially bad for uncacheable memory such as memory-mapped I/O. Intel, with the SSE4.1 extensions, introduced NTA loads. They are implemented using a small number of streaming load buffers; each buffer contains a cache line. The first `movntdqa` instruction for a given cache line will load a cache line into a buffer, possibly replacing another cache line. Subsequent 16-byte aligned accesses to the same cache line will be serviced from the load buffer at little cost. Unless there are other reasons to do so, the cache line will not be loaded into a cache, thus enabling the loading of large amounts of memory without polluting the caches. The compiler provides an intrinsic for this instruction:

```
#include <smmintrin.h>
__m128i _mm_stream_load_si128 (__m128i *p);
```

This intrinsic should be used multiple times, with addresses of 16-byte blocks passed as the parameter, until each cache line is read. Only then should the next cache line be started. Since there are a few streaming read buffers it might be possible to read from two memory locations at once.

What we should take away from this experiment is that modern CPUs very nicely optimize uncached write and (more recently) read accesses as long as they are sequential. This knowledge can come in very handy when handling large data structures which are used only once. Second, caches can help to cover up some—but not all—of the costs of random memory access. Random access in this example is 70% slower due to the implementation of RAM access. Until the implementation changes, random accesses should be avoided whenever possible.

In the section about prefetching we will again take a look at the non-temporal flag.

6.2 Cache Access

The most important improvements a programmer can make with respect to caches are those which affect the level 1 cache. We will discuss it first before including the other levels. Obviously, all the optimizations for the level 1 cache also affect the other caches. The theme for all memory access is the same: improve locality (spatial and temporal) and align the code and data.

6.2.1 Optimizing Level 1 Data Cache Access

In section Section 3.3 we have already seen how much the effective use of the L1d cache can improve performance. In this section we will show what kinds of code changes can help to improve that performance. Continuing from the previous section, we first concentrate on optimizations to access memory sequentially. As seen in the numbers of Section 3.3, the processor automatically prefetches data when memory is accessed sequentially.

The example code used is a matrix multiplication. We use two square matrices of **1000×1000** `double` elements. For those who have forgotten the math, given two matrices **A** and **B** with elements $\mathbf{a_{ij}}$ and $\mathbf{b_{ij}}$ with $\mathbf{0 \le i,j < N}$ the product is

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b_{kj} = a_{i1} b_{1j} + a_{i2} b_{2j} + \cdots + a_{i(N-1)} b_{(N-1)j}$$

A straight-forward C implementation of this can look like this:

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * mul2[k][j];
```

The two input matrices are `mul1` and `mul2`. The result matrix `res` is assumed to be initialized to all zeroes. It is a nice and simple implementation. But it should be obvious that we have exactly the problem explained in Figure 6.1. While `mul1` is accessed sequentially, the inner loop advances the row number of `mul2`. That means that `mul1` is handled like the left matrix in Figure 6.1 while `mul2` is handled like the right matrix. This cannot be good.

There is one possible remedy one can easily try. Since each element in the matrices is accessed multiple times it might be worthwhile to rearrange ("transpose," in mathematical terms) the second matrix `mul2` before using it.

$$(AB)_{ij} = \sum_{k=0}^{N-1} a_{ik} b^{\mathrm{T}}_{jk} = a_{i1} b^{\mathrm{T}}_{j1} + a_{i2} b^{\mathrm{T}}_{j2} + \cdots + a_{i(N-1)} b^{\mathrm{T}}_{j(N-1)}$$

After the transposition (traditionally indicated by a superscript 'T') we now iterate over both matrices sequentially. As far as the C code is concerned, it now looks like this:

```
double tmp[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    tmp[i][j] = mul2[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * tmp[j][k];
```

We create a temporary variable to contain the transposed matrix. This requires touching more memory, but this cost is, hopefully, recovered since the 1000 non-sequential accesses per column are more expensive (at least on

modern hardware). Time for some performance tests. The results on a Intel Core 2 with 2666MHz clock speed are (in clock cycles):

	Original	Transposed
Cycles	16,765,297,870	3,922,373,010
Relative	100%	23.4%

Through the simple transformation of the matrix we can achieve a 76.6% speed-up! The copy operation is more than made up. The 1000 non-sequential accesses really hurt.

The next question is whether this is the best we can do. We certainly need an alternative method anyway which does not require the additional copy. We will not always have the luxury to be able to perform the copy: the matrix can be too large or the available memory too small.

The search for an alternative implementation should start with a close examination of the math involved and the operations performed by the original implementation. Trivial math knowledge allows us to see that the order in which the additions for each element of the result matrix are performed is irrelevant as long as each addend appears exactly once. {*We ignore arithmetic effects here which might change the occurrence of overflows, underflows, or rounding.*} This understanding allows us to look for solutions which reorder the additions performed in the inner loop of the original code.

Now let us examine the actual problem in the execution of the original code.

The order in which the elements of `mul2` are accessed is: (0,0), (1,0), …,

(N-1,0), (0,1), (1,1), …. The elements (0,0) and (0,1) are in the same cache line but, by the time the inner loop completes one round, this cache line has long been evicted. For this example, each round of the inner loop requires, for each of the three matrices, 1000 cache lines (with 64 bytes for the Core 2 processor). This adds up to much more than the 32k of L1d available.

But what if we handle two iterations of the middle loop together while

executing the inner loop? In this case we use two `double` values from the

cache line which is guaranteed to be in L1d. We cut the L1d miss rate in half. That is certainly an improvement, but, depending on the cache line size, it still might not be as good as we can get it. The Core 2 processor has a L1d cache line size of 64 bytes. The actual value can be queried using

```
sysconf (_SC_LEVEL1_DCACHE_LINESIZE)
```

at runtime or using the `getconf` utility from the command line so that the program can be compiled for a specific cache line size.

With`sizeof(double)` being 8 this means that, to fully utilize the cache line, we should unroll the middle loop 8 times. Continuing this thought, to effectively use the `res` matrix as well, i.e., to write 8 results at the same time, we should unroll the outer loop 8 times as well. We assume here cache lines of size 64 but the code works also well on systems with 32-byte cache lines since both cache lines are also 100% utilized. In general it is best to hardcode cache line sizes at compile time by using the `getconf` utility as in:

```
gcc -DCLS=$(getconf LEVEL1_DCACHE_LINESIZE) ...
```

If the binaries are supposed to be generic, the largest cache line size should be used. With very small L1ds this might mean that not all the data fits into the cache but such processors are not suitable for high-performance programs anyway. The code we arrive at looks something like this:

```
#define SM (CLS / sizeof (double))

for (i = 0; i < N; i += SM)
    for (j = 0; j < N; j += SM)
        for (k = 0; k < N; k += SM)
            for (i2 = 0, rres = &res[i][j],
                 rmul1 = &mul1[i][k]; i2 < SM;
                 ++i2, rres += N, rmul1 += N)
                for (k2 = 0, rmul2 = &mul2[k][j];
                     k2 < SM; ++k2, rmul2 += N)
                    for (j2 = 0; j2 < SM; ++j2)
                        rres[j2] += rmul1[k2] * rmul2[j2];
```

This looks quite scary. To some extent it is but only because it incorporates some tricks. The most visible change is that we now have six nested loops. The outer loops iterate with intervals of `SM` (the cache line size divided by `sizeof(double)`). This divides the multiplication in several smaller problems which can be handled with more cache locality. The inner loops iterate over the missing indexes of the outer loops. There are, once again, three loops. The only tricky part here is that the `k2` and `j2` loops are in a

different order. This is done since, in the actual computation, only one expression depends on `k2` but two depend on `j2`.

The rest of the complication here results from the fact that gcc is not very smart when it comes to optimizing array indexing. The introduction of the additional variables `rres`, `rmul1`, and `rmul2` optimizes the code by pulling common expressions out of the inner loops, as far down as possible. The default aliasing rules of the C and C++ languages do not help the compiler making these decisions (unless `restrict` is used, all pointer accesses are potential sources of aliasing). This is why Fortran is still a preferred language for numeric programming: it makes writing fast code easier. {*In theory the `restrict` keyword introduced into the C language in the 1999 revision should solve the problem. Compilers have not caught up yet, though. The reason is mainly that too much incorrect code exists which would mislead the compiler and cause it to generate incorrect object code.*}

How all this work pays off can be seen in Table 6.2.

	Original	Transposed	Sub-Matrix	Vectorized
Cycles	16,765,297,870	3,922,373,010	2,895,041,480	1,588,711,750
Relative	100%	23.4%	17.3%	9.47%

Table 6.2: Matrix Multiplication Timing

By avoiding the copying we gain another 6.1% of performance. Plus, we do not need any additional memory. The input matrices can be arbitrarily large as long as the result matrix fits into memory as well. This is a requirement for a general solution which we have now achieved.

There is one more column in Table 6.2 which has not been explained. Most modern processors nowadays include special support for vectorization. Often branded as multi-media extensions, these special instructions allow processing of 2, 4, 8, or more values at the same time. These are often SIMD (Single Instruction, Multiple Data) operations, augmented by others to get the data in the right form. The SSE2 instructions provided by Intel processors can handle two `double` values in one operation. The instruction reference manual lists the intrinsic functions which provide access to these SSE2 instructions. If these intrinsics are used the program runs another 7.3% (relative to the original) faster. The result is a program which runs in 10% of

the time of the original code. Translated into numbers which people recognize, we went from 318 MFLOPS to 3.35 GFLOPS. Since we are here only interested in memory effects here, the program code is pushed out into Section 9.1.

It should be noted that, in the last version of the code, we still have some cache problems with `mul2`; prefetching still will not work. But this cannot be solved without transposing the matrix. Maybe the cache prefetching units will get smarter to recognize the patterns, then no additional change would be needed. 3.19 GFLOPS on a 2.66 GHz processor with single-threaded code is not bad, though.

What we optimized in the example of the matrix multiplication is the use of the loaded cache lines. All bytes of a cache line are always used. We just made sure they are used before the cache line is evacuated. This is certainly a special case.

It is much more common to have data structures which fill one or more cache lines where the program uses only a few members at any one time. In Figure 3.11 we have already seen the effects of large structure sizes if only few members are used.

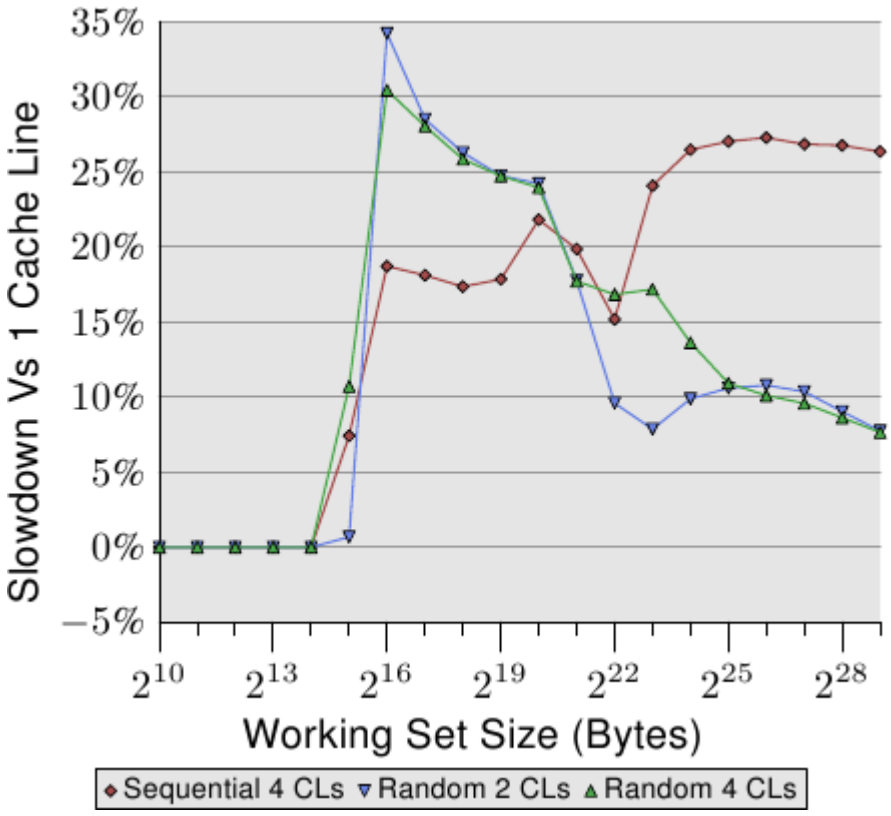


Figure 6.2: Spreading Over Multiple Cache Lines

Figure 6.2 shows the results of yet another set of benchmarks performed using the by now well-known program. This time two values of the same list element are added. In one case, both elements are in the same cache line; in the other case, one element is in the first cache line of the list element and the second is in the last cache line. The graph shows the slowdown we are experiencing.

Unsurprisingly, in all cases there are no negative effects if the working set fits into L1d. Once L1d is no longer sufficient, penalties are paid by using two cache lines in the process instead of one. The red line shows the data when the list is laid out sequentially in memory. We see the usual two step patterns: about 17% penalty when the L2 cache is sufficient and about 27% penalty when the main memory has to be used.

In the case of random memory accesses the relative data looks a bit different. The slowdown for working sets which fit into L2 is between 25% and 35%. Beyond that it goes down to about 10%. This is not because the penalties get smaller but, instead, because the actual memory accesses get disproportionally more costly. The data also shows that, in some cases, the distance between the elements does matter. The Random 4 CLs curve shows higher penalties because the first and fourth cache lines are used.

An easy way to see the layout of a data structure compared to cache lines is to use the pahole program (see [dwarves]). This program examines the data structures defined in a binary. Take a program containing this definition:

```
struct foo {
  int a;
  long fill[7];
  int b;
};
```

Compiled on a 64-bit machine, the output of pahole contains (among other things) the information shown in Figure 6.3.

```
struct foo {
        int                     a;                    /*     0
4 */

        /* XXX 4 bytes hole, try to pack */

        long int                fill[7];              /*     8
56 */
        /* --- cacheline 1 boundary (64 bytes) --- */
```

```
        int                     b;                      /*    64
4 */
}; /* size: 72, cachelines: 2 */
   /* sum members: 64, holes: 1, sum holes: 4 */
   /* padding: 4 */
   /* last cacheline: 8 bytes */
```

Figure 6.3: Output of pahole Run

This output tells us a lot. First, it shows that the data structure uses up more than one cache line. The tool assumes the currently-used processor's cache line size, but this value can be overridden using a command line parameter. Especially in cases where the size of the structure is barely over the limit of a cache line, and many objects of this type are allocated, it makes sense to seek a way to compress that structure. Maybe a few elements can have a smaller type, or maybe some fields are actually flags which can be represented using individual bits.

In the case of the example the compression is easy and it is hinted at by the program. The output shows that there is a hole of four bytes after the first element. This hole is caused by the alignment requirement of the structure and the `fill` element. It is easy to see that the element `b`, which has a size of four bytes (indicated by the 4 at the end of the line), fits perfectly into the gap. The result in this case is that the gap no longer exists and that the data structure fits onto one cache line. The pahole tool can perform this optimization itself. If the `—reorganize` parameter is used and the structure name is added at the end of the command line the output of the tool is the optimized structure and the cache line use. Besides moving elements to fill gaps, the tool can also optimize bit fields and combine padding and holes. For more details see [dwarves].

Having a hole which is just large enough for the trailing element is, of course, the ideal situation. For this optimization to be useful it is required that the object itself is aligned to a cache line. We get to that in a bit.

The pahole output also makes it easier to determine whether elements have to be reordered so that those elements which are used together are also stored together. Using the pahole tool, it is easily possible to determine which elements are in the same cache line and when, instead, the elements have to be reshuffled to achieve that. This is not an automatic process but the tool can help quite a bit.

The position of the individual structure elements and the way they are used is important, too. As we have seen in Section 3.5.2 the performance of code with the critical word late in the cache line is worse. This means a programmer should always follow the following two rules:

1. Always move the structure element which is most likely to be the critical word to the beginning of the structure.
2. When accessing the data structures, and the order of access is not dictated by the situation, access the elements in the order in which they are defined in the structure.

For small structures, this means that the programmer should arrange the elements in the order in which they are likely accessed. This must be handled in a flexible way to allow the other optimizations, such as filling holes, to be applied as well. For bigger data structures each cache line-sized block should be arranged to follow the rules.

Reordering elements is not worth the time it takes, though, if the object itself is not aligned. The alignment of an object is determined by the alignment requirement of the data type. Each fundamental type has its own alignment requirement. For structured types the largest alignment requirement of any of its elements determines the alignment of the structure. This is almost always smaller than the cache line size. This means even if the members of a structure are lined up to fit into the same cache line an allocated object might not have an alignment matching the cache line size. There are two ways to ensure that the object has the alignment which was used when designing the layout of the structure:

- the object can be allocated with an explicit alignment requirement.

  For dynamic allocation a call to `malloc` would only allocate the object with an alignment matching that of the most demanding standard type (usually `long double`). It is possible to use `posix_memalign`, though, to request higher alignments.

  ```
  #include <stdlib.h>
  int posix_memalign(void **memptr,
                     size_t align,
                     size_t size);
  ```

  The function stores the pointer to the newly-allocated memory in the pointer variable pointed to by `memptr`. The memory block is `size` bytes in size and is aligned on a `align`-byte boundary.

For objects allocated by the compiler (in .data, .bss, etc, and on the stack) a variable attribute can be used:

```
struct strtype variable
   __attribute((aligned(64)));
```

In this case the variable is aligned at a 64 byte boundary regardless of the alignment requirement of the strtype structure. This works for global variables as well as automatic variables.

This method does not work for arrays, though. Only the first element of the array would be aligned unless the size of each array element is a multiple of the alignment value. It also means that every single variable must be annotated appropriately. The use of posix_memalign is also not entirely free since the alignment requirements usually lead to fragmentation and/or higher memory consumption.

- the alignment requirement of a type can be changed using a type attribute:
  - `struct strtype {`
  - `  ...members...`
  - `} __attribute((aligned(64)));`

  This will cause the compiler to allocate all objects with the appropriate alignment, including arrays. The programmer has to take care of requesting the appropriate alignment for dynamically allocated objects, though. Here once again posix_memalign must be used. It is easy enough to use the alignof operator gcc provides and pass the value as the second parameter to posix_memalign.

The multimedia extensions previously mentioned in this section almost always require that the memory accesses are aligned. I.e., for 16 byte memory accesses the address is supposed to be 16 byte aligned. The x86 and x86-64 processors have special variants of the memory operations which can handle unaligned accesses but these are slower. This hard alignment requirement is nothing new for most RISC architectures which require full alignment for all memory accesses. Even if an architecture supports

unaligned accesses this is sometimes slower than using appropriate alignment, especially if the misalignment causes a load or store to use two cache lines instead of one.

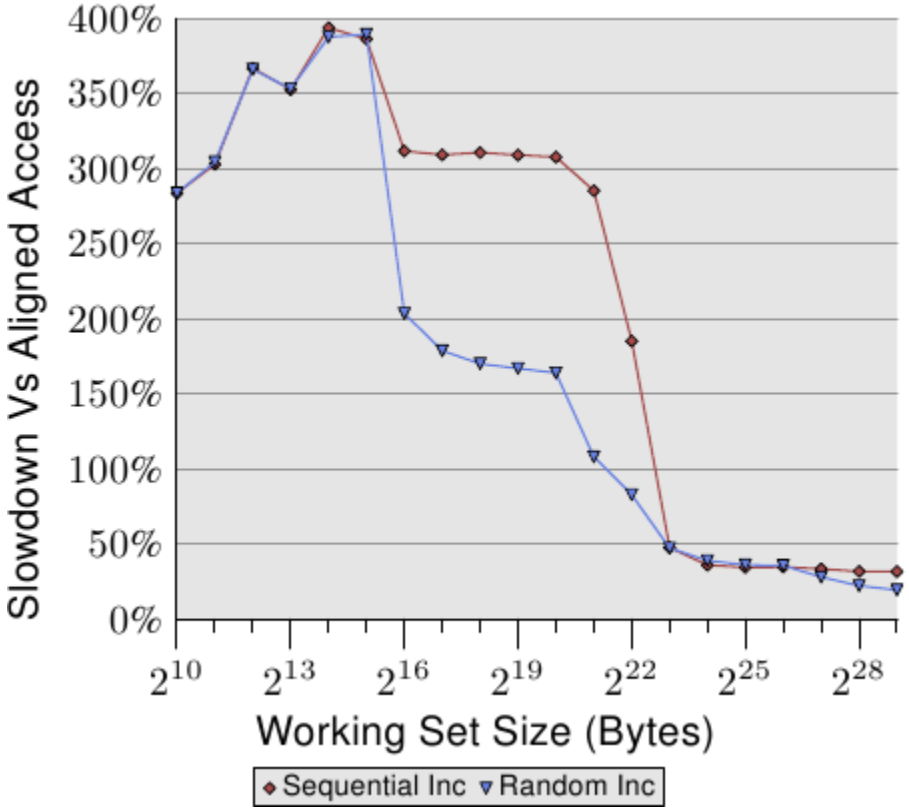


Figure 6.4: Overhead of Unaligned Accesses

Figure 6.4 shows the effects of unaligned memory accesses. The now well-known tests which increment a data element while visiting memory (sequentially or randomly) are measured, once with aligned list elements and once with deliberately misaligned elements. The graph shows the slowdown the program incurs because of the unaligned accesses. The effects are more dramatic for the sequential access case than for the random case because, in the latter case, the costs of unaligned accesses are partially hidden by the generally higher costs of the memory access. In the sequential case, for working set sizes which do fit into the L2 cache, the slowdown is about 300%. This can be explained by the reduced effectiveness of the L1 cache. Some increment operations now touch two cache lines, and beginning work on a list element now often requires reading of two cache lines. The connection between L1 and L2 is simply too congested.

For very large working set sizes, the effects of the unaligned access are still 20% to 30%—which is a lot given that the aligned access time for those sizes is long. This graph should show that alignment must be taken seriously.

Even if the architecture supports unaligned accesses, this must not be taken as “they are as good as aligned accesses”.

There is some fallout from these alignment requirements, though. If an automatic variable has an alignment requirement, the compiler has to ensure that it is met in all situations. This is not trivial since the compiler has no control over the call sites and the way they handle the stack. This problem can be handled in two ways:

1. The generated code actively aligns the stack, inserting gaps if necessary. This requires code to check for alignment, create alignment, and later undo the alignment.
2. Require that all callers have the stack aligned.

All of the commonly used application binary interfaces (ABIs) follow the second route. Programs will likely fail if a caller violates the rule and alignment is needed in the callee. Keeping alignment intact does not come for free, though.

The size of a stack frame used in a function is not necessarily a multiple of the alignment. This means padding is needed if other functions are called from this stack frame. The big difference is that the stack frame size is, in most cases, known to the compiler and, therefore, it knows how to adjust the stack pointer to ensure alignment for any function which is called from that stack frame. In fact, most compilers will simply round the stack frame size up and be done with it.

This simple way to handle alignment is not possible if variable length arrays (VLAs) or `alloca` are used. In that case, the total size of the stack frame is only known at runtime. Active alignment control might be needed in this case, making the generated code (slightly) slower.

On some architectures, only the multimedia extensions require strict alignment; stacks on those architectures are always minimally aligned for the normal data types, usually 4 or 8 byte alignment for 32- and 64-bit architectures respectively. On these systems, enforcing the alignment incurs unnecessary costs. That means that, in this case, we might want to get rid of the strict alignment requirement if we know that it is never depended upon. Tail functions (those which call no other functions) which do no multimedia operations do not need alignment. Neither do functions which only call functions which need no alignment. If a large enough set of functions can be identified, a program might want to relax the alignment requirement. For x86 binaries gcc has support for relaxed stack alignment requirements:

```
-mpreferred-stack-boundary=2
```

If this option is given a value of N, the stack alignment requirement will be set to 2^N bytes. So, if a value of 2 is used, the stack alignment requirement is reduced from the default (which is 16 bytes) to just 4 bytes. In most cases this means no additional alignment operation is needed since normal stack push and pop operations work on four-byte boundaries anyway. This machine-specific option can help to reduce code size and also improve execution speed. But it cannot be applied for many other architectures. Even for x86-64 it is generally not applicable since the x86-64 ABI requires that floating-point parameters are passed in an SSE register and the SSE instructions require full 16 byte alignment. Nevertheless, whenever the option is usable it can make a noticeable difference.

Efficient placement of structure elements and alignment are not the only aspects of data structures which influence cache efficiency. If an array of structures is used, the entire structure definition affects performance. Remember the results in Figure 3.11: in this case we had increasing amounts of unused data in the elements of the array. The result was that prefetching was increasingly less effective and the program, for large data sets, became less efficient.

For large working sets it is important to use the available cache as well as possible. To achieve this, it might be necessary to rearrange data structures. While it is easier for the programmer to put all the data which conceptually belongs together in the same data structure, this might not be the best approach for maximum performance. Assume we have a data structure as follows:

```
struct order {
  double price;
  bool paid;
  const char *buyer[5];
  long buyer_id;
};
```

Further assume that these records are stored in a big array and that a frequently-run job adds up the expected payments of all the outstanding bills.

In this scenario, the memory used for the `buyer` and `buyer_id` fields is

unnecessarily loaded into the caches. Judging from the data in Figure 3.11 the program will perform up to 5 times worse than it could.

It is much better to split the `order` data structure in two, storing the first two fields in one structure and the other fields elsewhere. This change certainly increases the complexity of the program, but the performance gains might justify this cost.

Finally, let's consider another cache use optimization which, while also applying to the other caches, is primarily felt in the L1d access. As seen in Figure 3.8 an increased associativity of the cache benefits normal operation. The larger the cache, the higher the associativity usually is. The L1d cache is too large to be fully associative but not large enough to have the same associativity as L2 caches. This can be a problem if many of the objects in the working set fall into the same cache set. If this leads to evictions due to overuse of a set, the program can experience delays even though much of the cache is unused. These cache misses are sometimes called *conflict misses*. Since the L1d addressing uses virtual addresses, this is actually something the programmer can have control over. If variables which are used together are also stored together the likelihood of them falling into the same set is minimized. Figure 6.5 shows how quickly the problem can hit.

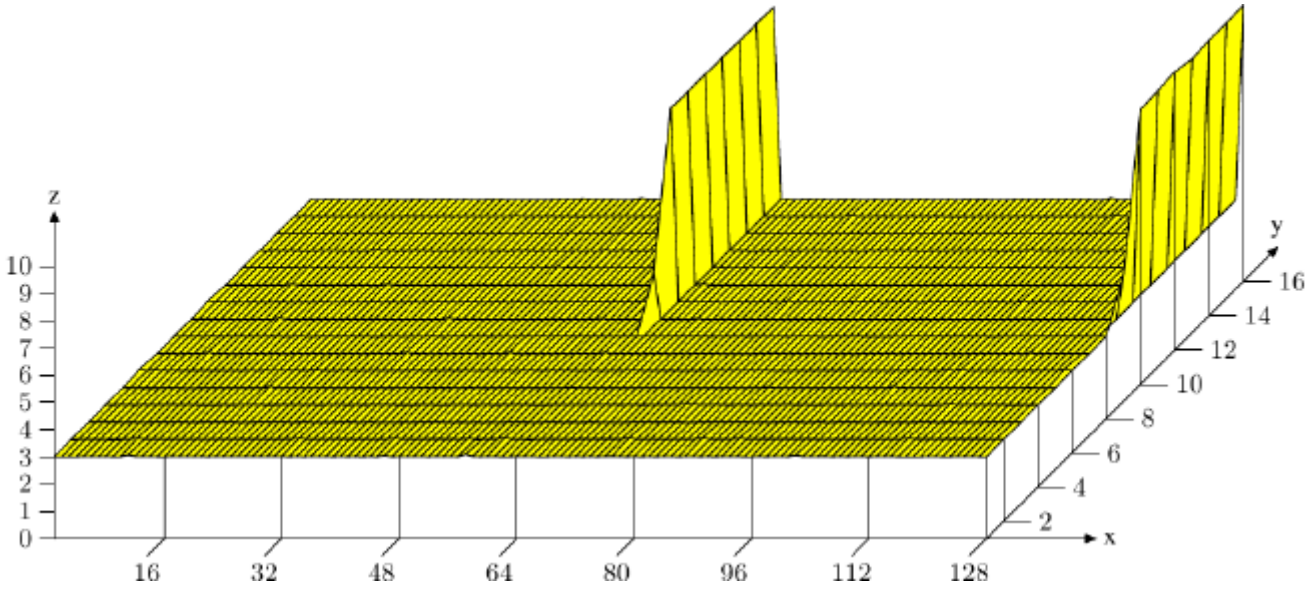


Figure 6.5: Cache Associativity Effects

In the figure, the now familiar Follow {*The test was performed on a 32-bit machine, hence `NPAD`=15 means one 64-byte cache line per list element.*}

with `NPAD`=15 test is measured with a special setup. The X–axis is the distance between two list elements, measured in empty list elements. In other words, a distance of 2 means that the next element's address is 128 bytes after the previous one. All elements are laid out in the virtual address

space with the same distance. The Y–axis shows the total length of the list. Only one to 16 elements are used, meaning that the total working set size is 64 to 1024 bytes. The z–axis shows the average number of cycles needed to traverse each list element.

The result shown in the figure should not be surprising. If few elements are used, all the data fits into L1d and the access time is only 3 cycles per list element. The same is true for almost all arrangements of the list elements: the virtual addresses are nicely mapped to L1d slots with almost no conflicts. There are two (in this graph) special distance values for which the situation is different. If the distance is a multiple of 4096 bytes (i.e., distance of 64 elements) and the length of the list is greater than eight, the average number of cycles per list element increases dramatically. In these situations all entries are in the same set and, once the list length is greater than the associativity, entries are flushed from L1d and have to be re-read from L2 the next round. This results in the cost of about 10 cycles per list element.

With this graph we can determine that the processor used has an L1d cache with associativity 8 and a total size of 32kB. That means that the test could, if necessary, be used to determine these values. The same effects can be measured for the L2 cache but, here, it is more complex since the L2 cache is indexed using physical addresses and it is much larger.

For programmers this means that associativity is something worth paying attention to. Laying out data at boundaries that are powers of two happens often enough in the real world, but this is exactly the situation which can easily lead to the above effects and degraded performance. Unaligned accesses can increase the probability of conflict misses since each access might require an additional cache line.

Figure 6.6: Bank Address of L1d on AMD

If this optimization is performed, another related optimization is possible, too. AMD's processors, at least, implement the L1d as several individual banks. The L1d can receive two data words per cycle but only if both words are stored in different banks or in a bank with the same index. The bank address is encoded in the low bits of the virtual address as shown in Figure 6.6. If variables which are used together are also stored together the likelihood that they are in different banks or the same bank with the same index is high.

6.2.2 Optimizing Level 1 Instruction Cache Access

Preparing code for good L1i use needs similar techniques as good L1d use. The problem is, though, that the programmer usually does not directly influence the way L1i is used unless s/he writes code in assembler. If compilers are used, programmers can indirectly determine the L1i use by guiding the compiler to create a better code layout.

Code has the advantage that it is linear between jumps. In these periods the processor can prefetch memory efficiently. Jumps disturb this nice picture because

- the jump target might not be statically determined;
- and even if it is static the memory fetch might take a long time if it misses all caches.

These problems create stalls in execution with a possibly severe impact on performance. This is why today's processors invest heavily in branch prediction (BP). Highly specialized BP units try to determine the target of a jump as far ahead of the jump as possible so that the processor can initiate loading the instructions at the new location into the cache. They use static and dynamic rules and are increasingly good at determining patterns in execution.

Getting data into the cache as soon as possible is even more important for the instruction cache. As mentioned in Section 3.1, instructions have to be decoded before they can be executed and, to speed this up (important on x86 and x86-64), instructions are actually cached in the decoded form, not in the byte/word form read from memory.

To achieve the best L1i use programmers should look out for at least the following aspects of code generation:

1. reduce the code footprint as much as possible. This has to be balanced with optimizations like loop unrolling and inlining.
2. code execution should be linear without bubbles. {*Bubbles describe graphically the holes in the execution in the pipeline of a processor which appear when the execution has to wait for resources. For more details the reader is referred to literature on processor design.*}
3. aligning code when it makes sense.

We will now look at some compiler techniques available to help with optimizing programs according to these aspects.

Compilers have options to enable levels of optimization; specific optimizations can also be individually enabled. Many of the optimizations enabled at high optimization levels (-O2 and -O3 for gcc) deal with loop

optimizations and function inlining. In general, these are good optimizations. If the code which is optimized in these ways accounts for a significant part of the total execution time of the program, overall performance can be improved. Inlining of functions, in particular, allows the compiler to optimize larger chunks of code at a time which, in turn, enables the generation of machine code which better exploits the processor's pipeline architecture. The handling of both code and data (through dead code elimination or value range propagation, and others) works better when larger parts of the program can be considered as a single unit.

A larger code size means higher pressure on the L1i (and also L2 and higher level) caches. This *can* lead to less performance. Smaller code can be faster. Fortunately gcc has an optimization option to specify this. If -Os is used the compiler will optimize for code size. Optimizations which are known to increase the code size are disabled. Using this option often produces surprising results. Especially if the compiler cannot really take advantage of loop unrolling and inlining, this option is a big win.

Inlining can be controlled individually as well. The compiler has heuristics and limits which guide inlining; these limits can be controlled by the programmer. The -finline-limit option specifies how large a function must be to be considered too large for inlining. If a function is called in multiple places, inlining it in all of them would cause an explosion in the code size.

But there is more. Assume a function `inlcand` is called in two

functions `f1` and `f2`. The functions `f1` and `f2` are themselves called in

sequence.

With inlining	Without inlining
start f1	start inlcand
code f1	code inlcand
inlined inlcand	end inlcand
more code f1	
end f1	start f1
	code f1
start f2	end f1
code f2	
inlined inlcand	start f2
more code f2	code f2
end f2	end f2

Table 6.3: Inlining Vs Not

Table 6.3 shows how the generated code could look like in the cases of no inline and inlining in both functions. If the function `inlcand` is inlined in both `f1` and `f2` the total size of the generated code is:

size `f1` + size `f2` + 2 × size `inlcand`

If no inlining happens, the total size is smaller by `size inlcand`. This is how much more L1i and L2 cache is needed if `f1` and `f2` are called shortly after one another. Plus: if `inlcand` is not inlined, the code might still be in L1i and it will not have to be decoded again. Plus: the branch prediction unit might do a better job of predicting jumps since it has already seen the code. If the compiler default for the upper limit on the size of inlined functions is not the best for the program, it should be lowered.

There are cases, though, when inlining always makes sense. If a function is only called once it might as well be inlined. This gives the compiler the opportunity to perform more optimizations (like value range propagation, which might significantly improve the code). That inlining might be thwarted by the selection limits. gcc has, for cases like this, an option to specify that a function is always inlined. Adding the `always_inline` function attribute instructs the compiler to do exactly what the name suggests.

In the same context, if a function should never be inlined despite being small enough, the `noinline` function attribute can be used. Using this attribute makes sense even for small functions if they are called often from different places. If the L1i content can be reused and the overall footprint is reduced this often makes up for the additional cost of the extra function call. Branch prediction units are pretty good these days. If inlining can lead to more aggressive optimizations things look different. This is something which must be decided on a case-by-case basis.

The `always_inline` attribute works well if the inline code is always used. But what if this is not the case? What if the inlined function is called only occasionally:

