On Memory Systems and Their Design

Memory is essential to the operation of a computer system, and nothing is more important to the development of the modern memory system than the concept of the memory hierarchy. While a flat memory system built of a single technology is attractive for its simplicity, a well-implemented hierarchy allows a memory system to approach simultaneously the performance of the fastest component, the cost per bit of the cheapest component, and the energy consumption of the most energy-efficient component.

For years, the use of a memory hierarchy has been very convenient, in that it has simplified the process of designing memory systems. The use of a hierarchy allowed designers to treat system design as a modularized process—to treat the memory system as an abstraction and to optimize individual subsystems (caches, DRAMs [dynamic RAM], disks) in isolation.

However, we are finding that treating the hierarchy in this way—as a set of disparate subsystems that interact only through well-defined functional interfaces and that can be optimized in isolation—no longer suffices for the design of modern memory systems. One trend becoming apparent is that many of the underlying implementation issues are becoming significant. These include the physics of device and interconnect scaling, the choice of signaling protocols and topologies to ensure signal integrity, design parameters such as granularity of access and support for concurrency, and communication-related issues such as scheduling algorithms and queueing. These low-level details have begun to affect the higher level design process quite dramatically, whereas they were considered transparent only a design-generation ago. Cache architectures are appearing that play to the limitations imposed by interconnect physics in deep submicron processes; modern DRAM design is driven by circuit-level limitations that create system-level headaches; and modern disk performance is dominated by the on-board caching and scheduling policies. This is a non-trivial environment in which to attempt optimal design.

This trend will undoubtedly become more important as time goes on, and even now it has tremendous impact on design results. As hierarchies and their components grow more complex, *systemic* behaviors—those arising from the complex interaction of the memory system's parts—have begun to dominate. The real loss of performance is not seen in the CPU or caches or DRAM devices or disk assemblies themselves, but in the subtle interactions between these subsystems and in the manner in which these subsystems are connected. Consequently, it is becoming increasingly foolhardy to attempt system-level optimization by designing/optimizing each of the parts in isolation (which, unfortunately, is often the approach taken in modern computer design). No longer can a designer remain oblivious to issues "outside the scope" and focus solely on designing a subsystem. It has now become the case that a memory-systems designer, wishing to build a properly behaved memory hierarchy, must be intimately familiar with issues involved at all levels of an implementation, from cache to DRAM to disk. Thus, we wrote this book.

Ov.1 Memory Systems

A memory hierarchy is designed to provide multiple functions that are seemingly mutually exclusive. We start at random-access memory (RAM): all microprocessors (and computer systems in general) expect a random-access memory out of which they operate. This is fundamental to the structure of modern software, built upon the von Neumann model in which code and data are essentially the same and reside in the same place (i.e., memory). All requests, whether for instructions or for data, go to this random-access memory. At any given moment, any particular datum in memory may be needed; there is no requirement that data reside next to the code that manipulates it, and there is no requirement that two instructions executed one after the other need to be adjacent in memory. Thus, the memory system must be able to handle randomly addressed[1] requests in a manner that favors no particular request. For instance, using a tape drive for this primary memory is unacceptable for performance reasons, though it might be acceptable in the Turing-machine sense.

Where does the mutually exclusive part come in? As we said, all microprocessors are built to expect a random-access memory out of which they can operate. Moreover, this memory must be *fast*, matching the machine's processing speed; otherwise, the machine will spend most of its time tapping its foot and staring at its watch. In addition, modern software is written to expect gigabytes of storage for data, and the modern consumer expects this storage to be cheap. How many memory technologies provide both tremendous speed and tremendous storage capacity at a low price? Modern processors execute instructions both out of order and speculatively—put simply, they execute instructions that, in some cases, are not meant to get executed—and system software is typically built to expect that certain changes to memory are permanent. How many memory technologies provide non-volatility and an *undo* operation?

While it might be elegant to provide all of these competing demands with a single technology (say, for example, a gigantic battery-backed SRAM [static RAM]), and though there is no engineering problem that cannot be solved (if ever in doubt about this, simply query a room full of engineers), the reality is that building a full memory system out of such a technology would be prohibitively expensive today.[2] The good news is that it is not necessary. Specialization and division of labor make possible all of these competing goals simultaneously. Modern memory systems often have a terabyte of storage on the desktop and provide instruction-fetch and data-access bandwidths of 128 GB/s or more. Nearly all of the storage in the system is non-volatile, and speculative execution on the part of the microprocessor is supported. All of this can be found in a memory system that has an average cost of roughly 1/100,000,000 pennies per bit of storage.

The reason all of this is possible is because of a phenomenon called *locality of reference* [Belady 1966, Denning 1970]. This is an observed behavior that computer applications tend to exhibit and that, when exploited properly, allows a small memory to serve in place of a larger one.

Ov.1.1 Locality of Reference Breeds the Memory Hierarchy

We think linearly (in steps), and so we program the computer to solve problems by working in steps. The practical implications of this are that a computer's use of the memory system tends to be non-random and highly predictable. Thus is born the concept of *locality of reference*, so named because memory references tend to be localized in time and space:

- If you use something once, you are likely to use it again.
- If you use something once, you are likely to use its neighbor.

The first of these principles is called *temporal locality*; the second is called *spatial locality*. We will discuss them (and another type of locality) in more detail in *Part I*: *Cache* of this book, but for now it suffices to

[1] Though "random" addressing is the commonly used term, authors actually mean *arbitrarily* addressed requests because, in most memory systems, a *randomly* addressed sequence is one of the most efficiently handled events.

[2] Even Cray machines, which were famous for using SRAM as their main memory, today are built upon DRAM for their main memory.

say that one can exploit the locality principle and render a single-level memory system, which we just said was expensive, unnecessary. If a computer's use of the memory system, given a small time window, is both predictable and limited in spatial extent, then it stands to reason that a program does not need all of its data immediately accessible. A program would perform nearly as well if it had, for instance, a *two-level* store, in which the first level provides immediate access to a subset of the program's data, the second level holds the remainder of the data but is slower and therefore cheaper, and some appropriate heuristic is used to manage the movement of data back and forth between the levels, thereby ensuring that the most-needed data is usually in the first-level store.

This generalizes to the *memory hierarchy*: multiple levels of storage, each optimized for its assigned task. By choosing these levels wisely a designer can produce a system that has the best of all worlds: performance approaching that of the fastest component, cost per bit approaching that of the cheapest component, and energy consumption per access approaching that of the least power-hungry component.

The modern hierarchy is comprised of the following components, each performing a particular function or filling a functional niche within the system:

- **Cache (SRAM):** Cache provides access to program instructions and data that has very low latency (e.g., 1/4 nanosecond per access) and very high bandwidth (e.g., a 16-byte instruction block and a 16-byte data block per cycle => 32 bytes per 1/4 nanosecond, or 128 bytes per nanosecond, or 128 GB/s). It is also important to note that cache, on a per-access basis, also has relatively low energy requirements compared to other technologies.
- **DRAM:** DRAM provides a random-access storage that is relatively large, relatively fast, and relatively cheap. It is large and cheap compared to cache, and it is fast compared to disk. Its main strength is that it is just fast enough and just cheap enough to act as an operating store.
- **Disk:** Disk provides permanent storage at an ultra-low cost per bit. As mentioned, nearly all computer systems expect *some* data to be modifiable yet permanent, so the memory system must have, at some level, a permanent store. Disk's advantage is its very reasonable cost (currently less than 50¢ per gigabyte), which is low enough for users to buy enough of it to store thousands of songs, video clips, photos, and other memory hogs that users are wont to accumulate in their accounts (authors included).

Table Ov.1 lists some rough order-of-magnitude comparisons for access time and energy consumption per access.

Why is it not feasible to build a flat memory system out of these technologies? Cache is far too expensive to be used as permanent storage, and its cost to store a single album's worth of audio would exceed that of the

TABLE Ov.1 Cost-performance for various memory technologies

Technology	Bytes per Access (typ.)	Latency per Access	Cost per Megabyte[a]	Energy per Access
On-chip Cache	10	100 of picoseconds	$1–100	1 nJ
Off-chip Cache	100	Nanoseconds	$1–10	10–100 nJ
DRAM	1000 (internally fetched)	10–100 nanoseconds	$0.1	1–100 nJ (per device)
Disk	1000	Milliseconds	$0.001	100–1000 mJ

[a] Cost of semiconductor memory is extremely variable, dependent much more on economic factors and sales volume than on manufacturing issues. In particular, on-chip caches (i.e., those integrated with a microprocessor core) can take up half of the die area, in which case their "cost" would be half of the selling price of that microprocessor. Depending on the market (e.g., embedded versus high end) and sales volume, microprocessor costs cover an enormous range of prices, from pennies per square millimeter to several dollars per square millimeter.

original music CD by several orders of magnitude. Disk is far too slow to be used as an operating store, and its average seek time for random accesses is measured in milliseconds. Of the three, DRAM is the closest to providing a flat memory system. DRAM is sufficiently fast enough that, without the support of a cache front-end, it can act as an operating store for many embedded systems, and with battery back-up it can be made to function as a permanent store. However, DRAM alone is not cheap enough to serve the needs of human users, who often want nearly a terabyte of permanent storage, and, even with random access times in the tens of nanoseconds, DRAM is not quite fast enough to serve as the only memory for modern general-purpose microprocessors, which would prefer a new block of instructions every fraction of a nanosecond.

So far, no technology has appeared that provides every desired characteristic: low cost, non-volatility, high bandwidth, low latency, etc. So instead we build a system in which each component is designed to offer one or more characteristics, and we manage the operation of the system so that the poorer characteristics of the various technologies are "hidden." For example, if most of the memory references made by the microprocessor are handled by the cache and/or DRAM subsystems, then the disk will be used only rarely, and, therefore, its extremely long latency will contribute very little to the average access time. If most of the data resides in the disk subsystem, and very little of it is needed at any given moment in time, then the cache and DRAM subsystems will not need much storage, and, therefore, their higher costs per bit will contribute very little to the average cost of the system. If done right, a memory system has an average cost approaching that of bottommost layer and an average access time and bandwidth approaching that of topmost layer.

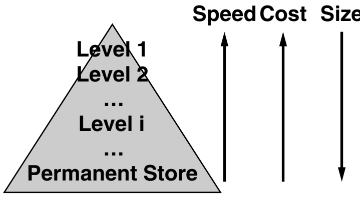


FIGURE Ov.1: A memory hierachy.

The memory hierarchy is usually pictured as a pyramid, as shown in Figure Ov.1. The higher levels in the hierarchy have better performance characteristics than the lower levels in the hierarchy; the higher levels have a higher cost per bit than the lower levels; and the system uses fewer bits of storage in the higher levels than found in the lower levels.

Though modern memory systems are comprised of SRAM, DRAM, and disk, these are simply technologies chosen to serve particular needs of the system, namely permanent store, operating store, and a fast store. Any technology set would suffice if it (a) provides permanent and operating stores and (b) satisfies the given computer system's performance, cost, and power requirements.

Permanent Store

The system's permanent store is where everything lives … meaning it is home to data that can be modified (potentially), but whose modifications must be remembered across invocations of the system (power-ups and power-downs). In general-purpose systems, this data typically includes the operating system's files, such as boot program, OS (operating system) executable, libraries, utilities, applications, etc., and the users' files, such as graphics, word-processing documents, spreadsheets, digital photographs, digital audio and video, email, etc. In embedded systems, this data typically includes the system's executable image and any installation-specific configuration information that it requires. Some embedded systems also maintain in permanent store the state of any partially completed transactions to withstand worst-case scenarios such as the system going down before the transaction is finished (e.g., financial transactions).

These all represent data that should not disappear when the machine shuts down, such as a user's saved email messages, the operating system's code and configuration information, and applications and their saved documents. Thus, the storage must be *non-volatile*, which in this context means not susceptible to power outages. Storage technologies chosen for permanent store include magnetic disk, flash memory, and even EEPROM (electrically erasable programmable read-only memory), of which flash memory is a special type. Other forms of programmable ROM (read-only memory) such as ROM, PROM (programmable ROM), or EPROM (erasable programmable ROM) are suitable for non-writable permanent information such as the executable image of an embedded system or a general-purpose system's boot code and BIOS.[3] Numerous exotic non-volatile technologies are in development, including magnetic RAM (MRAM), FeRAM (ferroelectric RAM), and phase-change RAM (PCRAM).

In most systems, the cost per bit of this technology is a very important consideration. In general-purpose systems, this is the case because these systems tend to have an enormous amount of permanent storage. A desktop can easily have more than 500 GB of permanent store, and a departmental server can have one hundred times that amount. The enormous number of bits in these systems translates even modest cost-per-bit increases into significant dollar amounts. In embedded systems, the cost per bit is important because of the significant number of units shipped. Embedded systems are often consumer devices that are manufactured and sold in vast quantities, e.g., cell phones, digital cameras, MP3 players, programmable thermostats, and disk drives. Each embedded system might not require more than a handful of megabytes of storage, yet a tiny 1¢ increase in the cost per megabyte of memory can translate to a $100,000 increase in cost per million units manufactured.

Operating (Random-Access) Store

As mentioned earlier, a typical microprocessor expects a new instruction or set of instructions on every clock cycle, and it can perform a data-read or data-write every clock cycle. Because the addresses of these instructions and data need not be sequential (or, in fact, related in any detectable way), the memory system must be able to handle *random access*—it must be able to provide instant access to any datum in the memory system.

The machine's operating store is the level of memory that provides random access at the microprocessor's data granularity. It is the storage level out of which the microprocessor could conceivably operate, i.e., it is the storage level that can provide random access to its storage, one data word at a time. This storage level is typically called "main memory." Disks cannot serve as main memory or operating store and cannot provide random access for two reasons: instant access is provided for only the data underneath the disk's head at any given moment, and the granularity of access is not what a typical processor requires. Disks are block-oriented devices, which means they read and write data only in large chunks; the typical granularity is 512 B. Processors, in contrast, typically operate at the granularity of 4 B or 8 B data words. To use a disk, a microprocessor must have additional buffering memory out of which it can read one instruction at a time and read or write one datum at a time. This buffering memory would become the *de facto* operating store of the system.

Flash memory and EEPROM (as well as the exotic non-volatile technologies mentioned earlier) are potentially viable as an operating store for systems that have small permanent-storage needs, and the non-volatility of these technologies provides them with a distinct advantage. However, not all are set up as an ideal operating store; for example, flash memory supports word-sized reads but supports only block-sized writes. If this type of issue can be handled in a manner that is transparent to the processor (e.g., in this case through additional data buffering), then the memory technology can still serve as a reasonable hybrid operating store.

Though the non-volatile technologies seem positioned perfectly to serve as operating store in all manner of devices and systems, DRAM is the most commonly used technology. Note that the only requirement of a memory system's operating store is that it provide random access with a small access granularity. Non-volatility is not a requirement, so long as it is provided by another level in the hierarchy. DRAM is a popular choice for operating store for several reasons: DRAM is faster than the various non-volatile technologies (in some cases *much* faster); DRAM supports an unlimited number of writes, whereas some non-volatile technologies start to fail after being erased and rewritten too many times (in some technologies, as few as 1–10,000 erase/write cycles); and DRAM processes are very similar to those used to build logic devices.

[3] BIOS = basic input/output system, the code that provides to software low-level access to much of the hardware.

DRAM can be fabricated using similar materials and (relatively) similar silicon-based process technologies as most microprocessors, whereas many of the various non-volatile technologies require new materials and (relatively) different process technologies.

Fast (and Relatively Low-Power) Store

If these storage technologies provide such reasonable operating store, why, then, do modern systems use cache? Cache is inserted between the processor and the main memory system whenever the access behavior of the main memory is not sufficient for the needs or goals of the system. Typical figures of merit include performance and energy consumption (or power dissipation). If the performance when operating out of main memory is insufficient, cache is interposed between the processor and main memory to decrease the average access time for data. Similarly, if the energy consumed when operating out of main memory is too high, cache is interposed between the processor and main memory to decrease the system's energy consumption.

The data in Table Ov.1 should give some intuition about the design choice. If a cache can reduce the number of accesses made to the next level down in the hierarchy, then it potentially reduces both execution time and energy consumption for an application. The gain is only potential because these numbers are valid only for certain technology parameters. For example, many designs use large SRAM caches that consume much more energy than several DRAM chips combined, but because the caches can reduce execution time they are used in systems where performance is critical, even at the expense of energy consumption.

It is important to note at this point that, even though the term "cache" is usually interpreted to mean SRAM, a cache is merely a concept and as such imposes no expectations on its implementation. Caches are best thought of as compact databases, as shown in Figure Ov.2. They contain data and, optionally, metadata such as the unique ID (address) of each data block in the array, whether it has been updated recently, etc. Caches can be built from SRAM, DRAM, disk, or virtually any storage technology. They can be managed completely in hardware and thus can be transparent to the running application and even to the memory system itself; and at the other extreme they can be explicitly managed by the running application. For instance, Figure Ov.2 shows that there is an optional block of metadata, which if implemented in hardware would be called the cache's *tags*. In that instance, a key is passed to the tags array, which produces either the location of the corresponding item in the data array (a *cache hit*) or an indication that the item is not in the data array (a *cache miss*). Alternatively, software can be written to index the array explicitly, using direct cache-array addresses, in which case the key lookup (as well as its associated tags array) is unnecessary. The configuration chosen for the cache is called its *organization*. Cache organizations exist at all spots along the continuum between these two extremes. Clearly, the choice of organization will significantly impact the cache's performance and energy consumption.

Predictability of access time is another common figure of merit. It is a special aspect of performance that is very important when building real-time systems or systems with highly orchestrated data movement. DRAM is occasionally in a state where it needs to ignore external requests so that it can guarantee the integrity of its stored data (this is called *refresh* and will be discussed in detail in Part II of the book). Such hiccups in data movement can be disastrous for some applications. For this reason, many microprocessors, such as digital signal processors (DSPs) and processors used in embedded control applications (called *microcontrollers*), often

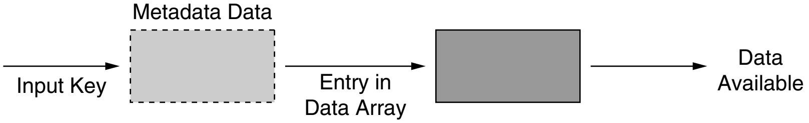


FIGURE Ov.2: An idealized cache lookup. A cache is logically comprised of two elements: the data array and some management information that indicates what is in the data array (labeled "metadata"). Note that the key information may be virtual, i.e., data addresses can be embedded in the software using the cache, in which case there is no explicit key lookup, and only the data array is needed.

have special caches that look like small main memories. These are *scratch-pad RAMs* whose implementation lies toward the end of the spectrum at which the running application manages the cache explicitly. DSPs typically have two of these scratch-pad SRAMs so that they can issue on every cycle a new *multiply-accumulate (MAC)* operation, an important DSP instruction whose repeated operation on a pair of data arrays produces its dot product. Performing a new MAC operation every cycle requires the memory system to load new elements from two different arrays simultaneously in the same cycle. This is most easily accomplished by having two separate data busses, each with its own independent data memory and each holding the elements of a different array.

Perhaps the most familiar example of a software-managed memory is the processor's *register file*, an array of storage locations that is indexed directly by bits within the instruction and whose contents are dictated entirely by software. Values are brought into the register file explicitly by software instructions, and old values are only overwritten if done so explicitly by software. Moreover, the register file is significantly smaller than most on-chip caches and typically consumes far less energy. Accordingly, software's best bet is often to optimize its use of the register file [Postiff & Mudge 1999].

Ov.1.2 Important Figures of Merit

The following issues have been touched on during the previous discussion, but at this point it would be valuable to formally present the various figures of merit that are important to a designer of memory systems. Depending on the environment in which the memory system will be used (supercomputer, departmental server, desktop, laptop, signal-processing system, embedded control system, etc.), each metric will carry more or less weight. Though most academic studies tend to focus on one axis at a time (e.g., performance), the design of a memory system is a multi-dimensional optimization problem, with all the adherent complexities of analysis. For instance, to analyze something in this design space or to consider one memory system over another, a designer should be familiar with concepts such as Pareto optimality (described later in this chapter). The various figures of merit, in no particular order other than performance being first due to its popularity, are performance, energy consumption and power dissipation, predictability of behavior (i.e., real time), manufacturing costs, and system reliability. This section describes them briefly, collectively. Later sections will treat them in more detail.

Performance

The term "performance" means many things to many people. The performance of a system is typically measured in the time it takes to execute a task (i.e., task *latency*), but it can also be measured in the number of tasks that can be handled in a unit time period (i.e., task *bandwidth*). Popular figures of merit for performance include the following:[4]

- Cycles per Instruction (CPI)

$$= \frac{\text{Total execution cycles}}{\text{Total user-level instructions committed}}$$

- Memory-system CPI overhead

$$= \text{Real CPI} - \text{CPI assuming perfect memory}$$

- Memory Cycles per Instruction (MCPI)

$$= \frac{\text{Total cycles spent in memory system}}{\text{Total user-level instructions committed}}$$

- Cache miss rate $= \dfrac{\text{Total cache misses}}{\text{Total cache accesses}}$

- Cache hit rate = 1 – Cache miss rate

- Average access time

$$= \begin{array}{l}(\text{hit rate} \cdot \text{average to service hit}) + \\ (\text{miss rate} \cdot \text{average to service miss})\end{array}$$

- Million Instructions per Second (MIPS)

$$= \frac{\text{Instructions executed (seconds)}}{10^6 \cdot \text{Average required for execution}}$$

[4] Note that the MIPS metric is easily abused. For instance, it is inappropriate for comparing different instruction-set architectures, and marketing literature often takes the definition of "instructions executed" to mean any particular given window of time as opposed to the full execution of an application. In such cases, the metric can mean the highest possible issue rate of instructions that the machine can achieve (but not necessarily sustain for any realistic period of time).

A cautionary note: using a metric of performance for the memory system that is independent of a processing context can be very deceptive. For instance, the MCPI metric does not take into account how much of the memory system's activity can be overlapped with processor activity, and, as a result, memory system A which has a worse MCPI than memory system B might actually yield a computer system with better total performance. As Figure Ov.5 in a later section shows, there can be significantly different amounts of overlapping activity between the memory system and CPU execution.

How to average a set of performance metrics correctly is still a poorly understood topic, and it is very sensitive to the weights chosen (either explicitly or implicitly) for the various benchmarks considered [John 2004]. Comparing performance is always the least ambiguous when it means the amount of time saved by using one design over another. When we ask the question *this machine is how much faster than that machine?* the implication is that we have been using *that* machine for some time and wish to know how much time we would save by using *this* machine instead. The true measure of performance is to compare the total execution time of one machine to another, with each machine running the benchmark programs that represent the user's typical workload as often as a user expects to run them. For instance, if a user compiles a large software application ten times per day and runs a series of regression tests once per day, then the total execution time should count the compiler's execution ten times more than the regression test.

Energy Consumption and Power Dissipation

Energy consumption is related to work accomplished (e.g., how much computing can be done with a given battery), whereas power dissipation is the rate of consumption. The instantaneous power dissipation of CMOS (complementary metal-oxide-semiconductor) devices, such as microprocessors, is measured in watts (W) and represents the sum of two components: *active power,* due to switching activity, and *static power,* due primarily to subthreshold leakage. To a first approximation, average power dissipation is equal to the following (we will present a more detailed model later):

$$P_{\text{avg}} = (P_{\text{dynamic}} + P_{\text{static}}) \equiv C_{\text{tot}}\, V^2{}_{\text{dd}} f + I_{\text{leak}} V_{\text{dd}} \tag{EQ Ov.1}$$

where C_{tot} is the total capacitance switched, V_{dd} is the power supply, f is the switching frequency, and I_{leak} is the leakage current, which includes such sources as subthreshold and gate leakage. With each generation in process technology, active power is decreasing on a device level and remaining roughly constant on a chip level. Leakage power, which used to be insignificant relative to switching power, increases as devices become smaller and has recently caught up to switching power in magnitude [Grove 2002]. In the future, leakage will be the primary concern.

Energy is related to power through time. The energy consumed by a computation that requires T seconds is measured in joules (J) and is equal to the integral of the instantaneous power over time T. If the power dissipation remains constant over T, the resultant energy consumption is simply the product of power and time.

$$E = (P_{\text{avg}} \cdot T) \equiv C_{\text{tot}} V^2{}_{\text{dd}} N + I_{\text{leak}} V_{\text{dd}} T \tag{EQ Ov.2}$$

where N is the number of switching events that occurs during the computation.

In general, if one is interested in extending battery life or reducing the electricity costs of an enterprise computing center, then *energy* is the appropriate metric to use in an analysis comparing approaches. If one is concerned with heat removal from a system or the thermal effects that a functional block can create, then *power* is the appropriate metric. In informal discussions (i.e., in common-parlance prose rather than in equations where units of measurement are inescapable), the two terms "power" and "energy" are frequently used interchangeably, though such use is technically incorrect. Beware, because this can lead to ambiguity and even misconception, which is usually unintentional, but not always so. For instance, microprocessor manufacturers will occasionally claim to have a "low-power" microprocessor that beats its predecessor by a factor of, say, two. This is easily accomplished by running the microprocessor at half the clock rate, which does reduce its power dissipation, but remember that power is the rate at which energy is consumed. However, to a first order, doing so doubles the time over which the processor dissipates that power. The net result is a processor that consumes the same amount of *energy* as before, though it is branded as having lower *power*, which is technically not a lie.

Popular figures of merit that incorporate both energy/power and performance include the following:

- Energy-Delay Product

$$= \begin{pmatrix}\text{Energy required}\\ \text{to perform task}\end{pmatrix} \cdot \begin{pmatrix}\text{Time required}\\ \text{to perform task}\end{pmatrix}$$

- Power-Delay Product

$$= \begin{pmatrix}\text{Power required}\\ \text{to perform task}\end{pmatrix}^{m} \cdot \begin{pmatrix}\text{Time required}\\ \text{to perform task}\end{pmatrix}^{n}$$

- MIPS per watt

$$= \frac{\text{Performance of benchmark in MIPS}}{\text{Average power dissipated by benchmark}}$$

The second equation was offered as a generalized form of the first (note that the two are equivalent when m = 1 and n = 2) so that designers could place more weight on the metric (time or energy/power) that is most important to their design goals [Gonzalez & Horowitz 1996, Brooks et al. 2000a].

Predictable (Real-Time) Behavior

Predictability of behavior is extremely important when analyzing real-time systems, because correctness of operation is often the primary design goal for these systems (consider, for example, medical equipment, navigation systems, anti-lock brakes, flight control systems, etc., in which failure to perform as predicted is not an option).

Popular figures of merit for expressing predictability of behavior include the following:

- Worst-Case Execution Time (WCET), taken to mean the longest amount of time a function could take to execute
- Response time, taken to mean the time between a stimulus to the system and the system's response (e.g., time to respond to an external interrupt)
- Jitter, the amount of deviation from an average timing value

These metrics are typically given as single numbers (average or worst case), but we have found that the probability density function makes a valuable aid in system analysis [Baynes et al. 2001, 2003].

Design (and Fabrication and Test) Costs

Cost is an obvious, but often unstated, design goal. Many consumer devices have cost as their primary consideration: if the cost to design and manufacture an item is not low enough, it is not worth the effort to build and sell it. Cost can be represented in many different ways (note that energy consumption is a measure of cost), but for the purposes of this book, by "cost" we mean the cost of producing an item: to wit, the cost of its design, the cost of testing the item, and/or the cost of the item's manufacture. Popular figures of merit for cost include the following:

- Dollar cost (best, but often hard to even approximate)
- Design size, e.g., die area (cost of manufacturing a VLSI (very large scale integration) design is proportional to its area cubed or more)
- Packaging costs, e.g., pin count
- Design complexity (can be expressed in terms of number of logic gates, number of transistors, lines of code, time to compile or synthesize, time to verify or run DRC (design-rule check), and many others, including a design's impact on clock cycle time [Palacharla et al. 1996])

Cost is often presented in a relative sense, allowing differing technologies or approaches to be placed on equal footing for a comparison.

- Cost per storage bit/byte/KB/MB/etc. (allows cost comparison between different storage technologies)
- Die area per storage bit (allows size-efficiency comparison within same process technology)

In a similar vein, cost is especially informative when combined with performance metrics. The following are variations on the theme:

- Bandwidth per package pin (total sustainable bandwidth to/from part, divided by total number of pins in package)
- Execution-time-dollars (total execution time multiplied by total cost; note that cost can be expressed in other units, e.g., pins, die area, etc.)

An important note: cost should incorporate *all* sources of that cost. Focusing on just one source of cost blinds the analysis in two ways: first, the true cost of the system is not considered, and second, solutions can be unintentionally excluded from the analysis. If cost is expressed in pin count, then all pins should be considered by the analysis; the analysis should not focus solely on data pins, for example. Similarly, if cost is expressed in die area, then all sources of die area should be considered by the analysis; the analysis should not focus solely on the number of banks, for example, but should also consider the cost of building control logic (decoders, muxes, bus lines, etc.) to select among the various banks.

Reliability

Like the term "performance," the term "reliability" means many things to many different people. In this book, we mean reliability of the data stored within the memory system: how easily is our stored data corrupted or lost, and how can it be protected from corruption or loss? Data integrity is dependent upon physical devices, and physical devices can fail.

Approaches to guarantee the integrity of stored data typically operate by storing redundant information in the memory system so that in the case of device failure, some but not all of the data will be lost or corrupted. If enough redundant information is stored, then the missing data can be reconstructed. Popular figures of merit for measuring reliability characterize both device fragility and robustness of a proposed solution. They include the following:

- Mean Time Between Failures (MTBF): [5] given in time (seconds, hours, etc.) or number of uses
- Bit-error tolerance, e.g., how many bit errors in a data word or packet the mechanism can correct, and how many it can detect (but not necessarily correct)
- Error-rate tolerance, e.g., how many errors per second in a data stream the mechanism can correct
- Application-specific metrics, e.g., how much radiation a design can tolerate before failure, etc.

Note that values given for MTBF often seem astronomically high. This is because they are not meant to apply to individual devices, but to system-wide device use, as in a large installation. For instance, if the expected service lifetime of a device is several years, then that device is expected to fail in several years. If an administrator swaps out devices every few years (before the service lifetime is up), then the administrator should expect to see failure frequencies consistent with the MTBF rating.

Ov.1.3 The Goal of a Memory Hierarchy

As already mentioned, a well-implemented hierarchy allows a memory system to approach simultaneously the performance of the fastest component, the cost per bit of the cheapest component, and the energy consumption of the most energy-efficient component. A modern memory system typically has performance close to that of on-chip cache, the fastest component in the system. The rate at which microprocessors fetch and execute their instructions is measured in nanoseconds or fractions of a nanosecond. A modern low-end desktop machine has several hundred gigabytes of storage and sells for under $500, roughly half of which goes to the on-chip caches, off-chip caches, DRAM, and disk. This represents an average cost of

[5] A common variation is "Mean Time To Failure (MTTF)."

several dollars per gigabyte—very close to that of disk, the cheapest component. Modern desktop systems have an energy cost that is typically in the low tens of nanojoules per instruction executed—close to that of on-chip SRAM cache, the least energy-costly component in the system (on a per-access basis).

The goal for a memory-system designer is to create a system that behaves, on average and from the point of view of the processor, like a big cache that has the price tag of a disk. A successful memory hierarchy is much more than the sum of its parts; moreover, successful memory-system design is non-trivial.

How the system is built, how it is used (and what parts of it are used more heavily than others), and on which issues an engineer should focus most of his effort at design time—all these are highly dependent on the target application of the memory system. Two common categories of target applications are (a) general-purpose systems, which are characterized by their need for universal applicability for just about any type of computation, and (b) embedded systems, which are characterized by their tight design restrictions along multiple axes (e.g., cost, correctness of design, energy consumption, reliability) and the fact that each executes only a single, dedicated software application its entire lifespan, which opens up possibilities for optimization that are less appropriate for general-purpose systems.

General-Purpose Computer Systems

General-purpose systems are what people normally think of as "computers." These are the machines on your desktop, the machines in the refrigerated server room at work, and the laptop on the kitchen table. They are designed to handle any and all tasks thrown at them, and the software they run on a day-to-day basis is radically different from machine to machine.

General-purpose systems are typically overbuilt. By definition they are expected by the consumer to run all possible software applications with acceptable speed, and therefore, they are built to handle the average case very well and the worst case at least tolerably well. Were they optimized for any particular task, they could easily become less than optimal for all dissimilar tasks. Therefore, general-purpose systems are optimized for everything, which is another way of saying that they are actually optimized for nothing in particular. However, they make up for this in raw performance, pure number-crunching. The average notebook computer is capable of performing orders of magnitude more operations per second than that required by a word processor or email client, tasks to which the average notebook is frequently relegated, but because the general-purpose system may be expected to handle virtually anything at any time, it must have significant spare number-crunching ability, just in case.

It stands to reason that the memory system of this computer must also be designed in a Swiss-army-knife fashion. Figure Ov.3 shows the organization of a typical personal computer, with the components of the memory system highlighted in grey boxes. The cache levels are found both on-chip (i.e., integrated on the same die as the microprocessor core) and off-chip (i.e., on a separate die). The DRAM system is comprised of a memory controller and a number of DRAM chips organized into DIMMs (dual in-line memory modules, printed circuit boards that contain a handful of DRAMs each). The memory controller can be located on-chip or off-chip, but the DRAMs are always separate from the CPU to allow memory upgrades. The disks in the system are considered peripheral devices, and so their access is made through one or more levels of controllers, each representing a potential chip-to-chip crossing (e.g., here a disk request passes through the system controller to the PCI (peripheral component interconnect) bus controller, to the SCSI (small computer system interface) controller, and finally to the disk itself).