```
void fct(void) {
```

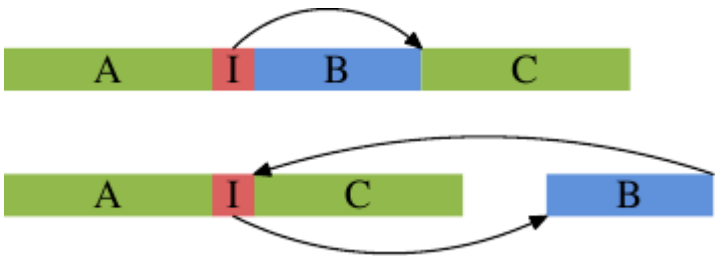
```
  ... code block A ...
 if (condition)
   inlfct()
 ... code block C ...
}
```

The code generated for such a code sequence in general matches the structure of the sources. That means first comes the code block A, then a conditional jump which, if the condition evaluates to false, jumps forward.

The code generated for the inlined `inlfct` comes next, and finally the code block C. This looks all reasonable but it has a problem.

If the `condition` is frequently false, the execution is not linear. There is a big chunk of unused code in the middle which not only pollutes the L1i due to prefetching, it also can cause problems with branch prediction. If the branch prediction is wrong the conditional expression can be very inefficient.

This is a general problem and not specific to inlining functions. Whenever conditional execution is used and it is lopsided (i.e., the expression far more often leads to one result than the other) there is the potential for false static branch prediction and thus bubbles in the pipeline. This can be prevented by telling the compiler to move the less often executed code out of the main code path. In that case the conditional branch generated for an `if` statement would jump to a place out of the order as can be seen in the following figure.



The upper parts represents the simple code layout. If the area B, e.g. generated from the inlined function `inlfct` above, is often not executed because the conditional I jumps over it, the prefetching of the processor will pull in cache lines containing block B which are rarely used. Using block reordering this can be changed, with a result that can be seen in the lower part of the figure. The often-executed code is linear in memory while the rarely-executed code is moved somewhere where it does not hurt prefetching and L1i efficiency.

gcc provides two methods to achieve this. First, the compiler can take profiling output into account while recompiling code and lay out the code blocks according to the profile. We will see how this works in Section 7. The second method is through explicit branch prediction. gcc recognizes `__builtin_expect`:

```
long __builtin_expect(long EXP, long C);
```

This construct tells the compiler that the expression `EXP` most likely will have the value `C`. The return value is `EXP`. `__builtin_expect` is meant to be used in an conditional expression. In almost all cases will it be used in the context of boolean expressions in which case it is much more convenient to define two helper macros:

```
#define unlikely(expr) __builtin_expect(!!(expr), 0)
#define likely(expr) __builtin_expect(!!(expr), 1)
```

These macros can then be used as in

```
if (likely(a > 1))
```

If the programmer makes use of these macros and then uses the `-freorder-blocks` optimization option gcc will reorder blocks as in the figure above. This option is enabled with `-O2` but disabled for `-Os`. There is another option to reorder block (`-freorder-blocks-and-partition`) but it has limited usefulness because it does not work with exception handling.

There is another big advantage of small loops, at least on certain processors. The Intel Core 2 front end has a special feature called Loop Stream Detector (LSD). If a loop has no more than 18 instructions (none of which is a call to a subroutine), requires only up to 4 decoder fetches of 16 bytes, has at most 4 branch instructions, and is executed more than 64 times, than the loop is sometimes locked in the instruction queue and therefore more quickly available when the loop is used again. This applies, for instance, to small inner loops which are entered many times through an outer loop. Even without such specialized hardware compact loops have advantages.

Inlining is not the only aspect of optimization with respect to L1i. Another aspect is alignment, just as for data. There are obvious differences: code is a

mostly linear blob which cannot be placed arbitrarily in the address space and it cannot be influenced directly by the programmer as the compiler generates the code. There are some aspects which the programmer can control, though.

Aligning each single instruction does not make any sense. The goal is to have the instruction stream be sequential. So alignment only makes sense in strategic places. To decide where to add alignments it is necessary to understand what the advantages can be. Having an instruction at the beginning of a cache line {*For some processors cache lines are not the atomic blocks for instructions. The Intel Core 2 front end issues 16 byte blocks to the decoder. They are appropriately aligned and so no issued block can span a cache line boundary. Aligning at the beginning of a cache line still has advantages since it optimizes the positive effects of prefetching.*} means that the prefetch of the cache line is maximized. For instructions this also means the decoder is more effective. It is easy to see that, if an instruction at the end of a cache line is executed, the processor has to get ready to read a new cache line and decode the instructions. There are things which can go wrong (such as cache line misses), meaning that an instruction at the end of the cache line is, on average, not as effectively executed as one at the beginning.

Combine this with the follow-up deduction that the problem is most severe if control was just transferred to the instruction in question (and hence prefetching is not effective) and we arrive at our final conclusion where alignment of code is most useful:

- at the beginning of functions;
- at the beginning of basic blocks which are reached only through jumps;
- to some extent, at the beginning of loops.

In the first two cases the alignment comes at little cost. Execution proceeds at a new location and, if we choose it to be at the beginning of a cache line, we optimize prefetching and decoding. {*For instruction decoding processors often use a smaller unit than cache lines, 16 bytes in case of x86 and x86-64.*} The compiler accomplishes this alignment through the insertion of a series of no-op instructions to fill the gap created by aligning the code. This “dead code” takes a little space but does not normally hurt performance.

The third case is slightly different: aligning beginning of each loop might create performance problems. The problem is that beginning of a loop often follows other code sequentially. If the circumstances are not very lucky there will be a gap between the previous instruction and the aligned

beginning of the loop. Unlike in the previous two cases, this gap cannot be completely dead. After execution of the previous instruction the first instruction in the loop must be executed. This means that, following the previous instruction, there either must be a number of no-op instructions to fill the gap or there must be an unconditional jump to the beginning of the loop. Neither possibility is free. Especially if the loop itself is not executed often, the no-ops or the jump might cost more than one saves by aligning the loop.

There are three ways the programmer can influence the alignment of code. Obviously, if the code is written in assembler the function and all instructions in it can be explicitly aligned. The assembler provides for all architectures the `.align` pseudo-op to do that. For high-level languages the compiler must be told about alignment requirements. Unlike for data types and variables this is not possible in the source code. Instead a compiler option is used:

```
-falign-functions=N
```

This option instructs the compiler to align all functions to the next power-of-two boundary greater than `N`. That means a gap of up to `N` bytes is created. For small functions using a large value for `N` is a waste. Equally for code which is executed only rarely. The latter can happen a lot in libraries which can contain both popular and not-so-popular interfaces. A wise choice of the option value can speed things up or save memory by avoiding alignment. All alignment is turned off by using one as the value of `N` or by using the `-fno-align-functions` option.

The alignment for the second case above—beginning of basic blocks which are not reached sequentially—can be controlled with a different option:

```
-falign-jumps=N
```

All the other details are equivalent, the same warning about waste of memory applies.

The third case also has its own option:

```
-falign-loops=N
```

Yet again, the same details and warnings apply. Except that here, as explained before, alignment comes at a runtime cost since either no-ops or a jump instruction has to be executed if the aligned address is reached sequentially.

gcc knows about one more option for controlling alignment which is mentioned here only for completeness. `-falign-labels` aligns every single label in the code (basically the beginning of each basic block). This, in all but a few exceptional cases, slows down the code and therefore should not be used.

6.2.3 Optimizing Level 2 and Higher Cache Access

Everything said about optimizations for using level 1 cache also applies to level 2 and higher cache accesses. There are two additional aspects of last level caches:

- cache misses are always very expensive. While L1 misses (hopefully) frequently hit L2 and higher cache, thus limiting the penalties, there is obviously no fallback for the last level cache.
- L2 caches and higher are often shared by multiple cores and/or hyper-threads. The effective cache size available to each execution unit is therefore usually less than the total cache size.

To avoid the high costs of cache misses, the working set size should be matched to the cache size. If data is only needed once this obviously is not necessary since the cache would be ineffective anyway. We are talking about workloads where the data set is needed more than once. In such a case the use of a working set which is too large to fit into the cache will create large amounts of cache misses which, even with prefetching being performed successfully, will slow down the program.

A program has to perform its job even if the data set is too large. It is the programmer's job to do the work in a way which minimizes cache misses. For last-level caches this is possible—just as for L1 caches—by working on the job in smaller pieces. This is very similar to the optimized matrix multiplication on Table 6.2. One difference, though, is that, for last level caches, the data blocks which are be worked on can be bigger. The code becomes yet more complicated if L1 optimizations are needed, too. Imagine a matrix multiplication where the data sets—the two input matrices and the output matrix—do not fit into the last level cache together. In this case it might be appropriate to optimize the L1 and last level cache accesses at the same time.

The L1 cache line size is usually constant over many processor generations; even if it is not, the differences will be small. It is no big problem to just assume the larger size. On processors with smaller cache sizes two or more cache lines will then be used instead of one. In any case, it is reasonable to hardcode the cache line size and optimize the code for it.

For higher level caches this is not the case if the program is supposed to be generic. The sizes of those caches can vary widely. Factors of eight or more are not uncommon. It is not possible to assume the larger cache size as a default since this would mean the code performs poorly on all machines except those with the biggest cache. The opposite choice is bad too: assuming the smallest cache means throwing away 87% of the cache or more. This is bad; as we can see from Figure 3.14 using large caches can have a huge impact on the program's speed.

What this means is that the code must dynamically adjust itself to the cache line size. This is an optimization specific to the program. All we can say here is that the programmer should compute the program's requirements correctly. Not only are the data sets themselves needed, the higher level caches are also used for other purposes; for example, all the executed instructions are loaded from cache. If library functions are used this cache usage might add up to a significant amount. Those library functions might also need data of their own which further reduces the available memory.

Once we have a formula for the memory requirement we can compare it with the cache size. As mentioned before, the cache might be shared with multiple other cores. Currently {*There definitely will sometime soon be a better way!*} the only way to get correct information without hardcoding knowledge is through the `/sys` filesystem. In Table 5.2 we have seen the what the kernel publishes about the hardware. A program has to find the directory:

```
/sys/devices/system/cpu/cpu*/cache
```

for the last level cache. This can be recognized by the highest numeric value in the `level` file in that directory. When the directory is identified the program should read the content of the `size` file in that directory and divide the numeric value by the number of bits set in the bitmask in the file `shared_cpu_map`.

The value which is computed this way is a safe lower limit. Sometimes a program knows a bit more about the behavior of other threads or processes. If those threads are scheduled on a core or hyper-thread sharing the cache, and the cache usage is known to not exhaust its fraction of the total cache size, then the computed limit might be too low to be optimal. Whether more than the fair share should be used really depends on the situation. The programmer has to make a choice or has to allow the user to make a decision.

6.2.4 Optimizing TLB Usage

There are two kinds of optimization of TLB usage. The first optimization is to reduce the number of pages a program has to use. This automatically results in fewer TLB misses. The second optimization is to make the TLB lookup cheaper by reducing the number higher level directory tables which must be allocated. Fewer tables means less memory usage which can result is higher cache hit rates for the directory lookup.

The first optimization is closely related to the minimization of page faults. We will cover that topic in detail in Section 7.5. While page faults usually are a one-time cost, TLB misses are a perpetual penalty given that the TLB cache is usually small and it is flushed frequently. Page faults are orders of magnitude more expensive than TLB misses but, if a program is running long enough and certain parts of the program are executed frequently enough, TLB misses can outweigh even page fault costs. It is therefore important to regard page optimization not only from the perspective of page faults but also from the TLB miss perspective. The difference is that, while page fault optimizations only require page-wide grouping of the code and data, TLB optimization requires that, at any point in time, as few TLB entries are in use as possible.

The second TLB optimization is even harder to control. The number of page directories which have to be used depends on the distribution of the address ranges used in the virtual address space of the process. Widely varying locations in the address space mean more directories. A complication is that Address Space Layout Randomization (ASLR) leads to exactly these situations. The load addresses of stack, DSOs, heap, and possibly executable are randomized at runtime to prevent attackers of the machine from guessing the addresses of functions or variables.

For maximum performance ASLR certainly should be turned off. The costs of the extra directories is low enough, though, to make this step unnecessary in all but a few extreme cases. One possible optimization the kernel could at any time perform is to ensure that a single mapping does not cross the

address space boundary between two directories. This would limit ASLR in a minimal fashion but not enough to substantially weaken it.

The only way a programmer is directly affected by this is when an address space region is explicitly requested. This happens when

using `mmap`with `MAP_FIXED`. Allocating new a address space region this way is

very dangerous and hardly ever done. It is possible, though, and, if it is used, the programmer should know about the boundaries of the last level page directory and select the requested address appropriately.

6.3 Prefetching

The purpose of prefetching is to hide the latency of a memory access. The command pipeline and out-of-order (OOO) execution capabilities of today's processors can hide some latency but, at best, only for accesses which hit the caches. To cover the latency of main memory accesses, the command queue would have to be incredibly long. Some processors without OOO try to compensate by increasing the number of cores, but this is a bad trade unless all the code in use is parallelized.

Prefetching can further help to hide latency. The processor performs prefetching on its own, triggered by certain events (hardware prefetching) or explicitly requested by the program (software prefetching).

6.3.1 Hardware Prefetching

The trigger for hardware prefetching is usually a sequence of two or more cache misses in a certain pattern. These cache misses can be to succeeding or preceding cache lines. In old implementations only cache misses to adjacent cache lines are recognized. With contemporary hardware, strides are recognized as well, meaning that skipping a fixed number of cache lines is recognized as a pattern and handled appropriately.

It would be bad for performance if every single cache miss triggered a hardware prefetch. Random memory accesses, for instance to global variables, are quite common and the resulting prefetches would mostly waste FSB bandwidth. This is why, to kickstart prefetching, at least two cache misses are needed. Processors today all expect there to be more than one stream of memory accesses. The processor tries to automatically assign each cache miss to such a stream and, if the threshold is reached, start hardware prefetching. CPUs today can keep track of eight to sixteen separate streams for the higher level caches.

The units responsible for the pattern recognition are associated with the respective cache. There can be a prefetch unit for the L1d and L1i caches. There is most probably a prefetch unit for the L2 cache and higher. The L2 and higher prefetch unit is shared with all the other cores and hyper-threads using the same cache. The number of eight to sixteen separate streams therefore is quickly reduced.

Prefetching has one big weakness: it cannot cross page boundaries. The reason should be obvious when one realizes that the CPUs support demand paging. If the prefetcher were allowed to cross page boundaries, the access might trigger an OS event to make the page available. This by itself can be bad, especially for performance. What is worse is that the prefetcher does not know about the semantics of the program or the OS itself. It might therefore prefetch pages which, in real life, never would be requested. That means the prefetcher would run past the end of the memory region the processor accessed in a recognizable pattern before. This is not only possible, it is very likely. If the processor, as a side effect of a prefetch, triggered a request for such a page the OS might even be completely thrown off its tracks if such a request could never otherwise happen.

It is therefore important to realize that, regardless of how good the prefetcher is at predicting the pattern, the program will experience cache misses at page boundaries unless it explicitly prefetches or reads from the new page. This is another reason to optimize the layout of data as described in Section 6.2 to minimize cache pollution by keeping unrelated data out.

Because of this page limitation the processors do not have terribly sophisticated logic to recognize prefetch patterns. With the still predominant 4k page size there is only so much which makes sense. The address range in which strides are recognized has been increased over the years, but it probably does not make much sense to go beyond the 512 byte window which is often used today. Currently prefetch units do not recognize non-linear access patterns. It is more likely than not that such patterns are truly random or, at least, sufficiently non-repeating that it makes no sense to try recognizing them.

If hardware prefetching is accidentally triggered there is only so much one can do. One possibility is to try to detect this problem and change the data and/or code layout a bit. This is likely to prove hard. There might be special localized solutions like using the `ud2` instruction {*Or non-instruction. It is the recommended undefined opcode.*} on x86 and x86-64 processors. This instruction, which cannot be executed itself, is used after an indirect jump instruction; it is used as a signal to the instruction fetcher that the processor should not waste efforts decoding the following memory since the execution

will continue at a different location. This is a very special situation, though. In most cases one has to live with this problem.

It is possible to completely or partially disable hardware prefetching for the entire processor. On Intel processors an Model Specific Register (MSR) is used for this (IA32_MISC_ENABLE, bit 9 on many processors; bit 19 disables only the adjacent cache line prefetch). This, in most cases, has to happen in the kernel since it is a privileged operation. If profiling shows that an important application running on a system suffers from bandwidth exhaustion and premature cache evictions due to hardware prefetches, using this MSR is a possibility.

6.3.2 Software Prefetching

The advantage of hardware prefetching is that programs do not have to be adjusted. The drawbacks, as just described, are that the access patterns must be trivial and that prefetching cannot happen across page boundaries. For these reasons we now have more possibilities, software prefetching the most important of them. Software prefetching does require modification of the source code by inserting special instructions. Some compilers support pragmas to more or less automatically insert prefetch instructions. On x86 and x86-64 Intel's convention for compiler intrinsics to insert these special instructions is generally used:

```
#include <xmmintrin.h>
enum _mm_hint
{
  _MM_HINT_T0 = 3,
  _MM_HINT_T1 = 2,
  _MM_HINT_T2 = 1,
  _MM_HINT_NTA = 0
};
void _mm_prefetch(void *p,
                  enum _mm_hint h);
```

Programs can use the `_mm_prefetch` intrinsic on any pointer in the program.

Most processors (certainly all x86 and x86-64 processors) ignore errors resulting from invalid pointers which make the life of the programmer significantly easier. If the passed pointer references valid memory, though, the prefetch unit will be instructed to load the data into cache and, if necessary, evict other data. Unnecessary prefetches should definitely be avoided since this might reduce the effectiveness of the caches and it consumes memory bandwidth (possibly for two cache lines in case the evicted cache line is dirty).

The different hints to be used with the _mm_prefetch intrinsic are implementation defined. That means each processor version can implement them (slightly) differently. What can generally be said is that _MM_HINT_T0 fetches data to all levels of the cache for inclusive caches and to the lowest level cache for exclusive caches. If the data item is in a higher level cache it is loaded into L1d. The _MM_HINT_T1 hint pulls the data into L2 and not into L1d. If there is an L3 cache the _MM_HINT_T2 hints can do something similar for it. These are details, though, which are weakly specified and need to be verified for the actual processor in use. In general, if the data is to be used right away using _MM_HINT_T0 is the right thing to do. Of course this requires that the L1d cache size is large enough to hold all the prefetched data. If the size of the immediately used working set is too large, prefetching everything into L1d is a bad idea and the other two hints should be used.

The fourth hint, _MM_HINT_NTA, is special in that it allows telling the processor to treat the prefetched cache line specially. NTA stands for non-temporal aligned which we already explained in Section 6.1. The program tells the processor that polluting caches with this data should be avoided as much as possible since the data is only used for a short time. The processor can therefore, upon loading, avoid reading the data into the lower level caches for inclusive cache implementations. When the data is evicted from L1d the data need not be pushed into L2 or higher but, instead, can be written directly to memory. There might be other tricks the processor designers can deploy if this hint is given. The programmer must be careful using this hint: if the immediate working set size is too large and forces eviction of a cache line loaded with the NTA hint, reloading from memory will occur.

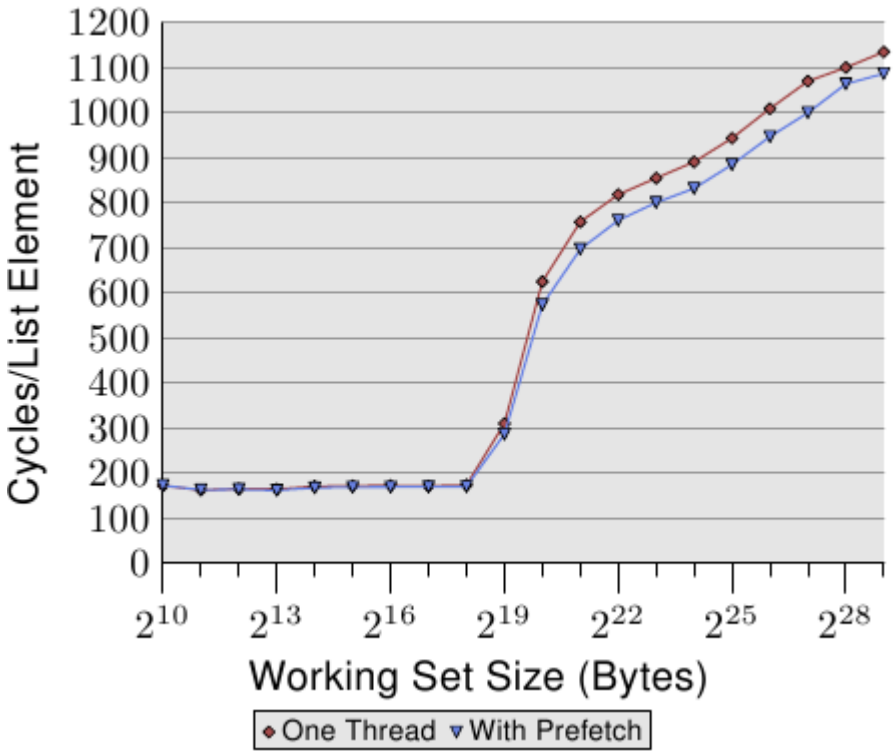


Figure 6.7: Average with Prefetch, NPAD=31

Figure 6.7 shows the results of a test using the now familiar pointer chasing framework. The list is randomized. The difference to previous test is that the program actually spends some time at each list node (about 160 cycles). As we learned from the data in Figure 3.15, the program's performance suffers badly as soon as the working set size is larger than the last-level cache.

We can now try to improve the situation by issuing prefetch requests ahead of the computation. I.e., in each round of the loop we prefetch a new element. The distance between the prefetched node in the list and the node which is currently worked on must be carefully chosen. Given that each node is processed in 160 cycles and that we have to prefetch two cache lines (`NPAD`=31), a distance of five list elements is enough.

The results in Figure 6.7 show that the prefetch does indeed help. As long as the working set size does not exceed the size of the last level cache (the machine has 512kB = 2^{19}B of L2) the numbers are identical. The prefetch instructions do not add a measurable extra burden. As soon as the L2 size is exceeded the prefetching saves between 50 to 60 cycles, up to 8%. The use of prefetch cannot hide all the penalties but it does help a bit.

AMD implements, in their family 10h of the Opteron line, another instruction: `prefetchw`. This instruction has so far no equivalent on the Intel side and is not available through intrinsics. The `prefetchw` instruction prefetches the cache line into L1 just like the other prefetch instructions. The difference is that the cache line is immediately put into 'M' state. This will be a disadvantage if no write to the cache line follows later. If there are one or more writes, they will be accelerated since the writes do not have to change the cache state—that already happened when the cache line was prefetched.

Prefetching can have bigger advantages than the meager 8% we achieved here. But it is notoriously hard to do right, especially if the same binary is supposed to perform well on a variety of machines. The performance counters provided by the CPU can help the programmer analyze prefetches. Events which can be counted and sampled include hardware prefetches, software prefetches, useful software prefetches, cache misses at the various levels, and more. In Section 7.1 we will introduce a number of these events. All these counters are machine specific.

When analyzing programs one should first look at the cache misses. When a large source of cache misses is located one should try to add a prefetch instruction for the problematic memory accesses. This should be done in one place at a time. The result of each modification should be checked by observing the performance counters measuring useful prefetch instructions. If those counters do not increase the prefetch might be wrong, it is not given enough time to load from memory, or the prefetch evicts memory from the cache which is still needed.

gcc today is able to emit prefetch instructions automatically in one situation. If a loop is iterating over an array the following option can be used:

```
-fprefetch-loop-arrays
```

The compiler will figure out whether prefetching makes sense and, if so, how far ahead it should look. For small arrays this can be a disadvantage and, if the size of the array is not known at compile time, the results might be worse. The gcc manual warns that the benefits highly depend on the form of the code and that in some situation the code might actually run slower. Programmers have to use this option carefully.

6.3.3 Special Kind of Prefetch: Speculation

The OOO execution capability of a processor allows moving instructions around if they do not conflict with each other. For instance (using this time IA-64 for the example):

```
st8         [r4] = 12
add         r5 = r6, r7;;
st8         [r18] = r5
```

This code sequence stores 12 at the address specified by register `r4`, adds the content of registers `r6` and `r7` and stores it in register `r5`. Finally it stores the sum at the address specified by register `r18`. The point here is that the add instruction can be executed before—or at the same time as—the first `st8` instruction since there is no data dependency. But what happens if one of the addends has to be loaded?