The software that runs on a general-purpose system typically executes in the context of a robust operating system, one that provides virtual memory. Virtual memory is a mechanism whereby the operating system can provide to all running user-level software (i.e., email clients, web browsers, spreadsheets, word-processing packages, graphics and video editing software, etc.) the illusion that the user-level software is in direct control of the computer, when in fact its use of the computer's resources is managed by the operating system. This is a very effective way for an operating system to provide simultaneous access by

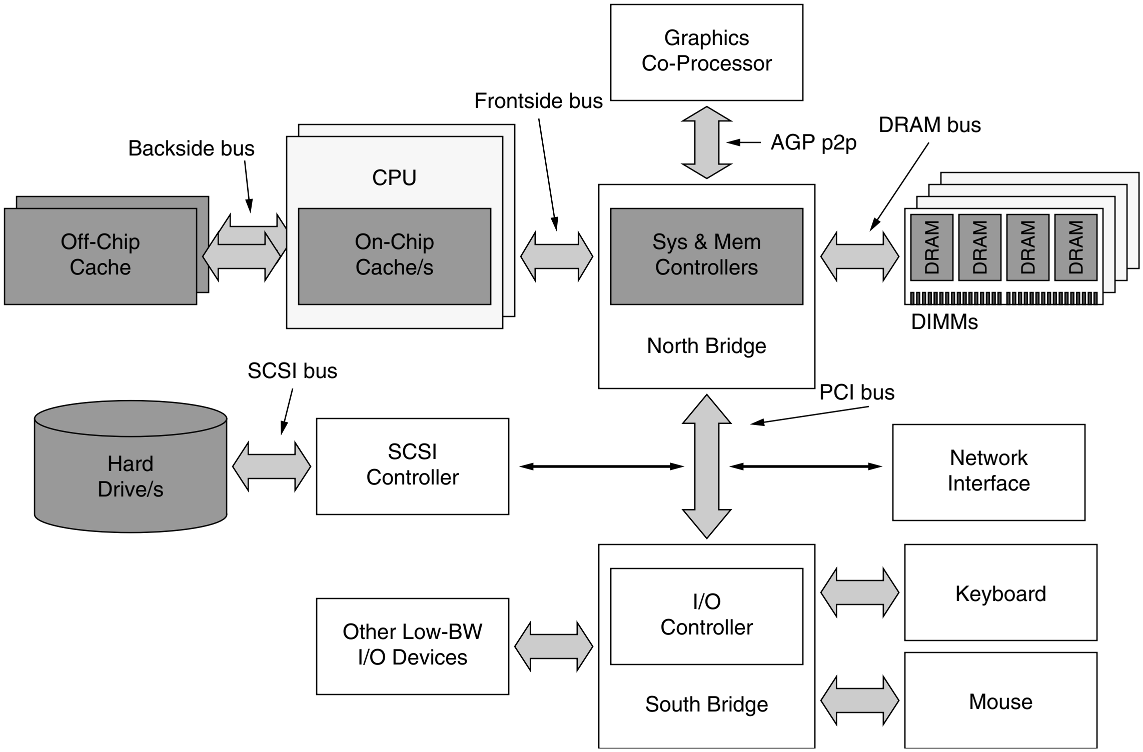


FIGURE Ov.3: Typical PC organization. The memory subsystem is one part of a relatively complex whole. This figure illustrates a two-way multiprocessor, with each processor having its own dedicated off-chip cache. The parts most relevant to this text are shaded in grey: the CPU and its cache system, the system and memory controllers, the DIMMs and their component DRAMs, and the hard drive/s.

large numbers of software packages to small numbers of limited-use resources (e.g., physical memory, the hard disk, the network, etc.).

The virtual memory system is the primary constituent of the memory system, in that it is the primary determinant of the manner/s in which the memory system's components are used by software running on the computer. Permanent data is stored on the disk, and the operating store, DRAM, is used as a cache for this permanent data. This DRAM-based cache is explicitly managed by the operating system. The operating system decides what data from the disk should be kept, what should be discarded, what should be sent back to the disk, and, for data retained, where it should be placed in the DRAM system. The primary and secondary caches are usually transparent to software, which means that they are managed by hardware, not software (note, however, the use of the word "usually"—later sections will delve into this in more detail). In general, the primary and secondary caches hold *demand-fetched* data, i.e., running software demands data, the hardware fetches it from memory, and the caches retain as much of it as possible. The DRAM system contains data that the operating system deems worthy of keeping around, and because fetching data from the disk and writing it back to the disk are such time-consuming processes, the operating system can exploit that lag time (during

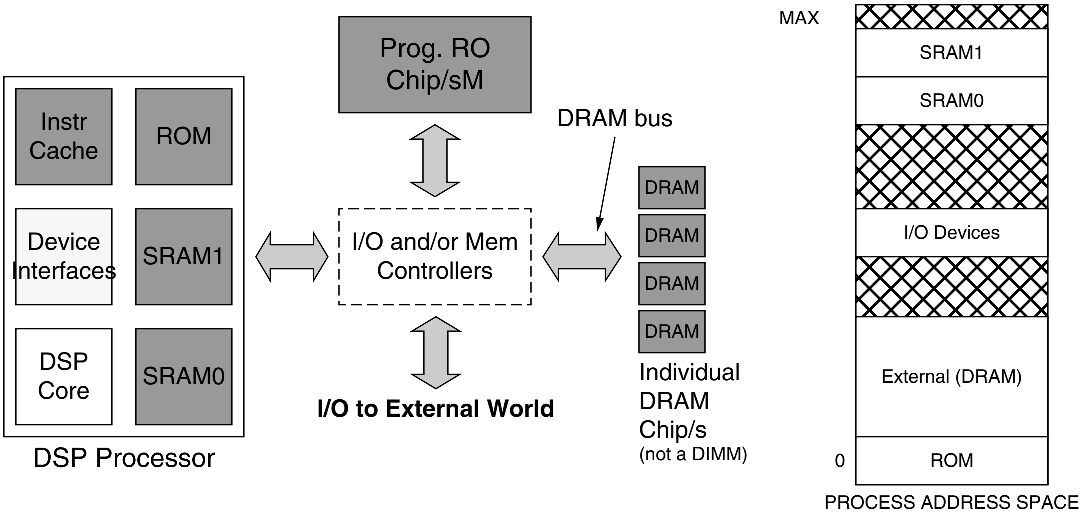


FIGURE Ov.4: DSP-style memory system. Example based on Texas Instruments' TMS320C3x DSP family.

which it would otherwise be stalled, doing nothing) to use sophisticated heuristics to decide what data to retain.

Embedded Computer Systems

Embedded systems differ from general-purpose systems in two main aspects. First and foremost, the two are designed to suit very different purposes. While general-purpose systems run a myriad of unrelated software packages, each having potentially very different performance requirements and dynamic behavior compared to the rest, embedded systems perform a single function their entire lifetime and thus execute the same code day in and day out until the system is discarded or a software upgrade is performed. Second, while performance is the primary (in many instances, the only) figure of merit by which a general-purpose system is judged, optimal embedded-system designs usually represent trade-offs between several goals, including manufacturing cost (e.g., die area), energy consumption, and performance.

As a result, we see two very different design strategies in the two camps. As mentioned, general-purpose systems are typically overbuilt; they are optimized for nothing in particular and must make up for this in raw performance. On the other hand, embedded systems are expected to handle only one task that is known at design time. Thus, it is not only possible, but highly beneficial to optimize an embedded design for its one suited task. If general-purpose systems are *overbuilt*, the goal for an embedded system is to be *appropriately* built. In addition, because effort spent at design time is amortized over the life of a product, and because many embedded systems have long lifetimes (tens of years), many embedded design houses will expend significant resources up front to optimize a design, using techniques not generally used in general-purpose systems (for instance, compiler optimizations that require many days or weeks to perform).

The memory system of a typical embedded system is less complex than that of a general-purpose system.[6] Figure Ov.4 illustrates an average digital signal-processing system with dual tagless SRAMs on-chip,

[6] Note that "less complex" does not necessarily imply "small," e.g., consider a typical iPod (or similar MP3 player), whose primary function is to store gigabytes' worth of a user's music and/or image files.

an off-chip programmable ROM (e.g., PROM, EPROM, flash ROM, etc.) that holds the executable image, and an off-chip DRAM that is used for computation and holding variable data. External memory and device controllers can be used, but many embedded microprocessors already have such controllers integrated onto the CPU die. This cuts down on the system's die count and thus cost. Note that it would be possible for the entire hierarchy to lie on the CPU die, yielding a single-chip solution called a *system-on-chip*. This is relatively common for systems that have limited memory requirements. Many DSPs and microcontrollers have programmable ROM embedded within them. Larger systems that require megabytes of storage (e.g., in Cisco routers, the instruction code alone is more than a 12 MB) will have increasing numbers of memory chips in the system.

On the right side of Figure Ov.4 is the software's view of the memory system. The primary distinction is that, unlike general-purpose systems, is that the SRAM caches are visible as separately addressable memories, whereas they are transparent to software in general-purpose systems.

Memory, whether SRAM or DRAM, usually represents one of the more costly components in an embedded system, especially if the memory is located on-CPU because once the CPU is fabricated, the memory size cannot be increased. In nearly all system-on-chip designs and many microcontrollers as well, memory accounts for the lion's share of available die area. Moreover, memory is one of the primary consumers of energy in a system, both on-CPU and off-CPU. As an example, it has been shown that, in many digital signal-processing applications, the memory system consumes more of both energy and die area than the processor datapath. Clearly, this is a resource on which significant time and energy is spent performing optimization.

Ov.2 Four Anecdotes on Modular Design

It is our observation that computer-system design in general, and memory-hierarchy design in particular, has reached a point at which it is no longer sufficient to design and optimize subsystems in isolation. Because memory systems and their subsystems are so complex, it is now the rule, and not the exception, that the subsystems we thought to be independent actually interact in unanticipated ways. Consequently, our traditional design methodologies no longer work because their underlying assumptions no longer hold. Modular design, one of the most widely adopted design methodologies, is an oft-praised engineering design principle in which clean functional interfaces separate subsystems (i.e., modules) so that subsystem design and optimization can be performed independently and in parallel by different designers. Applying the principles of modular design to produce a complex product can reduce the time and thus the cost for system-level design, integration, and test; optimization at the modular level guarantees optimization at the system level, provided that the system-level architecture and resulting module-to-module interfaces are optimal.

That last part is the sticking point: the principle of modular design assumes no interaction between module-level implementations and the choice of system-level architecture, but that is exactly the kind of interaction that we have observed in the design of modern, high-performance memory systems. Consequently, though modular design has been a staple of memory-systems design for decades, allowing cache designers to focus solely on caches, DRAM designers to focus solely on DRAMs, and disk designers to focus solely on disks, we find that, going forward, modular design is no longer an appropriate methodology.

Earlier we noted that, in the design of memory systems, many of the underlying implementation issues have begun to affect the higher level design process quite significantly: cache design is driven by interconnect physics; DRAM design is driven by circuit-level limitations that have dramatic system-level effects; and modern disk performance is dominated by the on-board caching and scheduling policies. As hierarchies and their components grow more complex, we find that the bulk of performance is lost not in the CPUs or caches or DRAM devices or disk assemblies themselves, but in the subtle interactions between these subsystems and in the manner in which these subsystems are connected. The bulk of lost performance is due to poor configuration of system-level parameters such as bus widths, granularity of access, scheduling policies, queue organizations, and so forth.

This is extremely important, so it bears repeating: the bulk of lost performance is not due to the number of CPU pipeline stages or functional units or choice of branch prediction algorithm or even CPU clock speed; the bulk of lost performance is due to poor configuration of system-level parameters such as bus widths, granularity of access, scheduling policies, queue organizations, etc. Today's computer-system performance is dominated by the manner in which data is moved between subsystems, i.e., the scheduling of transactions, and so it is not surprising that seemingly insignificant details can cause such a headache, as scheduling is known to be highly sensitive to such details.

Consequently, one can no longer attempt system-level optimization by designing/optimizing each of the parts in isolation (which, unfortunately, is often the approach taken in modern computer design). In subsystem design, nothing can be considered "outside the scope" and thus ignored. Memory-system design must become the purview of architects, and a subsystem designer must consider the system-level ramifications of even the slightest low-level design decision or modification. In addition, a designer must understand the low-level implications of system-level design choices. A simpler form of this maxim is as follows:

> *A designer must consider the system-level ramifications of circuit- and device-level decisions as well as the circuit- and device-level ramifications of system-level decisions.*

To illustrate what we mean and to motivate our point, we present several anecdotes. Though they focus on the DRAM system, their message is global, and we will show over the course of the book that the relationships they uncover are certainly not restricted to the DRAM system alone. We will return to these anecdotes and discuss them in much more detail in Chapter 27, *The Case for Holistic Design*, which follows the technical section of the book.

Ov.2.1 Anecdote I: Systemic Behaviors Exist

In 1999–2001, we performed a study of DRAM systems in which we explicitly studied only system-level effects—those that had nothing to do with the CPU architecture, DRAM architecture, or even DRAM interface protocol. In this study, we held constant the CPU and DRAM architectures and considered only a handful of parameters that would affect how well the two communicate with each other. Figure Ov.5 shows some of the results [Cuppu & Jacob 1999, 2001, Jacob 2003]. The varied parameters in Figure Ov.5 are all seemingly innocuous parameters, certainly not the type that would account for up to 20% differences in system performance (execution time) if one parameter was increased or decreased by a small amount, which is indeed the case. Moreover, considering the top two graphs, all of the choices represent intuitively "good" configurations. None of the displayed values represent strawmen, machine configurations that one would avoid putting on one's own desktop. Nonetheless, the performance variability is significant. When the analysis considers a wider range of bus speeds and burst lengths, the problematic behavior increases. As shown in the bottom graph, the ratio of best to worst execution times can be a factor of three, and the local optima are both more frequent and more exaggerated. Systems with relatively low bandwidth (e.g., 100, 200, 400 MB/s) and relatively slow bus speeds (e.g., 100, 200 MHz), if configured well, can match or exceed the performance of system configurations with much faster hardware that is poorly configured.

Intuitively, one would expect the design space to be relatively smooth: as system bandwidth increases, so should system performance. Yet the design space is far from smooth. Performance variations of 20% or more can be found in design points that are immediately adjacent to one another. The variations from best-performing to worst-performing design exceed a factor of three across the full space studied, and local minima and maxima abound. Moreover, the behaviors are related. Increasing one parameter by a factor of two toward higher expected performance (e.g., increasing the channel width) can move the system off a local optimum, but local optimality can be restored by changing other related parameters to follow suit,

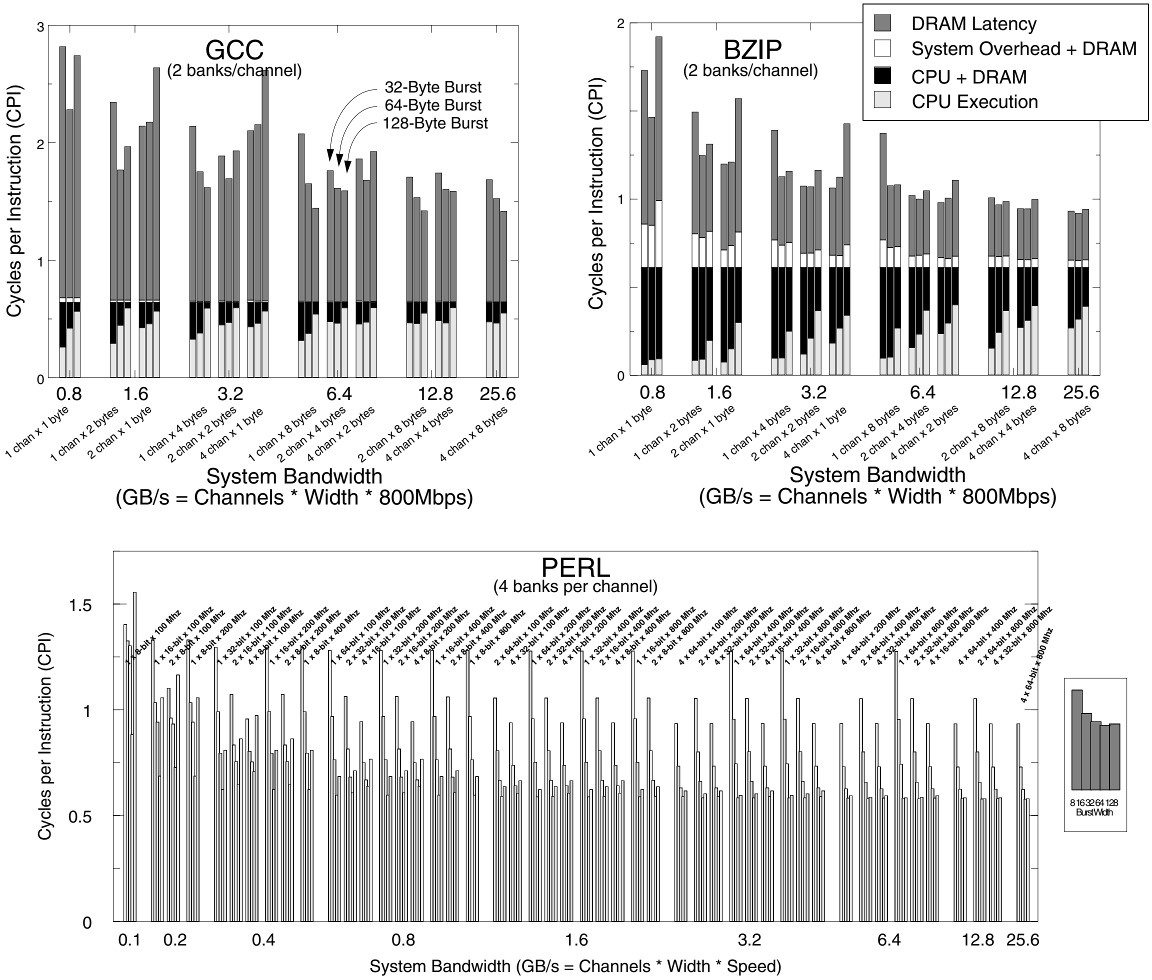


FIGURE Ov.5: Execution time as a function of bandwidth, channel organization, and granularity of access. Top two graphs from Cuppu & Jacob [2001] (© 2001 *IEEE*); bottom graph from Jacob [2003] (© 2003 *IEEE*).

such as increasing the burst length and cache block size to match the new channel width. This complex interaction between parameters previously thought to be independent arises because of the complexity of the system under study, and so we have named these "systemic" behaviors.[7] This study represents the moment we realized that systemic behaviors exist and that they are significant. Note that the behavior

[7] There is a distinction between this type of behavior and what in complex system theory is called "emergent system" behaviors or properties. Emergent system behaviors are those of individuals within a complex system, behaviors that an individual may perform in a group setting that the individual would never perform alone. In our environment, the behaviors are observations we have made of the design space, which is derived from the system as a whole.

is not restricted to the DRAM system. We have seen it in the disk system as well, where the variations in performance from one configuration to the next are even more pronounced.

Recall that this behavior comes from the varying of parameters that are seemingly unimportant in the grand scheme of things—at least they would certainly seem to be far less important than, say, the cache architecture or the number of functional units in the processor core. The bottom line, as we have observed, is that systemic behaviors—unanticipated interactions between seemingly innocuous parameters and mechanisms—cause significant losses in performance, requiring in-depth, detailed design-space exploration to achieve anything close to an optimal design given a set of technologies and limitations.

Ov.2.2 Anecdote II: The DLL in DDR SDRAM

Beginning with their first generation, DDR (double data rate) SDRAM devices have included a circuit-level mechanism that has generated significant controversy within JEDEC (Joint Electron Device Engineering Council), the industry consortium that created the DDR SDRAM standard. The mechanism is a delay-locked loop (DLL), whose purpose is to more precisely align the output of the DDR part with the clock on the system bus. The controversy stems from the cost of the technology versus its benefits.

The system's global clock signal, as it enters the chip, is delayed by the DLL so that the chip's internal clock signal, after amplification and distribution across the chip, is exactly in-phase with the original system clock signal. This more precisely aligns the DRAM part's output with the system clock. The trade-off is extra latency in the datapath as well as a higher power and heat dissipation because the DLL, a dynamic control mechanism, is continuously running. By aligning each DRAM part in a DIMM to the system clock, each DRAM part is effectively de-skewed with respect to the other parts, and the DLLs cancel out timing differences due to process variations and thermal gradients.

Figure Ov.6 illustrates a small handful of alternative solutions considered by JEDEC, who ultimately chose Figure Ov.6(b) for the standard. The interesting thing is that the data strobe is not used to capture data at the memory controller, bringing into question its purpose if the DLL is being used to help with data transfer to the memory controller. There is significant disagreement over the value of the chosen design; an anonymous JEDEC member, when

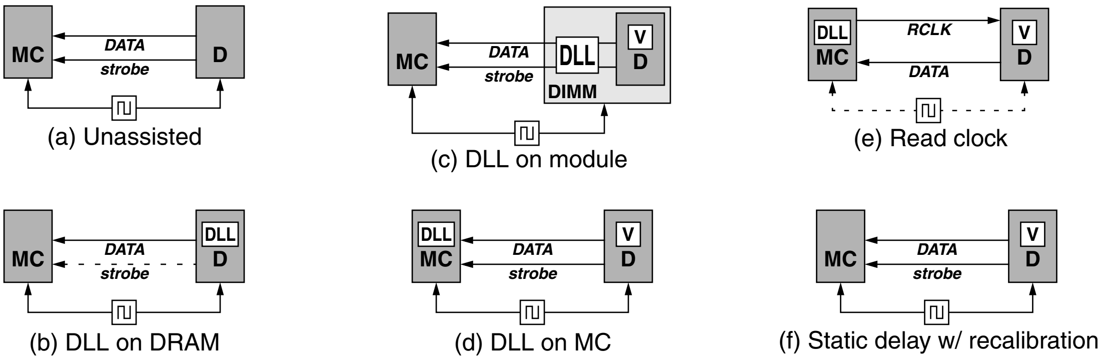


FIGURE Ov.6: Several alternatives to the per-DRAM DLL. The figure illustrates a half dozen different timing conventions (a dotted line indicates a signal is unused for capturing data): (a) the scheme in single data rate SDRAM; (b) the scheme chosen for DDR SDRAM; (c) moving the DLL onto the module, with a per-DRAM static delay element (Vernier); (d) moving the DLL onto the memory controller, with a per-DRAM static delay; (e) using a separate read clock per DRAM or per DIMM; and (f) using only a static delay element and recalibrating periodically to address dynamic changes.

asked "what is the DLL doing on the DDR chip?" answered with a grin, "burning power." In applications that require low latency and low power dissipation, designers turn off the DLL entirely and use only the data strobe for data capture, ignoring the system clock (as in Figure Ov.6(a)) [Kellogg 2002, Lee 2002, Rhoden 2002].

The argument for the DLL is that it de-skews the DRAM devices on a DIMM and provides a path for system design that can use a global clocking scheme, one of the simplest system designs known. The argument against the DLL is that it would be unnecessary if a designer learned to use the data strobe—this would require a more sophisticated system design, but it would achieve better performance at a lower cost. At the very least, it is clear that a DLL is a circuit-oriented solution to the problem of system-level skew, which could explain the controversy.

Ov.2.3 Anecdote III: A Catch-22 in the Search for Bandwidth

With every DRAM generation, timing parameters are added. Several have been added to the DDR specification to address the issues of power dissipation and synchronization.

- t_{FAW} (*Four-bank Activation Window*) and t_{RRD} (*Row-to-Row activation Delay*) put a ceiling on the maximum current draw of a single DRAM part. These are protocol-level limitations whose values are chosen to prevent a memory controller from exceeding circuit-related thresholds.
- t_{DQS} is our own name for the DDR system-bus turnaround time; one can think of it as the DIMM-to-DIMM switching time that has implications only at the system level (i.e., it has no meaning or effect if considering read requests in a system with but a single DIMM). By obeying t_{DQS}, one can ensure that a second DIMM will not drive the data bus at the same time as a first when switching from one DIMM to another for data output.

These are per-device timing parameters that were chosen to improve the behavior (current draw, timing uncertainty) of individual devices. However, they do so at the expense of a significant loss in system-level performance. When reading large amounts of data from the DRAM system, an application will have to read, and thus will have to *activate*, numerous DRAM rows. At this point, the t_{FAW} and t_{RRD} timing parameters kick in and limit the available read bandwidth. The t_{RRD} parameter specifies the minimum time between two successive row activation commands to the same DRAM device (which implies the same DIMM, because all the DRAMs on a DIMM are slaved together[8]). The t_{FAW} parameter represents a sliding window of time during which no more than four row activation commands to the same device may appear.

The parameters are specified in nanoseconds and not bus cycles, so they become increasingly problematic at higher bus frequencies. Their net effect is to limit the bandwidth available from a DIMM by limiting how quickly one can get the data out of the DRAM's storage array, irrespective of how fast the DRAM's I/O circuitry can ship the data back to the memory controller. At around 1 GBps, sustainable bandwidth hits a ceiling and remains flat no matter how fast the bus runs because the memory controller is limited in how quickly it can activate a new row and start reading data from it.

The obvious solution is to interleave data from different DIMMs on the bus. If one DIMM is limited in how quickly it can read data from its arrays, then one should populate the bus with many DIMMs and move through them in a round-robin fashion. This should bring the system bandwidth up to maximum. However, the function of t_{DQS} is to prevent exactly that: t_{DQS} is the bus turnaround time, inserted to account for skew on the bus and to prevent different bus masters from driving the bus at the same time.

[8] This is a minor oversimplification. We would like to avoid having to explain details of DRAM-system organization, such as the concept of *rank*, at this point.

To avoid such collisions, a second DIMM must wait at least t_{DQS} after a first DIMM has finished before driving the bus. So we have a catch:

- One set of parameters limits device-level bandwidth and expects a designer to go to the system level to reclaim performance.
- The other parameter limits system-level bandwidth and expects a designer to go to the device level to reclaim performance.

The good news is that the problem is solvable (see Chapter 15, Section 15.4.3, *DRAM Command Scheduling Algorithms*), but this is nonetheless a very good example of low-level design decisions that create headaches at the system level.

Ov.2.4 Anecdote IV: Proposals to Exploit Variability in Cell Leakage

The last anecdote is an example of a system-level design decision that ignores circuit- and device-level implications. Ever since DRAM was invented, it has been observed that different DRAM cells exhibit different data-retention time characteristics, typically ranging between hundreds of milliseconds to tens of seconds. DRAM manufacturers typically set the refresh requirement conservatively and require that every row in a DRAM device be refreshed at least once every 64 or 32 ms to avoid losing data. Though refresh might not seem to be a significant concern, in mobile devices researchers have observed that refresh can account for one-third of the power in otherwise idle systems, prompting action to address the issue. Several recent papers propose moving the refresh function into the memory controller and refreshing each row only when needed. During an initialization phase, the controller would characterize each row in the memory system, measuring DRAM data-retention time on a row-by-row basis, discarding leaky rows entirely, limiting its DRAM use to only those rows deemed non-leaky, and refreshing once every tens of seconds instead of once every tens of milliseconds.

The problem is that these proposals ignore another, less well-known phenomenon of DRAM cell variability, namely that a cell with a long retention time can suddenly (in the time frame of seconds) exhibit a short retention time [Yaney et al. 1987, Restle et al. 1992, Ueno et al. 1998, Kim 2004]. Such an effect would render these power-efficient proposals functionally erroneous. The phenomenon is called *variable retention time* (VRT), and though its occurrence is infrequent, it is non-zero. The occurrence rate is low enough that a system using one of these reduced-refresh proposals could protect itself against VRT by using error correcting codes (ECC, described in detail in Chapter 30, *Memory Errors and Error Correction*), but none of the proposals so far discuss VRT or ECC.

Ov.2.5 Perspective

To summarize so far:

Anecdote I: Systemic behaviors exist and are significant (they can be responsible for factors of two to three in execution time).

Anecdote II: The DLL in DDR SDRAM is a circuit-level solution chosen to address system-level skew.

Anecdote III: t_{DQS} represents a circuit-level solution chosen to address system-level skew in DDR SDRAM; t_{FAW} and t_{RRD} are circuit-level limitations that significantly limit system-level performance.

Anecdote IV: Several research groups have recently proposed system-level solutions to the DRAM-refresh problem, but fail to account for circuit-level details that might compromise the correctness of the resulting system.

Anecdotes II and III show that a common practice in industry is to focus at the level of devices and circuits, in some cases ignoring their system-level ramifications. Anecdote IV shows that a common practice in research is to design systems that have device- and circuit-level ramifications while abstracting away the details of the devices and circuits involved. Anecdote I illustrates that both approaches are doomed to failure in future memory-systems design.

It is clear that in the future we will have to move away from modular design; one can no longer safely abstract away details that were previously considered "out of scope." To produce a credible analysis, a designer must consider many different subsystems of a design and many different levels of abstraction—one must consider the forest when designing trees and consider the trees when designing the forest.

Ov.3 Cross-Cutting Issues

Though their implementation details might apply at a local level, most design decisions must be considered in terms of their system-level effects and side-effects before they become part of the system/hierarchy. For instance, power is a cross-cutting, system-level phenomenon, even though most power optimizations are specific to certain technologies and are applied locally; reliability is a system-level issue, even though each level of the hierarchy implements its own techniques for improving it; and, as we have shown, performance optimizations such as widening a bus or increasing support for concurrency rarely result in system performance that is globally optimal. Moreover, design decisions that locally optimize along one axis (e.g., power) can have even larger effects on the system level when all axes are considered. Not only can the global power dissipation be thrown off optimality by blindly making a local decision, it is even easier to throw the system off a global optimum when more than one axis is considered (e.g., power/performance).

Designing the best system given a set of constraints requires an approach that considers multiple axes simultaneously and measures the system-level effects of all design choices. Such a holistic approach requires an understanding of many issues, including cost and performance models, power, reliability, and software structure. The following sections provide overviews of these cross-cutting issues, and Part IV of the book will treat these topics in more detail.

Ov.3.1 Cost/Performance Analysis

To perform a cost/performance analysis correctly, the designer must define the problem correctly, use the appropriate tools for analysis, and apply those tools in the manner for which they were designed. This section provides a brief, intuitive look at the problem. Herein, we will use *cost* as an example of problem definition, *Pareto optimality* as an example of an appropriate tool, and *sampled averages* as an example to illustrate correct tool usage. We will discuss these issues in more detail with more examples in Chapter 28, *Analysis of Cost and Performance.*

Problem Definition: Cost

A designer must think in an all-inclusive manner when accounting for cost. For example, consider a cost-performance analysis of a DRAM system wherein performance is measured in sustainable bandwidth and cost is measured in pin count.

To represent the cost correctly, the analysis should consider *all* pins, including those for control, power, ground, address, and data. Otherwise, the resulting analysis can incorrectly portray the design space, and workable solutions can get left out of the analysis. For example, a designer can reduce latency in some cases by increasing the number of address and command pins, but if the cost analysis only considers data pins, then these optimizations would be cost-free. Consider DRAM addressing, which is done half of an address at a time. A 32-bit physical address is sent to the DRAM system 16 bits at a time in two different commands; one could potentially decrease DRAM latency by using an SRAM-like wide address bus and sending the entire 32 bits at once. This represents a *real* cost in design and manufacturing that would be higher, but an analysis that accounts only for data pins would not consider it as such.

Power and ground pins must also be counted in a cost analysis for similar reasons. High-speed chip-to-chip interfaces typically require more power and ground pins than slower interfaces. The extra power and ground signals help to isolate the I/O drivers from each other and the signal lines from each other, both improving signal integrity by reducing crosstalk, ground bounce, and related effects. I/O systems with higher switching speeds would have an unfair advantage over those with lower switching speeds (and thus fewer power/ground pins) in a cost-performance analysis if power and ground pins were to be excluded from the analysis. The inclusion of these pins would provide for an effective and easily quantified trade-off between cost and bandwidth.

Failure to include address, control, power, and ground pins in an analysis, meaning failure to be all-inclusive at the conceptual stages of design, would tend to blind a designer to possibilities. For example, an architecturally related family of solutions that at first glance gives up total system bandwidth so as to be more cost-effective might be thrown out at the conceptual stages for its intuitively lower performance. However, considering all sources of cost in the analysis would allow a designer to look more closely at this family and possibly to recover lost bandwidth through the addition of pins.

Comparing SDRAM and Rambus system architectures provides an excellent example of considering cost as the total number of pins leading to a continuum of designs. The Rambus memory system is a narrow-channel architecture, compared to SDRAM's wide-channel architecture, pictured in Figure Ov.7 Rambus uses fewer address and command pins than SDRAM and thus incurs an additional latency at the command level. Rambus also uses fewer data pins and occurs an additional latency when transmitting data as well. The trade-off is the ability to run the bus at a much higher bus frequency, or *pin-bandwidth* in bits per second per pin, than SDRAM. The longer channel of the DRDRAM (direct Rambus DRAM) memory system contributes directly to longer read-command latencies and longer bus turnaround times. However, the longer channel also allows for more devices to be connected to the memory system and reduces the likelihood that consecutive commands access the same device. The width and depth of the memory channels impact the bandwidth, latency, pin count, and various cost components of the respective memory systems. The effect that these organizational differences have on the DRAM access protocol is shown in Figure Ov.8 which illustrates a row activation and column read

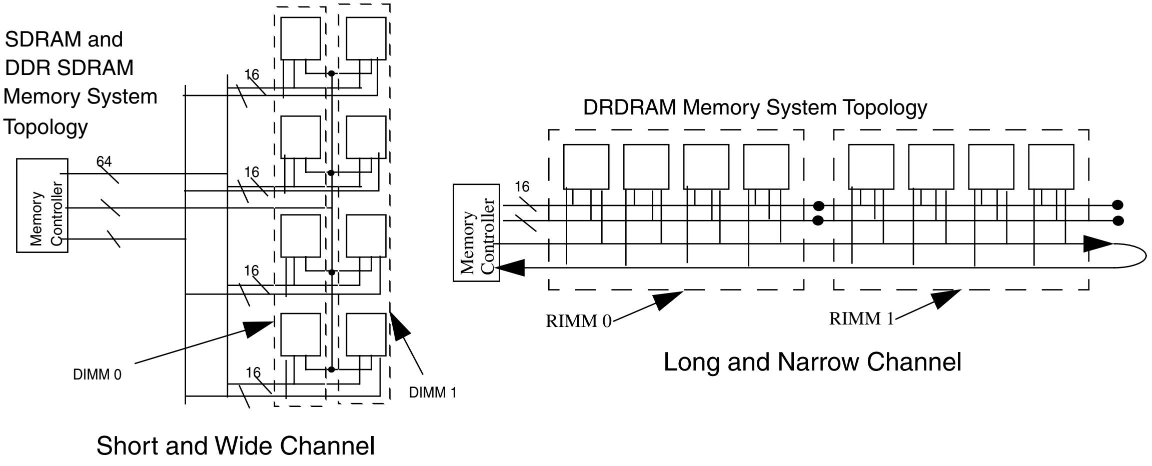


FIGURE Ov.7: Difference in topology between SDRAM and Rambus memory systems.