```
st8         [r4] = 12
ld8         r6 = [r8];;
add         r5 = r6, r7;;
st8         [r18] = r5
```

The extra `ld8` instruction loads the value from the address specified by the register `r8`. There is an obvious data dependency between this load instruction and the following `add` instruction (this is the reason for the `;;` after the instruction, thanks for asking). What is critical here is that the new `ld8` instruction—unlike the `add` instruction—cannot be moved in front of the first `st8`. The processor cannot determine quickly enough during the instruction decoding whether the store and load conflict, i.e., whether `r4` and `r8` might have same value. If they do have the same value, the `st8` instruction would determine the value loaded into `r6`. What is worse, the `ld8` might also bring with it a large latency in case the load misses the caches. The IA-64 architecture supports speculative loads for this case:

```
ld8.a       r6 = [r8];;
[... other instructions ...]
st8         [r4] = 12
ld8.c.clr   r6 = [r8];;
add         r5 = r6, r7;;
st8         [r18] = r5
```

The new `ld8.a` and `ld8.c.clr` instructions belong together and replace the `ld8` instruction in the previous code sequence. The `ld8.a` instruction is the speculative load. The value cannot be used directly but the processor can start the work. At the time when the `ld8.c.clr` instruction is reached the content might have been loaded already (given there is a sufficient number of instructions in the gap). The arguments for this instruction must match that for the `ld8.a` instruction. If the preceding `st8` instruction does not overwrite the value (i.e., `r4` and `r8` are the same), nothing has to be done. The speculative load does its job and the latency of the load is hidden. If the store and load do conflict the`ld8.c.clr` reloads the value from memory and we end up with the semantics of a normal `ld8` instruction.

Speculative loads are not (yet?) widely used. But as the example shows it is a very simple yet effective way to hide latencies. Prefetching is basically equivalent and, for processors with fewer registers, speculative loads probably do not make much sense. Speculative loads have the (sometimes big) advantage of loading the value directly into the register and not into the cache line where it might be evicted again (for instance, when the thread is descheduled). If speculation is available it should be used.

6.3.4 Helper Threads

When one tries to use software prefetching one often runs into problems with the complexity of the code. If the code has to iterate over a data structure (a list in our case) one has to implement two independent iterations in the same loop: the normal iteration doing the work and the second iteration, which looks ahead, to use prefetching. This easily gets complex enough that mistakes are likely.

Furthermore, it is necessary to determine how far to look ahead. Too little and the memory will not be loaded in time. Too far and the just loaded data

might have been evicted again. Another problem is that prefetch instructions, although they do not block and wait for the memory to be loaded, take time. The instruction has to be decoded, which might be noticeable if the decoder is too busy, for instance, due to well written/generated code. Finally, the code size of the loop is increased. This decreases the L1i efficiency. If one tries to avoid parts of this cost by issuing multiple prefetch requests in a row (in case the second load does not depend on the result of the first) one runs into problems with the number of outstanding prefetch requests.

An alternative approach is to perform the normal operation and the prefetch completely separately. This can happen using two normal threads. The threads must obviously be scheduled so that the prefetch thread is populating a cache accessed by both threads. There are two special solutions worth mentioning:

- Use hyper-threads (see Figure 3.22) on the same core. In this case the prefetch can go into L2 (or even L1d).
- Use “dumber” threads than SMT threads which can do nothing but prefetch and other simple operations. This is an option processor manufacturers might explore.

The use of hyper-threads is particularly intriguing. As we have seen on Figure 3.22, the sharing of caches is a problem if the hyper-threads execute independent code. If, instead, one thread is used as a prefetch helper thread this is not a problem. To the contrary, it is the desired effect since the lowest level cache is preloaded. Furthermore, since the prefetch thread is mostly idle or waiting for memory, the normal operation of the other hyper-thread is not disturbed much if it does not have to access main memory itself. The latter is exactly what the prefetch helper thread prevents.

The only tricky part is to ensure that the helper thread is not running too far ahead. It must not completely pollute the cache so that the oldest prefetched values are evicted again. On Linux, synchronization is easily done using the `futex` system call [futexes] or, at a little bit higher cost, using the POSIX thread synchronization primitives.

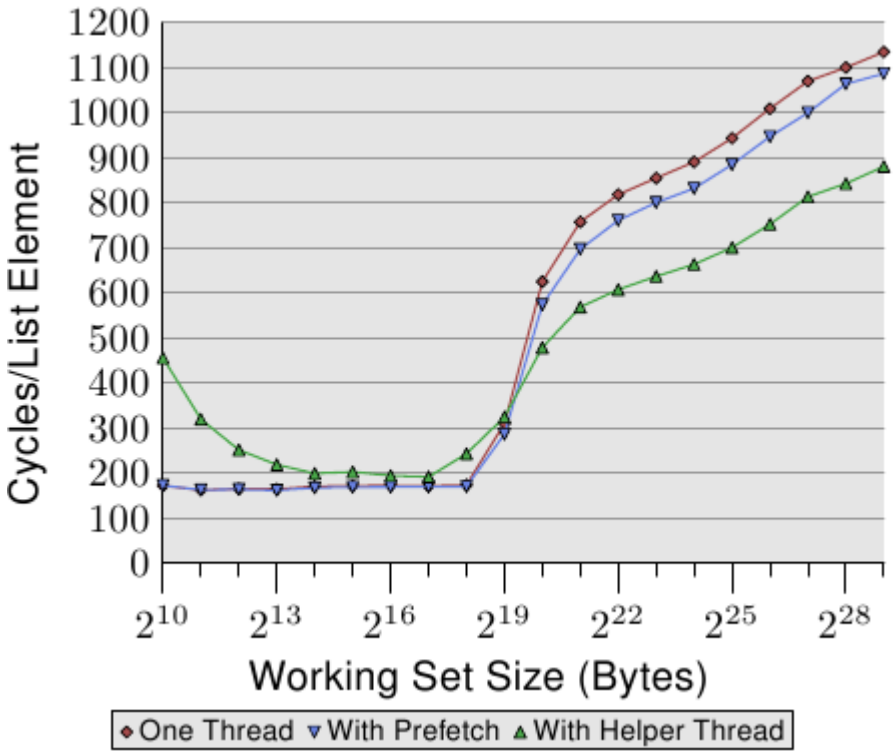


Figure 6.8: Average with Helper Thread, NPAD=31

The benefits of the approach can be seen in Figure 6.8. This is the same test as in Figure 6.7 only with the additional result added. The new test creates an additional helper thread which runs about 100 list entries ahead and reads (not only prefetches) all the cache lines of each list element. In this case we have two cache lines per list element (`NPAD`=31 on a 32-bit machine with 64 byte cache line size).

The two threads are scheduled on two hyper-threads of the same core. The test machine has only one core but the results should be about the same if there is more than one core. The affinity functions, which we will introduce in Section 6.4.3, are used to tie the threads down to the appropriate hyper-thread.

To determine which two (or more) processors the OS knows are hyper-threads, the `NUMA_cpu_level_mask` interface from libNUMA can be used (see Section 12).

```
#include <libNUMA.h>
ssize_t NUMA_cpu_level_mask(size_t destsize,
                            cpu_set_t *dest,
                            size_t srcsize,
```

```
                        const cpu_set_t*src,
                        unsigned int level);
```

This interface can be used to determine the hierarchy of CPUs as they are connected through caches and memory. Of interest here is level 1 which corresponds to hyper-threads. To schedule two threads on two hyper-threads the libNUMA functions can be used (error handling dropped for brevity):

```
cpu_set_t self;
NUMA_cpu_self_current_mask(sizeof(self),
                           &self);
cpu_set_t hts;
NUMA_cpu_level_mask(sizeof(hts), &hts,
                    sizeof(self), &self, 1);
CPU_XOR(&hts, &hts, &self);
```

After this code is executed we have two CPU bit sets. `self` can be used to set the affinity of the current thread and the mask in `hts` can be used to set the affinity of the helper thread. This should ideally happen before the thread is created. In Section 6.4.3 we will introduce the interface to set the affinity. If there is no hyper-thread available the `NUMA_cpu_level_mask` function will return 1. This can be used as a sign to avoid this optimization.

The result of this benchmark might be surprising (or maybe not). If the working set fits into L2, the overhead of the helper thread reduces the performance by between 10% and 60% (ignore the smallest working set sizes again, the noise is too high). This should be expected since, if all the data is already in the L2 cache, the prefetch helper thread only uses system resources without contributing to the execution.

Once the L2 size is exhausted the picture changes, though. The prefetch helper thread helps to reduce the runtime by about 25%. We still see a rising curve simply because the prefetches cannot be processed fast enough. The arithmetic operations performed by the main thread and the memory load operations of the helper thread do complement each other, though. The resource collisions are minimal which causes this synergistic effect.

The results of this test should be transferable to many other situations. Hyper-threads, often not useful due to cache pollution, shine in these situations and should be taken advantage of. The `sys` file system allows a

program to find the thread siblings (see the `thread_siblings`column in Table 5.3). Once this information is available the program just has to define the affinity of the threads and then run the loop in two modes: normal operation and prefetching. The amount of memory prefetched should depend on the size of the shared cache. In this example the L2 size is relevant and the program can query the size using

```
sysconf(_SC_LEVEL2_CACHE_SIZE)
```

Whether or not the progress of the helper thread must be restricted depends on the program. In general it is best to make sure there is some synchronization since scheduling details could otherwise cause significant performance degradations.

6.3.5 Direct Cache Access

One sources of cache misses in a modern OS is the handling of incoming data traffic. Modern hardware, like Network Interface Cards (NICs) and disk controllers, has the ability to write the received or read data directly into memory without involving the CPU. This is crucial for the performance of the devices we have today, but it also causes problems. Assume an incoming packet from a network: the OS has to decide how to handle it by looking at the header of the packet. The NIC places the packet into memory and then notifies the processor about the arrival. The processor has no chance to prefetch the data since it does not know when the data will arrive, and maybe not even where exactly it will be stored. The result is a cache miss when reading the header.

Intel has added technology in their chipsets and CPUs to alleviate this problem [directcacheaccess]. The idea is to populate the cache of the CPU which will be notified about the incoming packet with the packet's data. The payload of the packet is not critical here, this data will, in general, be handled by higher-level functions, either in the kernel or at user level. The packet header is used to make decisions about the way the packet has to be handled and so this data is needed immediately.

The network I/O hardware already has DMA to write the packet. That means it communicates directly with the memory controller which potentially is integrated in the Northbridge. Another side of the memory controller is the interface to the processors through the FSB (assuming the memory controller is not integrated into the CPU itself).

The idea behind Direct Cache Access (DCA) is to extend the protocol between the NIC and the memory controller. In Figure 6.9 the first figure

shows the beginning of the DMA transfer in a regular machine with North- and Southbridge.

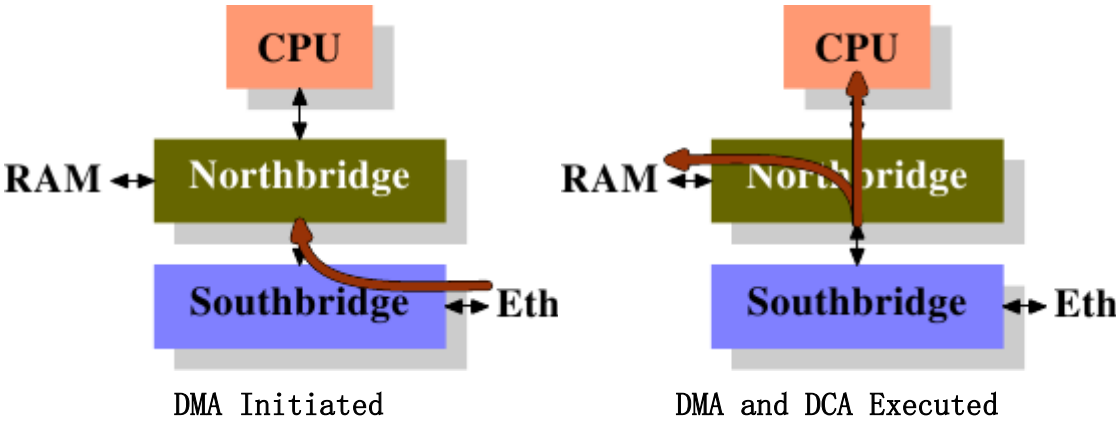


Figure 6.9: Direct Cache Access

The NIC is connected to (or is part of) the Southbridge. It initiates the DMA access but provides the new information about the packet header which should be pushed into the processor's cache.

The traditional behavior would be, in step two, to simply complete the DMA transfer with the connection to the memory. For the DMA transfers with the DCA flag set the Northbridge additionally sends the data on the FSB with a special, new DCA flag. The processor always snoops the FSB and, if it recognizes the DCA flag, it tries to load the data directed to the processor into the lowest cache. The DCA flag is, in fact, a hint; the processor can freely ignore it. After the DMA transfer is finished the processor is signaled.

The OS, when processing the packet, first has to determine what kind of packet it is. If the DCA hint is not ignored, the loads the OS has to perform to identify the packet most likely hit the cache. Multiply this saving of hundreds of cycles per packet with tens of thousands of packets which can be processed per second, and the savings add up to very significant numbers, especially when it comes to latency.

Without the integration of I/O hardware (NIC in this case), chipset, and CPUs such an optimization is not possible. It is therefore necessary to make sure to select the platform wisely if this technology is needed.

This article was contributed by Ulrich Drepper

[*Editor's note: this is part 6 of Ulrich Drepper's "What every programmer should know about memory"; this part contains the second half of section 6, covering the optimization of multi-threaded code. The first half of this*

section was published in part 5; please see part 1 for pointers to the other sections.]

6.4 Multi-Thread Optimizations

When it comes to multi-threading, there are three different aspects of cache use which are important:

- Concurrency
- Atomicity
- Bandwidth

These aspects also apply to multi-process situations but, because multiple processes are (mostly) independent, it is not so easy to optimize for them. The possible multi-process optimizations are a subset of those available for the multi-thread scenario. So we will deal exclusively with the latter here.

Concurrency in this context refers to the memory effects a process experiences when running more than one thread at a time. A property of threads is that they all share the same address space and, therefore, can all access the same memory. In the ideal case, the memory regions used by the threads are distinct, in which case those threads are coupled only lightly (common input and/or output, for instance). If more than one thread uses the same data, coordination is needed; this is when atomicity comes into play. Finally, depending on the machine architecture, the available memory and inter-processor bus bandwidth available to the processors is limited. We will handle these three aspects separately in the following sections—although they are, of course, closely linked.

6.4.1 Concurrency Optimizations

Initially, in this section, we will discuss two separate issues which actually require contradictory optimizations. A multi-threaded application uses common data in some of its threads. Normal cache optimization calls for keeping data together so that the footprint of the application is small, thus maximizing the amount of memory which fits into the caches at any one time.

There is a problem with this approach, though: if multiple threads write to a memory location, the cache line must be in ‘E’ (exclusive) state in the L1d of each respective core. This means that a lot of RFO requests are sent, in the worst case one for each write access. So a normal write will be suddenly very expensive. If the same memory location is used, synchronization is needed (maybe through the use of atomic operations, which is handled in the next section). The problem is also visible, though, when all the threads are using different memory locations and are supposedly independent.

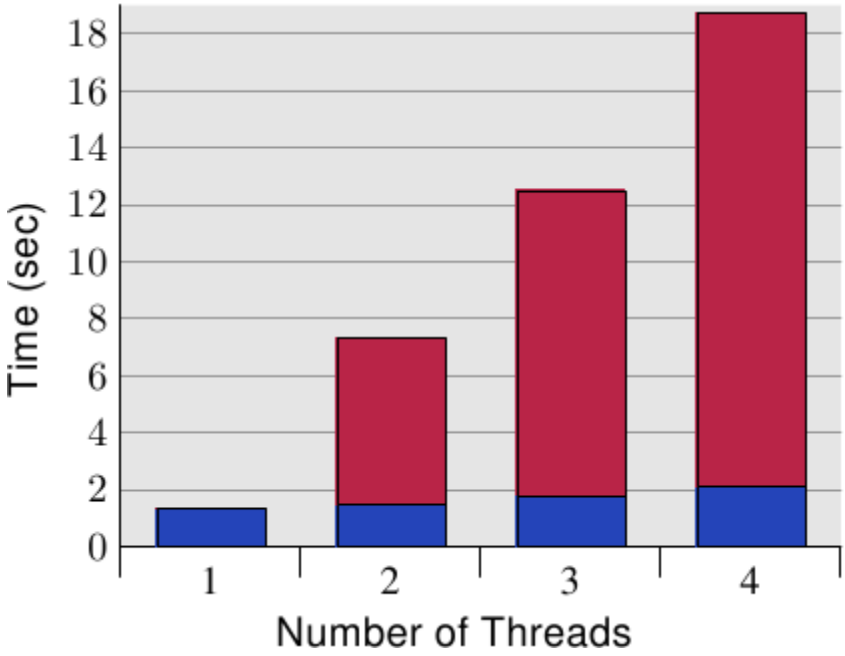


Figure 6.10: Concurrent Cache Line Access Overhead

Figure 6.10 shows the results of this "false sharing". The test program (shown in Section 9.3) creates a number of threads which do nothing but increment a memory location (500 million times). The measured time is from the program start until the program finishes after waiting for the last thread. The threads are pinned to individual processors. The machine has four P4 processors. The blue values represent runs where the memory allocations assigned to each thread are on separate cache lines. The red part is the penalty occurred when the locations for the threads are moved to just one cache line.

The blue measurements (when using individual cache lines) match what one would expect. The program scales without penalty to many threads. Each processor keeps its cache line in its own L1d and there are no bandwidth issues since not much code or data has to be read (in fact, it is all cached). The measured slight increase is really system noise and probably some prefetching effects (the threads use sequential cache lines).

The measured overhead, computed by dividing the time needed when using one cache line versus a separate cache line for each thread, is 390%, 734%, and 1,147% respectively. These large numbers might be surprising at first sight but, when thinking about the cache interaction needed, it should be obvious. The cache line is pulled from one processor's cache just after it has finished writing to the cache line. All processors, except the one which has the cache line at any given moment, are delayed and cannot do anything. Each additional processor will just cause more delays.

It is clear from these measurements that this scenario must be avoided in programs. Given the huge penalty, this problem is, in many situations, obvious (profiling will show the code location, at least) but there is a pitfall with modern hardware. Figure 6.11 shows the equivalent measurements when running the code on a single processor, quad core machine (Intel Core 2 QX 6700). Even with this processor's two separate L2s the test case does not show any scalability issues. There is a slight overhead when using the same cache line more than once but it does not increase with the number of cores. {*I cannot explain the lower number when all four cores are used but it is reproducible.*} If more than one of these processors were used we would, of course, see results similar to those in Figure 6.10. Despite the increasing use of multi-core processors, many machines will continue to use multiple processors and, therefore, it is important to handle this scenario correctly, which might mean testing the code on real SMP machines.

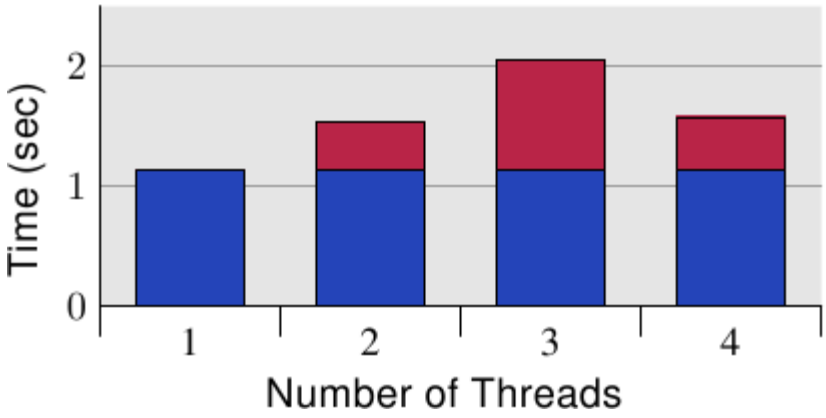


Figure 6.11: Overhead, Quad Core

There is a very simple “fix” for the problem: put every variable on its own cache line. This is where the conflict with the previously mentioned optimization comes into play, specifically, the footprint of the application would increase a lot. This is not acceptable; it is therefore necessary to come up with a more intelligent solution.

What is needed is to identify which variables are used by only one thread at a time, those used by only one thread ever, and maybe those which are contested at times. Different solutions for each of these scenarios are possible and useful. The most basic criterion for the differentiation of variables is: are they ever written to and how often does this happen.

Variables which are never written to and those which are only initialized once are basically constants. Since RFO requests are only needed for write operations, constants can be shared in the cache (‘S’ state). So, these variables do not have to be treated specially; grouping them together is fine.

If the programmer marks the variables correctly with `const`, the tool chain will move the variables away from the normal variables into the `.rodata` (read-only data) or `.data.rel.ro` (read-only after relocation) section {*Sections, identified by their names are the atomic units containing code and data in an ELF file.*} No other special action is required. If, for some reason, variables cannot be marked correctly with `const`, the programmer can influence their placement by assigning them to a special section.

When the linker constructs the final binary, it first appends the sections with the same name from all input files; those sections are then arranged in an order determined by the linker script. This means that, by moving all variables which are basically constant but are not marked as such into a special section, the programmer can group all of those variables together. There will not be a variable which is often written to between them. By aligning the first variable in that section appropriately, it is possible to guarantee that no false sharing happens. Assume this little example:

```
int foo = 1;
int bar __attribute__((section(".data.ro"))) = 2;
int baz = 3;
int xyzzy __attribute__((section(".data.ro"))) = 4;
```

If compiled, this input file defines four variables. The interesting part is that the variables `foo` and `baz`, and `bar` and `xyzzy` are grouped together respectively. Without the attribute definitions the compiler would allocate all four variables in the sequence in which they are defined in the source code the a section named `.data`. {*This is not guaranteed by the ISO C standard but it is how gcc works.*} With the code as-is the variables `bar` and `xyzzy` are placed in a section named `.data.ro`. The section name `.data.ro` is more or less arbitrary. A prefix of `.data.` guarantees that the GNU linker will place the section together with the other data sections.

The same technique can be applied to separate out variables which are mostly read but occasionally written. Simply choose a different section name. This separation seems to make sense in some cases like the Linux kernel.

If a variable is only ever used by one thread, there is another way to specify the variable. In this case it is possible and useful to use thread-local variables (see [mytls]). The C and C++ language in gcc allow variables to be defined as per-thread using the `__thread` keyword.

```
int foo = 1;
__thread int bar = 2;
int baz = 3;
__thread int xyzzy = 4;
```

The variables `bar` and `xyzzy` are not allocated in the normal data segment; instead each thread has its own separate area where such variables are stored. The variables can have static initializers. All thread-local variables are addressable by all other threads but, unless a thread passes a pointer to a thread-local variable to those other threads, there is no way the other threads can find that variable. Due to the variable being thread-local, false sharing is not a problem—unless the program artificially creates a problem. This solution is easy to set up (the compiler and linker do all the work), but it has its cost. When a thread is created, it has to spend some time on setting up the thread-local variables, which requires time and memory. In addition, addressing thread-local variables is usually more expensive than using global or automatic variables (see [mytls] for explanations of how the costs are minimized automatically, if possible).

One drawback of using thread-local storage (TLS) is that, if the use of the variable shifts over to another thread, the current value of the variable in the old thread is not available to new thread. Each thread's copy of the variable is distinct. Often this is not a problem at all and, if it is, the shift over to the new thread needs coordination, at which time the current value can be copied.

A second, bigger problem is possible waste of resources. If only one thread ever uses the variable at any one time, all threads have to pay a price in terms of memory. If a thread does not use any TLS variables, the lazy allocation of the TLS memory area prevents this from being a problem (except for TLS in the application itself). If a thread uses just one TLS variable in a DSO, the memory for all the other TLS variables in this object will be allocated, too. This could potentially add up if TLS variables are used on a large scale.

In general the best advice which can be given is

1. Separate at least read-only (after initialization) and read-write variables. Maybe extend this separation to read-mostly variables as a third category.
2. Group read-write variables which are used together into a structure. Using a structure is the only way to ensure the memory locations for all of those variables are close together in a way which is translated consistently by all gcc versions..
3. Move read-write variables which are often written to by different threads onto their own cache line. This might mean adding padding at the end to fill a remainder of the cache line. If combined with step 2, this is often not really wasteful. Extending the example above, we might end up with code as follows (assuming `bar` and `xyzzy` are meant to be used together):

```
int foo = 1;
int baz = 3;
struct {
  struct all {
    int bar;
    int xyzzy;
  };
  char pad[CLSIZE - sizeof(struct all)];
} rwstruct __attribute__((aligned(CLSIZE))) =
  { { .bar = 2, .xyzzy = 4 } };
```

   Some code changes are needed (references to `bar` have to be replaced with `rwstruct.bar`, likewise for `xyzzy`) but that is all. The compiler and linker do all the rest. {*This code has to be compiled with `-fms-extensions`} on the command line.*}

14. If a variable is used by multiple threads, but every use is independent, move the variable into TLS.

6.4.2 Atomicity Optimizations

If multiple threads modify the same memory location concurrently, processors do not guarantee any specific result. This is a deliberate decision made to avoid costs which are unnecessary in 99.999% of all cases. For instance, if a memory location is in the ‘S’ state and two threads concurrently have to increment its value, the execution pipeline does not have to wait for the cache line to be available in the ‘E’ state before reading the old value from the cache to perform the addition. Instead it reads the

value currently in the cache and, once the cache line is available in state ‘E’, the new value is written back. The result is not as expected if the two cache reads in the two threads happen simultaneously; one addition will be lost.

To assure this does not happen, processors provide atomic operations. These atomic operations would, for instance, not read the old value until it is clear that the addition could be performed in a way that the addition to the memory location appears as atomic. In addition to waiting for other cores and processors, some processors even signal atomic operations for specific addresses to other devices on the motherboard. All this makes atomic operations slower.

Processor vendors decided to provide different sets of atomic operations. Early RISC processors, in line with the ‘R’ for reduced, provided very few atomic operations, sometimes only an atomic bit set and test. {*HP Parisc still does not provide more…*} At the other end of the spectrum, we have x86 and x86-64 which provide a large number of atomic operations. The generally available atomic operations can be categorized in four classes:

Bit Test

These operations set or clear a bit atomically and return a status indicating whether the bit was set before or not.

Load Lock/Store Conditional (LL/SC)

{*Some people use “linked” instead of “lock”, it is all the same.*}

These operations work as a pair where the special load instruction is used to start an transaction and the final store will only succeed if the location has not been modified in the meantime. The store operation indicates success or failure, so the program can repeat its efforts if necessary.

Compare-and-Swap (CAS)

This is a ternary operation which writes a value provided as a parameter into an address (the second parameter) only if the current value is the same as the third parameter value;

Atomic Arithmetic

These operations are only available on x86 and x86-64, which can perform arithmetic and logic operations on memory locations. These processors have support for non-atomic versions of these operations but RISC architectures do not. So it is no wonder that their availability is limited.

An architecture supports either the LL/SC or the CAS instruction, not both. Both approaches are basically equivalent; they allow the implementation of atomic arithmetic operations equally well, but CAS seems to be the

preferred method these days. All other operations can be indirectly implemented using it. For instance, an atomic addition:

```
int curval;
int newval;
do {
  curval = var;
  newval = curval + addend;
} while (CAS(&var, curval, newval));
```

The result of the `CAS` call indicates whether the operation succeeded or not. If it returns failure (non-zero value), the loop is run again, the addition is performed, and the `CAS` call is tried again. This repeats until it is successful. Noteworthy about the code is that the address of the memory location has to be computed in two separate instructions. {*The `CAS` opcode on x86 and x86-64 can avoid the load of the value in the second and later iterations but, on this platform, we can write the atomic addition in a simpler way, with a single addition opcode.*} For LL/SC the code looks about the same.

```
int curval;
int newval;
do {
  curval = LL(var);
  newval = curval + addend;
} while (SC(var, newval));
```

Here we have to use a special load instruction (`LL`) and we do not have to pass the current value of the memory location to `SC` since the processor knows if the memory location has been modified in the meantime.

The big differentiators are x86 and x86-64 where we have the atomic operations and, here, it is important to select the proper atomic operation to achieve the best result. Figure 6.12 shows three different ways to implement an atomic increment operation.

```
    for (i = 0; i < N; ++i)
      __sync_add_and_fetch(&var,1);
```

1. Add and Read Result

```
    for (i = 0; i < N; ++i)
```

```
    __sync_fetch_and_add(&var,1);

2. Add and Return Old Value
  for (i = 0; i < N; ++i) {
    long v, n;
    do {
      v = var;
      n = v + 1;
    } while (!__sync_bool_compare_and_swap(&var, v, n));
  }

3. Atomic Replace with New Value
```

Figure 6.12: Atomic Increment in a Loop

All three produce different code on x86 and x86-64 while the code might be identical on other architectures. There are huge performance differences. The following table shows the execution time for 1 million increments by four concurrent threads. The code uses the built-in primitives of gcc (`__sync_*`).

1. Exchange Add	2. Add Fetch	3. CAS
0.23s	0.21s	0.73s

The first two numbers are similar; we see that returning the old value is a little bit faster. The important piece of information is the highlighted field, the cost when using CAS. It is, unsurprisingly, a lot more expensive. There are several reasons for this: 1. there are two memory operations, 2. the CAS operation by itself is more complicated and requires even conditional operation, and 3. the whole operation has to be done in a loop in case two concurrent accesses cause a CAS call to fail.

Now a reader might ask a question: why would somebody use the complicated and longer code which utilizes CAS? The answer to this is: the complexity is usually hidden. As mentioned before, CAS is currently the unifying atomic operation across all interesting architectures. So some people think it is sufficient to define all atomic operations in terms of CAS. This makes programs simpler. But as the numbers show, the results can be everything but optimal. The memory handling overhead of the CAS solution is huge. The following illustrates the execution of just two threads, each on its own core.

```
Thread #1 Thread #2 var  Cache State
```

```
v = var                 'E'  on Proc 1
n = v + 1  v = var      'S'  on Proc 1+2
CAS(var)   n = v + 1    'E'  on Proc 1
           CAS(var)     'E'  on Proc 2
```

We see that, within this short period of execution, the cache line status changes at least three times; two of the changes are RFOs. Additionally, the second CAS will fail, so that thread has to repeat the whole operation. During that operation the same can happen again.

In contrast, when the atomic arithmetic operations are used, the processor can keep the load and store operations needed to perform the addition (or whatever) together. It can ensure that concurrently-issued cache line requests are blocked until the atomic operation is done. Each loop iteration in the example therefore results in, at most, one RFO cache request and nothing else.

What all this means is that it is crucial to define the machine abstraction at a level at which atomic arithmetic and logic operations can be utilized. CAS should not be universally used as the unification mechanism.

For most processors, the atomic operations are, by themselves, always atomic. One can avoid them only by providing completely separate code paths for the case when atomicity is not needed. This means more code, a conditional, and further jumps to direct execution appropriately.

For x86 and x86-64 the situation is different: the same instructions can be used in both atomic and non-atomic ways. To make them atomic, a special prefix for the instruction is used: the `lock` prefix. This opens the door for atomic operations to avoid the high costs if the atomicity requirement in a given situation is not needed. Generic code in libraries, for example, which always has to be thread-safe if needed, can benefit from this. No information is needed when writing the code, the decision can be made at runtime. The trick is to jump over the `lock`prefix. This trick applies to all the instructions which the x86 and x86-64 processor allow to prefix with `lock`.

```
      cmpl $0, multiple_threads
      je   1f
      lock
1:    add  $1, some_var
```

If this assembler code appears cryptic, do not worry, it is simple. The first instruction checks whether a variable is zero or not. Nonzero in this case indicates that more than one thread is running. If the value is zero, the second instruction jumps to label `1`. Otherwise, the next instruction is executed. This is the tricky part. If the `je` instruction does not jump, the `add` instruction is executed with the `lock` prefix. Otherwise it is executed without the `lock` prefix.

Adding a relatively expensive operation like a conditional jump (expensive in case the branch prediction fails) seems to be counter productive. Indeed it can be: if multiple threads are running most of the time, the performance is further decreased, especially if the branch prediction is not correct. But if there are many situations where only one thread is in use, the code is significantly faster. The alternative of using an if-then-else construct introduces an additional unconditional jump in both cases which can be slower. Given that an atomic operation costs on the order of 200 cycles, the cross-over point for using the trick (or the if-then-else block) is pretty low. This is definitely a technique to be kept in mind. Unfortunately this means gcc's `__sync_*` primitives cannot be used.

6.4.3 Bandwidth Considerations

When many threads are used, and they do not cause cache contention by using the same cache lines on different cores, there still are potential problems. Each processor has a maximum bandwidth to the memory which is shared by all cores and hyper-threads on that processor. Depending on the machine architecture (e.g., the one in Figure 2.1), multiple processors might share the same bus to memory or the Northbridge.

The processor cores themselves run at frequencies where, at full speed, even in perfect conditions, the connection to the memory cannot fulfill all load and store requests without waiting. Now, further divide the available bandwidth by the number of cores, hyper-threads, and processors sharing a connection to the Northbridge and suddenly parallelism becomes a big problem. Programs which are, in theory, very efficient may be limited by the memory bandwidth.

In Figure 3.32 we have seen that increasing the FSB speed of a processor can help a lot. This is why, with growing numbers of cores on a processor, we will also see an increase in the FSB speed. Still, this will never be

enough if the program uses large working sets and it is sufficiently optimized. Programmers have to be prepared to recognize problems due to limited bandwidth.

The performance measurement counters of modern processors allow the observation of FSB contention. On Core 2 processors the NUS_BNR_DRV event counts the number of cycles a core has to wait because the bus is not ready. This indicates that the bus is highly used and loads from or stores to main memory take even longer than usual. The Core 2 processors support more events which can count specific bus actions like RFOs or the general FSB utilization. The latter might come in handy when investigating the possibility of scalability of an application during development. If the bus utilization rate is already close to 1.0 then the scalability opportunities are minimal.

If a bandwidth problem is recognized, there are several things which can be done. They are sometimes contradictory so some experimentation might be necessary. One solution is to buy faster computers, if there are some available. Getting more FSB speed, faster RAM modules, and possibly memory local to the processor, can—and probably will—help. It can cost a lot, though. If the program in question is only needed on one (or a few machines) the one-time expense for the hardware might cost less than reworking the program. In general, though, it is better to work on the program.

After optimizing the program itself to avoid cache misses, the only option left to achieve better bandwidth utilization is to place the threads better on the available cores. By default, the scheduler in the kernel will assign a thread to a processor according to its own policy. Moving a thread from one core to another is avoided when possible. The scheduler does not really know anything about the workload, though. It can gather information from cache misses etc but this is not much help in many situations.

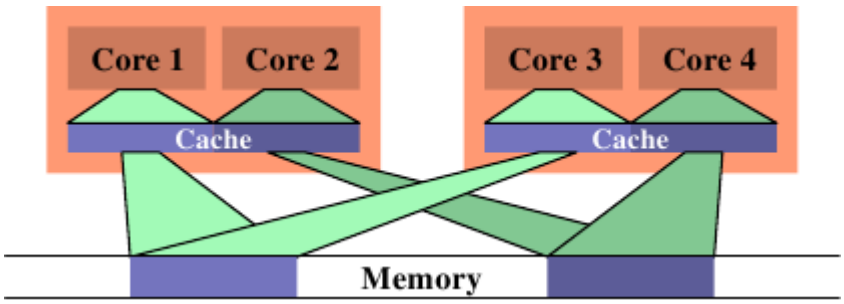


Figure 6.13: Inefficient Scheduling

One situation which can cause big FSB usage is when two threads are scheduled on different processors (or cores which do not share a cache) and they use the same data set. Figure 6.13 shows such a situation. Core 1 and 3 access the same data (indicated by the same color for the access indicator and the memory area). Similarly core 2 and 4 access the same data. But the threads are scheduled on different processors. This means each data set has to be read twice from memory. This situation can be handled better.

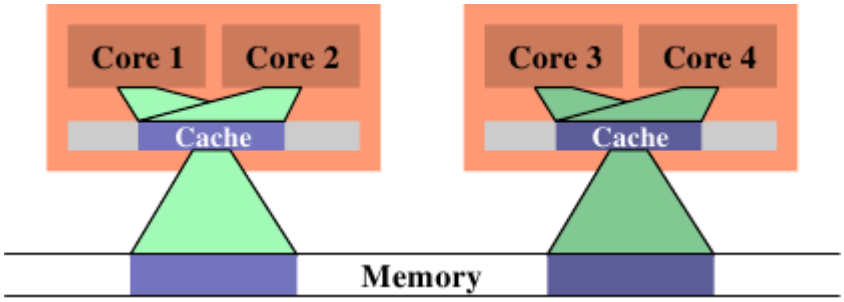


Figure 6.14: Efficient Scheduling

In Figure 6.14 we see how it should ideally look like. Now the total cache size in use is reduced since now core 1 and 2 and core 3 and 4 work on the same data. The data sets have to be read from memory only once.

This is a simple example but, by extension, it applies to many situations. As mentioned before, the scheduler in the kernel has no insight into the use of data, so the programmer has to ensure that scheduling is done efficiently. There are not many kernel interfaces available to communicate this requirement. In fact, there is only one: defining thread affinity.

Thread affinity means assigning a thread to one or more cores. The scheduler will then choose among those cores (only) when deciding where to run the thread. Even if other cores are idle they will not be considered. This might sound like a disadvantage, but it is the price one has to pay. If too many threads exclusively run on a set of cores the remaining cores might mostly be idle and there is nothing one can do except change the affinity. By default threads can run on any core.

There are a number of interfaces to query and change the affinity of a thread:

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t size, const cpu_set_t *cpuset);
int sched_getaffinity(pid_t pid, size_t size, cpu_set_t *cpuset);
```

These two interfaces are meant to be used for single-threaded code.

The pid argument specifies which process's affinity should be changed or determined. The caller obviously needs appropriate privileges to do this. The second and third parameter specify the bitmask for the cores. The first function requires the bitmask to be filled in so that it can set the affinity. The second fills in the bitmask with the scheduling information of the selected thread. The interfaces are declared in <sched.h>.

The cpu_set_t type is also defined in that header, along with a number of macros to manipulate and use objects of this type.

```
#define _GNU_SOURCE
#include <sched.h>

#define CPU_SETSIZE
#define CPU_SET(cpu, cpusetp)
#define CPU_CLR(cpu, cpusetp)
#define CPU_ZERO(cpusetp)
#define CPU_ISSET(cpu, cpusetp)
#define CPU_COUNT(cpusetp)
```

CPU_SETSIZE specifies how many CPUs can be represented in the data structure. The other three macros manipulate cpu_set_t objects. To initialize an object CPU_ZERO should be used; the other two macros should be used to select or deselect individual cores. CPU_ISSET tests whether a specific processor is part of the set. CPU_COUNT returns the number of cores selected in the set. The cpu_set_t type provide a reasonable default value for the upper limit on the number of CPUs. Over time it certainly will prove too small; at that point the type will be adjusted. This means programs always have to keep the size in mind. The above convenience macros implicitly handle the size according to the definition of cpu_set_t. If more dynamic size handling is needed an extended set of macros should be used:

```
#define _GNU_SOURCE
#include <sched.h>
```

```
#define CPU_SET_S(cpu, setsize, cpusetp)
#define CPU_CLR_S(cpu, setsize, cpusetp)
#define CPU_ZERO_S(setsize, cpusetp)
#define CPU_ISSET_S(cpu, setsize, cpusetp)
#define CPU_COUNT_S(setsize, cpusetp)
```

These interfaces take an additional parameter with the size. To be able to allocate dynamically sized CPU sets three macros are provided:

```
#define _GNU_SOURCE
#include <sched.h>

#define CPU_ALLOC_SIZE(count)
#define CPU_ALLOC(count)
#define CPU_FREE(cpuset)
```

The `CPU_ALLOC_SIZE` macro returns the number of bytes which have to be allocated for a `cpu_set_t` structure which can handle `count` CPUs. To allocate such a block the `CPU_ALLOC` macro can be used. The memory allocated this way should be freed with `CPU_FREE`. The functions will likely use `malloc`and `free` behind the scenes but this does not necessarily have to remain this way.

Finally, a number of operations on CPU set objects are defined:

```
#define _GNU_SOURCE
#include <sched.h>

#define CPU_EQUAL(cpuset1, cpuset2)
#define CPU_AND(destset, cpuset1, cpuset2)
#define CPU_OR(destset, cpuset1, cpuset2)
#define CPU_XOR(destset, cpuset1, cpuset2)
#define CPU_EQUAL_S(setsize, cpuset1, cpuset2)
#define CPU_AND_S(setsize, destset, cpuset1, cpuset2)
#define CPU_OR_S(setsize, destset, cpuset1, cpuset2)
#define CPU_XOR_S(setsize, destset, cpuset1, cpuset2)
```

These two sets of four macros can check two sets for equality and perform logical AND, OR, and XOR operations on sets. These operations come in handy when using some of the libNUMA functions (see Section 12).

A process can determine on which processor it is currently running using the `sched_getcpu` interface:

```
#define _GNU_SOURCE
#include <sched.h>
int sched_getcpu(void);
```

The result is the index of the CPU in the CPU set. Due to the nature of scheduling this number cannot always be 100% correct. The thread might have been moved to a different CPU between the time the result was returned and when the thread returns to userlevel. Programs always have to take this possibility of inaccuracy into account. More important is, in any case, the set of CPUs the thread is allowed to run on. This set can be retrieved using `sched_getaffinity`. The set is inherited by child threads and processes. Threads cannot rely on the set to be stable over the lifetime. The affinity mask can be set from the outside (see the `pid` parameter in the prototypes above); Linux also supports CPU hot-plugging which means CPUs can vanish from the system—and, therefore, also from the affinity CPU set.

In multi-threaded programs, the individual threads officially have no process ID as defined by POSIX and, therefore, the two functions above cannot be used. Instead `<pthread.h>` declares four different interfaces:

```
#define _GNU_SOURCE
#include <pthread.h>

int pthread_setaffinity_np(pthread_t th, size_t size,
                           const cpu_set_t *cpuset);
int pthread_getaffinity_np(pthread_t th, size_t size, cpu_set_t
*cpuset);
int pthread_attr_setaffinity_np(pthread_attr_t *at,
                                size_t size, const cpu_set_t *cpuset);
int pthread_attr_getaffinity_np(pthread_attr_t *at, size_t size,
                                cpu_set_t *cpuset);
```

The first two interfaces are basically equivalent to the two we have already seen, except that they take a thread handle in the first parameter instead of a process ID. This allows addressing individual threads in a process. It also means that these interfaces cannot be used from another process, they are strictly for intra-process use. The third and fourth interfaces use a thread attribute. These attributes are used when creating a new thread. By setting the attribute, a thread can be scheduled from the start on a specific set of CPUs. Selecting the target processors this early—instead of after the thread already started—can be of advantage on many different levels, including (and especially) memory allocation (see NUMA in Section 6.5).

Speaking of NUMA, the affinity interfaces play a big role in NUMA programming, too. We will come back to that case shortly.

So far, we have talked about the case where the working set of two threads overlaps such that having both threads on the same core makes sense. The opposite can be true, too. If two threads work on separate data sets, having them scheduled on the same core can be a problem. Both threads fight for the same cache, thereby reducing each others effective use of the cache. Second, both data sets have to be loaded into the same cache; in effect this increases the amount of data that has to be loaded and, therefore, the available bandwidth is cut in half.

The solution in this case is to set the affinity of the threads so that they cannot be scheduled on the same core. This is the opposite from the previous situation, so it is important to understand the situation one tries to optimize before making any changes.

Optimizing for cache sharing to optimize bandwidth is in reality an aspect of NUMA programming which is covered in the next section. One only has to extend the notion of "memory" to the caches. This will become ever more important once the number of levels of cache increases. For this reason, the solution to multi-core scheduling is available in the NUMA support library. See the code samples in Section 12 for ways to determine the affinity masks without hardcoding system details or diving into the depth of

the `/sys` filesystem.

6.5 NUMA Programming

For NUMA programming everything said so far about cache optimizations applies as well. The differences only start below that level. NUMA introduces different costs when accessing different parts of the address space. With uniform memory access we can optimize to minimize page faults (see Section 7.5) but that is about it. All pages are created equal.

NUMA changes this. Access costs can depend on the page which is accessed. Differing access costs also increase the importance of optimizing for memory page locality. NUMA is inevitable for most SMP machines since both Intel with CSI (for x86,x86-64, and IA-64) and AMD (for Opteron) use it. With an increasing number of cores per processor we are likely to see a sharp reduction of SMP systems being used (at least outside data centers and offices of people with terribly high CPU usage requirements). Most home machines will be fine with just one processor and hence no NUMA issues. But this a) does not mean programmers can ignore NUMA and b) it does not mean there are not related issues.

If one thinks about generalizations to NUMA one quickly realizes the concept extends to processor caches as well. Two threads on cores using the same cache will collaborate faster than threads on cores not sharing a cache. This is not a fabricated case:

- early dual-core processors had no L2 sharing.
- Intel's Core 2 QX 6700 and QX 6800 quad core chips, for instance, have two separate L2 caches.
- as speculated early, with more cores on a chip and the desire to unify caches, we will have more levels of caches.

Caches form their own hierarchy, and placement of threads on cores becomes important for sharing (or not) of caches. This is not very different from the problems NUMA is facing and, therefore, the two concepts can be unified. Even people only interested in non-SMP machines should therefore read this section.

In Section 5.3 we have seen that the Linux kernel provides a lot of information which is useful—and needed—in NUMA programming. Collecting this information is not that easy, though. The currently available NUMA library on Linux is wholly inadequate for this purpose. A much more suitable version is currently under construction by the author.

The existing NUMA library, `libnuma`, part of the numactl package, provides no access to system architecture information. It is only a wrapper around the available system calls together with some convenience interfaces for commonly used operations. The system calls available on Linux today are:

`mbind`
: Select binding for specified memory pages to nodes.

`set_mempolicy`
: Set the default memory binding policy.

`get_mempolicy`

Get the default memory binding policy.

`migrate_pages`
: Migrate all pages of a process on a given set of nodes to a different set of nodes.

`move_pages`
: Move selected pages to given node or request node information about pages.

These interfaces are declared in `<numaif.h>` which comes along with the `libnuma` library. Before we go into more details we have to understand the concept of memory policies.

6.5.1 Memory Policy

The idea behind defining a memory policy is to allow existing code to work reasonably well in a NUMA environment without major modifications. The policy is inherited by child processes, which makes it possible to use the numactl tool. This tool can be used to, among other things, start a program with a given policy.

The Linux kernel supports the following policies:

`MPOL_BIND`
: Memory is allocated only from the given set of nodes. If this is not possible allocation fails.

`MPOL_PREFERRED`
: Memory is preferably allocated from the given set of nodes. If this fails memory from other nodes is considered.

`MPOL_INTERLEAVE`
: Memory is allocated equally from the specified nodes. The node is selected either by the offset in the virtual memory region for VMA-based policies, or through a free-running counter for task-based policies.

`MPOL_DEFAULT`
: Choose the allocation based on the default for the region.

This list seems to recursively define policies. This is half true. In fact, memory policies form a hierarchy (see Figure 6.15).

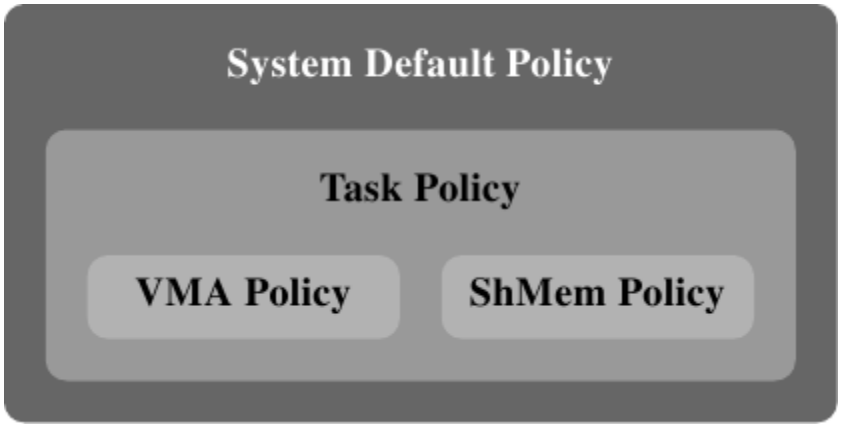


Figure 6.15: Memory Policy Hierarchy

If an address is covered by a VMA policy then this policy is used. A special kind of policy is used for shared memory segments. If no policy for the specific address is present, the task's policy is used. If this is also not present the system's default policy is used.

The system default is to allocate memory local to the thread requesting the memory. No task and VMA policies are provided by default. For a process with multiple threads the local node is the “home” node, the one which first ran the process. The system calls mentioned above can be used to select different policies.

6.5.2 Specifying Policies

The `set_mempolicy` call can be used to set the task policy for the current thread (task in kernel-speak). Only the current thread is affected, not the entire process.

```
#include <numaif.h>

long set_mempolicy(int mode,
                   unsigned long *nodemask,
                   unsigned long maxnode);
```

The `mode` parameter must be one of the `MPOL_*` constants introduced in the previous section. The `nodemask` parameter specifies the memory nodes to use and `maxnode` is the number of nodes (i.e., bits) in `nodemask`. If `MPOL_DEFAULT` is

used the nodemask parameter is ignored. If a null pointer is passed as nodemask for MPOL_PREFERRED the local node is selected.

Otherwise MPOL_PREFERRED uses the lowest node number with the corresponding bit set in nodemask.

Setting a policy does not have any effect on already-allocated memory. Pages are not automatically migrated; only future allocations are affected. Note the difference between memory allocation and address space reservation: an address space region established using mmap is usually not automatically allocated. The first read or write operation on the memory region will allocate the appropriate page. If the policy changes between accesses to different pages of the same address space region, or if the policy allows allocation of memory from different nodes, a seemingly uniform address space region might be scattered across many memory nodes.

6.5.3 Swapping and Policies

If physical memory runs out, the system has to drop clean pages and save dirty pages to swap. The Linux swap implementation discards node information when it writes pages to swap. That means when the page is reused and paged in the node which is used will be chosen from scratch. The policies for the thread will likely cause a node which is close to the executing processors to be chosen, but the node might be different from the one used before.

This changing association means that the node association cannot be stored by a program as a property of the page. The association can change over time. For pages which are shared with other processes this can also happen because a process asks for it (see the discussion of mbindbelow). The kernel by itself can migrate pages if one node runs out of space while other nodes still have free space.

Any node association the user-level code learns about can therefore be true for only a short time. It is more of a hint than absolute information. Whenever accurate knowledge is required the get_mempolicy interface should be used (see Section 6.5.5).

6.5.4 VMA Policy

To set the VMA policy for an address range a different interface has to be used:

```
#include <numaif.h>

long mbind(void *start, unsigned long len,
           int mode,
           unsigned long *nodemask,
           unsigned long maxnode,
           unsigned flags);
```

This interface registers a new VMA policy for the address range [start, start + len). Since memory handling operates on pages the start address must be page-aligned. The len value is rounded up to the next page size.

The mode parameter specifies, again, the policy; the values must be chosen from the list in Section 6.5.1. As with set_mempolicy, the nodemaskparameter is only used for some policies. Its handling is identical.

The semantics of the mbind interface depends on the value of the flags parameter. By default, if flags is zero, the system call sets the VMA policy for the address range. Existing mappings are not affected. If this is not sufficient there are currently three flags to modify this behavior; they can be selected individually or together:

MPOL_MF_STRICT

The call to mbind will fail if not all pages are on the nodes specified by nodemask. In case this flag is used together with MPOL_MF_MOVEand/or MPOL_MF_MOVEALL the call will fail if any page cannot be moved.

MPOL_MF_MOVE

The kernel will try to move any page in the address range allocated on a node not in the set specified by nodemask. By default, only pages used exclusively by the current process's page tables are moved.

MPOL_MF_MOVEALL

Like MPOL_MF_MOVE but the kernel will try to move all pages, not just those used by the current process's page tables alone. This has system-wide implications since it influences the memory access of other processes—which are possibly not owned by the same user—as well. Therefore MPOL_MF_MOVEALL is a privileged operation

(CAP_NICE capability is needed).

Note that support for MPOL_MF_MOVE and MPOL_MF_MOVEALL was added only in the 2.6.16 Linux kernel.

Calling mbind without any flags is most useful when the policy for a newly reserved address range has to be specified before any pages are actually allocated.

```
void *p = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_ANON, -1, 0);
if (p != MAP_FAILED)
  mbind(p, len, mode, nodemask, maxnode, 0);
```

This code sequence reserve an address space range of len bytes and

specifies that the policy mode referencing the memory nodes

in nodemask should be used. Unless the MAP_POPULATE flag is used with mmap,

no memory will have been allocated by the time of the mbind call and, therefore, the new policy applies to all pages in that address space region.

The MPOL_MF_STRICT flag alone can be used to determine whether any page

in the address range described by the start and len parameters to mbind is

allocated on nodes other than those specified by nodemask. No allocated pages are changed. If all pages are allocated on the specified nodes, the VMA policy for the address space region will be changed according to mode.

Sometimes the rebalancing of memory is needed, in which case it might be necessary to move pages allocated on one node to another node.

Calling`mbind` with `MPOL_MF_MOVE` set makes a best effort to achieve that. Only pages which are solely referenced by the process's page table tree are considered for moving. There can be multiple users in the form of threads or other processes which share that part of the page table tree. It is not possible to affect other processes which happen to map the same data. These pages do not share the page table entries.

If both `MPOL_MF_STRICT` and `MPOL_MF_MOVE` are passed to `mbind` the kernel will try to move all pages which are not allocated on the specified nodes. If this is not possible the call will fail. Such a call might be useful to determine whether there is a node (or set of nodes) which can house all the pages. Several combinations can be tried in succession until a suitable node is found.

The use of `MPOL_MF_MOVEALL` is harder to justify unless running the current process is the main purpose of the computer. The reason is that even pages that appear in multiple page tables are moved. That can easily affect other processes in a negative way. This operation should thus be used with caution.

6.5.5 Querying Node Information

The `get_mempolicy` interface can be used to query a variety of facts about the state of NUMA for a given address.

```
#include <numaif.h>
long get_mempolicy(int *policy,
             const unsigned long *nmask,
             unsigned long maxnode,
             void *addr, int flags);
```

When `get_mempolicy` is called without a flag set in `flags`, the information about the policy for address `addr` is stored in the word pointed to by`policy` and in the bitmask for the nodes pointed to by `nmask`. If `addr` falls into an address space region for which a VMA policy has been specified,

information about that policy is returned. Otherwise information about the task policy or, if necessary, system default policy will be returned.

If the `MPOL_F_NODE` flag is set in `flags`, and the policy governing `addr` is `MPOL_INTERLEAVE`, the value stored in the word pointed to by `policy` is the index of the node on which the next allocation is going to happen. This information can potentially be used to set the affinity of a thread which is going to work on the newly-allocated memory. This might be a less costly way to achieve proximity, especially if the thread has yet to be created.

The `MPOL_F_ADDR` flag can be used to retrieve yet another completely different data item. If this flag is used, the value stored in the word pointed to by `policy` is the index of the memory node on which the memory for the page containing `addr` has been allocated. This information can be used to make decisions about possible page migration, to decide which thread could work on the memory location most efficiently, and many more things.

The CPU—and therefore memory node—a thread is using is much more volatile than its memory allocations. Memory pages are, without explicit requests, only moved in extreme circumstances. A thread can be assigned to another CPU as the result of rebalancing the CPU loads. Information about the current CPU and node might therefore be short-lived. The scheduler will try to keep the thread on the same CPU, and possibly even on the same core, to minimize performance losses due to cold caches. This means it is useful to look at the current CPU and node information; one only must avoid assuming the association will not change.

libNUMA provides two interfaces to query the node information for a given virtual address space range:

```
#include <libNUMA.h>

int NUMA_mem_get_node_idx(void *addr);
int NUMA_mem_get_node_mask(void *addr,
                           size_t size,
                           size_t __destsize,
                           memnode_set_t *dest);
```

NUMA_mem_get_node_mask sets in dest the bits for all memory nodes on which the pages in the range [addr, addr+size) are (or would be) allocated, according to the governing policy. NUMA_mem_get_node only looks at the address addr and returns the index of the memory node on which this address is (or would be) allocated. These interfaces are simpler to use than get_mempolicy and probably should be preferred.

The CPU currently used by a thread can be queried using sched_getcpu (see Section 6.4.3). Using this information, a program can determine the memory node(s) which are local to the CPU using the NUMA_cpu_to_memnode interface from libNUMA:

```
#include <libNUMA.h>

int NUMA_cpu_to_memnode(size_t cpusetsize,
                        const cpu_set_t *cpuset,
                        size_t memnodesize,
                        memnode_set_t *
                        memnodeset);
```

A call to this function will set (in the memory node set pointed to by the fourth parameter) all the bits corresponding to memory nodes which are local to any of the CPUs in the set pointed to by the second parameter. Just like CPU information itself, this information is only correct until the configuration of the machine changes (for instance, CPUs get removed and added).

The bits in the memnode_set_t objects can be used in calls to the low-level functions like get_mempolicy. It is more convenient to use the other functions in libNUMA. The reverse mapping is available through:

```
#include <libNUMA.h>

int NUMA_memnode_to_cpu(size_t memnodesize,
                        const memnode_set_t *
                        memnodeset,
```

```
                size_t cpusetsize,
                cpu_set_t *cpuset);
```

The bits set in the resulting `cpuset` are those of the CPUs local to any of the memory nodes with corresponding bits set in `memnodeset`. For both interfaces, the programmer has to be aware that the information can change over time (especially with CPU hot-plugging). In many situations, a single bit is set in the input bit set, but it is also meaningful, for instance, to pass the entire set of CPUs retrieved by a call to`sched_getaffinity` to `NUMA_cpu_to_memnode` to determine which are the memory nodes the thread ever can have direct access to.

6.5.6 CPU and Node Sets

Adjusting code for SMP and NUMA environments by changing the code to use the interfaces described so far might be prohibitively expensive (or impossible) if the sources are not available. Additionally, the system administrator might want to impose restrictions on the resources a user and/or process can use. For these situations the Linux kernel supports so-called CPU sets. The name is a bit misleading since memory nodes are also covered. They also have nothing to do with the `cpu_set_t` data type.

The interface to CPU sets is, at the moment, a special filesystem. It is usually not mounted (so far at least). This can be changed with

```
mount -t cpuset none /dev/cpuset
```

Of course the mount point `/dev/cpuset` must exist. The content of this directory is a description of the default (root) CPU set. It comprises initially all CPUs and all memory nodes. The `cpus` file in that directory shows the CPUs in the CPU set, the `mems` file the memory nodes, the `tasks`file the processes.

To create a new CPU set one simply creates a new directory somewhere in the hierarchy. The new CPU set will inherit all settings from the parent. Then the CPUs and memory nodes for new CPU set can be changed by

writing the new values into the `cpus` and `mems` pseudo files in the new directory.

If a process belongs to a CPU set, the settings for the CPUs and memory nodes are used as masks for the affinity and memory policy bitmasks. That means the program cannot select any CPU in the affinity mask which is not in the `cpus` file for the CPU set the process is using (i.e., where it is listed in the `tasks` file). Similarly for the node masks for the memory policy and the `mems` file.

The program will not experience any errors unless the bitmasks are empty after the masking, so CPU sets are an almost-invisible means to control program execution. This method is especially efficient on large machines with lots of CPUs and/or memory nodes. Moving a process into a new CPU set is as simple as writing the process ID into the `tasks` file of the appropriate CPU set.

The directories for the CPU sets contain a number of other files which can be used to specify details like behavior under memory pressure and exclusive access to CPUs and memory nodes. The interested reader is referred to the file `Documentation/cpusets.txt` in the kernel source tree.

6.5.7 Explicit NUMA Optimizations

All the local memory and affinity rules cannot help out if all threads on all the nodes need access to the same memory regions. It is, of course, possible to simply restrict the number of threads to a number supportable by the processors which are directly connected to the memory node. This does not take advantage of SMP NUMA machines, though, and is therefore not a real option.

If the data in question is read-only there is a simple solution: replication. Each node can get its own copy of the data so that no inter-node accesses are necessary. Code to do this can look like this:

```
void *local_data(void) {
  static void *data[NNODES];
  int node =
    NUMA_memnode_self_current_idx();
```

```
  if (node == -1)
    /* Cannot get node, pick one.  */
    node = 0;
  if (data[node] == NULL)
    data[node] = allocate_data();
  return data[node];
}
void worker(void) {
  void *data = local_data();
  for (...)
    compute using data
}
```

In this code the function `worker` prepares by getting a pointer to the local copy of the data by a call to `local_data`. Then it proceeds with the loop, which uses this pointer. The `local_data` function keeps a list of the already allocated copies of the data around. Each system has a limited number of memory nodes, so the size of the array with the pointers to the per-node memory copies is limited in size. The `NUMA_memnode_system_count`function from libNUMA returns this number. If the pointer for the current node, as determined by the `NUMA_memnode_self_current_idx` call, is not yet known a new copy is allocated.

It is important to realize that nothing terrible happens if the threads get scheduled onto another CPU connected to a different memory node after the `sched_getcpu` system call. It just means that the accesses using the `data` variable in `worker` access memory on another memory node. This slows the program down until `data` is computed anew, but that is all. The kernel will always avoid gratuitous rebalancing of the per-CPU run queues. If such a transfer happens it is usually for a good reason and will not happen again for the near future.

Things are more complicated when the memory area in question is writable. Simple duplication will not work in this case. Depending on the exact situation there might a number of possible solutions.

For instance, if the writable memory region is used to accumulate results, it might be possible to first create a separate region for each memory node in which the results are accumulated. Then, when this work is done, all the per-node memory regions are combined to get the total result. This technique can work even if the work never really stops, but intermediate results are needed. The requirement for this approach is that the accumulation of a result is stateless, i.e., it does not depend on the previously collected results.

It will always be better, though, to have direct access to the writable memory region. If the number of accesses to the memory region is substantial, it might be a good idea to force the kernel to migrate the memory pages in question to the local node. If the number of accesses is really high, and the writes on different nodes do not happen concurrently, this could help. But be aware that the kernel cannot perform miracles: the page migration is a copy operation and as such it is not cheap. This cost has to be amortized.

6.5.8 Utilizing All Bandwidth

The numbers in Figure 5.4 show that access to remote memory when the caches are ineffective is not measurably slower than access to local memory. This means a program could possibly save bandwidth to the local memory by writing data it does not have to read again into memory attached to another processor. The bandwidth of the connection to the DRAM modules and the bandwidth of the interconnects are mostly independent, so parallel use could improve overall performance.

Whether this is really possible depends on many factors. One really has to be sure that caches are ineffective since otherwise the slowdown related to remote accesses is measurable. Another big problem is whether the remote node has any needs for its own memory bandwidth. This possibility must be examined in detail before the approach is taken. In theory, using all the bandwidth available to a processor can have positive effects. A family 10h Opteron processor can be directly connected to up to four other processors. Utilizing all that additional bandwidth, perhaps coupled with appropriate prefetches (especially `prefetchw`) could lead to improvements if the rest of the system plays along.

This article was contributed by Ulrich Drepper

[*Editor's note: welcome to part 7 of Ulrich Drepper's "What every programmer should know about memory"; this section is concerned with tools which can help programmers optimize the memory performance of*

their code. Those who have not seen the previous parts of this document can find them from part 1.]

7 Memory Performance Tools

A wide variety of tools is available to help programmers understand the cache and memory use of a program. Modern processors have performance monitoring hardware that can be used. Some events are hard to measure exactly, so there is also room for simulation. When it comes to higher-level functionality, there are special tools to monitor the execution of a process. We will introduce a set of commonly used tools available on most Linux systems.

7.1 Memory Operation Profiling

Profiling memory operations requires collaboration from the hardware. It is possible to gather some information in software alone, but this is either coarse-grained or merely a simulation. Examples of simulation will be shown in Section 7.2 and Section 7.5. Here we will concentrate on measurable memory effects.

Access to performance monitoring hardware on Linux is provided by oprofile. Oprofile provides continuous profiling capabilities as first described in [continuous]; it performs statistical, system-wide profiling with an easy-to-use interface. Oprofile is by no means the only way the performance measurement functionality of processors can be used; Linux developers are working on pfmon which might at some point be sufficiently widely deployed to warrant being described here, too.

The interface oprofile provides is simple and minimal but also pretty low-level, even if the optional GUI is used. The user has to select among the events the processor can record. The architecture manuals for the processors describe the events but, oftentimes, it requires extensive knowledge about the processors themselves to interpret the data. Another problem is the interpretation of the collected data. The performance measurement counters are absolute values and can grow arbitrarily. How high is too high for a given counter?