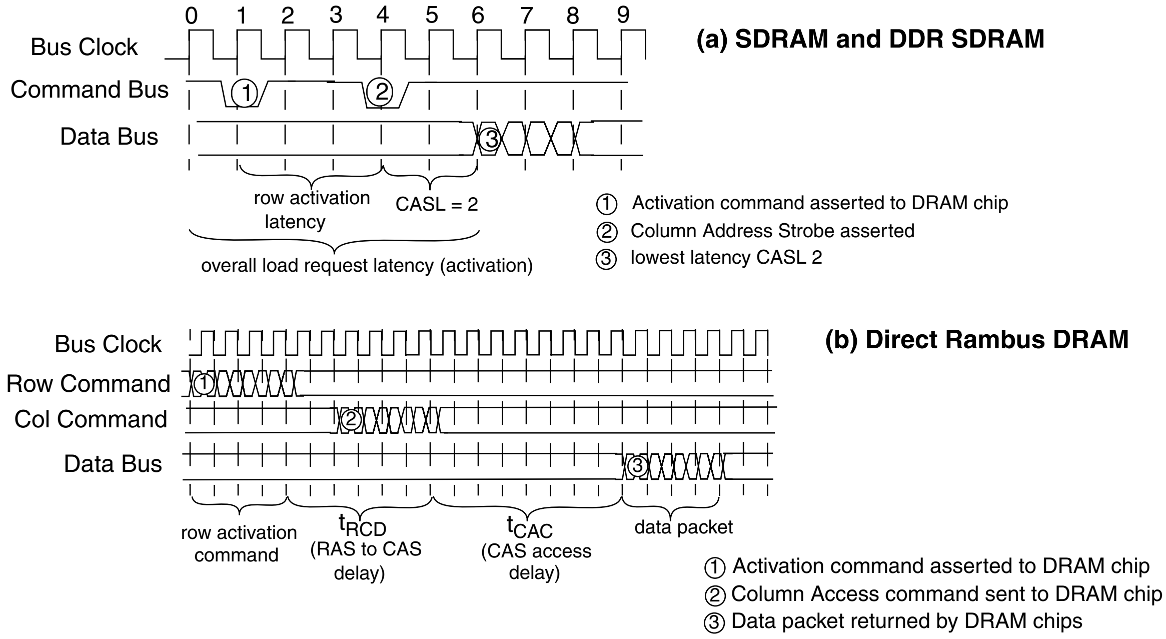

0 1 2 3 4 5 6 7 8 9
Bus Clock
Command Bus
Data Bus
(a) SDRAM and DDR SDRAM
row activation latency
CASL = 2
overall load request latency (activation)
① Activation command asserted to DRAM chip
② Column Address Strobe asserted
③ lowest latency CASL 2
Bus Clock
Row Command
Col Command
Data Bus
(b) Direct Rambus DRAM
row activation command
t_{RCD} (RAS to CAS delay)
t_{CAC} (CAS access delay)
data packet
① Activation command asserted to DRAM chip
② Column Access command sent to DRAM chip
③ Data packet returned by DRAM chips

FIGURE Ov.8: Memory access latency in SDRAM and DDR SDRAM memory systems (top) and DRDRAM (bottom).

command for both DDR SDRAM and Direct Rambus DRAM.

Contemporary SDRAM and DDR SDRAM memory chips operating at a frequency of 200 MHz can activate a row in 3 clock cycles. Once the row is activated, memory controllers in SDRAM or DDR SDRAM memory systems can retrieve data using a simple column address strobe command with a latency of 2 or 3 clock cycles. In Figure Ov.8(a), Step 1 shows the assertion of a row activation command, and Step 2 shows the assertion of the column address strobe signal. Step 3 shows the relative timing of a high-performance DDR SDRAM memory module with a CASL (CAS latency) of 2 cycles. For a fair comparison against the DRDRAM memory system, we include the bus cycle that the memory controller uses to assert the load command to the memory chips. With this additional cycle included, a DDR SDRAM memory system has a read latency of 6 clock cycles (to critical data). In a SDRAM or DDR SDRAM memory system that operates at 200 MHz, 6 clock cycles translate to 30 ns of latency for a memory load command with row activation latency inclusive. These latency values are the same for high-performance SDRAM and DDR SDRAM memory systems.

The DRDRAM memory system behaves very differently from SDRAM and DDR SDRAM memory systems. Figure Ov.8(b) shows a row activation command in Step 1, followed by a column access command in Step 2. The requested data is then returned by the memory chip to the memory controller in Step 3. The row activation command in Step 1 is transmitted by the memory controller to the memory chip in a packet format that spans 4 clock cycles. The minimum delay between the row activation and column access is 7 clock cycles, and, after an additional (also minimum) CAS (column address strobe) latency of 8 clock cycles, the DRDRAM chip begins to transmit the data to the memory controller. One caveat to the computation of the access latency in the DRDRAM memory system is that CAS delay in the DRDRAM memory system is a function of the number of devices on a single DRDRAM memory channel. On a DRDRAM memory system with a full load of 32 devices

TABLE Ov.2 Peak bandwidth statistics of SDRAM, DDR SDRAM, and DRDRAM memory systems

	Operating Frequency (Data)	Data Channel Pin Count	Data Channel Bandwidth	Control Channel Pin Count	Command Channel Bandwidth	Address Channel Pin Count	Address Channel Bandwidth
SDRAM controller	133	64	1064 MB/s	28	465 MB/s	30	500 MB/s
DDR SDRAM controller	2 * 200	64	3200 MB/s	42	1050 MB/s	30	750 MB/s
DRDRAM controller	2 * 600	16	2400 MB/s	9	1350 MB/s	8	1200 MB/s
x16 SDRAM chip	133	16	256 MB/s	9	150 MB/s	15	250 MB/s
x16 DDR SDRAM chip	2 *200	16	800 MB/s	11	275 MB/s	15	375 MB/s

TABLE Ov.3 Cross-comparison of SDRAM, DDR SDRAM, and DRDRAM memory systems

DRAM Technology	Operating Frequency (Data Bus)	Pin Count per Channel	Peak Band-width	Sustained BW on StreamAdd	Bits per Pin per Cycle (Peak)	Bits per Pin per Cycle (Sustained)
SDRAM	133	152	1064 MB/s	540 MB/s	0.4211	0.2139
DDR SDRAM	2 * 200	171	3200 MB/s	1496 MB/s	0.3743	0.1750
DRDRAm	2 * 600	117	2400 MB/s	1499 MB/s	0.1368	0.0854

on the data bus, the CAS-latency delay may be as large as 12 clock cycles. Finally, it takes 4 clock cycles for the DRDRAM memory system to transport the data packet. Note that we add half the transmission time of the data packet in the computation of the latency of a memory request in a DRDRAM memory system due to the fact that the DRDRAM memory system does not support critical word forwarding, and the critically requested data may exist in the latter parts of the data packet; on average, it will be somewhere in the middle. This yields a total latency of 21 cycles, which, in a DRDRAM memory system operating at 600 MHz, translates to a latency of 35 ns.

The Rambus memory system trades off a longer latency for fewer pins and higher pin bandwidth (in this example, three times higher bandwidth). How do the systems compare in performance?

Peak bandwidth of any interface depends solely on the channel width and the operating frequency of the channel. In Table Ov.2, we summarize the statistics of the interconnects and compute the peak bandwidths of the memory systems at the interface of the memory controller and at the interface of the memory chips as well.

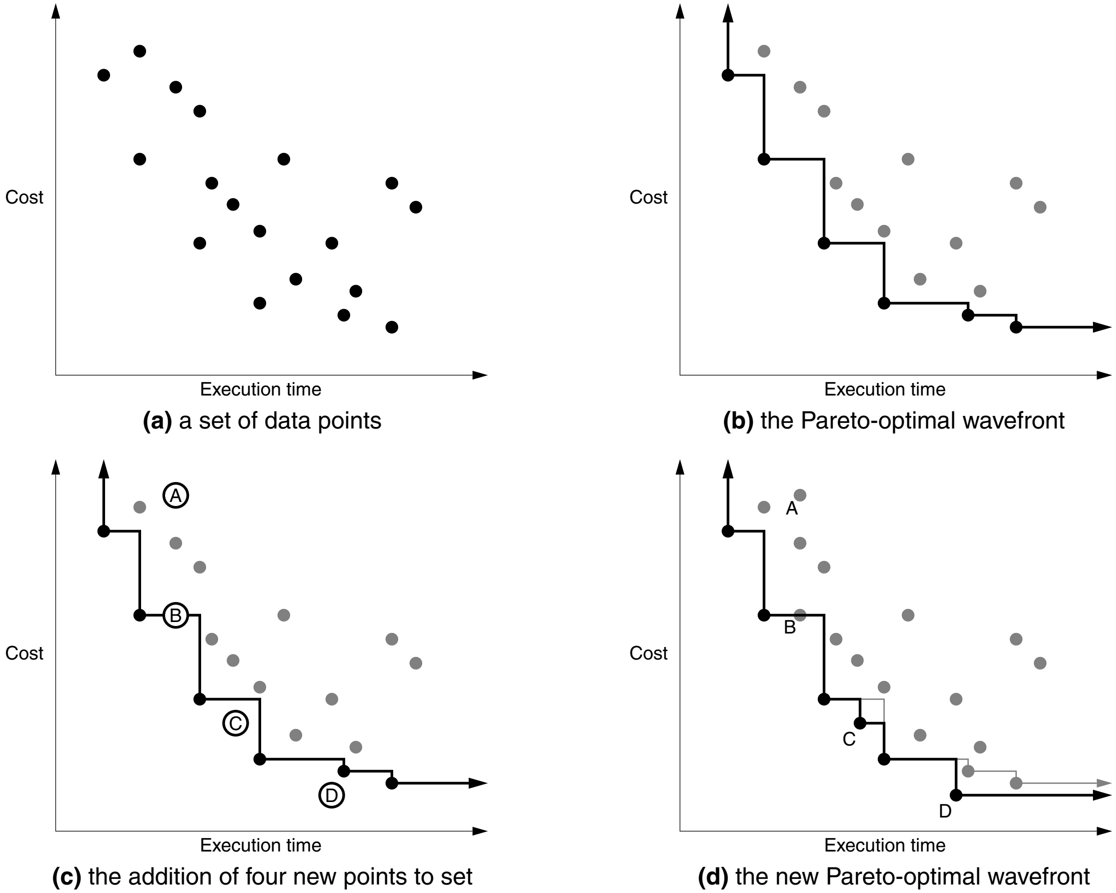
Table Ov.3 compares a 133-MHz SDRAM, a 200-MHz DDR SDRAM system, and a 600-MHz DRDRAM system. The 133-MHz SDRAM system, as represented by a PC-133 compliant SDRAM memory system on an AMD Athlon-based computer system, has a theoretical peak bandwidth of 1064 MB/s. The maximum sustained bandwidth for the single channel of SDRAM, as measured by the use of the add kernel in the STREAM benchmark, reaches 540 MB/s. The maximum sustained bandwidth for DDR SDRAM and DRDRAM was also measured on STREAM, yielding 1496 and 1499 MB/s, respectively. The pin cost of each system is factored in, yielding bandwidth per pin on both a per-cycle basis and a per-nanosecond basis.

Appropriate Tools: Pareto Optimality

It is convenient to represent the "goodness" of a design solution, a particular system configuration, as a single number so that one can readily compare the number with the goodness ratings of other candidate design solutions and thereby quickly find the "best" system configuration. However, in the design of memory systems, we are inherently dealing with a multi-dimensional design space (e.g., one that encompasses performance, energy consumption, cost, etc.), and so using a single number to represent a solution's worth is not really appropriate, unless we can assign exact weights to the various figures of merit (which is dangerous and will be discussed in more detail later) or we care about one aspect to the exclusion of all others (e.g., performance at any cost).

Assuming that we do not have exact weights for the figures of merit and that we do care about more than one aspect of the system, a very powerful tool to aid in system analysis is the concept of *Pareto optimality* or *Pareto efficiency,* named after the Italian economist Vilfredo Pareto, who invented it in the early 1900s.

Pareto optimality asserts that one candidate solution to a problem is better than another candidate solution only if the first *dominates* the second, i.e., if the first is better than or equal to the second in *all* figures of merit. If one solution has a better value in one dimension but a worse value in another, then the two candidates are Pareto equivalent. The best solution is actually a set



(a) a set of data points

(b) the Pareto-optimal wavefront

(c) the addition of four new points to set

(d) the new Pareto-optimal wavefront

FIGURE Ov.9: Pareto optimality. Members of the Pareto-optimal set are shown in solid black; non-optimal points are grey.

of candidate solutions: the set of Pareto-equivalent solutions that is not dominated by any solution.

Figure Ov.9(a) shows a set of candidate solutions in a two-dimensional space that represent a cost/performance metric. The *x*-axis represents system performance in execution time (smaller numbers are better), and the *y*-axis represents system cost in dollars (smaller numbers are better). Figure Ov.9(b) shows the Pareto-optimal set in solid black and connected by a line; non-optimal data points are shown in grey. The Pareto-optimal set forms a wavefront that approaches both axes simultaneously. Figures Ov.9(c) and (d) show the effect of adding four new candidate solutions to the space: one lies inside the wavefront, one lies on the wavefront, and two lie outside the wavefront. The first two new additions, A and B, are both dominated by at least one member of the Pareto-optimal set, and so neither is considered Pareto optimal. Even though B lies on the wavefront, it is not considered Pareto optimal. The point to the left of B has better performance than B at equal cost. Thus, it dominates B.

Point C is not dominated by any member of the Pareto-optimal set, nor does it dominate any member of the Pareto-optimal set. Thus, candidate-solution C is added to the optimal set, and its addition changes the shape of the wavefront slightly. The last of the additional points, D, is dominated by no members of the optimal set, but it *does* dominate several members of the optimal set, so D's inclusion in the optimal set excludes those dominated members from the set. As a result, candidate-solution D changes the shape of the wave front more significantly than candidate-solution C.

Tool Use: Taking Sampled Averages Correctly

In many fields, including the field of computer engineering, it is quite popular to find a *sampled average,* i.e., the average of a sampled set of numbers, rather than the average of the entire set. This is useful when the entire set is unavailable, difficult to obtain, or expensive to obtain. For example, one might want to use this technique to keep a running performance average for a real microprocessor, or one might want to sample several windows of execution in a terabyte-size trace file. Provided that the sampled subset is representative of the set as a whole, and provided that the technique used to collect the samples is correct, this mechanism provides a low-cost alternative that can be very accurate.

The discussion will use as an example a mechanism that samples the miles-per-gallon performance of an automobile under way. The trip we will study is an out and back trip with a brief pit stop, as shown in Figure Ov.10. The automobile will follow a simple course that is easily analyzed:

1. The auto will travel over even ground for 60 miles at 60 mph, and it will achieve 30 mpg during this window of time.
2. The auto will travel uphill for 20 miles at 60 mph, and it will achieve 10 mpg during this window of time.

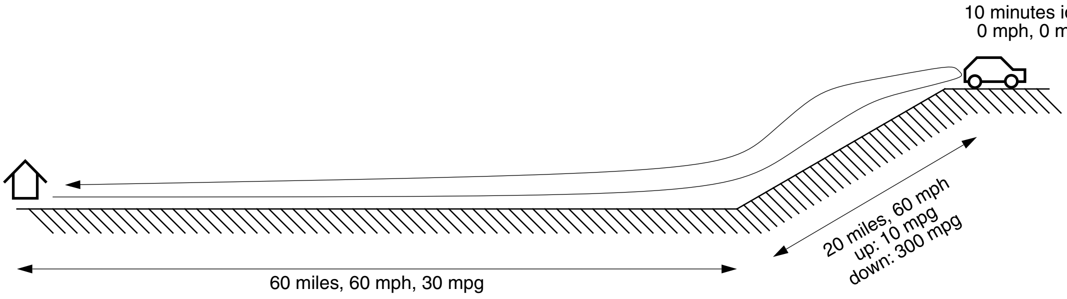


FIGURE Ov.10: Course taken by the automobile in the example.

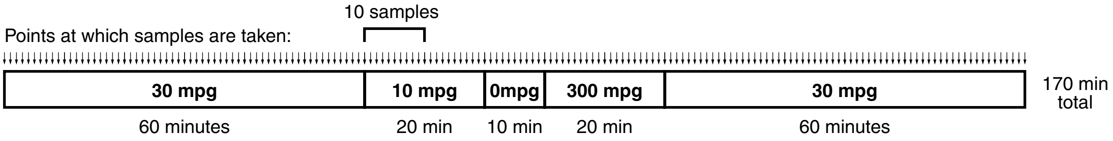


FIGURE Ov.11: Sampling miles-per-gallon (mpg) over time. The figure shows the trip in time, with each segment of time labeled with the average miles-per-gallon for the car during that segment of the trip. Thus, whenever the sampling algorithm samples miles-per-gallon during a window of time, it will add that value to the running average.

3. The auto will travel downhill for 20 miles at 60 mph, and it will achieve 300 mpg during this window of time.
4. The auto will travel back home over even ground for 60 miles at 60 mph, and it will achieve 30 mpg during this window of time.
5. In addition, before returning home, the driver will sit at the top of the hill for 10 minutes, enjoying the view, with the auto idling, consuming gasoline at the rate of 1 gallon every 5 hours. This is equivalent to 1/300 gallon per minute or 1/30 of a gallon during the 10-minute respite. Note that the auto will achieve 0 mpg during this window of time.

Our car's algorithm samples evenly in time, so for our analysis we need to break down the segments of the trip by the amount of time that they take:

- Outbound: 60 minutes
- Uphill: 20 minutes
- Idling: 10 minutes
- Downhill: 20 minutes
- Return: 60 minutes

This is displayed graphically in Figure Ov.11, in which the time for each segment is shown to scale. Assume, for the sake of simplicity, that the sampling algorithm samples the car's miles-per-gallon every minute and adds that sampled value to the running average (it could just as easily sample every second or millisecond). Then the algorithm will sample the value 30 mpg 60 times during the first segment of the trip, the value 10 mpg 20 times during the second segment of the trip, the value 0 mpg 10 times during the third segment of the trip, and so on. Over the trip, the car is operating for a total of 170 minutes. Thus, we can derive the sampling algorithm's results as follows:

$$\frac{60}{170}30 + \frac{20}{170}10 + \frac{10}{170}0 + \frac{20}{170}300 + \frac{60}{170}30 = 57.5\text{mpg} \qquad \textbf{(EQ Ov.3)}$$

The sampling algorithm tells us that the auto achieved 57.5 mpg during our trip. However, a quick reality check will demonstrate that this cannot be correct; somewhere in our analysis we have made an invalid assumption. What is the correct answer, the correct approach? In Part IV of the book we will revisit this example and provide a complete picture. In the meantime, the reader is encouraged to figure the answer out for him- or herself.