A partial answer to this problem is to avoid looking at the absolute values and, instead, relate multiple counters to each other. Processors can monitor more than one event; the ratio of the collected absolute values can then be examined. This gives nice, comparable results. Often the divisor is a measure of processing time, the number of clock cycles or the number of

instructions. As an initial stab at program performance, relating just these two numbers by themselves is useful.

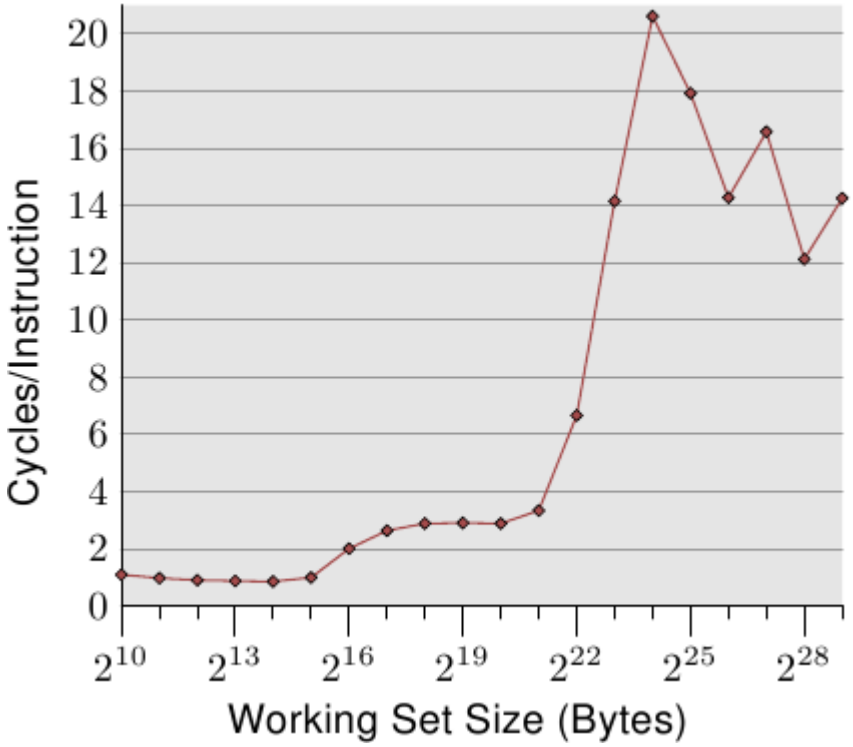


Figure 7.1: Cycles per Instruction (Follow Random)

Figure 7.1 shows the Cycles Per Instruction (CPI) for the simple random "Follow" test case for the various working set sizes. The names of the events to collect this information for most Intel processor

are `CPU_CLK_UNHALTED` and `INST_RETIRED`. As the names suggest, the former

counts the clock cycles of the CPU and the latter the number of instructions. We see a picture similar to the cycles per list element measurements we used. For small working set sizes the ratio is 1.0 or even lower. These measurements were made on a Intel Core 2 processor, which is multi-scalar and can work on several instructions at once. For a program which is not limited by memory bandwidth, the ratio can be significantly below 1.0 but, in this case, 1.0 is pretty good.

Once the L1d is no longer large enough to hold the working set, the CPI jumps to just below 3.0. Note that the CPI ratio averages the penalties for accessing L2 over all instructions, not just the memory instructions. Using the cycles for list element data, it can be worked out how many instructions per list element are needed. If even the L2 cache is not sufficient, the CPI ratio jumps to more than 20. These are expected results.

But the performance measurement counters are supposed to give more insight into what is going on in the processor. For this we need to think about processor implementations. In this document, we are concerned with cache handling details, so we have to look at events related to the caches. These events, their names, and what they count, are processor-specific. This is where oprofile is currently hard to use, irrespective of the simple user interface: the user has to figure out the performance counter details by her/himself. In Section 10 we will see details about some processors.

For the Core 2 processor the events to look for are `L1D_REPL`, `DTLB_MISSES`, and `L2_LINES_IN`. The latter can measure both all misses and misses caused by instructions instead of hardware prefetching. The results for the random "Follow" test can be seen in Figure 7.2.

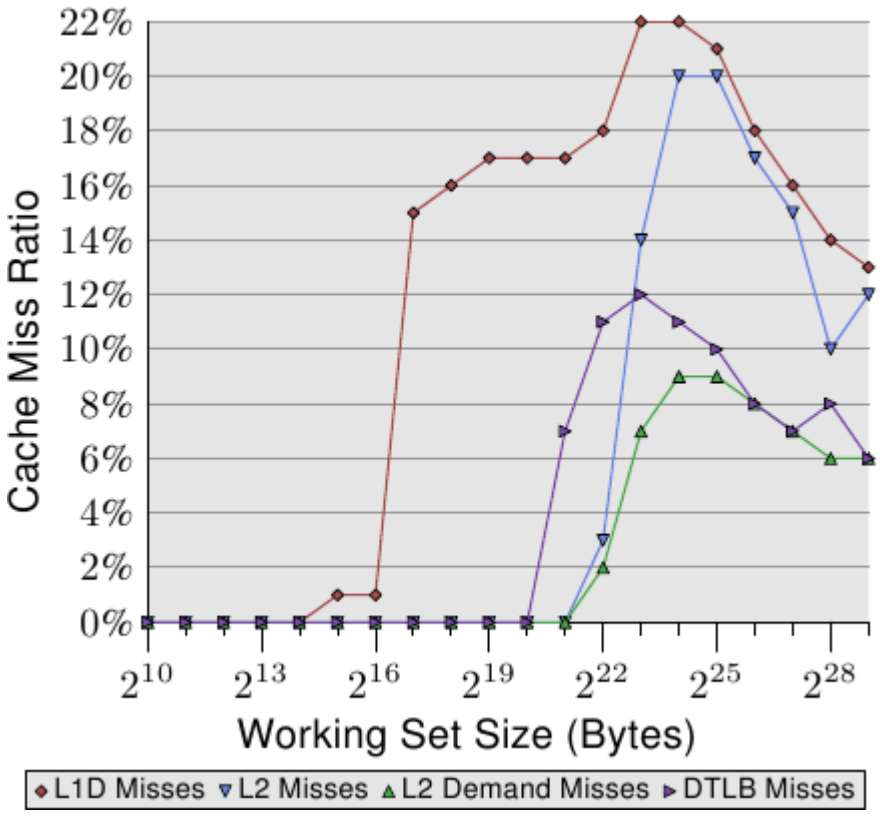


Figure 7.2: Measured Cache Misses (Follow Random)

All ratios are computed using the number of retired instructions (`INST_RETIRED`). This means that instructions not touching memory are also counted, which, in turn, means that the number of instructions which do touch memory and which suffer a cache miss is even higher than shown in the graph.

The L1d misses tower over all the others since an L2 miss implies, for Intel processors, an L1d miss due to the use of inclusive caches. The processor has 32k of L1d and so we see, as expected, the L1d rate go up from zero at about that working set size (there are other uses of the cache beside the list data structure, which means the increase happens between the 16k and 32k mark). It is interesting to see that the hardware prefetching can keep the miss rate at 1% for a working set size up to and including 64k. After that the L1d rate skyrockets.

The L2 miss rate stays zero until the L2 is exhausted; the few misses due to other uses of L2 do not influence the numbers much. Once the size of L2 (2^{21} bytes) is exceeded, the miss rates rise. It is important to notice that the L2 demand miss rate is nonzero. This indicates that the hardware prefetcher does not load all the cache lines needed by instructions later. This is expected, the randomness of the accesses prevents perfect prefetching. Compare this with the data for the sequential read in Figure 7.3.

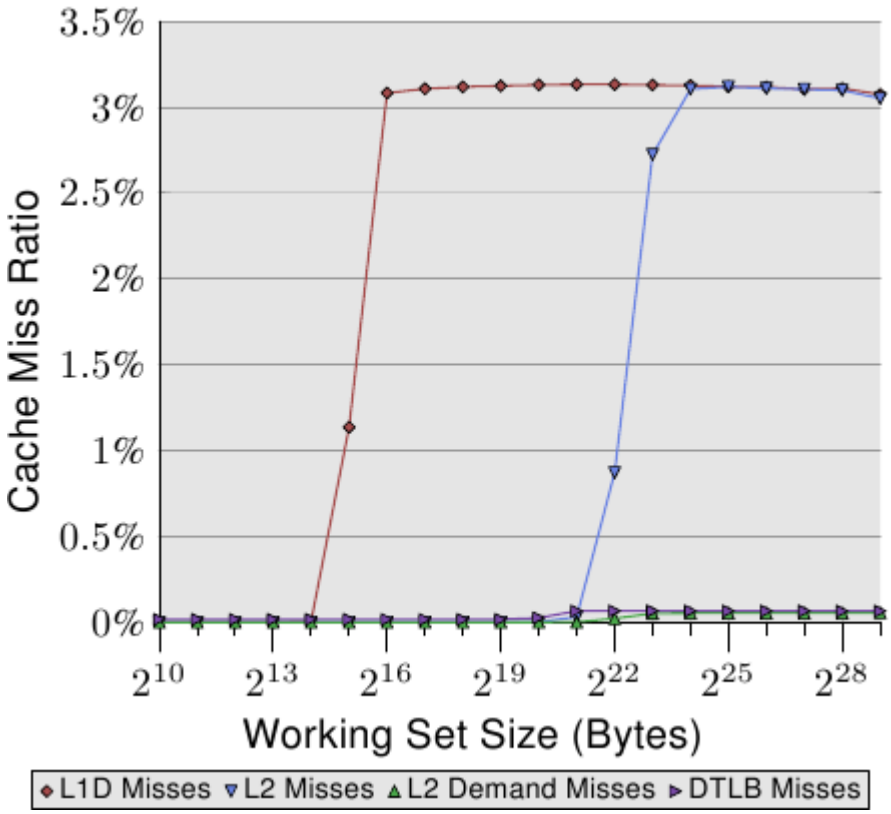


Figure 7.3: Measured Cache Misses (Follow Sequential)

In this graph we can see that the L2 demand miss rate is basically zero (note the scale of this graph is different from Figure 7.2). For the sequential access case, the hardware prefetcher works perfectly: almost all L2 cache misses are caused by the prefetcher. The fact that the L1d and L2 miss rates are the same shows that all L1d cache misses are handled by the L2 cache without

further delays. This is the ideal case for all programs but it is, of course, hardly ever achievable.

The fourth line in both graphs is the DTLB miss rate (Intel has separate TLBs for code and data, DTLB is the data TLB). For the random access case, the DTLB miss rate is significant and contributes to the delays. What is interesting is that the DTLB penalties set in before the L2 misses. For the sequential access case the DTLB costs are basically zero.

Going back to the matrix multiplication example in Section 6.2.1 and the example code in Section 9.1, we can make use of three more counters.

The `SSE_PRE_MISS`, `SSE_PRE_EXEC`, and `LOAD_HIT_PRE` counters can be used to see how effective the software prefetching is. If the code in Section 9.1 is run we get the following results:

Description	Ratio
Useful NTA prefetches	2.84%
Late NTA prefetches	2.65%

The low useful NTA (non-temporal aligned) prefetch ratio indicates that many prefetch instructions are executed for cache lines which are already loaded, so no work is needed. This means the processor wastes time to decode the prefetch instruction and look up the cache. One cannot judge the code too harshly, though. Much depends on the size of the caches of the processor used; the hardware prefetcher also plays a role.

The low late NTA prefetch ratio is misleading. The ratio means that 2.65% of all prefetch instructions are issued too late. The instruction which needs the data is executed before the data could be prefetched into the cache. It must be kept in mind that only 2.84%+2.65%=5.5% of the prefetch instructions were of any use. Of the NTA prefetch instructions which are useful, 48% did not finish in time. The code therefore can be optimized further:

- most of the prefetch instructions are not needed.
- the use of the prefetch instruction can be adjusted to match the hardware better.

It is left as an exercise to the reader to determine the best solution for the available hardware. The exact hardware specification plays a big role. On Core 2 processors the latency of the SSE arithmetic operations is 1 cycle. Older versions had a latency of 2 cycles, meaning that the hardware prefetcher and the prefetch instructions had more time to bring in the data.

To determine where prefetches might be needed—or are unnecessary—one can use the opannotate program. It lists the source or assembler code of the program and shows the instructions where the event was recognized. Note that there are two sources of vagueness:

1. Oprofile performs stochastic profiling. Only every N^{th} event (where N is a per-event threshold with an enforced minimum) is recorded to avoid slowing down operation of the system too much. There might be lines which cause 100 events and yet they might not show up in the report.
2. Not all events are recorded accurately. For example, the instruction counter at the time a specific event was recorded might be incorrect. Processors being multi-scalar makes it hard to give a 100% correct answer. A few events on some processors are exact, though.

The annotated listings are useful for more than determining the prefetching information. Every event is recorded with the instruction pointer; it is therefore also possible to pinpoint other hot spots in the program. Locations which are the source of many `INST_RETIRED` events are executed frequently and deserve to be tuned. Locations where many cache misses are reported might warrant a prefetch instruction to avoid the cache miss.

One type of event which can be measured without hardware support is page faults. The OS is responsible for resolving page faults and, on those occasions, it also counts them. It distinguishes two kinds of page faults:

Minor Page Faults
: These are page faults for anonymous (i.e., not file-backed) pages which have not been used so far, for copy-on-write pages, and for other pages whose content is already in memory somewhere.

Major Page Faults
: Resolving them requires access to disk to retrieve the file-backed (or swapped-out) data.

Obviously, major page faults are significantly more expensive than minor page faults. But the latter are not cheap either. In either case an entry into the kernel is necessary, a new page must be found, the page must be cleared or populated with the appropriate data, and the page table tree must be modified accordingly. The last step requires synchronization with other tasks reading or modifying the page table tree, which might introduce further delays.

The easiest way to retrieve information about the page fault counts is to use the time tool. Note: use the real tool, not the shell builtin. The output can be

seen in Figure 7.4. {*The leading backslash prevents the use of the built-in command.*}

```
$ \time ls /etc
[...]
0.00user 0.00system 0:00.02elapsed 17%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (1major+335minor)pagefaults 0swaps
```

Figure 7.4: Output of the time utility

The interesting part here is the last line. The time tool reports one major and 335 minor page faults. The exact numbers vary; in particular, repeating the run immediately will likely show that there are now no major page faults at all. If the program performs the same action, and nothing changes in the environment, the total page fault count will be stable.

An especially sensitive phase with respect to page faults is program start-up. Each page which is used will produce a page fault; the visible effect (especially for GUI applications) is that the more pages that are used, the longer it takes for the program to start working. In Section 7.5 we will see a tool to measure this effect specifically.

Under the hood, the time tool uses the `rusage` functionality.

The `wait4` system call fills in a `struct rusage` object when the parent waits for a child to terminate; that is exactly what is needed for the time tool. But it is also possible for a process to request information about its own resource usage (that is where the name `rusage` comes from) or the resource usage of its terminated children.

```
#include <sys/resource.h>
int getrusage(__rusage_who_t who, struct rusage *usage)
```

The `who` parameter specifies which process the information is requested for.

Currently, `RUSAGE_SELF` and `RUSAGE_CHILDREN` are defined. The resource usage of the child processes is accumulated when each child terminates. It is a total value, not the usage of an individual child process. Proposals to allow requesting thread-specific information exist, so it is likely that we will see `RUSAGE_THREAD` in the near future. The `rusage`structure is defined to

contain all kinds of metrics, including execution time, the number of IPC messages sent and memory used, and the number of page faults. The latter information is available in the `ru_minflt` and `ru_majflt` members of the structure.

A programmer who tries to determine where her program loses performance due to page faults could regularly request the information and then compare the returned values with the previous results.

From the outside, the information is also visible if the requester has the necessary privileges. The pseudo file `/proc/<PID>/stat`, where `<PID>` is the process ID of the process we are interested in, contains the page fault numbers in the tenth to fourteenth fields. They are pairs of the process's and its childrens' cumulative minor and major page faults, respectively.

7.2 Simulating CPU Caches

While the technical description of how a cache works is relatively easy to understand, it is not so easy to see how an actual program behaves with respect to cache. Programmers are not directly concerned with the values of addresses, be they absolute nor relative. Addresses are determined, in part, by the linker and, in part, at runtime by the dynamic linker and the kernel. The generated assembly code is expected to work with all possible addresses and, in the source language, there is not even a hint of absolute address values left. So it can be quite difficult to get a sense for how a program is making use of memory. {*When programming close to the hardware this might be different, but this is of no concern to normal programming and, in any case, is only possible for special addresses such as memory-mapped devices.*}

CPU-level profiling tools such as oprofile (as described in Section 7.1) can help to understand the cache use. The resulting data corresponds to the actual hardware, and it can be collected relatively quickly if fine-grained collection is not needed. As soon as more fine-grained data is needed, oprofile is not usable anymore; the thread would have to be interrupted too often. Furthermore, to see the memory behavior of the program on different processors, one actually has to have such machines and execute the program on them. This is sometimes (often) not possible. One example is the data from Figure 3.8. To collect such data with oprofile one would have to have 24 different machines, many of which do not exist.

The data in that graph was collected using a cache simulator. This program, cachegrind, uses the valgrind framework, which was initially developed to

check for memory handling related problems in a program. The valgrind framework simulates the execution of a program and, while doing this, it allows various extensions, such as cachegrind, to hook into the execution framework. The cachegrind tool uses this to intercept all uses of memory addresses; it then simulates the operation of L1i, L1d, and L2 caches with a given size, cache line size, and associativity.

To use the tool a program must be run using valgrind as a wrapper:

```
valgrind --tool=cachegrind command arg
```

In this simplest form the program `command` is executed with the

parameter `arg` while simulating the three caches using sizes and associativity

corresponding to that of the processor it is running on. One part of the output is printed to standard error when the program is running; it consists of statistics of the total cache use as can be seen in Figure 7.5.

```
==19645== I   refs:      152,653,497
==19645== I1  misses:         25,833
==19645== L2i misses:          2,475
==19645== I1  miss rate:        0.01%
==19645== L2i miss rate:        0.00%
==19645==
==19645== D   refs:       56,857,129  (35,838,721 rd + 21,018,408 wr)
==19645== D1  misses:         14,187  (    12,451 rd +      1,736 wr)
==19645== L2d misses:          7,701  (     6,325 rd +      1,376 wr)
==19645== D1  miss rate:         0.0% (       0.0%   +        0.0%  )
==19645== L2d miss rate:         0.0% (       0.0%   +        0.0%  )
==19645==
==19645== L2 refs:            40,020  (    38,284 rd +      1,736 wr)
==19645== L2 misses:          10,176  (     8,800 rd +      1,376 wr)
==19645== L2 miss rate:          0.0% (       0.0%   +        0.0%  )
```

Figure 7.5: Cachegrind Summary Output

The total number of instructions and memory references is given, along with the number of misses they produce for the L1i/L1d and L2 cache, the miss rates, etc. The tool is even able to split the L2 accesses into instruction and data accesses, and all data cache uses are split in read and write accesses.

It becomes even more interesting when the details of the simulated caches

are changed and the results compared. Through the use of the —`I1`, —`D1`,

and —L2 parameters, cachegrind can be instructed to disregard the processor's cache layout and use that specified on the command line. For example:

```
valgrind --tool=cachegrind --L2=8388608,8,64 command arg
```

would simulate an 8MB L2 cache with 8-way set associativity and 64 byte cache line size. Note that the —L2 option appears on the command line before the name of the program which is simulated.

This is not all cachegrind can do. Before the process exits cachegrind writes out a file named cachegrind.out.XXXXX where XXXXX is the PID of the process. This file contains the summary information and detailed information about the cache use in each function and source file. The data can be viewed using the cg_annotate program.

The output this program produces contains the cache use summary which was printed when the process terminated, along with a detailed summary of the cache line use in each function of the program. Generating this per-function data requires that cg_annotate is able to match addresses to functions. This means debug information should be available for best results. Failing that, the ELF symbol tables can help a bit but, since internal symbols are not listed in the dynamic symbol table, the results are not complete. Figure 7.6 shows part of the output for the same program run as Figure 7.5.

```
--------------------------------------------------------------------------------
           Ir  I1mr I2mr          Dr   D1mr D2mr         Dw D1mw D2mw
file:function
--------------------------------------------------------------------------------
53,684,905     9     8  9,589,531     13     3 5,820,373    14
0  ???:_IO_file_xsputn@@GLIBC_2.2.5
36,925,729 6,267   114 11,205,241     74    18 7,123,370    22
0  ???:vfprintf
11,845,373    22     2  3,126,914     46    22 1,563,457     0
0  ???:__find_specmb
 6,004,482    40    10     697,872 1,744   484         0     0
0  ???:strlen
 5,008,448     3     2  1,450,093    370   118         0     0
0  ???:strcmp
```

```
 3,316,589      24      4      757,523       0      0     540,952      0
0   ???:_IO_padn
 2,825,541       3      3      290,222       5      1     216,403      0
0   ???:_itoa_word
 2,628,466       9      6      730,059       0      0     358,215      0
0   ???:_IO_file_overflow@@GLIBC_2.2.5
 2,504,211       4      4      762,151       2      0     598,833      3
0   ???:_IO_do_write@@GLIBC_2.2.5
 2,296,142      32      7      616,490      88      0     321,848      0      0
dwarf_child.c:__libdw_find_attr
 2,184,153  2,876     20      503,805      67      0     435,562      0
0   ???:__dcigettext
 2,014,243       3      3      435,512       1      1     272,195      4
0   ???:_IO_file_write@@GLIBC_2.2.5
 1,988,697  2,804      4      656,112     380      0      47,847      1
1   ???:getenv
 1,973,463      27      6      597,768      15      0     420,805      0      0
dwarf_getattrs.c:dwarf_getattrs
```

Figure 7.6: cg_annotate Output

The Ir, Dr, and Dw columns show the total cache use, not cache misses, which are shown in the following two columns. This data can be used to identify the code which produces the most cache misses. First, one probably would concentrate on L2 cache misses, then proceed to optimizing L1i/L1d cache misses.

cg_annotate can provide the data in more detail. If the name of a source file is given, it also annotates (hence the program's name) each line of the source file with the number of cache hits and misses corresponding to that line. This information allows the programmer to drill down to the exact line where cache misses are a problem. The program interface is a bit raw: as of this writing, the cachegrind data file and the source file must be in the same directory.

It should, at this point, be noted again: cachegrind is a simulator which does *not* use measurements from the processor. The actual cache implementation in the processor might very well be quite different. cachegrind simulates Least Recently Used (LRU) eviction, which is likely to be too expensive for caches with large associativity. Furthermore, the simulation does not take context switches and system calls into account, both of which can destroy large parts of L2 and must flush L1i and L1d. This causes the total number of cache misses to be lower than experienced in reality. Nevertheless, cachegrind is a nice tool to learn about a program's memory use and its problems with memory.

7.3 Measuring Memory Usage

Knowing how much memory a program allocates and possibly where the allocation happens is the first step to optimizing its memory use There are, fortunately, some easy-to-use programs available which do not even require that the program be recompiled or specifically modified.

For the first tool, called massif, it is sufficient to not strip the debug information which the compiler can automatically generate. It provides an overview of the accumulated memory use over time. Figure 7.7 shows an example of the generated output.

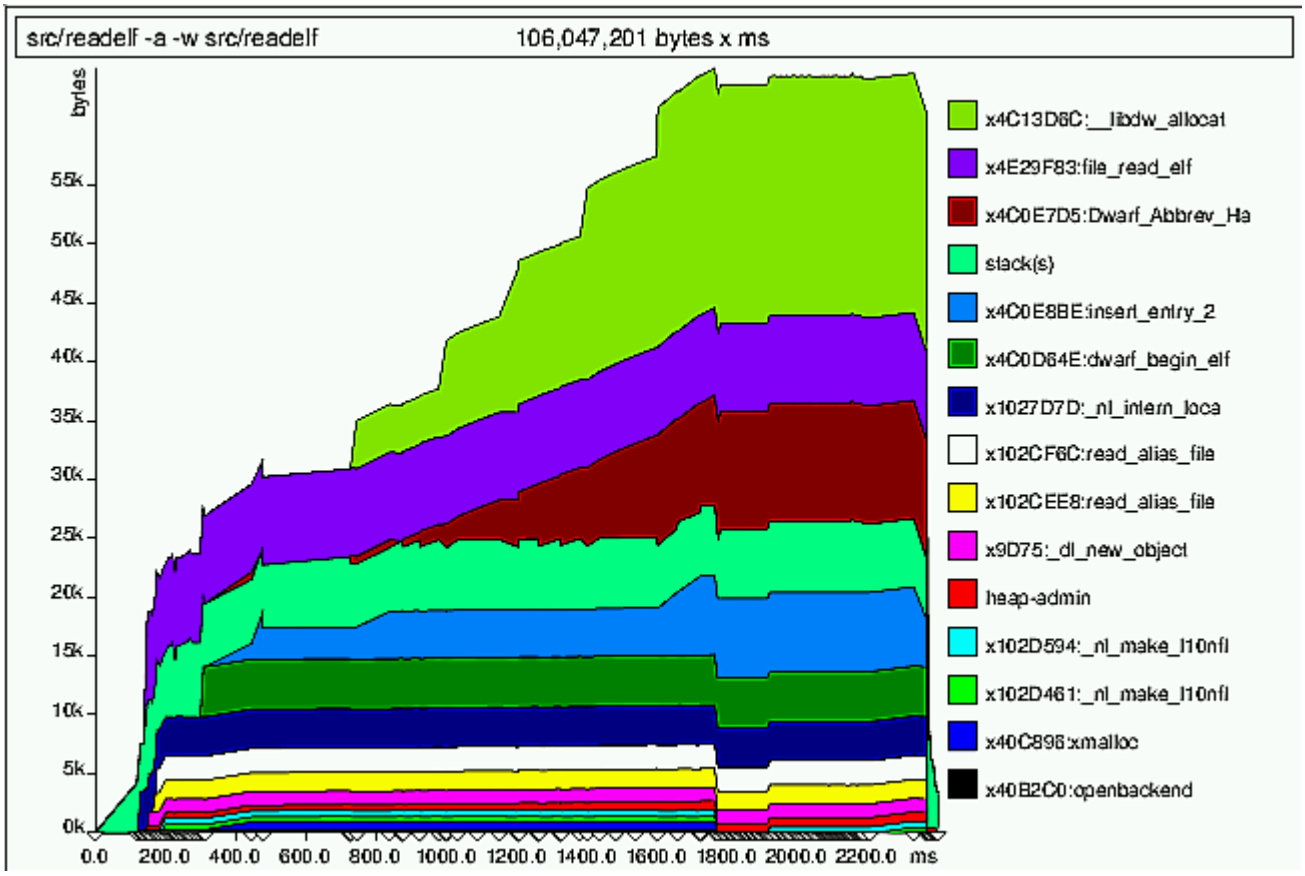


Figure 7.7: Massif Output

Like cachegrind (Section 7.2), massif is a tool using the valgrind infrastructure. It is started using

```
valgrind --tool=massif command arg
```

where `command arg` is the program which is observed and its parameter(s),

The program will be simulated and all calls to memory allocation functions are recognized. The call site is recorded along with a timestamp value; the

new allocation size is added to both the whole-program total and total for the specific call site. The same applies to the functions which free memory where, obviously, the size of the freed block is subtracted from the appropriated sums. This information can then be used to create a graph showing the memory use over the lifetime of the program, splitting each time value according to the location which requested the allocation. Before the process is terminated massif creates two files: `massif.XXXXX.txt` and `massif.XXXXX.ps`, where `XXXXX` in both cases is the PID of the process. The `.txt` file is a summary of the memory use for all call sites and the `.ps` is what can be seen in Figure 7.7.

Massif can also record the program's stack usage, which can be useful to determine the total memory footprint of an application. But this is not always possible. In some situations (some thread stacks or when `signaltstack` is used) the valgrind runtime cannot know about the limits of the stack . In these situations, it also does not make much sense to add these stacks' sizes to the total. There are several other situations where it makes no sense. If a program is affected by this, massif should be started with the addition option `—stacks=no`. Note, this is an option for valgrind and therefore must come before the name of the program which is being observed.

Some programs provide their own memory allocation functions or wrapper functions around the system's allocation functions. In the first case, allocations are normally missed; in the second case, the recorded call sites hide information, since only the address of the call in the wrapper function is recorded. For this reason, it is possible to add additional functions to the list of allocation functions. The `—alloc-fn=xmalloc` parameter would specify that the function `xmalloc` is also an allocation function, which is often the case in GNU programs. Calls to`xmalloc` are recorded, but not the allocation calls made from within `xmalloc`.

The second tool is called memusage; it is part of the GNU C library. It is a simplified version of massif (but existed a long time before massif). It only records the total memory use for heap (including possible calls to `mmap` etc. if

the `-m` option is given) and, optionally, the stack. The results can be shown as a graph of the total memory use over time or, alternatively, linearly over the calls made to allocation functions. The graphs are created separately by the memusage script which, just as with valgrind, has to be used to start the application:

```
memusage command arg
```

The `-p IMGFILE` option must be used to specify that the graph should be generated in the file `IMGFILE`, which will be a PNG file. The code to collect the data is run in the actual program itself, it is not an simulation like valgrind. This means memusage is much faster than massif and usable in situations where massif would be not useful. Besides total memory consumption, the code also records allocation sizes and, on program termination, it shows a histogram of the used allocation sizes. This information is written to standard error.

Sometimes it is not possible (or feasible) to call the program which is supposed to be observed directly. An example is the compiler stage of gcc, which is started by the gcc driver program. In this case the name of the program which should be observed must be provided to the memusage script using the `-n NAME` parameter. This parameter is also useful if the program which is observed starts other programs. If no program name is specified all started programs will be profiled.

Both programs, massif and memusage, have additional options. A programmer finding herself in the position needing more functionality should first consult the manual or help messages to make sure the additional functionality is not already implemented.

Now that we know how the data about memory allocation can be captured, it is necessary to discuss how this data can be interpreted in the context of memory and cache use. The main aspects of efficient dynamic memory allocation are linear allocation and compactness of the used portion. This goes back to making prefetching efficient and reducing cache misses.

A program which has to read in an arbitrary amount of data for later processing could do this by creating a list where each of the list elements contains a new data item. The overhead for this allocation method might be minimal (one pointer for a single-linked list) but the cache effects when using the data can reduce the performance dramatically.

One problem is, for instance, that there is no guarantee that sequentially allocated memory is laid out sequentially in memory. There are many possible reasons for this:

- memory blocks inside a large memory chunk administrated by the memory allocator are actually returned from the back to the front;
- a memory chunk is exhausted and a new one is started in a different part of the address space;
- the allocation requests are for different sizes which are served from different memory pools;
- the interleaving of allocations in the various threads of multi-threaded programs.

If data must be allocated up front for later processing, the linked-list approach is clearly a bad idea. There is no guarantee (or even likelihood) that the consecutive elements in the list are laid out consecutively in memory. To ensure contiguous allocations, that memory must not be allocated in small chunks. Another layer of memory handling must be used; it can easily be implemented by the programmer. An alternative is to use the obstack implementation available in the GNU C library. This allocator requests large blocks of memory from the system's allocator and then hands arbitrarily large or small blocks of memory out. These allocations are always sequential unless the large memory chunk is exhausted, which is, depending on the requested allocation sizes, pretty rare. Obstacks are not a complete replacement for a memory allocator, they have limited abilities to free objects. See the GNU C library manual for details.