Ov.3.2 Power and Energy

Power has become a "first-class" design goal in recent years within the computer architecture and design community. Previously, low-power circuit, chip, and system design was considered the purview of specialized communities, but this is no longer the case, as even high-performance chip manufacturers can be blindsided by power dissipation problems.

Power Dissipation in Computer Systems

Power dissipation in CMOS circuits arises from two different mechanisms: *static power*, which is primarily *leakage power* and is caused by the transistor not completely turning off, and *dynamic power*, which is largely the result of switching capacitive loads between two different voltage states. Dynamic power is dependent on frequency of circuit activity, since no power is dissipated if the node values do not change, while static power is independent of the frequency of activity and exists whenever the chip is powered on. When CMOS circuits were first used, one of their main advantages was the negligible leakage current flowing with the gate at DC or steady state. Practically all of the power consumed by CMOS gates was due to dynamic power consumed during the transition of the gate. But as transistors become increasingly smaller, the CMOS leakage current starts to become significant and is projected to be larger than the dynamic power, as shown in Figure Ov.12.

In charging a load capacitor C up ΔV volts and discharging it to its original voltage, a gate pulls an amount of current equal to $C \cdot \Delta V$ from the V_{dd} supply to charge up the capacitor and then sinks this charge to ground discharging the node. At the end of a charge/discharge cycle, the gate/capacitor combination has moved $C \cdot \Delta V$ of charge from V_{dd} to ground, which uses an amount of energy equal to $C \cdot \Delta V \cdot V_{dd}$ that is independent of the cycle time. The average dynamic power of this node, the average rate of its energy consumption, is given by the following equation [Chandrakasan & Brodersen 1995]:

$$P_{dynamic} = C \cdot \Delta V \cdot V_{dd} \cdot \alpha \cdot f \qquad \textbf{(EQ Ov.4)}$$

Dividing by the charge/discharge period (i.e., multiplying by the clock frequency f) produces the rate of energy consumption over that period. Multiplying by the expected *activity ratio* α, the probability that the node will switch (in which case it dissipates dynamic power; otherwise, it does not), yields an average power dissipation over a larger window of time for which the activity ratio holds (e.g., this can yield average power for an entire hour of computation, not just a nanosecond). The dynamic power for the whole chip is the sum of this equation over all nodes in the circuit.

It is clear from EQ Ov.4 what can be done to reduce the dynamic power dissipation of a system. We can either reduce the capacitance being switched, the voltage swing, the power supply voltage, the activity ratio, or the operating frequency. Most of these options are available to a designer at the architecture level.

Note that, for a specific chip, the voltage swing ΔV is usually proportional to V_{dd}, so EQ Ov.4 is often simplified to the following:

$$P_{dynamic} = C \cdot V^2_{dd} \cdot \alpha \cdot f \qquad \textbf{(EQ Ov.5)}$$

Moreover, the activity ratio α is often approximated as 1/2, giving the following form:

$$P_{dynamic} = \frac{1}{2} \cdot C \cdot V^2_{dd} \cdot f \qquad \textbf{(EQ Ov.6)}$$

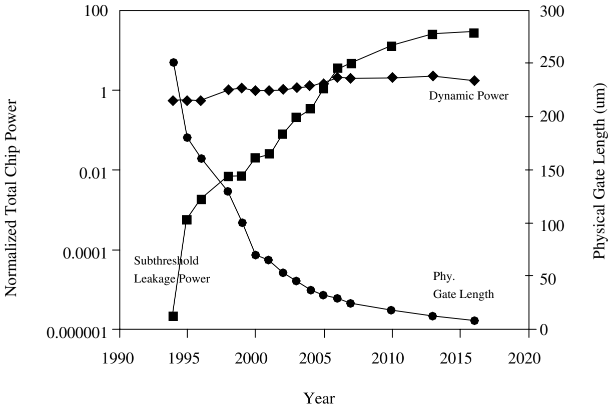


FIGURE Ov.12: Projections for dynamic and leakage, along with gate length. (Figure taken from Kim et al. [2004a]).

Static leakage power is due to our inability to completely turn off the transistor, which leaks current in the subthreshold operating region [Taur & Ning 1998]. The gate couples to the active channel mainly through the gate oxide capacitance, but there are other capacitances in a transistor that couple the gate to a "fixed charge" (charge which cannot move) present in the bulk and not associated with current flow [Peckerar et al. 1979, 1982]. If these extra capacitances are large (note that they increase with each process generation as physical dimensions shrink), then changing the gate bias merely alters the densities of the fixed charge and will not turn the channel off. In this situation, the transistor becomes a leaky faucet; it does not turn off no matter how hard you turn it.

Leakage power is proportional to V_{dd}. It is a linear, not a quadratic, relationship. For a particular process technology, the per-device leakage power is given as follows [Butts & Sohi 2000]:

$$\mathrm{P}_{static} = I_{leakage} \cdot V^2{}_{dd} \qquad \textbf{(EQ Ov.7)}$$

Leakage energy is the product of leakage power times the duration of operation.

It is clear from EQ Ov.7 what can be done to reduce the leakage power dissipation of a system: reduce leakage current and/or reduce the power supply voltage. Both options are available to a designer at the architecture level.

Heat in VLSI circuits is becoming a significant and related problem. The rate at which physical dimensions such as gate length and gate oxide thickness have been reduced is faster than for other parameters, especially voltage, resulting in higher power densities on the chip surface. To lower leakage power and maintain device operation, voltage levels are set according to the silicon bandgap and intrinsic built-in potentials, in spite of the conventional scaling algorithm. Thus, power densities are increasing exponentially for next-generation chips. For instance, the power density of Intel's Pentium chip line has already surpassed that of a hot plate with the introduction of the Pentium Pro [Gelsinger 2001]. The problem of power and heat dissipation now extends to the DRAM system, which traditionally has represented low power densities and low costs. Today, higher end DRAMs are dynamically throttled when, due to repeated high-speed access to the same devices, their operating temperatures surpass design thresholds. The next-generation memory system embraced by the DRAM community, the Fully Buffered DIMM architecture, specifies a per-module controller that, in many implementations, requires a heatsink. This is a cost previously unthinkable in DRAM-system design.

Disks have many components that dissipate power, including the spindle motor driving the platters, the actuator that positions the disk heads, the bus interface circuitry, and the microcontroller/s and memory chips. The spindle motor dissipates the bulk of the power, with the entire disk assembly typically dissipating power in the tens of watts.

Schemes for Reducing Power and Energy

There are numerous mechanisms in the literature that attack the power dissipation and/or energy consumption problem. Here, we will briefly describe three: dynamic voltage scaling, the powering down of unused blocks, and circuit-level approaches for reducing leakage power.

Dynamic Voltage Scaling Recall that total energy is the sum of switching energy and leakage energy, which, to a first approximation, is equal to the following:

$$\begin{aligned} E_{tot} = {} & [(C_{tot} \cdot V^2{}_{dd} \cdot \alpha \cdot f) \\ & + (N_{tot} \cdot I_{leakage} \cdot V_{dd})] \cdot T \end{aligned} \qquad \textbf{(EQ Ov.8)}$$

T is the time required for the computation, and N_{tot} is the total number of devices leaking current. Variations in processor utilization affect the amount of switching activity (the activity ratio α). However, a light workload produces an idle processor that wastes clock cycles and energy because the clock signal continues propagating and the operating voltage remains the same. Gating the clock during idle cycles reduces the switched capacitance C_{tot} during idle cycles. Reducing the frequency f during periods of low workload eliminates most idle cycles altogether.

None of the approaches, however, affects $C_{\text{tot}}V^2_{\text{dd}}$ for the actual computation or substantially reduces the energy lost to leakage current. Instead, reducing the supply voltage V_{dd} in conjunction with the frequency f achieves savings in switching energy and reduces leakage energy. For high-speed digital CMOS, a reduction in supply voltage increases the circuit delay as shown by the following equation [Baker et al. 1998, Baker 2005]:

$$T_d = \frac{C_L V_{\text{dd}}}{\mu C_{ox}(W/L)(V_{\text{dd}} - V_t)^2} \quad \textbf{(EQ Ov.9)}$$

where

- T_d is the delay or the reciprocal of the frequency f
- V_{dd} is the supply voltage
- C_L is the total node capacitance
- μ is the carrier mobility
- C_{ox} is the oxide capacitance
- V_t is the threshold voltage
- *W/L* is the width-to-length ratio of the transistors in the circuit

This can be simplified to the following form, which gives the maximum operating frequency as a function of supply and threshold voltages:

$$f_{\text{MAX}} \sim \frac{(V_{\text{dd}} - V_t)^2}{V_{\text{dd}}} \quad \textbf{(EQ Ov.10)}$$

As mentioned earlier, the threshold voltage is closely tied to the problem of leakage power, so it cannot be arbitrarily lowered. Thus, the right-hand side of the relation ends up being a constant proportion of the operating voltage for a given process technology. Microprocessors typically operate at the maximum speed at which their operating voltage level will allow, so there is not much headroom to arbitrarily lower V_{dd} by itself. However, V_{dd} can be lowered if the clock frequency is also lowered in the same proportion. This mechanism is called *dynamic voltage scaling* (*DVS)* [Pering & Broderson 1998] and is appearing in nearly every modern microprocessor. The technique sets the microprocessor's frequency to the most appropriate level for performing each task at hand, thus avoiding hurry-up-and-wait scenarios that consume more energy than is required for the computation (see Figure Ov.13). As Weiser points out,

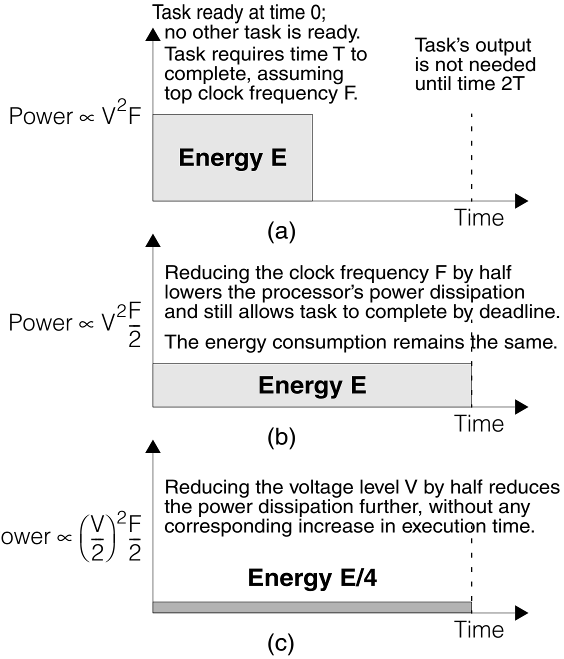

Task ready at time 0; no other task is ready. Task requires time T to complete, assuming top clock frequency F.
Task's output is not needed until time 2T
Power ∝ V²F
Energy E
Time
(a)
Reducing the clock frequency F by half lowers the processor's power dissipation and still allows task to complete by deadline.
The energy consumption remains the same.
Power ∝ V² F/2
Energy E
Time
(b)
Reducing the voltage level V by half reduces the power dissipation further, without any corresponding increase in execution time.
Power ∝ (V/2)² F/2
Energy E/4
Time
(c)

FIGURE Ov.13: Dynamic voltage scaling. Not every task needs the CPU's full computational power. In many cases, for example, the processing of video and audio streams, the only performance requirement is that the task meet a deadline, see (a). Such cases create opportunities to run the CPU at a lower performance level and achieve the same perceived performance while consuming less energy. As (b) shows, reducing the clock frequency of a processor reduces power dissipation but simply spreads a computation out over time, thereby consuming the same total energy as before. As (c) shows,reducing the voltage level as well as the clock frequency achieves the desired goal of reduced energy consumption and appropriate performance level. Figure and caption from Varma et al. [2003].

idle time represents wasted energy, even if the CPU is stopped [Weiser et al. 1994].

Note that it is not sufficient to merely have a chip that *supports* voltage scaling. There must be a heuristic, either implemented in hardware or software, that decides when to scale the voltage and by how much to scale it. This decision is essentially a prediction of the near-future computational needs of the system and is generally made on the basis of the recent computing requirements of all tasks and threads running at the time. The development of good heuristics is a tricky problem (pointed out by Weiser et al. [1994]). Heuristics that closely track performance requirements save little energy, while those that save the most energy tend to do so at the expense of performance, resulting in poor response time, for example.

Most research quantifies the effect that DVS has on reducing dynamic power dissipation because dynamic power follows V_{dd} in a quadratic relationship: reducing V_{dd} can significantly reduce dynamic power. However, lowering V_{dd} also reduces leakage power, which is becoming just as significant as dynamic power. Though the reduction is only linear, it is nonetheless a reduction.

Note also that even though DVS is commonly applied to microprocessors, it is perfectly well suited to the memory system as well. As a processor's speed is decreased through application of DVS, it requires less speed out of its associated SRAM caches, whose power supply can be scaled to keep pace. This will reduce both the dynamic and the static power dissipation of the memory circuits.

Powering-Down Unused Blocks A popular mechanism for reducing power is simply to turn off functional blocks that are not needed. This is done at both the circuit level and the chip or I/O-device level.

At the circuit level, the technique is called *clock gating*. The clock signal to a functional block (e.g., an adder, multiplier, or predictor) passes through a gate, and whenever a control circuit determines that the functional block will be unused for several cycles, the gate halts the clock signal and sends a non-oscillating voltage level to the functional block instead. The latches in the functional block retain their information; do not change their outputs; and, because the data is held constant to the combinational logic in the circuit, do not switch. Therefore, it does not draw current or consume energy.

Note that, in the naïve implementation, the circuits in this instance are still powered up, so they still dissipate static power; clock gating is a technique that only reduces dynamic power. Other gating techniques can reduce leakage as well. For example, in caches, unused blocks can be powered down using Gated-V_{dd} [Powell et al. 2000] or Gated-ground [Powell et al. 2000] techniques. Gated-V_{dd} puts the power supply of the SRAM in a series with a transistor as shown in Figure Ov.14. With the stacking effect introduced by this transistor, the leakage current is reduced drastically. This technique benefits from having both low-leakage current and a simpler fabrication process requirement since only a single threshold voltage is conceptually required (although, as shown in Figure Ov.14, the gating transistor can also have a high threshold to decrease the leakage even further at the expense of process complexity).

At the device level, for instance in DRAM chips or disk assemblies, the mechanism puts the device into a low-activity, low-voltage, and/or low-frequency mode such as *sleep* or *doze* or, in the case of disks, *spin-down*. For example, microprocessors can dissipate anywhere from a fraction of a watt to over 100 W of power; when not in use, they can be put into a low-power sleep or doze mode that consumes milli-watts. The processor typically expects an interrupt to cause it to resume normal operation, for instance, a clock interrupt, the interrupt output of a watchdog timer, or an external device interrupt. DRAM chips typically consume on the order of 1 W each; they have a low-power mode that will reduce this by more than an order of magnitude. Disks typically dissipate power in the tens of watts, the bulk of which is in the spindle motor. When the disk is placed in the "spin-down" mode (i.e., it is not rotating, but it is still responding to the disk controller),

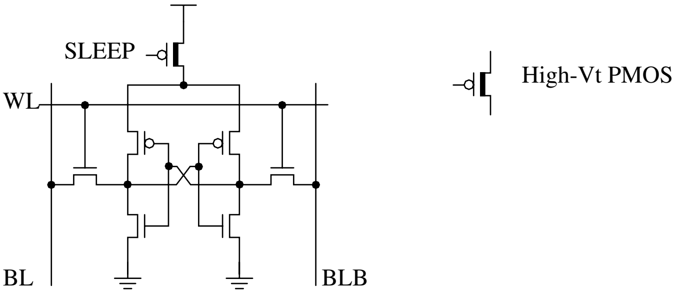


FIGURE Ov.14: Gated-V_{dd} technique using a high-V_t transistor to gate V_{dd}.

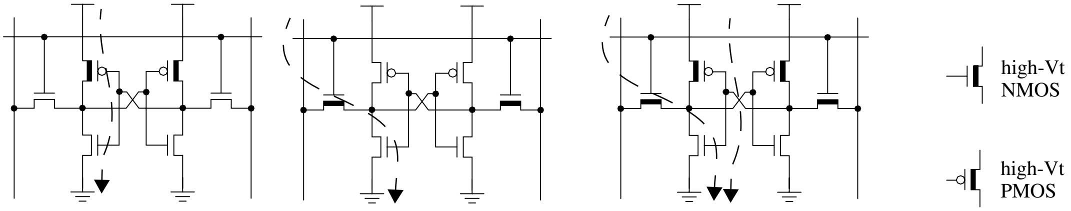


FIGURE Ov.15: Different multi-V_t configurations for the 6T memory cell showing which leakage currents are reduced for each configuration.

the disk assembly consumes a total of a handful of watts [Gurumurthi et al. 2003].

Leakage Power in SRAMs Low-power SRAM techniques provide good examples of approaches for lowering leakage power. SRAM designs targeted for low power have begun to account for the increasingly larger amount of power consumed by leakage currents.

One conceptually simple solution is the use of multi-threshold CMOS circuits. This involves using process-level techniques to increase the threshold voltage of transistors to reduce the leakage current. Increasing this threshold serves to reduce the gate overdrive and reduces the gate's drive strength, resulting in increased delay. Because of this, the technique is mostly used on the non-critical paths of the logic, and fast, low-V_t transistors are used for the critical paths. In this way the delay penalty involved in using higher V_t transistors can be hidden in the non-critical paths, while reducing the leakage currents drastically. For example, multi-V_t transistors are selectively used for memory cells since they represent a majority of the circuit, reaping the most benefit in leakage power consumption with a minor penalty in the access time. Different multi-V_t configurations are shown in Figure Ov.15, along with the leakage current path that each configuration is designed to minimize.

Another technique that reduces leakage power in SRAMs is the Drowsy technique [Kim et al. 2004a]. This is similar to gated-V_{dd} and gated-ground techniques in that it uses a transistor to conditionally enable the power supply to a given part of the SRAM. The difference is that this technique puts infrequently accessed parts of the SRAM into a

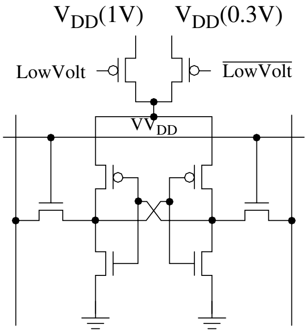


FIGURE Ov.16: A drowsy SRAM cell containing the transistors that gate the desired power supply.

state-preserving, low-power mode. A second power supply with a lower voltage than the regular supply provides power to memory cells in the "drowsy" mode. Leakage power is effectively reduced because of its dependence on the value of the power supply. An SRAM cell of a drowsy cache is shown in Figure Ov.16.

Ov.3.3 Reliability

Like performance, reliability means many things to many people. For example, embedded systems are computer systems, typically small, that run dedicated software and are embedded within the context of a larger system. They are increasingly appearing in the place of traditional electromechanical systems, whose function they are replacing because one can now find chip-level computer systems which can be programmed to perform virtually any function at a price of pennies per system. The reliability problem stems from the fact that the embedded system is a state machine (piece of software) executing within the context of a relatively complex state machine (real-time operating system) executing within the context of an extremely complex state machine (microprocessor and its memory system). We are replacing simple electromechanical systems with ultra-complex systems whose correct function cannot be guaranteed. This presents an enormous problem for the future, in which systems will only get more complex and will be used increasingly in safety-critical situations, where incorrect functioning can cause great harm.

This is a very deep problem, and one that is not likely to be solved soon. A smaller problem that we *can* solve right now—one that engineers currently do—is to increase the reliability of data within the memory system. If a datum is stored in the memory system, whether in a cache, in a DRAM, or on disk, it is reasonable to expect that the next time a processor reads that datum, the processor will get the value that was written.

How could the datum's value change? Solid-state memory devices (e.g., SRAMs and DRAMs) are susceptible to both hard failures and soft errors in the same manner that other semiconductor-based electronic devices are susceptible to both hard failures and soft failures. Hard failures can be caused by electromigration, corrosion, thermal cycling, or electrostatic shock. In contrast to hard failures, soft errors are failures where the physical device remains functional, but random and transient electronic noises corrupt the value of the stored information in the memory system. Transient noise and upset comes from a multitude of sources, including circuit noise (e.g., crosstalk, ground bounce, etc.), ambient radiation (e.g., even from sources within the computer chassis), clock jitter, or substrate interactions with high-energy particles. Which of these is the most common is obviously very dependent on the operating environment.

Figure Ov.17 illustrates the last of these examples. It pictures the interactions between high-energy alpha particles and neutrons with the silicon lattice. The figure shows that when high-energy alpha particles pass through silicon, the alpha particle leaves an ionized trail, and the length of that ionized trail depends on the energy of the alpha particle. The figure also illustrates that when high-energy neutrons pass through silicon, some neutrons pass through without affecting operations of the semiconductor device, but some neutrons collide with nuclei in the silicon lattice. The atomic collision can result in the creation of multiple ionized trails as the secondary particles generated in the collision scatter in the silicon lattice. In the presence of an electric field, the ionized trails of

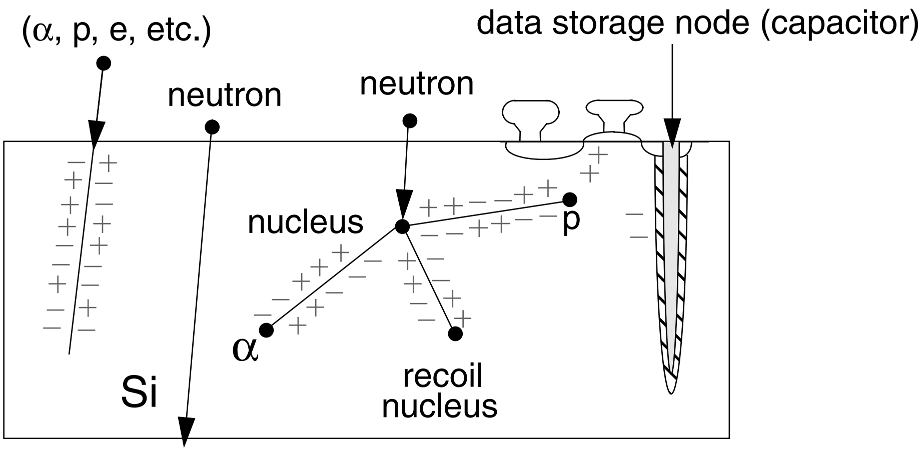


FIGURE Ov.17: Generation of electron-hole pairs in silicon by alpha particles and high-energy neutrons.

TABLE Ov.4 Cross-comparison of failure rates for SRAM, DRAM, and disk

Technology	Failure Rate[a] (SRAM & DRAM: at 0.13 µm)	Frequency of Multi-bit Errors (Relative to Single-bit Errors)	Expected Service Life
SRAM	100 per million device-hours		Several years
DRAM	1 per million device-hours	10–20%	Several years
Disk	1 per million device-hours		Several years

[a] Note that failure rate, i.e., a variation of mean-time-between-failures, says nothing about the expected performance of a single device. However, taken with the expected service life of a device, it can give a designer or administrator an idea of expected performance. If the service life of a device is 5 years, then the part will last about 5 years. A very large installation of those devices (e.g., in the case of disks or DRAMs, hundreds or more) will collectively see the expected failure rate: i.e., several hundred disks will collectively see several million device hours of operation before a single disk fails.

electron-hole pairs behave as temporary surges in current or as charges that can change the data values in storage cells. In addition, charge from the ionized trails of electron-hole pairs can impact the voltage level of bit lines as the value of the stored data is resolved by the sense amplifiers. The result is that the *soft error rate (SER)* of a memory-storage device depends on a combination of factors including the type, number, and energy distribution of the incident particles as well as the process technology design of the storage cells, design of the bit lines and sense amplifiers, voltage level of the device, as well as the design of the logic circuits that control the movement of data in the DRAM device.

Table Ov.4 compares the failure rates for SRAM, DRAM, and disk. SRAM device error rates have historically tracked DRAM devices and did so up until the 180-nm process generation. The combination of reduced supply voltage and reduced critical cell charge means that SRAM SERs have climbed dramatically for the 180-nm and 130-nm process generations. In a recent publication, Monolithic System

Technology, Inc. (MoSys) claimed that for the 250-nm process generation, SRAM SERs were reported to be in the range of 100 failures per million device-hours per megabit, while SERs were reported to be in the range of 100,000 failures per megabit for the 130-nm process generation. The generalized trend is expected to continue to increase as the demand for low power dissipation forces a continued reduction in supply voltage and reduced critical charge per cell.

Solid-state memory devices (SRAMs and DRAMs) are typically protected by error detection codes and/or ECC. These are mechanisms wherein data redundancy is used to detect and/or recover from single- and even multi-bit errors. For instance, parity is a simple scheme that adds a bit to a protected word, indicating the number of even or odd bits in the word. If the read value of the word does not match the parity value, then the processor knows that the read value does not equal the value that was initially written, and an error has occurred. Error correction is achieved by encoding a word such that a bit error moves the resulting word some distance away from the original word (in the Hamming-distance sense) into an invalid encoding. The encoding space is chosen such that the new, invalid word is closest in the space to the original, valid word. Thus, the original word can always be derived from an invalid codeword, assuming a maximum number of bit errors.

Due to SRAM's extreme sensitivity to soft errors, modern processors now ship with parity and single-bit error correction for the SRAM caches. Typically, the tag arrays are protected by parity, whereas the data arrays are protected by single-bit error correction. More sophisticated multi-bit ECC algorithms are typically not deployed for on-chip SRAM caches in modern processors since the addition of sophisticated computation circuitry can add to the die size and cause significant delay relative to the timing demands of the on-chip caches. Moreover, caches store frequently accessed data, and in case an uncorrectable error is detected, a processor simply has to re-fetch the data from memory. In this sense, it can be considered unnecessary to detect and correct multi-bit errors, but sufficient to simply detect multi-bit errors. However, in the physical design of modern SRAMs, often designers will intentionally place capacitors above the SRAM cell to improve SER.

Disk reliability is a more-researched area than data reliability in disks, because data stored in magnetic disks tends to be more resistant to transient errors than data stored in solid-state memories. In other words, whereas reliability in solid-state memories is largely concerned with correcting soft errors, reliability in hard disks is concerned with the fact that disks occasionally die, taking most or all of their data with them. Given that the disk drive performs the function of permanent store, its reliability is paramount, and, as Table Ov.4 shows, disks tend to last several years. This data is corroborated by a recent study from researchers at Google [Pinheiro et al. 2007]. The study tracks the behavior and environmental parameters of a fleet of over 100,000 disks for five years.

Reliability in the disk system is improved in much the same manner as ECC: data stored in the disk system is done so in a redundant fashion. RAID (redundant array of inexpensive disks) is a technique wherein encoded data is striped across multiple disks, so that even in the case of a disk's total failure the data will always be available.

Ov.3.4 Virtual Memory

Virtual memory is the mechanism by which the operating system provides executing software access to the memory system. In this regard, it is the primary consumer of the memory system: its procedures, data structures, and protocols dictate how the components of the memory system are used by all software that runs on the computer. It therefore behooves the reader to know what the virtual memory system does and how it does it. This section provides a brief overview of the mechanics of virtual memory. More detailed treatments of the topic can also be found on-line in articles by the author [Jacob & Mudge 1998a–c].

In general, programs today are written to run on no particular hardware configuration. They have no knowledge of the underlying memory system. Processes execute in imaginary address spaces that are mapped onto the memory system (including the DRAM system and disk system) by the operating system. Processes generate instruction fetches and loads and stores using imaginary or "virtual" names for their instructions and data. The ultimate home for the process's address space is nonvolatile *permanent store*, usually a disk drive; this is where the process's instructions and data come from and where all of its permanent changes go to. Every hardware memory structure between the CPU and the permanent store is a cache for the instructions and data in the process's address space. This includes main memory—main memory is really nothing more than a cache for a process's virtual address space. A cache operates on the principle that a small, fast storage device can hold the most important data found on a larger, slower storage device, effectively making the slower device look fast. The large storage area in this case is the process address space, which can range from kilobytes to gigabytes or more in size. Everything in the address space initially comes from the program file stored on disk or is created on demand and defined to be zero. This is illustrated in Figure Ov.18.

Address Translation

Translating addresses from virtual space to physical space is depicted in Figure Ov.19. Addresses are mapped at the granularity of *pages*. Virtual memory is

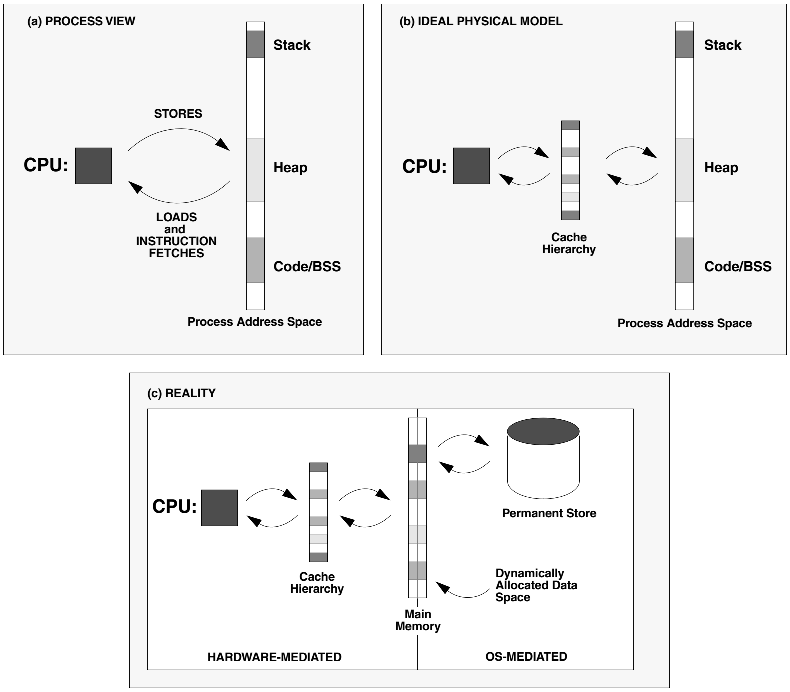


FIGURE Ov.18: Caching the process address space. In the first view, a process is shown referencing locations in its address space. Note that all loads, stores, and fetches use virtual names for objects. The second view illustrates that a process references locations in its address space indirectly through a hierarchy of caches. The third view shows that the address space is not a linear object stored on some device, but is instead scattered across hard drives and dynamically allocated when necessary.

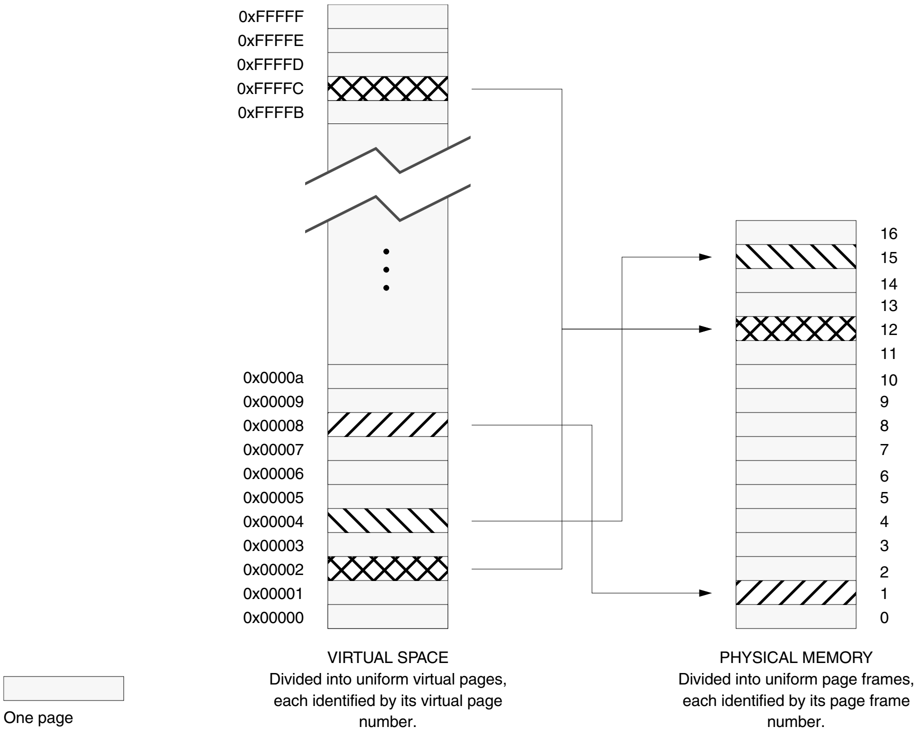


FIGURE Ov.19: Mapping virtual pages into physical page frames.

essentially a mapping of *virtual page numbers* (*VPNs*) to *page frame numbers* (*PFNs*). The mapping is a function, and any virtual page can have only one location. However, the inverse map is not necessarily a function. It is possible and sometimes advantageous to have several virtual pages mapped to the same page frame (to share memory between processes or threads or to allow different views of data with different protections, for example). This is depicted in Figure Ov.19 by mapping two virtual pages (0x00002 and 0xFFFFC) to PFN 12.

If DRAM is a cache, what is its organization? For example, an idealized *fully associative* cache (one in which any datum can reside at any location within the cache's data array) is pictured in Figure Ov.20. A data tag is fed into the cache. The first stage compares the input tag to the tag of every piece of data in the cache. The matching tag points to the data's location in the cache. However, DRAM is not physically built like a cache. For example, it has no inherent concept of a tags array: one merely tells memory what data location one wishes to read or write, and the datum at that location is read out or overwritten. There is no attempt to match the address against a tag to verify the contents of the data location. However, if main memory is to be an effective cache for the virtual address space, the tags mechanism must be implemented *somewhere*. There is clearly a myriad of possibilities, from special DRAM designs that include a hardware tag feature to software algorithms that make several memory references to look up one datum. Traditional virtual memory has the tags array implemented in software, and this software structure often holds more entries than there are entries in the data array (i.e., pages in main memory). The software structure is called a *page table;* it is a database of mapping information.

The page table performs the function of the tags array depicted in Figure Ov.20. For any given memory reference, it indicates where in main memory (corresponding to "data array" in the figure) that page can be found. There are many different possible organizations for page tables, most of which require only a few memory references to find the appropriate tag entry. However, requiring more than one memory reference for a page table lookup can be very costly, and so access to the page table is sped up by caching its entries in a special cache called the *translation lookaside buffer* (*TLB*) [Lee 1960], a hardware structure that typically has far fewer entries than there are pages in main memory. The TLB is a hardware cache which is usually implemented as a content addressable memory (CAM), also called a fully associative cache.