So, how can a situation where the use of obstacks (or similar techniques) is advisable be recognized from the graphs? Without consulting the source, possible candidates for the changes cannot be identified, but the graph can provide an entry point for the search. If many allocations are made from the same location, this could mean that allocation in bulk might help. In Figure 7.7, we can see such a possible candidate in the allocations at address 0x4c0e7d5. From about 800ms into the run until 1,800ms into the run this is the only area (except the top, green one) which grows. Moreover, the slope is not steep, which means we have a large number of relatively small allocations. This is, indeed, a candidate for the use of obstacks or similar techniques.

Another problem the graphs can show is when the total number of allocations is high. This is especially easy to see if the graph is not drawn linearly over time but, instead, linearly over the number of calls (the default with memusage). In that case, a gentle slope in the graph means a lot of small allocations. memusage will not say where the allocations took place, but the comparison with massif's output can say that, or the programmer

might recognize it right away. Many small allocations should be consolidated to achieve linear memory use.

But there is another, equally important, aspect to this latter class of cases: many allocations also means higher overhead in administrative data. This by itself might not be that problematic. The red area named “heap-admin” represents this overhead in the massif graph and it is quite small. But, depending on the `malloc` implementation, this administrative data is allocated along with the data blocks, in the same memory. For the current `malloc` implementation in the GNU C library, this is the case: every allocated block has at least a 2-word header (8 bytes for 32-bit platforms, 16 bytes for 64-bit platforms). In addition, block sizes are often a bit larger than necessary due to the way memory is administrated (rounding up block sizes to specific multiples).

This all means that memory used by the program is interspersed with memory only used by the allocator for administrative purposes. We might see something like this:

Each block represents one memory word and, in this small region of memory, we have four allocated blocks. The overhead due to the block header and padding is 50%. Due to the placement of the header, this automatically means that the effective prefetch rate of the processor is lowered by up to 50% as well. If the blocks were be processed sequentially (to take maximum advantage of prefetching), the processor would read all the header and padding words into the cache, even though they are never supposed to be read from or written to by the application itself. Only the runtime uses the header words, and the runtime only comes into play when the block is freed.

Now, one could argue that the implementation should be changed to put the administrative data somewhere else. This is indeed done in some implementations, and it might prove to be a good idea. There are many aspects to be kept in mind, though, security not being the least of them. Regardless of whether we might see a change in the future, the padding issue will never go away (amounting to 16% of the data in the example, when ignoring the headers). Only if the programmer directly takes control of allocations can this be avoided. When alignment requirements come into

play there might still be holes, but this is also something under control of the programmer.

7.4 Improving Branch Prediction

In Section 6.2.2, two methods to improve L1i use through branch prediction and block reordering were mentioned: static prediction through `__builtin_expect` and profile guided optimization (PGO). Correct branch prediction has performance impacts, but here we are interested in the memory usage improvements.

The use of `__builtin_expect` (or better the `likely` and `unlikely` macros) is simple. The definitions are placed in a central header and the compiler takes care of the rest. There is a little problem, though: it is easy enough for a programmer to use `likely` when really `unlikely` was meant and vice versa. Even if somebody uses a tool like oprofile to measure incorrect branch predictions and L1i misses these problems are hard to detect.

There is one easy method, though. The code in Section 9.2 shows an alternative definition of the `likely` and `unlikely` macros which measure actively, at runtime, whether the static predictions are correct or not. The results can then be examined by the programmer or tester and adjustments can be made. The measurements do not actually take the performance of the program into account, they simply test the static assumptions made by the programmer. More details can be found, along with the code, in the section referenced above.

PGO is quite easy to use with gcc these days. It is a three-step process, though, and certain requirements must be fulfilled. First, all source files must be compiled with the additional `-fprofile-generate` option. This option must be passed to all compiler runs and to the command which links the program. Mixing object files compiled with and without this option is possible, but GPO will not do any good for those that do not have it enabled.

The compiler generates a binary which behaves normally except that it is significantly larger and slower since it records (and emits) all kinds of information about branches taken or not. The compiler also emits a file with the extension `.gcno` for each input file. This file contains information related to the branches in the code. It must be preserved for later.

Once the program binary is available, it should be used to run a representative set of workloads. Whatever workload is used, the final binary will be optimized to do this task well. Consecutive runs of the program are possible and, in general necessary; all the runs will contribute to the same output file. Before the program terminates, the data collected during the program run is written out into files with the extension. `gcda`. These files are created in the directory which contains the source file. The program can be executed from any directory, and the binary can be copied, but the directory with the sources must be available and writable. Again, one output file is created for each input source file. If the program is run multiple times, it is important that the . `gcda` files of the previous run are found in the source directories since otherwise the data of the runs cannot be accumulated in one file.

When a representative set of tests has been run, it is time to recompile the application. The compiler has to be able to find the . `gcda` files in the same directory which holds the source files. The files cannot be moved since the compiler would not find them and the embedded checksum for the files would not match anymore. For the recompilation, replace the `-fprofile-generate` parameter with `-fprofile-use`. It is essential that the sources do not change in any way that would change the generated code. That means: it is OK to change white spaces and edit comments, but adding more branches or basic blocks invalidates the collected data and the compilation will fail.

This is all the programmer has to do; it is a fairly simple process. The most important thing to get right is the selection of representative tests to perform the measurements. If the test workload does not match the way the program is actually used, the performed optimizations might actually do more harm than good. For this reason, is it often hard to use PGO for libraries. Libraries can be used in many—sometimes widely different—scenarios. Unless the use cases are indeed similar, it is usually better to rely exclusively on static branch prediction using `__builtin_expect`.

A few words on the . `gcno` and . `gcda` files. These are binary files which are not immediately usable for inspection. It is possible, though, to use the gcov tool, which is also part of the gcc package, to examine them. This tool is mainly used for coverage analysis (hence the name) but the file format used is the same as for PGO. The gcov tool generates output files with the

extension . `gcov` for each source file with executed code (this might include system headers). The files are source listings which are annotated, according to the parameters given to gcov, with branch counter, probabilities, etc.

7.5 Page Fault Optimization

On demand-paged operating systems like Linux, an `mmap` call only modifies the page tables. It makes sure that, for file-backed pages, the underlying data can be found and, for anonymous memory, that, on access, pages initialized with zeros are provided. No actual memory is allocated at the time of the `mmap` call. {*If you want to say "Wrong!" wait a second, it will be qualified later that there are exceptions.*}

The allocation part happens when a memory page is first accessed, either by reading or writing data, or by executing code. In response to the ensuing page fault, the kernel takes control and determines, using the page table tree, the data which has to be present on the page. This resolution of the page fault is not cheap, but it happens for every single page which is used by a process.

To minimize the cost of page faults, the total number of used pages has to be reduced. Optimizing the code for size will help with this. To reduce the cost of a specific code path (for instance, the start-up code), it is also possible to rearrange code so that, in that code path, the number of touched pages is minimized. It is not easy to determine the right order, though.

The author wrote a tool, based on the valgrind toolset, to measure page faults as they happen. Not the number of page faults, but the reason why they happen. The pagein tool emits information about the order and timing of page faults. The output, written to a file named `pagein.<PID>`, looks as in Figure 7.8.

```
   0 0x3000000000 C            0 0x3000000B50: (within
/lib64/ld-2.5.so)
   1 0x 7FF000000 D         3320 0x3000000B53: (within /lib64/ld-2.5.so)
   2 0x3000001000 C        58270 0x3000001080: _dl_start (in
/lib64/ld-2.5.so)
   3 0x3000219000 D       128020 0x30000010AE: _dl_start (in
/lib64/ld-2.5.so)
   4 0x300021A000 D       132170 0x30000010B5: _dl_start (in
/lib64/ld-2.5.so)
```

```
   5 0x3000008000 C      10489930 0x3000008B20: _dl_setup_hash (in
/lib64/ld-2.5.so)
   6 0x3000012000 C      13880830 0x3000012CC0: _dl_sysdep_start (in
/lib64/ld-2.5.so)
   7 0x3000013000 C     18091130 0x3000013440: brk (in /lib64/ld-2.5.so)
   8 0x3000014000 C      19123850 0x3000014020: strlen (in
/lib64/ld-2.5.so)
   9 0x3000002000 C      23772480 0x3000002450: dl_main (in
/lib64/ld-2.5.so)
```

Figure 7.8: Output of the pagein Tool

The second column specifies the address of the page which is paged-in. Whether it is a code or data page is indicated in the third column, which contains `C' or `D' respectively. The fourth column specifies the number of cycles which passed since the first page fault. The rest of the line is valgrind's attempt to find a name for the address which caused the page fault. The address value itself is correct but the name is not always accurate if no debug information is available.

In the example in Figure 7.8, execution starts at address 0x3000000B50, which forces the page at address 0x3000000000 to be paged in. Shortly after that, the page after this is also brought in; the function called on that page is `_dl_start`. The initial code accesses a variable on page 0x7FF000000.

This happens just 3,320 cycles after the first page fault and is most likely the second instruction of the program (just three bytes after the first instruction). If one looks at the program, one will notice that there is something peculiar about this memory access. The instruction in question is a `call` instruction, which does not explicitly load or store data. It does store the return address on the stack, though, and this is exactly what happens here. This is not the official stack of the process, though, it is valgrind's internal stack of the application. This means when interpreting the results of pagein it is important to keep in mind that valgrind introduces some artifacts.

The output of pagein can be used to determine which code sequences should ideally be adjacent in the program code. A quick look at the `/lib64/ld-2.5.so` code shows that the first instructions immediately call the function `_dl_start`, and that these two places are on different pages.

Rearranging the code to move the code sequences onto the same page can avoid—or at least delay—a page fault. It is, so far, a cumbersome process to determine what the optimal code layout should be. Since the second use of a

page is, by design, not recorded, one needs to use trial and error to see the effects of a change. Using call graph analysis, it is possible to guess about possible call sequences; this might help speed up the process of sorting the functions and variables.

At a very coarse level, the call sequences can be seen by looking a the object files making up the executable or DSO. Starting with one or more entry points (i.e., function names), the chain of dependencies can be computed. Without much effort this works well at the object file level. In each round, determine which object files contain needed functions and variables. The seed set has to be specified explicitly. Then determine all undefined references in those object files and add them to the set of needed symbols. Repeat until the set is stable.

The second step in the process is to determine an order. The various object files have to be grouped together to fill as few pages as possible. As an added bonus, no function should cross over a page boundary. A complication in all this is that, to best arrange the object files, it has to be known what the linker will do later. The important fact here is that the linker will put the object files into the executable or DSO in the same order in which they appear in the input files (e.g., archives), and on the command line. This gives the programmer sufficient control.

For those who are willing to invest a bit more time, there have been successful attempts at reordering made using automatic call tracing via the `__cyg_profile_func_enter` and `__cyg_profile_func_exit` hooks gcc inserts when called with the `-finstrument-functions` option [oooreorder].

See the gcc manual for more information on these `__cyg_*` interfaces. By creating a trace of the program execution, the programmer can more accurately determine the call chains. The results in [oooreorder] are a 5% decrease in start-up costs, just through reordering of the functions. The main benefit is the reduced number of page faults, but the TLB cache also plays a role—an increasingly important role given that, in virtualized environments, TLB misses become significantly more expensive.

By combining the analysis of the pagein tool with the call sequence information, it should be possible to optimize certain phases of the program (such as start-up) to minimize the number of page faults.

The Linux kernel provides two additional mechanisms to avoid page faults. The first one is a flag for `mmap` which instructs the kernel to not only modify

the page table but, in fact, to pre-fault all the pages in the mapped area. This is achieved by simply adding the `MAP_POPULATE` flag to the fourth parameter of the `mmap` call. This will cause the `mmap` call to be significantly more expensive, but, if all pages which are mapped by the call are being used right away, the benefits can be large. Instead of having a number of page faults, which each are pretty expensive due to the overhead incurred by synchronization requirements etc., the program would have one, more expensive, `mmap` call. The use of this flag has disadvantages, though, in cases where a large portion of the mapped pages are not used soon (or ever) after the call. Mapped, unused pages are obviously a waste of time and memory. Pages which are immediately pre-faulted and only much later used also can clog up the system. The memory is allocated before it is used and this might lead to shortages of memory in the meantime. On the other hand, in the worst case, the page is simply reused for a new purpose (since it has not been modified yet), which is not that expensive but still, together with the allocation, adds some cost.

The granularity of `MAP_POPULATE` is simply too coarse. And there is a second possible problem: this is an optimization; it is not critical that all pages are, indeed, mapped in. If the system is too busy to perform the operation the pre-faulting can be dropped. Once the page is really used the program takes the page fault, but this is not worse than artificially creating resource scarcity. An alternative is to use the`POSIX_MADV_WILLNEED` advice with the `posix_madvise` function. This is a hint to the operating system that, in the near future, the program will need the page described in the call. The kernel is free to ignore the advice, but it also can pre-fault pages. The advantage here is that the granularity is finer. Individual pages or page ranges in any mapped address space area can be pre-faulted. For memory-mapped files which contain a lot of data which is not used at runtime, this can have huge advantages over using `MAP_POPULATE`.

Beside these active approaches to minimizing the number of page faults, it is also possible to take a more passive approach which is popular with the hardware designers. A DSO occupies neighboring pages in the address space, one range of pages each for the code and the data. The smaller the page size, the more pages are needed to hold the DSO. This, in turn, means more page faults, too. Important here is that the opposite is also true. For

larger page sizes, the number of necessary pages for the mapping (or anonymous memory) is reduced; with it falls the number of page faults.

Most architectures support page sizes of 4k. On IA-64 and PPC64, page sizes of 64k are also popular. That means the smallest unit in which memory is given out is 64k. The value has to be specified when compiling the kernel and cannot be changed dynamically (at least not at the moment). The ABIs of the multiple-page-size architectures are designed to allow running an application with either page size. The runtime will make the necessary adjustments, and a correctly-written program will not notice a thing. Larger page sizes mean more waste through partially-used pages, but, in some situations, this is OK.

Most architectures also support very large page sizes of 1MB or more. Such pages are useful in some situations, too, but it makes no sense to have all memory given out in units that large. The waste of physical RAM would simply be too large. But very large pages have their advantages: if huge data sets are used, storing them in 2MB pages on x86-64 would require 511 fewer page faults (per large page) than using the same amount of memory with 4k pages. This can make a big difference. The solution is to selectively request memory allocation which, just for the requested address range, uses huge memory pages and, for all the other mappings in the same process, uses the normal page size.

Huge page sizes come with a price, though. Since the physical memory used for large pages must be continuous, it might, after a while, not be possible to allocate such pages due to memory fragmentation. prevent this. People are working on memory defragmentation and fragmentation avoidance, but it is very complicated. For large pages of, say, 2MB the necessary 512 consecutive pages are always hard to come by, except at one time: when the system boots up. This is why the current solution for large pages requires

the use of a special filesystem, `hugetlbfs`. This pseudo filesystem is

allocated on request by the system administrator by writing the number of huge pages which should be reserved to

```
/proc/sys/vm/nr_hugepages
```

the number of huge pages which should be reserved. This operation might fail if not enough continuous memory can be located. The situation gets especially interesting if virtualization is used. A system virtualized using the VMM model does not directly access physical memory and, therefore,

cannot by itself allocate the `hugetlbfs`. It has to rely on the VMM, and this

feature is not guaranteed to be supported. For the KVM model, the Linux

kernel running the KVM module can perform the hugetlbfs allocation and possibly pass a subset of the pages thus allocated on to one of the guest domains.

Later, when a program needs a large page, there are multiple possibilities:

- the program can use System V shared memory with the SHM_HUGETLB flag.
- the hugetlbfs filesystem can actually be mounted and the program can then create a file under the mount point and use mmap to map one or more pages as anonymous memory.

In the first case, the hugetlbfs need not be mounted. Code requesting one or more large pages could look like this:

```
key_t k = ftok("/some/key/file", 42);
int id = shmget(k, LENGTH, SHM_HUGETLB|IPC_CREAT|SHM_R|SHM_W);
void *a = shmat(id, NULL, 0);
```

The critical parts of this code sequence are the use of the SHM_HUGETLB flag and the choice of the right value for LENGTH, which must be a multiple of the huge page size for the system. Different architectures have different values. The use of the System V shared memory interface has the nasty problem of depending on the key argument to differentiate (or share) mappings. The ftok interface can easily produce conflicts which is why, if possible, it is better to use other mechanisms.

If the requirement to mount the hugetlbfs filesystem is not a problem, it is better to use it instead of System V shared memory. The only real problems with using the special filesystem are that the kernel must support it, and that there is no standardized mount point yet. Once the filesystem is mounted, for instance at /dev/hugetlb, a program can make easy use of it:

```
int fd = open("/dev/hugetlb/file1", O_RDWR|O_CREAT, 0700);
void *a = mmap(NULL, LENGTH, PROT_READ|PROT_WRITE, fd, 0);
```

By using the same file name in the `open` call, multiple processes can share the same huge pages and collaborate. It is also possible to make the pages executable, in which case the `PROT_EXEC` flag must also be set in the `mmap` call.

As in the System V shared memory example, the value of`LENGTH` must be a multiple of the system's huge page size.

A defensively-written program (as all programs should be) can determine the mount point at runtime using a function like this:

```
char *hugetlbfs_mntpoint(void) {
  char *result = NULL;
  FILE *fp = setmntent(_PATH_MOUNTED, "r");
  if (fp != NULL) {
    struct mntent *m;
    while ((m = getmntent(fp)) != NULL)
       if (strcmp(m->mnt_fsname, "hugetlbfs") == 0) {
         result = strdup(m->mnt_dir);
         break;
       }
    endmntent(fp);
  }
  return result;
}
```

More information for both these cases can be found in the hugetlbpage.txt file which comes as part of the kernel source tree. The file also describes the special handling needed for IA-64.

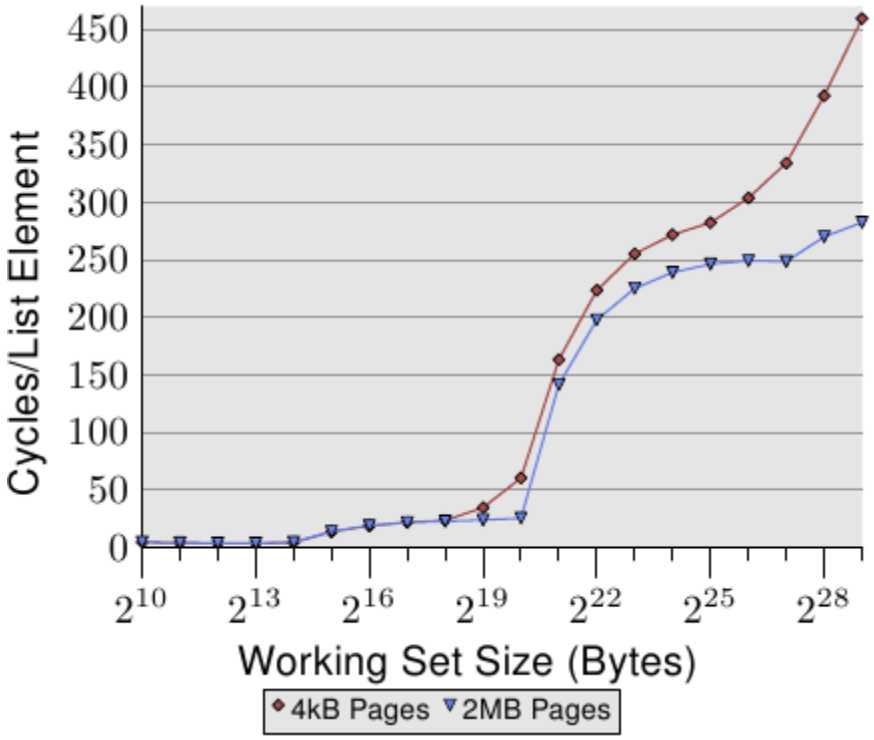


Figure 7.9: Follow with Huge Pages, NPAD=0

To illustrate the advantages of huge pages, Figure 7.9 shows the results of running the random Follow test for `NPAD`=0. This is the same data shown in Figure 3.15, but, this time, we measure the data also with memory allocated in huge pages. As can be seen the performance advantage can be huge. For 2^{20} bytes the test using huge pages is 57% faster. This is due to the fact that this size still fits completely into one single 2MB page and, therefore, no DTLB misses occur.

After this point, the winnings are initially smaller but grow again with increasing working set size. The huge pages test is 38% faster for the 512MB working set size. The curve for the huge page test has a plateau at around 250 cycles. Beyond working sets of 2^{27} bytes, the numbers rise significantly again. The reason for the plateau is that 64 TLB entries for 2MB pages cover 2^{27} bytes.

As these numbers show, a large part of the costs of using large working set sizes comes from TLB misses. Using the interfaces described in this section can pay off big-time. The numbers in the graph are, most likely, upper limits, but even real-world programs show a significant speed-up. Databases, since they use large amounts of data, are among the programs which use huge pages today.

There is currently no way to use large pages to map file-backed data. There is interest in implementing this capability, but the proposals made so far all involve explicitly using large pages, and they rely on

the `hugetlbfs` filesystem. This is not acceptable: large page use in this case

must be transparent. The kernel can easily determine which mappings are large and automatically use large pages. A big problem is that the kernel does not always know about the use pattern. If the memory, which could be mapped as a large page, later requires 4k-page granularity (for instance, because the protection of parts of the memory range is changed

using `mprotect`) a lot of precious resources, in particular the linear physical

memory, will have been wasted. So it will certainly be some more time before such an approach is successfully implemented.

This article was contributed by Ulrich Drepper

[*Editor's note: Here, at last, is the final segment of Ulrich Drepper's "What every programmer should know about memory." This eight-part series began back in September. The conclusion of this document looks at how future technologies may help to improve performance as the memory bottleneck continues to worsen.*

We would like to thank Ulrich one last time for giving LWN the opportunity to help shape this document and bring it to our readers. Ulrich plans to post the entire thing in PDF format sometime in the near future; we'll carry an announcement when that happens.]

8 Upcoming Technology

In the preceding sections about multi-processor handling we have seen that significant performance problems must be expected if the number of CPUs or cores is scaled up. But this scaling-up is exactly what has to be expected in the future. Processors will get more and more cores, and programs must be ever more parallel to take advantage of the increased potential of the CPU, since single-core performance will not rise as quickly as it used to.

8.1 The Problem with Atomic Operations

Synchronizing access to shared data structures is traditionally done in two ways:

- through mutual exclusion, usually by using functionality of the system runtime to achieve just that;
- by using lock-free data structures.

The problem with lock-free data structures is that the processor has to provide primitives which can perform the entire operation atomically. This support is limited. On most architectures support is limited to atomically read and write a word. There are two basic ways to implement this (see Section 6.4.2):

- using atomic compare-and-exchange (CAS) operations;
- using a load lock/store conditional (LL/SC) pair.

It can be easily seen how a CAS operation can be implemented using LL/SC instructions. This makes CAS operations the building block for most atomic operations and lock free data structures.

Some processors, notably the x86 and x86-64 architectures, provide a far more elaborate set of atomic operations. Many of them are optimizations of the CAS operation for specific purposes. For instance, atomically adding a value to a memory location can be implemented using CAS and LL/SC operations, but the native support for atomic increments on x86/x86-64 processors is faster. It is important for programmers to know about these operations, and the intrinsics which make them available when programming, but that is nothing new.

The extraordinary extension of these two architectures is that they have double-word CAS (DCAS) operations. This is significant for some applications but not all (see [dcas]). As an example of how DCAS can be used, let us try to write a lock-free array-based stack/LIFO data structure. A first attempt using gcc's intrinsics can be seen in Figure 8.1.

```
struct elem {
  data_t d;
  struct elem *c;
};
struct elem *top;
void push(struct elem *n) {
  n->c = top;
  top = n;
}
struct elem *pop(void) {
  struct elem *res = top;
  if (res != NULL)
    top = res->c;
  return res;
}
```

Figure 8.1: Not Thread-Safe LIFO

This code is clearly not thread-safe. Concurrent accesses in different threads will modify the global variable `top` without consideration of other thread's modifications. Elements could be lost or removed elements can magically reappear. It is possible to use mutual exclusion but here we will try to use only atomic operations.

The first attempt to fix the problem uses CAS operations when installing or removing list elements. The resulting code looks like Figure 8.2.

```
#define CAS __sync_bool_compare_and_swap
struct elem {
  data_t d;
  struct elem *c;
};
struct elem *top;
void push(struct elem *n) {
  do
    n->c = top;
  while (!CAS(&top, n->c, n));
}
struct elem *pop(void) {
  struct elem *res;
  while ((res = top) != NULL)
    if (CAS(&top, res, res->c))
      break;
  return res;
}
```

Figure 8.2: LIFO using CAS

At first glance this looks like a working solution. `top` is never modified unless it matches the element which was at the top of the LIFO when the operation started. But we have to take concurrency at all levels into account. It might be that another thread working on the data structure is scheduled at the worst possible moment. One such case here is the so-called ABA problem. Consider what happens if a second thread is scheduled right before the CAS operation in `pop` and it performs the following operation:

1. `l = pop()`
2. `push(newelem)`

3. push(l)

The end effect of this operation is that the former top element of the LIFO is back at the top but the second element is different. Back in the first thread, because the top element is unchanged, the CAS operation will succeed. But the value res->c is not the right one. It is a pointer to the second element of the original LIFO and not newelem. The result is that this new element is lost.

In the literature [lockfree] you find suggestions to use a feature found on some processors to work around this problem. Specifically, this is about the ability of the x86 and x86-64 processors to perform DCAS operations. This is used in the third incarnation of the code in Figure 8.3.

```
#define CAS __sync_bool_compare_and_swap
struct elem {
  data_t d;
  struct elem *c;
};
struct lifo {
  struct elem *top;
  size_t gen;
} l;
void push(struct elem *n) {
  struct lifo old, new;
  do {
    old = l;
    new.top = n->c = old.top;
    new.gen = old.gen + 1;
  } while (!CAS(&l, old, new));
}
struct elem *pop(void) {
  struct lifo old, new;
  do {
    old = l;
    if (old.top == NULL) return NULL;
    new.top = old.top->c;
    new.gen = old.gen + 1;
  } while (!CAS(&l, old, new));
  return old.top;
}
```

Figure 8.3: LIFO using double-word CAS

Unlike the other two examples, this is (currently) pseudo-code since gcc does not grok the use of structures in the CAS intrinsics. Regardless, the example should be sufficient understand the approach. A generation counter is added to the pointer to the top of the LIFO. Since it is changed on every operation, `push` or `pop`, the ABA problem described above is no longer a problem. By the time the first thread is resuming its work by actually exchanging the `top` pointer, the generation counter has been incremented three times. The CAS operation will fail and, in the next round of the loop, the correct first and second element of the LIFO are determined and the LIFO is not corrupted. Voil à

Is this really the solution? The authors of [lockfree] certainly make it sound like it and, to their credit, it should be mentioned that it is possible to construct data structures for the LIFO which would permit using the code above. But, in general, this approach is just as doomed as the previous one. We still have concurrency problems, just now in a different place. Let us assume a thread executes `pop` and is interrupted after the test for `old.top == NULL`. Now a second thread uses `pop` and receives ownership of the previous first element of the LIFO. It can do anything with it, including changing all values or, in case of dynamically allocated elements, freeing the memory.

Now the first thread resumes. The `old` variable is still filled with the previous top of the LIFO. More specifically, the `top` member points to the element popped by the second thread. In `new.top = old.top->c` the first thread dereferences a pointer in the element. But the element this pointer references might have been freed. That part of the address space might be inaccessible and the process could crash. This cannot be allowed for a generic data type implementation. Any fix for this problem is terribly expensive: memory must never be freed, or at least it must be verified that no thread is referencing the memory anymore before it is freed. Given that lock-free data structures are supposed to be faster and more concurrent, these additional requirements completely destroy any advantage. In languages which support it, memory handling through garbage collection can solve the problem, but this comes with its price.

The situation is often worse for more complex data structures. The same paper cited above also describes a FIFO implementation (with refinements

in a successor paper). But this code has all the same problems. Because CAS operations on existing hardware (x86, x86-64) are limited to modifying two words which are consecutive in memory, they are no help at all in other common situations. For instance, atomically adding or removing elements anywhere in a double-linked list is not possible. {*As a side note, the developers of the IA-64 did* not *include this feature. They allow* comparing *two words, but replacing only one.*}

The problem is that more than one memory address is generally involved, and only if none of the values of these addresses is changed concurrently can the entire operation succeed. This is a well-known concept in database handling, and this is exactly where one of the most promising proposals to solve the dilemma comes from.

8.2 Transactional Memory

In their groundbreaking 1993 paper [transactmem] Herlihy and Moss propose to implement transactions for memory operations in hardware since software alone cannot deal with the problem efficiently. Digital Equipment Corporation, at that time, was already battling with scalability problems on their high-end hardware, which featured a few dozen processors. The principle is the same as for database transactions: the result of a transaction becomes visible all at once or the transaction is aborted and all the values remain unchanged.

This is where memory comes into play and why the previous section bothered to develop algorithms which use atomic operations. Transactional memory is meant as a replacement for—and extension of—atomic operations in many situations, especially for lock-free data structures. Integrating a transaction system into the processor sounds like a terribly complicated thing to do but, in fact, most processors, to some extent, already have something similar.

The LL/SC operations implemented by some processors form a transaction. The SC instruction aborts or commits the transaction based on whether the memory location was touched or not. Transactional memory is an extension of this concept. Now, instead of a simple pair of instructions, multiple instructions take part in the transaction. To understand how this can work, it is worthwhile to first see how LL/SC instructions can be implemented. {*This does not mean it is actually implemented like this.*}

8.2.1 Load Lock/Store Conditional Implementation

If the LL instruction is issued, the value of the memory location is loaded into a register. As part of that operation, the value is loaded into L1d. The

SC instruction later can only succeed if this value has not been tampered with. How can the processor detect this? Looking back at the description of the MESI protocol in Figure 3.18 should make the answer obvious. If another processor changes the value of the memory location, the copy of the value in L1d of the first processor must be revoked. When the SC instruction is executed on the first processor, it will find it has to load the value again into L1d. This is something the processor must already detect.

There are a few more details to iron out with respect to context switches (possible modification on the same processor) and accidental reloading of the cache line after a write on another processor. This is nothing that policies (cache flush on context switch) and extra flags, or separate cache lines for LL/SC instructions, cannot fix. In general, the LL/SC implementation comes almost for free with the implementation of a cache coherence protocol like MESI.

8.2.2 Transactional Memory Operations

For transactional memory to be generally useful, a transaction must not be finished with the first store instruction. Instead, an implementation should allow a certain number of load and store operations; this means we need separate commit and abort instructions. In a bit we will see that we need one more instruction which allows checking on the current state of the transaction and whether it is already aborted or not.

There are three different memory operations to implement:

- Read memory
- Read memory which is written to later
- Write memory

When looking at the MESI protocol it should be clear how this special second type of read operation can be useful. The normal read can be satisfied by a cache line in the `E' and `S' state. The second type of read operation needs a cache line in state `E'. Exactly why the second type of memory read is necessary can be glimpsed from the following discussion, but, for a more complete description, the interested reader is referred to literature about transactional memory, starting with [transactmem].

In addition, we need transaction handling which mainly consists of the commit and abort operation we are already familiar with from database transaction handling. There is one more operation, though, which is optional in theory but required for writing robust programs using transactional memory. This instruction lets a thread test whether the transaction is still on

track and can (perhaps) be committed later, or whether the transaction already failed and will in any case be aborted.

We will discuss how these operations actually interact with the CPU cache and how they match to bus operation. But before we do that we take a look at some actual code which uses transactional memory. This will hopefully make the remainder of this section easier to understand.

8.2.3 Example Code Using Transactional Memory

For the example we revisit our running example and show a LIFO implementation which uses transactional memory.

```
struct elem {
  data_t d;
  struct elem *c;
};
struct elem *top;
void push(struct elem *n) {
  while (1) {
    n->c = LTX(top);
    ST(&top, n);
    if (COMMIT())
      return;
    ... delay ...
  }
}
struct elem *pop(void) {
  while (1) {
    struct elem *res = LTX(top);
    if (VALIDATE()) {
      if (res != NULL)
        ST(&top, res->c);
      if (COMMIT())
        return res;
    }
    ... delay ...
  }
}
```

Figure 8.4: LIFO Using Transactional Memory

This code looks quite similar to the not-thread-safe code, which is an additional bonus as it makes writing code using transactional memory easier.

The new parts of the code are the `LTX`, `ST`, `COMMIT`, and `VALIDATE` operations. These four operations are the way to request accesses to transactional memory. There is actually one more operation, `LT`, which is not used here. `LT` requests non-exclusive read access, `LTX` requests exclusive read access, and `ST` is a store into transactional memory. The `VALIDATE` operation is the operation which checks whether the transaction is still on track to be committed. It returns true if this transaction is still OK. If the transaction is already marked as aborting, it will be actually aborted and the next transactional memory instruction will start a new transaction. For this reason, the code uses a new `if` block in case the transaction is still going on.

The `COMMIT` operation finishes the transaction; if the transaction is finished successfully the operation returns true. This means that this part of the program is done and the thread can move on. If the operation returns a false value, this usually means the whole code sequence must be repeated. This is what the outer `while` loop is doing here. This is not absolutely necessary, though, in some cases giving up on the work is the right thing to do.

The interesting point about the `LT`, `LTX`, and `ST` operations is that they can fail without signaling this failure in any direct way. The way the program can request this information is through the `VALIDATE` or `COMMIT` operation. For the load operation, this can mean that the value actually loaded into the register might be bogus; that is why it is necessary in the example above to use `VALIDATE` before dereferencing the pointer. In the next section, we will see why this is a wise choice for an implementation. It might be that, once transactional memory is actually widely available, the processors will implement something different. The results from [transactmem] suggest what we describe here, though.

The `push` function can be summarized as this: the transaction is started by reading the pointer to the head of the list. The read requests exclusive ownership since, later in the function, this variable is written to. If another thread has already started a transaction, the load will fail and mark the still-born transaction as aborted; in this case, the value actually loaded might

be garbage. This value is, regardless of its status, stored in the `next` field of the new list member. This is fine since this member is not yet in use, and it is accessed by exactly one thread. The pointer to the head of the list is then assigned the pointer to the new element. If the transaction is still OK, this write can succeed. This is the normal case, it can only fail if a thread uses some code other than the provided `push` and `pop` functions to access this pointer. If the transaction is already aborted at the time the `ST` is executed, nothing at all is done. Finally, the thread tries to commit the transaction. If this succeeds the work is done; other threads can now start their transactions. If the transaction fails, it must be repeated from the beginning. Before doing that, however, it is best to insert an delay. If this is not done the thread might run in a busy loop (wasting energy, overheating the CPU).

The `pop` function is slightly more complex. It also starts with reading the variable containing the head of the list, requesting exclusive ownership. The code then immediately checks whether the `LTX` operation succeeded or not. If not, nothing else is done in this round except delaying the next round. If the `top` pointer was read successfully, this means its state is good; we can now dereference the pointer. Remember, this was exactly the problem with the code using atomic operations; with transactional memory this case can be handled without any problem. The following `ST` operation is only performed when the LIFO is not empty, just as in the original, thread-unsafe code. Finally the transaction is committed. If this succeeds the function returns the old pointer to the head; otherwise we delay and retry. The one tricky part of this code is to remember that the `VALIDATE` operation aborts the transaction if it has already failed. The next transactional memory operation would start a new transaction and, therefore, we must skip over the rest of the code in the function.

How the delay code works will be something to see when implementations of transactional memory are available in hardware. If this is done badly system performance might suffer significantly.

8.2.4 Bus Protocol for Transactional Memory

Now that we have seen the basic principles behind transactional memory, we can dive into the details of the implementation. Note that this is*not* based

on actual hardware. It is based on the original design of transactional memory and knowledge about the cache coherency protocol. Some details are omitted, but it still should be possible to get insight into the performance characteristics.

Transactional memory is not actually implemented as separate memory; that would not make any sense given that transactions on any location in a thread's address space are wanted. Instead, it is implemented at the first cache level. The implementation could, in theory, happen in the normal L1d but, as [transactmem] points out, this is not a good idea. We will more likely see the transaction cache implemented in parallel to L1d. All accesses will use the higher level cache in the same way they use L1d. The transaction cache is likely much smaller than L1d. If it is fully associative its size is determined by the number of operations a transaction can comprise. Implementations will likely have limits for the architecture and/or specific processor version. One could easily imagine a transaction cache with 16 elements or even less. In the above example we only needed one single memory location; algorithms with a larger transaction working sets get very complicated. It is possible that we will see processors which support more than one active transaction at any one time. The number of elements in the cache then multiplies, but it is still small enough to be fully associative.

The transaction cache and L1d are exclusive. That means a cache line is in, at most, one of the caches but never in both. Each slot in the transaction cache is in, at any one time, one of the four MESI protocol states. In addition to this, a slot has a transaction state. The states are as follows (names according to [transactmem]):

EMPTY
: the cache slot contains no data. The MESI state is always 'I'.

NORMAL
: the cache slot contains committed data. The data could as well exist in L1d. The MESI state can be 'M', 'E', and 'S'. The fact that the 'M' state is allowed means that transaction commits do *not* force the data to be written into the main memory (unless the memory region is declared as uncached or write-through). This can significantly help to increase performance.

XABORT
: the cache slot contains data which is discarded on abort. This is obviously the opposite of XCOMMIT. All the data created during a transaction is kept in the transaction cache, nothing is written to main memory before a commit. This limits the maximum transaction size but it means that, beside the transaction cache, no other memory has to be aware of the XCOMMIT/XABORT duality for a single memory location. The possible MESI states are 'M', 'E', and 'S'.

XCOMMIT

the cache slot contains data which is discarded on commit. This is a possible optimization processors could implement. If a memory location is changed using a transaction operation, the old content cannot be just dropped: if the transaction fails the old content needs to be restored. The MESI states are the same as for XABORT. One difference with regard to XABORT is that, if the transaction cache is full, any XCOMMIT entries in the 'M' state could be written back to memory and then, for all states, discarded.

When an `LT` operation is started, the processor allocates two slots in the cache. Victims are chosen by first looking for NORMAL slots for the address of the operation, i.e., a cache hit. If such an entry is found, a second slot is located, the value copied, one entry is marked XABORT, and the other one is marked XCOMMIT.

If the address is not already cached, EMPTY cache slots are located. If none can be found, NORMAL slots are looked for. The old content must then be flushed to memory if the MESI state is 'M'. If no NORMAL slot is available either, it is possible to victimize XCOMMIT entries. This is likely going to be an implementation detail, though. The maximum size of a transaction is determined by the size of the transaction cache, and, since the number of slots which are needed for each operation in the transaction is fixed, the number of transactions can be capped before having to evict XCOMMIT entries.

If the address is not found in the transactional cache, a T_READ request is issued on the bus. This is just like the normal READ bus request, but it indicates that this is for the transactional cache. Just like for the normal READ request, the caches in all other processors first get the chance to respond. If none does the value is read from the main memory. The MESI protocol determines whether the state of the new cache line is 'E' or 'S'. The difference between T_READ and READ comes into play when the cache line is currently in use by an active transaction on another processor or core. In this case the T_READ operation plainly fails, no data is transmitted. The transaction which generated the T_READ bus request is marked as failed and the value used in the operation (usually a simple register load) is undefined. Looking back to the example, we can see that this behavior does not cause problems if the transactional memory operations are used correctly. Before a value loaded in a transaction is used, it must be verified with `VALIDATE`. This is, in almost no cases, an extra burden. As we have seen in the attempts to create a FIFO implementation using atomic operations, the

check which we added is the one missing feature which would make the lock-free code work.

The LTX operation is almost identical to LT. The one difference is that the bus operation is T_RFO instead of T_READ. T_RFO, like the normal RFO bus request, requests exclusive ownership of the cache line. The state of the resulting cache line is 'E'. Like the T_READ bus request, T_RFO can fail, in which case the used value is undefined, too. If the cache line is already in the local transaction cache with 'M' or 'E' state, nothing has to be done. If the state in the local transaction cache is 'S' the bus request has to go out to invalidate all other copies.

The ST operation is similar to LTX. The value is first made available exclusively in the local transaction cache. Then the ST operation makes a copy of the value into a second slot in the cache and marks the entry as XCOMMIT. Lastly, the other slot is marked as XABORT and the new value is written into it. If the transaction is already aborted, or is newly aborted because the implicit LTX fails, nothing is written.

Neither the VALIDATE nor COMMIT operations automatically and implicitly create bus operations. This is the huge advantage transactional memory has over atomic operations. With atomic operations, concurrency is made possible by writing changed values back into main memory. If you have read this document thus far, you should know how expensive this is. With transactional memory, no accesses to the main memory are forced. If the cache has no EMPTY slots, current content must be evicted, and for slots in the 'M' state, the content must be written to main memory. This is not different from regular caches, and the write-back can be performed without special atomicity guarantees. If the cache size is sufficient, the content can survive for a long time. If transactions are performed on the same memory location over and over again, the speed improvements can be astronomical since, in the one case, we have one or two main memory accesses in each round while, for transactional memory, all accesses hit the transactional cache, which is as fast as L1d.

All the VALIDATE and COMMIT operations do for an aborted transaction is to mark the cache slots marked XABORT as empty and mark the XCOMMIT slots as NORMAL. Similarly, when COMMIT successfully finishes a

transaction, the XCOMMIT slots are marked empty and the XABORT slots are marked NORMAL. These are very fast operations on the transaction cache. No explicit notification to other processors which want to perform transactions happens; those processors just have to keep trying. Doing this efficiently is another matter. In the example code above we simply have `...delay...` in the appropriate place. We might see actual processor support for delaying in a useful way.

To summarize, transactional memory operations cause bus operation only when a new transaction is started and when a new cache line, which is not already in the transaction cache, is added to a still-successful transaction. Operations in aborted transactions do not cause bus operations. There will be no cache line ping-pong due to multiple threads trying to use the same memory.

8.2.5 Other Considerations

In Section 6.4.2, we already discussed how the `lock` prefix, available on x86 and x86-64, can be used to avoid the coding of atomic operations in some situations. The proposed tricks falls short, though, when there are multiple threads in use which do not contend for the same memory. In this case, the atomic operations are used unnecessarily. With transactional memory this problem goes away. The expensive RFO bus requests are issued only if memory is used on different CPUs concurrently or in succession; this is only the case when they are needed. It is almost impossible to do any better.

The attentive reader might have wondered about delays. What is the expected worst case scenario? What if the thread with the active transaction is descheduled, or if it receives a signal and is possibly terminated, or decides to use `siglongjmp` to jump to an outer scope? The answer to this is: the transaction will be aborted. It is possible to abort a transaction whenever a thread makes a system call or receives a signal (i.e., a ring level change occurs). It might also be that aborting the transaction is part of the OS's duties when performing system calls or handling signals. We will have to wait until implementations become available to see what is actually done.

The final aspect of transactional memory which should be discussed here is something which people might want to think about even today. The transaction cache, like other caches, operates on cache lines. Since the transaction cache is an exclusive cache, using the same cache line for transactions and non-transaction operation will be a problem. It is therefore important to

- move non-transactional data off of the cache line
- have separate cache lines for data used in separate transactions

The first point is not new, the same effort will pay off for atomic operations today. The second is more problematic since today objects are hardly ever aligned to cache lines due to the associated high cost. If the data used, along with the words modified using atomic operations, is on the same cache line, one less cache line is needed. This does not apply to mutual exclusion (where the mutex object should always have its own cache line), but one can certainly make cases where atomic operations go together with other data. With transactional memory, using the cache line for two purposes will most likely be fatal. Every normal access to data {*From the cache line in question. Access to arbitrary other cache lines does not influence the transaction.*} would remove the cache line from the transactional cache, aborting the transaction in the process. Cache alignment of data objects will be in future not only a matter of performance but also of correctness.

It is possible that transactional memory implementations will use more precise accounting and will, as a result, not suffer from normal accesses to data on cache lines which are part of a transaction. This requires a lot more effort, though, since then the MESI protocol information is not sufficient anymore.

8.3 Increasing Latency

One thing about future development of memory technology is almost certain: latency will continue to creep up. We already discussed, in Section 2.2.4, that the upcoming DDR3 memory technology will have higher latency than the current DDR2 technology. FB-DRAM, if it should get deployed, also has potentially higher latency, especially when FB-DRAM modules are daisy-chained. Passing through the requests and results does not come for free.

The second source of latency is the increasing use of NUMA. AMD's Opterons are NUMA machines if they have more than one processor. There is some local memory attached to the CPU with its own memory controller but, on SMP motherboards, the rest of the memory has to be accessed through the Hypertransport bus. Intel's CSI technology will use almost the same technology. Due to per-processor bandwidth limitations and the requirement to keep (for instance) multiple 10Gb/s Ethernet ports busy, multi-socket motherboards will not vanish, even if the number of cores per socket increases.

A third source of latency are co-processors. We thought that we got rid of them after math co-processors for commodity processors were no longer

necessary at the beginning of the 1990's, but they are making a comeback. Intel's Geneseo and AMD's Torrenza are extensions of the platform to allow third-party hardware developers to integrate their products into the motherboards. I.e., the co-processors will not have to sit on a PCIe card but, instead, are positioned much closer to the CPU. This gives them more bandwidth.

IBM went a different route (although extensions like Intel's and AMD's are still possible) with the Cell CPU. The Cell CPU consists, beside the PowerPC core, of 8 Synergistic Processing Units (SPUs) which are specialized processors mainly for floating-point computation.

What co-processors and SPUs have in common is that they, most likely, have even slower memory logic than the real processors. This is, in part, caused by the necessary simplification: all the cache handling, prefetching etc is complicated, especially when cache coherency is needed, too. High-performance programs will increasingly rely on co-processors since the performance differences can be dramatic. Theoretical peak performance for a Cell CPU is 210 GFLOPS, compared to 50-60 GFLOPS for a high-end CPU. The Graphics Processing Units (GPUs, processors on graphics cards) in use today achieve even higher numbers (north of 500 GFLOPS) and those could probably, with not too much effort, be integrated into the Geneseo/Torrenza systems.

As a result of all these developments, a programmer must conclude that prefetching will become ever more important. For co-processors it will be absolutely critical. For CPUs, especially with more and more cores, it is necessary to keep the FSB busy all the time instead of piling on the requests in batches. This requires giving the CPU as much insight into future traffic as possible through the efficient use of prefetching instructions.

8.4 Vector Operations

The multi-media extensions in today's mainstream processors implement vector operations only in a limited fashion. Vector instructions are characterized by large numbers of operations which are performed together. Compared with scalar operations, this can be said about the multi-media instructions, but it is a far cry from what vector computers like the Cray-1 or vector units for machines like the IBM 3090 did.

To compensate for the limited number of operations performed for one instruction (four `float` or two `double` operations on most machines) the

surrounding loops have to be executed more often. The example in Section 9.1 shows this clearly, each cache line requires `SM` iterations.

With wider vector registers and operations, the number of loop iterations can be reduced. This results in more than just improvements in the instruction decoding etc.; here we are more interested in the memory effects. With a single instruction loading or storing more data, the processor has a better picture about the memory use of the application and does not have to try to piece together the information from the behavior of individual instructions. Furthermore, it becomes more useful to provide load or store instructions which do not affect the caches. With 16 byte wide loads of an SSE register in an x86 CPU, it is a bad idea to use uncached loads since later accesses to the same cache line have to load the data from memory again (in case of cache misses). If, on the other hand, the vector registers are wide enough to hold one or more cache lines, uncached loads or stores do not have negative impacts. It becomes more practical to perform operations on data sets which do not fit into the caches.

Having large vector registers does not necessarily mean the latency of the instructions is increased; vector instructions do not have to wait until all data is read or stored. The vector units could start with the data which has already been read if it can recognize the code flow. That means, if, for instance, a vector register is to be loaded and then all vector elements multiplied by a scalar, the CPU could start the multiplication operation as soon as the first part of the vector has been loaded. It is just a matter of sophistication of the vector unit. What this shows is that, in theory, the vector registers can grow really wide, and that programs could potentially be designed today with this in mind. In practice, there are limitations imposed on the vector register size by the fact that the processors are used in multi-process and multi-thread OSes. As a result, the context switch times, which include storing and loading register values, is important.

With wider vector registers there is the problem that the input and output data of the operations cannot be sequentially laid out in memory. This might be because a matrix is sparse, a matrix is accessed by columns instead of rows, and many other factors. Vector units provide, for this case, ways to access memory in non-sequential patterns. A single vector load or store can be parametrized and instructed to load data from many different places in the address space. Using today's multi-media instructions, this is not possible at all. The values would have to be explicitly loaded one by one and then painstakingly combined into one vector register.

The vector units of the old days had different modes to allow the most useful access patterns:

- using *striding*, the program can specify how big the gap between two neighboring vector elements is. The gap between all elements must be the same but this would, for instance, easily allow to read the column of a matrix into a vector register in one instruction instead of one instruction per row.
- using indirection, arbitrary access patterns can be created. The load or store instruction would receive a pointer to an array which contains addresses or offsets of the real memory locations which have to be loaded.

It is unclear at this point whether we will see a revival of true vector operations in future versions of mainstream processors. Maybe this work will be relegated to co-processors. In any case, should we get access to vector operations, it is all the more important to correctly organize the code performing such operations. The code should be self-contained and replaceable, and the interface should be general enough to efficiently apply vector operations. For instance, interfaces should allow adding entire matrixes instead of operating on rows, columns, or even groups of elements. The larger the building blocks, the better the chance of using vector operations.

In [vectorops] the authors make a passionate plea for the revival of vector operations. They point out many advantages and try to debunk various myths. They paint an overly simplistic image, though. As mentioned above, large register sets mean high context switch times, which have to be avoided in general purpose OSes. See the problems of the IA-64 processor when it comes to context switch-intensive operations. The long execution time for vector operations is also a problem if interrupts are involved. If an interrupt is raised, the processor must stop its current work and start working on handling the interrupt. After that, it must resume executing the interrupted code. It is generally a big problem to interrupt an instruction in the middle of the work; it is not impossible, but it is complicated. For long running instructions this has to happen, or the instructions must be implemented in a restartable fashion, since otherwise the interrupt reaction time is too high. The latter is not acceptable.

Vector units also were forgiving as far as alignment of the memory access is concerned, which shaped the algorithms which were developed. Some of today's processors (especially RISC processors) require strict alignment so the extension to full vector operations is not trivial. There are big potential upsides to having vector operations, especially when striding and indirection are supported, so that we can hope to see this functionality in the future.

Appendices and bibliography

The appendices and bibliography page contains, among other things, the source code for a number of the benchmark programs used for this document, more information on oprofile, some discussion of memory types, an introduction to libNUMA, and the bibliography.

This article was contributed by Ulrich Drepper
[*Editor's note: what follows is the set of appendices and the bibliography from What every programmer should know about memory by Ulrich Drepper.*]

9 Examples and Benchmark Programs

9.1 Matrix Multiplication

This is the complete benchmark program for the matrix multiplication in section Section 6.2.1. For details on the intrinsics used the reader is referred to Intel's reference manual.

```
#include <stdlib.h>
#include <stdio.h>
#include <emmintrin.h>
#define N 1000
double res[N][N] __attribute__ ((aligned (64)));
double mul1[N][N] __attribute__ ((aligned (64)));
double mul2[N][N] __attribute__ ((aligned (64)));
#define SM (CLS / sizeof (double))



int

main (void)

{

  // ... Initialize mul1 and mul2



  int i, i2, j, j2, k, k2;

  double *restrict rres;

  double *restrict rmul1;
```

```
  double *restrict rmul2;
  for (i = 0; i < N; i += SM)
    for (j = 0; j < N; j += SM)
      for (k = 0; k < N; k += SM)
        for (i2 = 0, rres = &res[i][j], rmul1 = &mul1[i][k]; i2 < SM;
             ++i2, rres += N, rmul1 += N)
          {
            _mm_prefetch (&rmul1[8], _MM_HINT_NTA);
            for (k2 = 0, rmul2 = &mul2[k][j]; k2 < SM; ++k2, rmul2 += N)
              {
                __m128d m1d = _mm_load_sd (&rmul1[k2]);
                m1d = _mm_unpacklo_pd (m1d, m1d);
                for (j2 = 0; j2 < SM; j2 += 2)
                  {
                    __m128d m2 = _mm_load_pd (&rmul2[j2]);
                    __m128d r2 = _mm_load_pd (&rres[j2]);
                    _mm_store_pd (&rres[j2],
                                  _mm_add_pd (_mm_mul_pd (m2, m1d),
r2));
                  }
              }
          }

  // ... use res matrix
```

```
  return 0;
}
```

The structure of the loops is pretty much the same as in the final incarnation in Section 6.2.1. The one big change is that loading the `rmul1[k2]` value has been pulled out of the inner loop since we have to create a vector where both elements have the same value. This is what the `_mm_unpacklo_pd()` intrinsic does.

The only other noteworthy thing is that we explicitly aligned the three arrays so that the values we expect to be in the same cache line actually are found there.

9.2 Debug Branch Prediction

If, as recommended, the definitions of `likely` and `unlikely` from Section 6.2.2 are used, it is easy {*At least with the GNU toolchain.*} to have a debug mode to check whether the assumptions are really true. The definitions of the macros could be replaced with this:

```
#ifndef DEBUGPRED
# define unlikely(expr) __builtin_expect (!!(expr), 0)
# define likely(expr) __builtin_expect (!!(expr), 1)
#else
asm (".section predict_data, \"aw\"; .previous\n"
     ".section predict_line, \"a\"; .previous\n"
     ".section predict_file, \"a\"; .previous");
# ifdef __x86_64__
#  define debugpred__(e, E) \
  ({ long int _e = !!(e); \
     asm volatile (".pushsection predict_data\n" \
                   "..predictcnt%=: .quad 0; .quad 0\n" \
                   ".section predict_line; .quad %c1\n" \
                   ".section predict_file; .quad %c2; .popsection\n" \
                   "addq $1,..predictcnt%=(,%0,8)" \
                   : : "r" (_e == E), "i" (__LINE__), "i" (__FILE__));
\
     __builtin_expect (_e, E); \
  })
```

```
# elif defined __i386__
#  define debugpred__(e, E) \
  ({ long int _e = !!(e); \
     asm volatile (".pushsection predict_data\n" \
                   "..predictcnt%=: .long 0; .long 0\n" \
                   ".section predict_line; .long %c1\n" \
                   ".section predict_file; .long %c2; .popsection\n" \
                   "incl ..predictcnt%=(,%0,4)" \
                   : : "r" (_e == E), "i" (__LINE__), "i" (__FILE__));
\
     __builtin_expect (_e, E); \
  })
# else
#  error "debugpred__ definition missing"
# endif
# define unlikely(expt) debugpred__ ((expr), 0)
# define likely(expr) debugpred__ ((expr), 1)
#endif
```

These macros use a lot of functionality provided by the GNU assembler and linker when creating ELF files. The first `asm` statement in the`DEBUGPRED` case defines three additional sections; it mainly gives the assembler information about how the sections should be created. All sections are available at runtime, and the `predict_data` section is writable. It is important that all section names are valid C identifiers. The reason will be clear shortly.

The new definitions of the `likely` and `unlikely` macros refer to the machine-specific `debugpred__` macro. This macro has the following tasks:

1. allocate two words in the `predict_data` section to contain the counts for correct and incorrect predictions. These two fields get a unique name through the use of `%=`; the leading dots makes sure the symbols do not pollute the symbol table.
2. allocate one word in the `predict_line` section to contain the line number of the `likely` or `unlikely` macro use.

3. allocate one word in the `predict_file` section to contain a pointer to the file name of the `likely` or `unlikely` macro use.
4. increment the “correct” or “incorrect” counter created for this macro according to the actual value of the expression `e`. We do not use atomic operations here because they are massively slower and absolute precision in the unlikely case of a collision is not that important. It is easy enough to change if wanted.

The `.pushsection` and `.popsection` pseudo-ops are described in the assembler manual. The interested reader is asked to explore the details of these definition with the help of the manuals and some trial and error.

These macros automatically and transparently take care of collecting the information about correct and incorrect branch predictions. What is missing is a method to get to the results. The simplest way is to define a destructor for the object and print out the results there. This can be achieved with a function defined like this:

```
extern long int __start_predict_data;
extern long int __stop_predict_data;
extern long int __start_predict_line;
extern const char *__start_predict_file;



static void

__attribute__ ((destructor))

predprint(void)

{

  long int *s = &__start_predict_data;

  long int *e = &__stop_predict_data;

  long int *sl = &__start_predict_line;

  const char **sf = &__start_predict_file;

  while (s < e) {
```

```
    printf("%s:%ld: incorrect=%ld, correct=%ld%s\n", *sf, *sl, s[0],
s[1],

           s[0] > s[1] ? "   <==== WARNING" : "");

    ++sl;

    ++sf;

    s += 2;

  }

}
```

Here the fact that the section names are valid C identifiers comes into play; it is used by the GNU linker to automatically define, if needed, two symbols for the section. The `__start_`*XYZ* symbols corresponds to the beginning of the section *XYZ* and `__stop_`*XYZ* is the location of the first byte following section *XYZ*. These symbols make it possible to iterate over the section content at runtime. Note that, since the content of the sections can come from all the files the linker uses at link time, the compiler and assembler do not have enough information to determine the size of the section. Only with these magic linker-generated symbols is it possible to iterate over the section content.

The code does not iterate over one section only, though; there are three sections involved. Since we know that, for every two words added to the `predict_data` section we add one word to each of the `predict_line` and `predict_file` sections, we do not have to check the boundaries of these two sections. We just carry pointers along and increment them in unison.

The code prints out a line for every prediction which appears in the code. It highlights those uses where the prediction is incorrect. Of course, this can be changed, and the debug mode could be restricted to flag only the entries which have more incorrect predictions than correct ones. Those are candidates for change. There are details which complicate the issue; for example, if the branch prediction happens inside a macro which is used in

multiple places, all the macro uses must be considered together before making a final judgment.

Two last comments: the data required for this debugging operation is not small, and, in case of DSOs, expensive (the `predict_file` section must be relocated). Therefore the debugging mode should not be enabled in production binaries. Finally, each executable and DSO creates it own output, this must be kept in mind when analyzing the data.

9.3 Measure Cache Line Sharing Overhead

This section contains the test program to measure the overhead of using variables on the same cache line versus variables on separate cache lines.

```
#include <error.h>
#include <pthread.h>
#include <stdlib.h>

#define N (atomic ? 10000000 : 500000000)

static int atomic;
static unsigned nthreads;
static unsigned disp;
static long **reads;

static pthread_barrier_t b;

static void *
tf(void *arg)
{
  long *p = arg;

  if (atomic)
    for (int n = 0; n < N; ++n)
      __sync_add_and_fetch(p, 1);
  else
    for (int n = 0; n < N; ++n)
      {
        *p += 1;
        asm volatile("" : : "m" (*p));
      }

  return NULL;
}
```

```
int
main(int argc, char *argv[])
{
  if (argc < 2)
    disp = 0;
  else
    disp = atol(argv[1]);

  if (argc < 3)
    nthreads = 2;
  else
    nthreads = atol(argv[2]) ?: 1;

  if (argc < 4)
    atomic = 1;
  else
    atomic = atol(argv[3]);

  pthread_barrier_init(&b, NULL, nthreads);

  void *p;
  posix_memalign(&p, 64, (nthreads * disp ?: 1) * sizeof(long));
  long *mem = p;

  pthread_t th[nthreads];
  pthread_attr_t a;
  pthread_attr_init(&a);
  cpu_set_t c;
  for (unsigned i = 1; i < nthreads; ++i)
    {
      CPU_ZERO(&c);
      CPU_SET(i, &c);
      pthread_attr_setaffinity_np(&a, sizeof(c), &c);
      mem[i * disp] = 0;
      pthread_create(&th[i], &a, tf, &mem[i * disp]);
    }

  CPU_ZERO(&c);
  CPU_SET(0, &c);
  pthread_setaffinity_np(pthread_self(), sizeof(c), &c);
  mem[0] = 0;
  tf(&mem[0]);
```

```
  if ((disp == 0 && mem[0] != nthreads * N)
      || (disp != 0 && mem[0] != N))
    error(1,0,"mem[0] wrong: %ld instead of %d",
          mem[0], disp == 0 ? nthreads * N : N);

  for (unsigned i = 1; i < nthreads; ++i)
    {
      pthread_join(th[i], NULL);
      if (disp != 0 && mem[i * disp] != N)
        error(1,0,"mem[%u] wrong: %ld instead of %d", i, mem[i * disp],
N);
    }

  return 0;
}
```

The code is provided here mainly as an illustration of how to write a program which measures effects like cache line overhead. The interesting parts are the bodies of the loops in `tf`. The `__sync_add_and_fetch` intrinsic, known to the compiler, generates an atomic add instruction. In the second loop we have to “consume” the result of the increment (through the inline `asm` statement). The `asm` does not introduce any actual code; instead, it prevents the compiler from lifting the increment operation out of the loop.

The second interesting part is that the program pins the threads onto specific processors. The code assumes the processors are numbered 0 to 3, which is usually the case if the machine has four or more logical processors. The code could have used the interfaces from libNUMA to determine the numbers of the usable processors, but this test program should be widely usable without introducing this dependency. It is easy enough to fix up one way or another.

10 Some OProfile Tips

The following is not meant as a tutorial on how to use oprofile. There are entire documents written on that topic. Instead it is meant to give a few higher-level hints on how to look at one's programs to find possible trouble spots. But before that we must at least have a minimal introduction.

10.1 Oprofile Basics

Oprofile works in two phases: collection and then analysis. The collection is performed by the kernel; it cannot be done at userlevel since the measurements use the performance counters of the CPU. These counters require access to MSRs which, in turn, requires privileges.

Each modern processor provides its own set of performance counters. On some architectures a subset of the counters are provided by all processor implementations while the others differ from version to version. This makes giving general advice about the use of oprofile hard. There is not (yet) a higher-level abstraction for the counters which could hide these details.

The processor version also controls how many events can be traced at any one time, and in which combination. This adds yet more complexity to the picture.

If the user knows the necessary details about the performance counters, the opcontrol program can be used to select the events which should be counted. For each event it is necessary to specify the “overrun number” (the number of events which must occur before the CPU is interrupted to record an event), whether the event should be counted for userlevel and/or the kernel, and finally a “unit mask” (it selects sub-functions of the performance counter).

To count the CPU cycles on x86 and x86-64 processors, one has to issue the following command:

```
opcontrol --event CPU_CLK_UNHALTED:30000:0:1:1
```

The number 30000 is the overrun number. Choosing a reasonable value is important for the behavior of the system and the collected data. It is a bad idea ask to receive data about every single occurrence of the event. For many events, this would bring the machine to a standstill since all it would do is work on the data collection for the event overrun; this is why oprofile enforces a minimum value. The minimum values differ for each event since different events have a different probability of being triggered in normal code.

Choosing a very high number reduces the resolution of the profile. At each overrun oprofile records the address of the instruction which is executed at that moment; for x86 and PowerPC it can, under some circumstances, record the backtrace as well. {*Backtrace support will hopefully be available for all architectures at some point.*} With a coarse resolution, the hot spots might not get a representative number of hits; it is all about probabilities, which is why oprofile is called a probabilistic profiler. The lower the overrun number

is the higher the impact on the system in terms of slowdown but the higher the resolution.

If a specific program is to be profiled, and the system is not used for production, it is often most useful to use the lowest possible overrun value. The exact value for each event can be queried using

```
opcontrol --list-events
```

This might be problematic if the profiled program interacts with another process, and the slowdown causes problems in the interaction. Trouble can also result if a process has some realtime requirements which cannot be met when it is interrupted often. In this case a middle ground has to be found. The same is true if the entire system is to be profiled for extended periods of time. A low overrun number would mean the massive slowdowns. In any case, oprofile, like any other profiling mechanism, introduces uncertainty and inaccuracy.

The profiling has to be started with `opcontrol --start` and can be stopped with `opcontrol --stop`. While oprofile is active it collects data; this data is first collected in the kernel and then send to a userlevel daemon in batches, where it is decoded and written to a filesystem. With `opcontrol --dump` it is possible to request all information buffered in the kernel to be released to userlevel.

The collected data can contain events from different performance counters. The numbers are all kept in parallel unless the user selects to wipe the stored data in between separate oprofile runs. It is possible to accumulate data from the same event at different occasions. If an event is encountered during different profiling runs the numbers are added if this is what is selected by the user.

The userlevel part of the data collection process demultiplexes the data. Data for each file is stored separately. It is even possible to differentiate DSOs used by individual executable and, even, data for individual threads.

The data thus produced can be archived using `oparchive`. The file produced by this command can be transported to another machine and the analysis can be performed there.

With the opreport program one can generate reports from the profiling results. Using opannotate it is possible to see where the various events

happened: which instruction and, if the data is available, in which source line. This makes it easy to find hot spots. Counting CPU cycles will point out where the most time is spent (this includes cache misses) while counting retired instructions allows finding where most of the executed instructions are—there is a big difference between the two.

A single hit at an address usually has no meaning. A side effect of statistical profiling is that instructions which are only executed a few times, or even only once, might be attributed with a hit. In such a case it is necessary to verify the results through repetition.

10.2 What It Looks Like

An oprofile session can look as simple as this:

```
$ opcontrol -i cachebench
$ opcontrol -e INST_RETIRED:6000:0:0:1 --start
$ ./cachebench ...
$ opcontrol -h
```

Note that these commands, including the actual program, are run as root. Running the program as root is done here only for simplicity; the program can be executed by any user and oprofile would pick up on it. The next step is analyzing the data. With opreport we see:

```
CPU: Core 2, speed 1596 MHz (estimated)
Counted INST_RETIRED.ANY_P events (number of instructions retired) with
a unit mask of
0x00 (No unit mask) count 6000
INST_RETIRED:6000|
  samples|      %|
------------------
   116452 100.000 cachebench
```

This means we collected a bunch of events; opannotate can now be used to look at the data in more detail. We can see where in the program the most events were recorded. Part of the `opannotate --source` output looks like this:

```
               :static void
               :inc (struct l *l, unsigned n)
               :{
               :  while (n-- > 0) /* inc total:  13980 11.7926 */
               :    {
     5  0.0042 :      ++l->pad[0].l;
```

```
 13974 11.7875 :      l = l->n;
     1 8.4e-04 :      asm volatile ("" :: "r" (l));
               :    }
               :}
```

That is the inner function of the test, where a large portion of the time is spent. We see the samples spread out over all three lines of the loop. The main reason for this is that the sampling is not always 100% accurate with respect to the recorded instruction pointer. The CPU executes instructions out of order; reconstructing the exact sequence of execution to produce a correct instruction pointer is hard. The most recent CPU versions try to do this for a select few events but it is, in general, not worth the effort—or simply not possible. In most cases it does not really matter. The programmer should be able to determine what is going on even if there is a normally-distributed set of samples.

10.3 Starting To Profile

When starting to analyze a body of code, one certainly can start looking at the places in the program where the most time is spent. That code should certainly be optimized as well as possible. But what happens next? Where is the program spending *unnecessary* time? This question is not so easy to answer.

One of the problems in this situation is that absolute values do not tell the real story. One loop in the program might demand the majority of the time, and this is fine. There are many possible reasons for the high CPU utilization, though. But what is more common, is that CPU usage is more evenly spread throughout the program. In this case, the absolute values point to many places, which is not useful.

In many situations it is helpful to look at ratios of two events. For instance, the number of mispredicted branches in a function can be meaningless if there is no measure for how often a function was executed. Yes, the absolute value is relevant for the program's performance. The ratio of mispredictions per call is more meaningful for the code quality of the function. Intel's optimization manual for x86 and x86-64 [intelopt] describes ratios which should be investigated (Appendix B.7 in the cited document for Core 2 events). A few of the ratios relevant for memory handling are the following.

Instruction Fetch Stall	CYCLES_L1I_MEM_STALLED/CPU_CLK_UNHALTED.CORE	Ratio of cycles during which in instruction decoder is

		waiting for new data due to cache or ITLB misses.
ITLB Miss Rate	ITLB_MISS_RETIRED /INST_RETIRED.ANY	ITLB misses per instruction. If this ratio is high the code is spread over too many pages.
L1I Miss Rate	L1I_MISSES /INST_RETIRED.ANY	L1i misses per instruction. The execution flow is unpredictable or the code size is too large. In the former case avoiding indirect jumps might help. In the latter case block reordering or avoiding inlining might help.
L2 Instruction Miss Rate	L2_IFETCH.SELF.I_STATE/ INST_RETIRED.ANY	L2 misses for program code per instruction. Any value larger than zero indicates code locality problems which are even worse than L1i

		misses.
Load Rate	L1D_CACHE_LD.MESI /CPU_CLK_UNHALTED.CORE	Read operations per cycle. A Core 2 core can service one load operation. A high ratio means the execution is bound by memory reads.
Store Order Block	STORE_BLOCK.ORDER /CPU_CLK_UNHALTED.CORE	Ratio if stores blocked by previous stores which miss the cache.
L1d Rate Blocking Loads	LOAD_BLOCK.L1D /CPU_CLK_UNHALTED.CORE	Loads from L1d blocked by lack of resources. Usually this means too many concurrent L1d accesses.
L1D Miss Rate	L1D_REPL /INST_RETIRED.ANY	L1d misses per instruction. A high rate means that prefetching is not effective and L2 is used too often.
L2 Data Miss Rate	L2_LINES_IN.SELF.ANY /INST_RETIRED.ANY	L2 misses for data per instruction. If the value is significantly

		greater than zero, hardware and software prefetching is ineffective. The processor needs more (or earlier) software prefetching help.
L2 Demand Miss Rate	L2_LINES_IN.SELF.DEMAND/ INST_RETIRED.ANY	L2 misses for data per instruction for which the hardware prefetcher was not used at all. That means, prefetching has not even started.
Useful NTA Prefetch Rate	SSE_PRE_MISS.NTA /SSS_PRE_EXEC.NTA	Ratio of useful non-temporal prefetch relative to the total number of all non-temporal prefetches. A low value means many values are already in the cache. This ratio can be computed for the other prefetch types as well.

Late NTA Prefetch Rate	LOAD_HIT_PRE /SSS_PRE_EXEC.NTA	Ratio of load requests for data with ongoing prefetch relative to the total number of all non-temporal prefetches. A high value means the software prefetch instruction is issued too late. This ratio can be computed for the other prefetch types as well.

For all these ratios, the program should be run with oprofile being instructed to measure both events. This guarantees the two counts are comparable. Before the division, one has to make sure that the possibly different overrun values are taken into account. The simplest way is to ensure this is by multiplying each events counter by the overrun value.

The ratios are meaningful for whole programs, at the executable/DSO level, or even at the function level. The deeper one looks into the program, the more errors are included in the value.

What is needed to make sense of the ratios are baseline values. This is not as easy as it might seem. Different types of code has different characteristics and a ratio value which is bad in one program might be normal in another program.

11 Memory Types

Though it is not necessary knowledge for efficient programming, it might be useful to describe some more technical details of available memory types.

Specifically we are here interested in the difference of “registered” versus “unregistered” and ECC versus non-ECC DRAM types.

The terms “registered” and “buffered” are used synonymously when describing a DRAM type which has one additional component on the DRAM module: a buffer. All DDR memory types can come in registered and unregistered form. For the unregistered modules, the memory controller is directly connected to all the chips on the module. Figure 11.1 shows the setup.

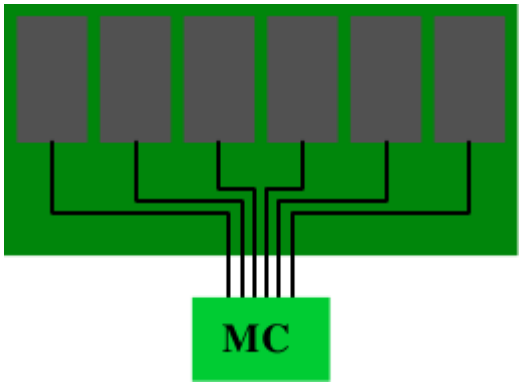


Figure 11.1: Unregistered DRAM Module

Electrically this is quite demanding. The memory controller must be able to deal with the capacities of all the memory chips (there are more than the six shown in the figure). If the memory controller (MC) has a limitation, or if many memory modules are to be used, this setup is not ideal.

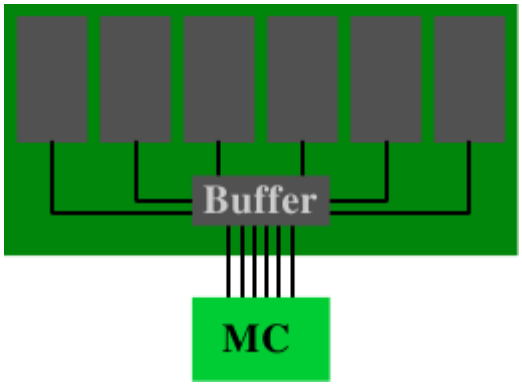


Figure 11.2: Registered DRAM Module

Buffered (or registered) memory changes the situation: instead of directly connecting the RAM chips on the DRAM module to the memory, they are connected to a buffer which, in turn, is then connected to the memory controller. This significantly reduces the complexity of the electrical

connections. The ability of the memory controllers to drive DRAM modules increases by a factor corresponding to the number of connections saved.

With these advantages the question is: why aren't all DRAM modules buffered? There are several reasons. Obviously, buffered modules are a bit more complicated and, hence, more expensive. Cost is not the only factor, though. The buffer delays the signals from the RAM chips a bit; the delay must be high enough to ensure that all signals from the RAM chips are buffered. The result is that the latency of the DRAM module increases. A last factor worth mentioning here is that the additional electrical component increases the energy cost. Since the buffer has to operate at the frequency of the bus this component's energy consumption can be significant.

With the other factors of the use of DDR2 and DDR3 modules it is usually not possible to have more than two DRAM modules per bank. The number of pins of the memory controller limit the number of banks (to two in commodity hardware). Most memory controllers are able to drive four DRAM modules and, therefore, unregistered modules are sufficient. In server environments with high memory requirements the situation might be different.

A different aspect of some server environments is that they cannot tolerate errors. Due to the minuscule charges held by the capacitors in the RAM cells, errors are possible. People often joke about cosmic radiation but this is indeed a possibility. Together with alpha decays and other natural phenomena, they lead to errors where the content of RAM cell changes from 0 to 1 or vice versa. The more memory is used, the higher the probability of such an event.

If such errors are not acceptable, ECC (Error Correction Code) DRAM can be used. Error correction codes enable the hardware to recognize incorrect cell contents and, in some cases, correct the errors. In the old days, parity checks only recognized errors, and the machine had to be stopped when one was detected. With ECC, instead, a small number of erroneous bits can be automatically corrected. If the number of errors is too high, though, the memory access cannot be performed correctly and the machine still stops. This is a rather unlikely case for working DRAM modules, though, since multiple errors must happen on the same module.

When we speak about ECC memory we are actually not quite correct. It is not the memory which performs the error checking; instead, it is the memory controller. The DRAM modules simply provide more storage and transport the additional non-data bits along with the real data. Usually, ECC memory stores one additional bit for each 8 data bits. Why 8 bits are used will be explained a bit later.

Upon writing data to a memory address, the memory controller computes the ECC for the new content on the fly before sending that data and ECC onto the memory bus. When reading, the data plus the ECC is received, the memory controller computes the ECC for the data, and compares it with the ECC transmitted from the DRAM module. If the ECCs match everything is fine. If they do not match, the memory controller tries to correct the error. If this correction is not possible, the error is logged and the machine is possibly halted.

	SEC		SEC/DED	
Data Bits W	ECC Bits E	Overhead	ECC Bits E	Overhead
4	3	75.0%	4	100.0%
8	4	50.0%	5	62.5%
16	5	31.3%	6	37.5%
32	6	18.8%	7	21.9%
64	7	10.9%	8	12.5%

Table 11.1: ECC and Data Bits Relationship

Several techniques for error correction are in use but, for DRAM ECC, usually Hamming codes are used. Hamming codes originally were used to encode four data bits with the ability to recognize and correct one flipped bit (SEC, Single Error Correction). The mechanism can easily be extended to more data bits. The relationship between the number of data bits W and the number of bits for the error code **E** is described by the equation

$$E = \lceil \log_2 (W+E+1) \rceil$$

Solving this equation iteratively results in the values shown in the second column of Table 11.1. With an additional bit, we can recognize two flipped bits using a simple parity bit. This is then called SEC/DED, Single Error Correction/Double Error Detection. With this additional bit we arrive at the values in the fourth column of Table 11.1. The overhead for W=64 is sufficiently low and the numbers (64, 8) are multiples of 8, so this is a natural selection for ECC. On most modules, each RAM chip produces 8 bits and, therefore, any other combination would lead to less efficient solution.

	7	6	5	4	3	2	1
ECC Word	D	D	D	P	D	P	P

P_1	Parity	D	—	D	—	D	—	P
P_2	Parity	D	D	—	—	D	P	—
P_4	Parity	D	D	D	P	—	—	—

Figure 11.3: Hamming Generation Matrix Construction

The Hamming code computation is easy to demonstrate with a code using W=4 and E=3. We compute parity bits at strategic places in the encoded word. Figure 11.3 shows the principle. At the bit positions corresponding to the powers of two the parity bits are added. The parity sum for the first parity bit P_1 contains every second bit. The parity sum for the second parity bit P_2 contains data bits 1, 3, and 4 (encoded here as 3, 6, and 7). Similar ly P_4 is computed.

The computation of the parity bits can be more elegantly described using a matrix multiplication. We construction a matrix G = [I|A] where I is the identity matrix and A is the parity generation matrix we can determine from Figure 11.3.

$$G = \left[\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array}\right]$$

The columns of A are constructed from the bits used in the computation of P_1, P_2, and P_4. If we now represent each input data item as a 4-dimensional vector d we can compute r=d·G and get a 7-dimensional vector r. This is the data which in the case of ECC DDR is stored.

To decode the data we construct a new matrix $H=[A^T|I]$ where A^T is the transposed parity generation matrix from the computation of G. That means:

$$H = \left[\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right]$$

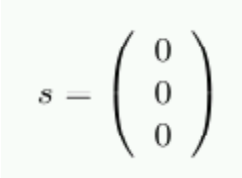
The result of H·r shows whether the stored data is defective. If this is not the case, the product is the 3-dimensional vector $(0\ 0\ 0)^T$. Otherwise the value of

the product, when interpreted as the binary representation of a number, indicates the column number with the flipped bit.

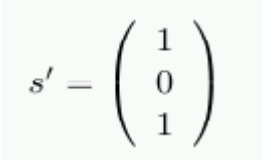
As an example, assume d=(1 0 0 1). This results in

r = (1 0 0 1 0 0 1)

Performing the test using the multiplication with H results in



Now, assume we have a corruption of the stored data and read back from memory r' = (1 0 **1** 1 0 0 1). In this case we get



The vector is not the null vector and, when interpreted as a number, s' has the value 5. This is the number of the bit we flipped in r' (starting to count the bits from 1). The memory controller can correct the bit and the programs will not notice that there has been a problem.

Handling the extra bit for the DED part is only slightly more complex. With more effort is it possible to create codes which can correct two flipped bits and more. It is probability and risk which decide whether this is needed. Some memory manufacturers say an error can occur in 256MB of RAM every 750 hours. By doubling the amount of memory the time is reduced by 75%. With enough memory the probability of experiencing an error in a short time can be significant and ECC RAM becomes a requirement. The time frame could even be so small that the SEC/DED implementation is not sufficient.

Instead of implementing even more error correction capabilities, server motherboards have the ability to automatically read all memory over a given timeframe. That means, whether or not the memory was actually requested by the processor, the memory controller reads the data and, if the ECC check fails, writes the corrected data back to memory. As long as the probablity of incurring less than two memory errors in the time frame

needed to read all of memory and write it back is acceptable, SEC/DED error correction is a perfectly reasonable solution.

As with registered DRAM, the question has to be asked: why is ECC DRAM not the norm? The answer to this question is the same as for the equivalent question about registered RAM: the extra RAM chip increases the cost and the parity computation increases the delay. Unregistered, non-ECC memory can be significantly faster. Because of the similarity of the problems of registered and ECC DRAM, one usually only finds registered, ECC DRAM and not registered, non-ECC DRAM.

There is another method to overcome memory errors. Some manufacturers offer what is often incorrectly called “memory RAID” where the data is distributed redundantly over multiple DRAM modules, or at least RAM chips. Motherboards with this feature can use unregistered DRAM modules, but the increased traffic on the memory busses is likely to negate the difference in access times for ECC and non-ECC DRAM modules.

12 libNUMA Introduction

Although much of the information programmers need to schedule threads optimally, allocate memory appropriately, etc. is available, this information is cumbersome to get at. The existing NUMA support library (libnuma, in the numactl package on RHEL/Fedora systems) does not, by a long shot, provide adequate functionality.

As a response, the author has proposed a new library which provides all the functionality needed for NUMA. Due to the overlap of memory and cache hierarchy handling, this library is also useful for non-NUMA systems with multi-thread and multi-core processors—almost every currently-available machine.

The functionality of this new library is urgently needed to follow the advice given in this document. This is the only reason why it is mentioned here. The library (as of this writing) is not finished, not reviewed, not polished, and not (widely) distributed. It might change significantly in future. It is currently available at http://people.redhat.com/drepper/libNUMA.tar.bz2.

The interfaces of this library depend heavily on the information exported by the `/sys` filesystem. If this filesystem is not mounted, many functions will simply fail or provide inaccurate information. This is particularly important to remember if a process is executed in a `chroot`jail.

The interface header for the library contains currently the following definitions:

```
typedef memnode_set_t;


#define MEMNODE_ZERO_S(setsize, memnodesetp)

#define MEMNODE_SET_S(node, setsize, memnodesetp)

#define MEMNODE_CLR_S(node, setsize, memnodesetp)

#define MEMNODE_ISSET_S(node, setsize, memnodesetp)

#define MEMNODE_COUNT_S(setsize, memnodesetp)


#define MEMNODE_EQUAL_S(setsize, memnodesetp1, memnodesetp2)


#define MEMNODE_AND_S(setsize, destset, srcset1, srcset2)

#define MEMNODE_OR_S(setsize, destset, srcset1, srcset2)

#define MEMNODE_XOR_S(setsize, destset, srcset1, srcset2)


#define MEMNODE_ALLOC_SIZE(count)

#define MEMNODE_ALLOC(count)

#define MEMNODE_FREE(memnodeset)


int NUMA_cpu_system_count(void);

int NUMA_cpu_system_mask(size_t destsize, cpu_set_t *dest);


int NUMA_cpu_self_count(void);
```

```
int NUMA_cpu_self_mask(size_t destsize, cpu_set_t *dest);


int NUMA_cpu_self_current_idx(void);

int NUMA_cpu_self_current_mask(size_t destsize, cpu_set_t *dest);


ssize_t NUMA_cpu_level_mask(size_t destsize, cpu_set_t *dest,

                            size_t srcsize, const cpu_set_t *src,

                            unsigned int level);


int NUMA_memnode_system_count(void);

int NUMA_memnode_system_mask(size_t destsize, memnode_set_t *dest);


int NUMA_memnode_self_mask(size_t destsize, memnode_set_t *dest);


int NUMA_memnode_self_current_idx(void);

int NUMA_memnode_self_current_mask(size_t destsize, memnode_set_t
*dest);


int NUMA_cpu_to_memnode(size_t cpusetsize, const cpu_set_t *cpuset,

                        size_t __memnodesize, memnode_set_t
*memnodeset);

int NUMA_memnode_to_cpu(size_t memnodesize, const memnode_set_t
*memnodeset,

                        size_t cpusetsize, cpu_set_t *cpuset);
```

```
int NUMA_mem_get_node_idx(void *addr);

int NUMA_mem_get_node_mask(void *addr, size_t size,

                           size_t destsize, memnode_set_t *dest);
```

The MEMNODE_* macros are similar in form and functionality to the CPU_* macros introduced in section Section 6.4.3. There are no non-_S variants of the macros, they all require a size parameter.

The memnode_set_t type is the equivalent of cpu_set_t, but this time for memory nodes. Note that the number of memory nodes need not have anything to do with the number of CPUs and vice versa. It is possible to have many CPUs per memory node or even no CPU at all. The size of dynamically allocated memory node bit sets should, therefore, not be determined by the number of CPUs.

Instead, the NUMA_memnode_system_count interface should be used. It returns the number of nodes currently registered. This number might grow or shrink over time. More often than not, though, it will remain constant, and is therefore a good value to use for sizing memory node bit sets. The allocation, again similar to the CPU_ macros, happens using MEMNODE_ALLOC_SIZE, MEMNODE_ALLOC and MEMNODE_FREE.

As a last parallel with the CPU_* macros, the library also provides macros to compare memory node bit sets for equality and to perform logical operations.

The NUMA_cpu_* functions provide functionality to handle CPU sets. In part, the interfaces only make existing functionality available under a new name. NUMA_cpu_system_count returns the number of CPUs in the system, the NUMA_CPU_system_mask variant returns a bit mask with the appropriate bits set—functionality which is not otherwise available.

NUMA_cpu_self_count and NUMA_cpu_self_mask return information about the CPUs the current thread is currently allowed to run on.NUMA_cpu_self_current_idx returns the index of the currently used CPU. This information might already be stale when returned, due to scheduling decisions the kernel can make; it always has to be assumed to be inaccurate. The NUMA_cpu_self_current_mask returns the same information and sets the appropriate bit in the bit set.

NUMA_memnode_system_count has already been introduced. NUMA_memnode_system_mask is the equivalent function which fills in a bit set.NUMA_memnode_self_mask fills in a bit set according to the memory nodes which are directly attached to any of the CPUs the thread can currently run on.

Even more specialized information is returned by NUMA_memnode_self_current_idx and NUMA_memnode_self_current_mask. The information returned is the memory node which is connected to the processor the thread is currently running on. Just as for the NUMA_cpu_self_current_* functions, this information can already be stale when the function returns; it can only be used as a hint.

The NUMA_cpu_to_memnode function can be used to map a set of CPUs to the set of directly-attached memory nodes. If only a single bit is set in the CPU set, one can determine which memory node each CPU belongs to. Currently, there is no support in Linux for a single CPU belonging to more than one memory node; this could, theoretically, change in future. To map in the other direction the NUMA_memnode_to_cpu function can be used.

If memory is already allocated, it is sometimes useful to know where it is allocated. This is what the NUMA_mem_get_node_idx andNUMA_mem_get_node_mask allow the programmer to determine. The former function returns the index of the memory node on which the page corresponding to the address specified by the parameter is allocated—or will be allocated according to the currently

installed policy if the page is not yet allocated. The second function can perform the work for a whole address range; it returns the information in the form of a bit set. The function's return value is the number of different memory nodes which are used.

In the remainder of this section we will see a few example for use cases of these interfaces. In all cases we skip the error handling and the case where the number of CPUs and/or memory nodes is too large for the `cpu_set_t` and `memnode_set_t` types respectively. Making the code robust is left as an exercise to the reader.

12.1 Determine Thread Sibling of Given CPU

To schedule helper threads, or other threads which benefit from being scheduled on a thread of a given CPU, a code sequence like the following can be used.

```
cpu_set_t cur;
CPU_ZERO(&cur);
CPU_SET(cpunr, &cur);
cpu_set_t hyperths;
NUMA_cpu_level_mask(sizeof(hyperths), &hyperths, sizeof(cur), &cur, 1);
CPU_CLR(cpunr, &hyperths);
```

The code first generates a bit set for the CPU specified by `cpunr`. This bit set is then passed to `NUMA_cpu_level_mask` along with the fifth parameter specifying that we are looking for hyper-threads. The result is returned in the `hyperths` bit set. All that remains to be done is to clear the bit corresponding to the original CPU.

12.2 Determine Core Siblings of Given CPU

If two threads should not be scheduled on two hyper-threads, but can benefit from cache sharing, we need to determine the other cores of the processor. The following code sequence does the trick.

```
  cpu_set_t cur;
  CPU_ZERO(&cur);
  CPU_SET(cpunr, &cur);
  cpu_set_t hyperths;
```

```
  int nhts = NUMA_cpu_level_mask(sizeof(hyperths), &hyperths,
sizeof(cur), &cur, 1);
  cpu_set_t coreths;
  int ncs = NUMA_cpu_level_mask(sizeof(coreths), &coreths, sizeof(cur),
&cur, 2);
  CPU_XOR(&coreths, &coreths, &hyperths);
  ncs -= nhts;
```

The first part of the code is identical to the code to determine hyper-threads. This is no coincidence since we have to distinguish the hyper-threads of the given CPU from the other cores. This is implemented in the second part which calls `NUMA_cpu_level_mask` again, but, this time, with a level of 2. All that remains to be done is to remove all hyper-threads of the given CPU from the result. The variables `nhts` and `ncs` are used to keep track of the number of bits set in the respective bit sets.

The resulting mask can be used to schedule another thread. If no other thread has to be explicitly scheduled, the decision about the core to use can be left to the OS. Otherwise one can iteratively run the following code:

```
  while (ncs > 0) {
    size_t idx = 0;
    while (! CPU_ISSET(idx, &ncs))
      ++idx;
    CPU_ZERO(&cur);
    CPU_SET(idx, &cur);
    nhts = NUMA_cpu_level_mask(sizeof(hyperths), &hyperths, sizeof(cur),
&cur, 1);
    CPU_XOR(&coreths, &coreths, hyperths);
    ncs -= nhts;



    ... schedule thread on CPU idx ...

  }
```

The loop picks, in each iteration, a CPU number from the remaining, used cores. It then computes all the hyper-threads for the this CPU. The resulting bit set is then subtracted (using `CPU_XOR`) from the bit set of the available cores. If the XOR operation does not remove anything, something is really

wrong. The `ncs` variable is updated and we are ready for the next round, but not before the scheduling decisions are made. At the end, any of `idx`, `cur`, or `hyperths` can be used to schedule a thread, depending on the requirements of the program. Often it is best to leave the OS as much freedom as possible and, therefore, to use the `hyperths` bit set so that the OS can select the best hyper-thread.

Bibliography

[amdccnuma]
Performance guidelines for amd athlon™ 64 and amd opteron™ ccnuma multiprocessor systems. Advanced Micro Devices, 2006.

[arstechtwo]
Stokes, Jon ``Hannibal". Ars Technica RAM Guide, Part II: Asynchronous and Synchronous DRAM.http://arstechnica.com/paedia/r/ram_guide/ram_guide.part2-1.html, 2004.

[continuous]
Anderson, Jennifer M., Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger and William E. Weihl. Continuous profiling: Where have all the cycles gone. *Proceedings of the 16th acm symposium of operating systems principles*, pages 1--14. 1997.

[dcas]
Doherty, Simon, David L. Detlefs, Lindsay Grove, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit and Jr. Guy L. Steele. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. *Spaa '04: proceedings of the sixteenth annual acm symposium on parallelism in algorithms and architectures*, pages 216--224. New York, NY, USA, 2004. ACM Press.

[ddrtwo]
Dowler, M. Introduction to DDR-2: The DDR Memory Replacement. http://www.pcstats.com/articleview.cfm?articleID=1573, 2004.

[directcacheaccess]
Huggahalli, Ram, Ravi Iyer and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. , 2005.

[dwarves]

Melo, Arnaldo Carvalho de. The 7 dwarves: debugging information beyond gdb. *Proceedings of the linux symposium*. 2007.

[futexes]
Drepper, Ulrich. Futexes Are Tricky., 2005. http://people.redhat.com/drepper/futex.pdf.

[goldberg]
Goldberg, David. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5--48, 1991.

[highperfdram]
Cuppu, Vinodh, Bruce Jacob, Brian Davis and Trevor Mudge. High-Performance DRAMs in Workstation Environments. *IEEE Transactions on Computers*, 50(11):1133--1153, 2001.

[htimpact]
Margo, William, Paul Petersen and Sanjiv Shah. Hyper-Threading Technology: Impact on Compute-Intensive Workloads. *Intel Technology Journal*, 6(1), 2002.

[intelopt]
Intel 64 and ia-32 architectures optimization reference manual. Intel Corporation, 2007.

[lockfree]
Fober, Dominique, Yann Orlarey and Stephane Letz. Lock-Free Techiniques for Concurrent Access to Shared Objects. In GMEM, editor, *Actes des journées d'informatique musicale jim2002, marseille*, pages 143--150. 2002.

[micronddr]
Double Data Rate (DDR) SDRAM MT46V. Micron Technology, 2003.

[mytls]
Drepper, Ulrich. ELF Handling For Thread-Local Storage. Technical report, Red Hat, Inc., 2003.

[nonselsec]
Drepper, Ulrich. Security Enhancements in Red Hat Enterprise Linux. , 2004.

[oooreorder]
McNamara, Caolán. Controlling symbol ordering. http://blogs.linux.ie/caolan/2007/04/24/controlling-symbol-ordering/, 2007.

[sramwiki]
Wikipedia. Static random access memory. http://en.wikipedia.org/wiki/Static_Random_Access_Memory, 2006.

[transactmem]

Herlihy, Maurice and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *Proceedings of 20th international symposium on computer architecture*. 1993.

[vectorops]

Gebis, Joe and David Patterson. Embracing and Extending 20th-Century Instruction Set Architectures. *Computer*, 40(4):68--75, 2007.