The TLB takes as input a VPN, possibly extended by an address-space identifier, and returns the corresponding PFN and protection information. This is illustrated in Figure Ov.21. The address-space identifier, if used, extends the virtual address to distinguish it from similar virtual addresses produced by other processes. For a load or store to complete successfully, the

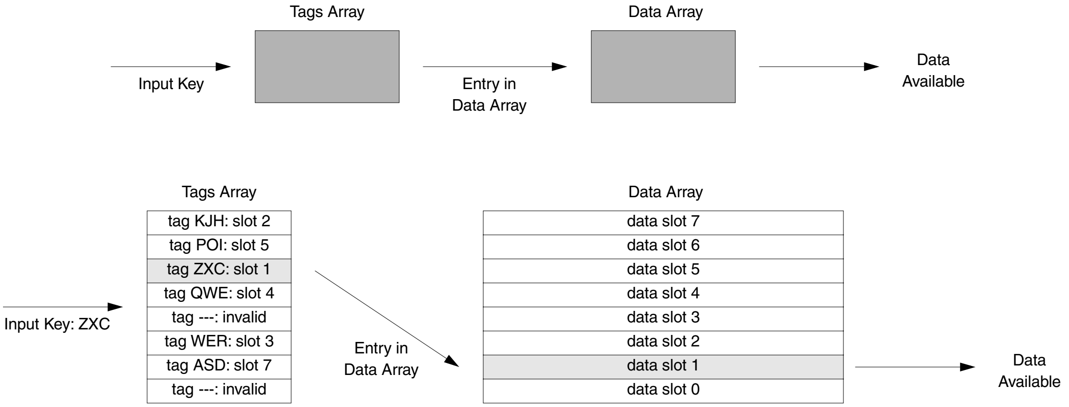


FIGURE Ov.20: An idealized cache lookup. A cache is comprised of two parts: the tag's array and the data array. In the example organization, the tags act as a database. They accept as input a key (an address) and output either the location of the item in the data array or an indication that the item is not in the data array.

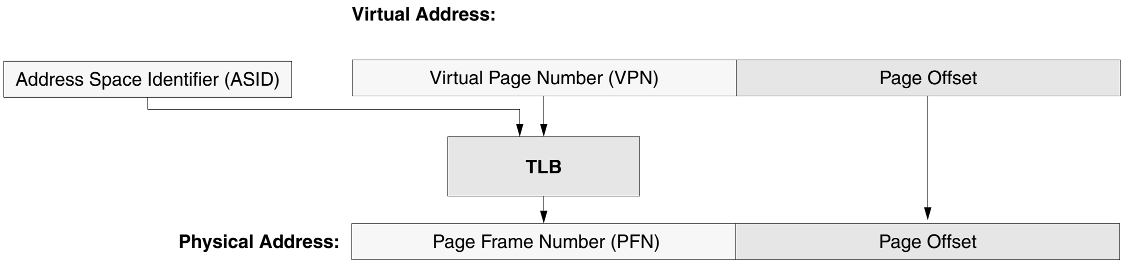


FIGURE Ov.21: Virtual-to-physical address translation using a TLB.

TLB must contain the mapping information for that virtual location. If it does not, a *TLB miss* occurs, and the system[9] must search the page table for the appropriate entry and place it into the TLB. If the system fails to find the mapping information in the page table, or if it finds the mapping but it indicates that the desired page is on disk, a *page fault* occurs. A page fault interrupts the OS, which must then retrieve the page from disk and place it into memory, create a new page if the page does not yet exist (as when a process allocates a new stack frame in virgin territory), or send the process an error signal if the access is to illegal space.

Shared Memory

Shared memory is a feature supported by virtual memory that causes many problems and gives rise to cache-management issues. It is a mechanism whereby two address spaces that are normally protected from each other are allowed to intersect at points, still retaining protection over the non-intersecting regions. Several processes sharing portions of their address spaces are pictured in Figure Ov.22. The shared memory mechanism only opens up a pre-defined portion of a process's address space; the rest of the address space is still protected, and even the shared portion is only unprotected for those processes sharing the memory. For instance, in Figure Ov.22, the region of A's address space that is shared with process B is unprotected from whatever actions B might want to take, but it is safe from the actions of any other processes. It is therefore useful as a simple, secure means for inter-process communication. Shared memory also reduces requirements for physical memory, as when the text regions of processes are shared whenever multiple instances of a single program are run or when multiple instances of a common library are used in different programs.

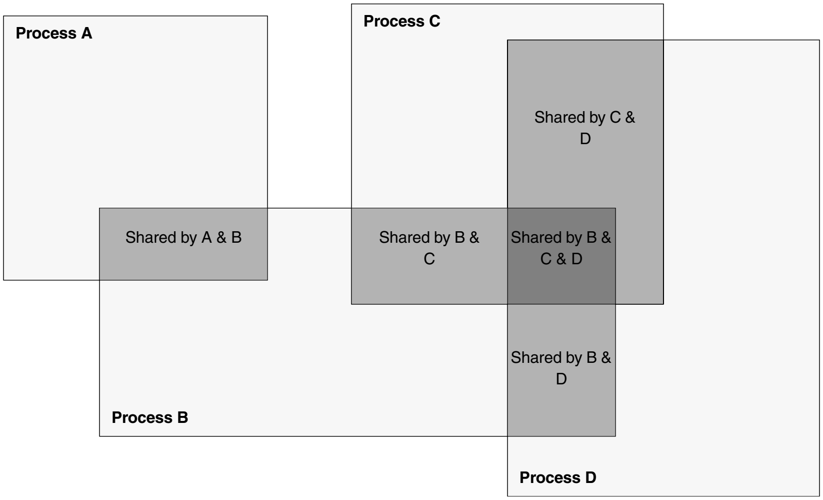


FIGURE Ov.22: Shared memory. Shared memory allows processes to overlap portions of their address space while retaining protection for the nonintersecting regions. This is a simple and effective method for inter-process communication. Pictured are four process address spaces that have overlapped. The darker regions are shared by more than one process, while the lightest regions are still protected from other processes.

[9] In the discussions, we will use the generic term "system" when the acting agent is implementation-dependent and can refer to either a hardware state machine or the operating system. For example, in some implementations, the page table search immediately following a TLB miss is performed by the operating system (MIPS, Alpha); in other implementations, it is performed by the hardware (PowerPC, x86).

The mechanism works by ensuring that shared pages map to the same physical page. This can be done by simply placing the same PFN in the page tables of two processes sharing a page. An example is shown in Figure Ov.23. Here, two very small address spaces are shown overlapping at several places, and one address space overlaps with itself; two of its virtual pages map to the same physical page. This is not just a contrived example. Many operating systems allow this, and it is useful, for example, in the implementation of user-level threads.

Some Commercial Examples

A few examples of what has been done in industry can help to illustrate some of the issues involved.

MIPS Page Table Design MIPS [Heinrich 1995, Kane & Heinrich 1992] eliminated the page table-walking hardware found in traditional memory management units and, in doing so, demonstrated that software can table-walk with reasonable efficiency. It also presented a simple hierarchical page table design, shown in Figure Ov.24. On a TLB miss, the VPN of the

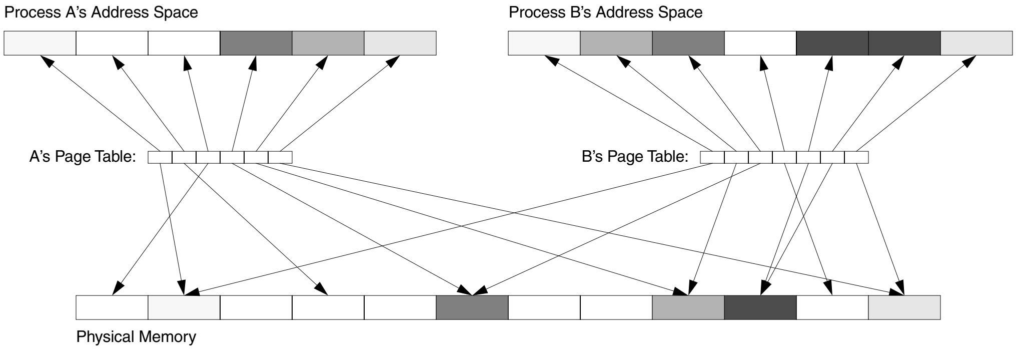


FIGURE Ov.23: An example of shared memory. Two process address spaces—one comprised of six virtual pages and the other of seven virtual pages—are shown sharing several pages. Their page tables maintain information on where virtual pages are located in physical memory. The darkened pages are mapped to several locations; note that the darkest page is mapped at two locations in the same address space.

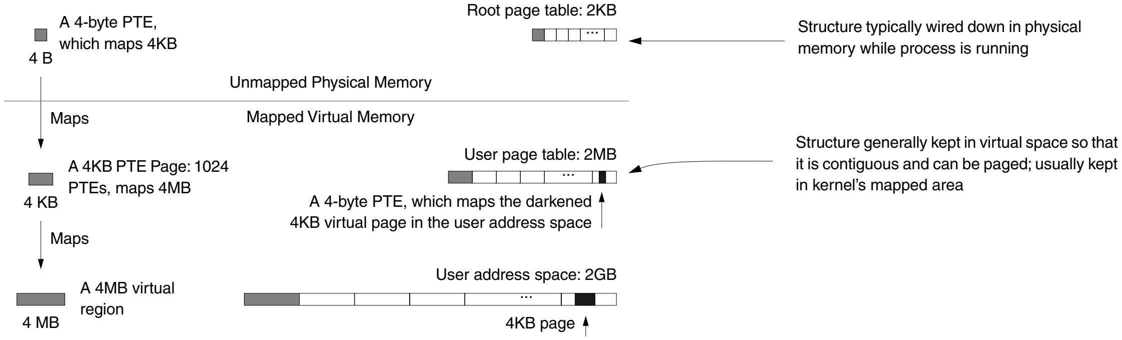

A 4-byte PTE, which maps 4KB
4 B
Root page table: 2KB
Structure typically wired down in physical memory while process is running
Unmapped Physical Memory
Mapped Virtual Memory
Maps
A 4KB PTE Page: 1024 PTEs, maps 4MB
4 KB
User page table: 2MB
Structure generally kept in virtual space so that it is contiguous and can be paged; usually kept in kernel's mapped area
A 4-byte PTE, which maps the darkened 4KB virtual page in the user address space
Maps
A 4MB virtual region
4 MB
User address space: 2GB
4KB page

FIGURE Ov.24: The MIPS 32-bit hierarchical page table. MIPS hardware provides support for a 2-MB linear virtual page table that maps the 2-GB user address space by constructing a virtual address from a faulting virtual address that indexes the mapping PTE (page-table entry) in the user page table. This 2-MB page table can easily be mapped by a 2-KB user root page table.

address that missed the TLB is used as an index into the user page table, which is accessed using a virtual address. The architecture provides hardware support for this activity, storing the virtual address of the base of the user-level page table in a hardware register and forming the concatenation of the base address with the VPN. This is illustrated in Figure Ov.25. On a TLB miss, the hardware creates a virtual address for the mapping PTE in the user page table, which must be aligned on a 2-MB virtual boundary for the hardware's lookup address to work. The base pointer, called *PTEBase*, is stored in a hardware register and is usually changed on context switch.

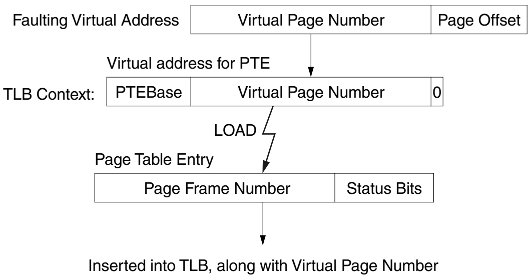


FIGURE Ov.25: The use of the MIPS TLB context register. The VPN of the faulting virtual address is placed into the context register, creating the virtual address of the mapping PTE. This PTE goes directly into the TLB.

PowerPC Segmented Translation The IBM 801 introduced a segmented design that persisted through the POWER and PowerPC architectures [Chang & Mergen 1988, IBM & Motorola 1993, May et al. 1994, Weiss & Smith 1994]. It is illustrated in Figure Ov.26. Applications generate 32-bit "effective" addresses that are mapped onto a larger "virtual" address space at the granularity of *segments*, 256-MB virtual regions. Sixteen segments comprise an application's address space. The top four bits of the effective address select a segment identifier from a set of 16 registers. This segment ID is concatenated with the bottom 28 bits of the effective address to form an extended virtual address. This extended address is used in the TLB and page table. The operating system performs data movement and relocation at the granularity of pages, not segments.

The architecture does not use explicit address-space identifiers; the segment registers ensure address space protection. If two processes duplicate an identifier in their segment registers, they share that virtual segment by definition. Similarly, protection is guaranteed if identifiers are *not* duplicated. If memory is shared through global addresses, the TLB and cache need not be flushed on context switch[10] because the system behaves like a single address space operating system. For more details, see Chapter 31, Section 31.1.7, *Perspective: Segmented Addressing Solves the Synonym Problem.*

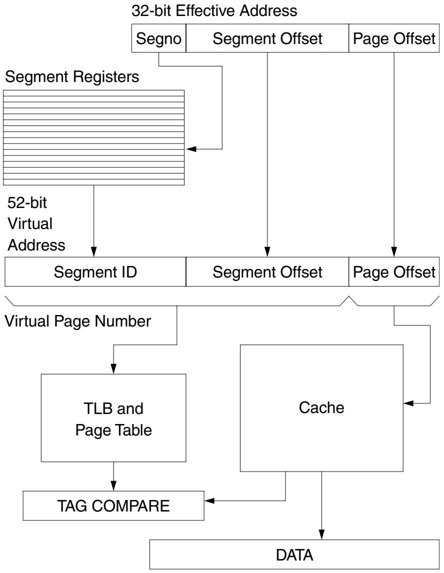


FIGURE Ov.26: PowerPC segmented address translation. Processes generate 32-bit effective addresses that are mapped onto a 52-bit address space via 16 segment registers, using the top 4 bits of the effective address as an index. It is this extended virtual address that is mapped by the TLB and page table. The segments provide address space protection and can be used for shared memory.

[10] Flushing is avoided until the system runs out of identifiers and must reuse them. For example, the address-space identifiers on the MIPS R3000 and Alpha 21064 are six bits wide, with a maximum of 64 active processes [Digital 1994, Kane & Heinrich 1992]. If more processes are desired, identifiers must be constantly reassigned, requiring TLB and virtual-cache flushes.

Ov.4 An Example Holistic Analysis

Disk I/O accounts for a substantial fraction of an application's execution time and power dissipation. A new DRAM technology called *Fully Buffered DIMM* (*FB-DIMM*) has been in development in the industry [Vogt 2004a, b, Haas & Vogt 2005], and, though it provides storage scalability significantly beyond the current DDRx architecture, FB-DIMM has met with some resistance due to its high power dissipation. Our modeling results show that the energy consumed in a moderate-size FB-DIMM system is indeed quite large, and it can easily approach the energy consumed by a disk.

This analysis looks at a trade-off between storage in the DRAM system and in the disk system, focusing on the disk-side write buffer; if configured and managed correctly, the write buffer enables a system to approach the performance of a large DRAM installation at half the energy. Disk-side caches and write buffers have been proposed and studied, but their effect upon total system behavior has not been studied. We present the impact on total system execution time, CPI, and memory-system power, including the effects of the operating system. Using a full-system, execution-based simulator that combines Bochs, Wattch, CACTI, DRAMsim, and DiskSim and boots the RedHat Linux 6.0 kernel, we have investigated the memory-system behavior of the SPEC CPU2000 applications. We study the disk-side cache in both single-disk and RAID-5 organizations. Cache parameters include size, organization, whether the cache supports write caching or not, and whether it prefetches read blocks or not. Our results are given in terms of L1/L2 cache accesses, power dissipation, and energy consumption; DRAM-system accesses, power dissipation, and energy consumption; disk-system accesses, power dissipation, and energy consumption; and execution time of the application plus operating system, in seconds. The results are not from sampling, but rather from a simulator that calculates these values on a cycle-by-cycle basis over the entire execution of the application.

Ov.4.1 Fully-Buffered DIMM vs. the Disk Cache

It is common knowledge that disk I/O is expensive in both power dissipated and time spent waiting on it. What is less well known is the system-wide breakdown of disk power versus cache power versus DRAM power, especially in light of the newest DRAM architecture adopted by industry, the FB-DIMM. This new DRAM standard replaces the conventional memory bus with a narrow, high-speed interface between the memory controller and the DIMMs. It has been shown to provide performance similar to that of DDRx systems, and thus, it represents a relatively low-overhead mechanism (in terms of execution time) for scaling DRAM-system capacity. FB-DIMM's latency degradation is not severe. It provides a noticeable bandwidth improvement, and it is relatively insensitive to scheduling policies [Ganesh et al. 2007].

FB-DIMM was designed to solve the problem of storage scalability in the DRAM system, and it provides scalability well beyond the current JEDEC-style DDRx architecture, which supports at most two to four DIMMs in a fully populated dual-channel system (DDR2 supports up to two DIMMs per channel; proposals for DDR3 include limiting a channel to a single DIMM). The daisy-chained architecture of FB-DIMM supports up to eight DIMMs per channel, and its narrow bus requires roughly one-third the pins of a DDRx SDRAM system. Thus, an FB-DIMM system supports an order of magnitude more DIMMs than DDRx. This scalability comes at a cost, however. The DIMM itself dissipates almost an order of magnitude more power than a traditional DDRx DIMM. Couple this with an order-of-magnitude increase in DIMMs per system, and one faces a serious problem.

To give an idea of the problem, Figure Ov.27 shows the simulation results of an entire execution of the *gzip* benchmark from SPEC CPU2000 on a complete-system simulator. The memory system is only moderate in size: one channel and four DIMMs, totalling a half-gigabyte. The graphs demonstrate numerous important issues, but in this book we are concerned with two items in particular:

- Program initialization is lengthy and represents a significant portion of an application's run time. As the CPI graph shows, the first two-thirds of execution time are spent dealing with the disk, and the corresponding CPI (both average and instantaneous) ranges from the 100s to the 1000s. After this initialization phase, the application settles into a

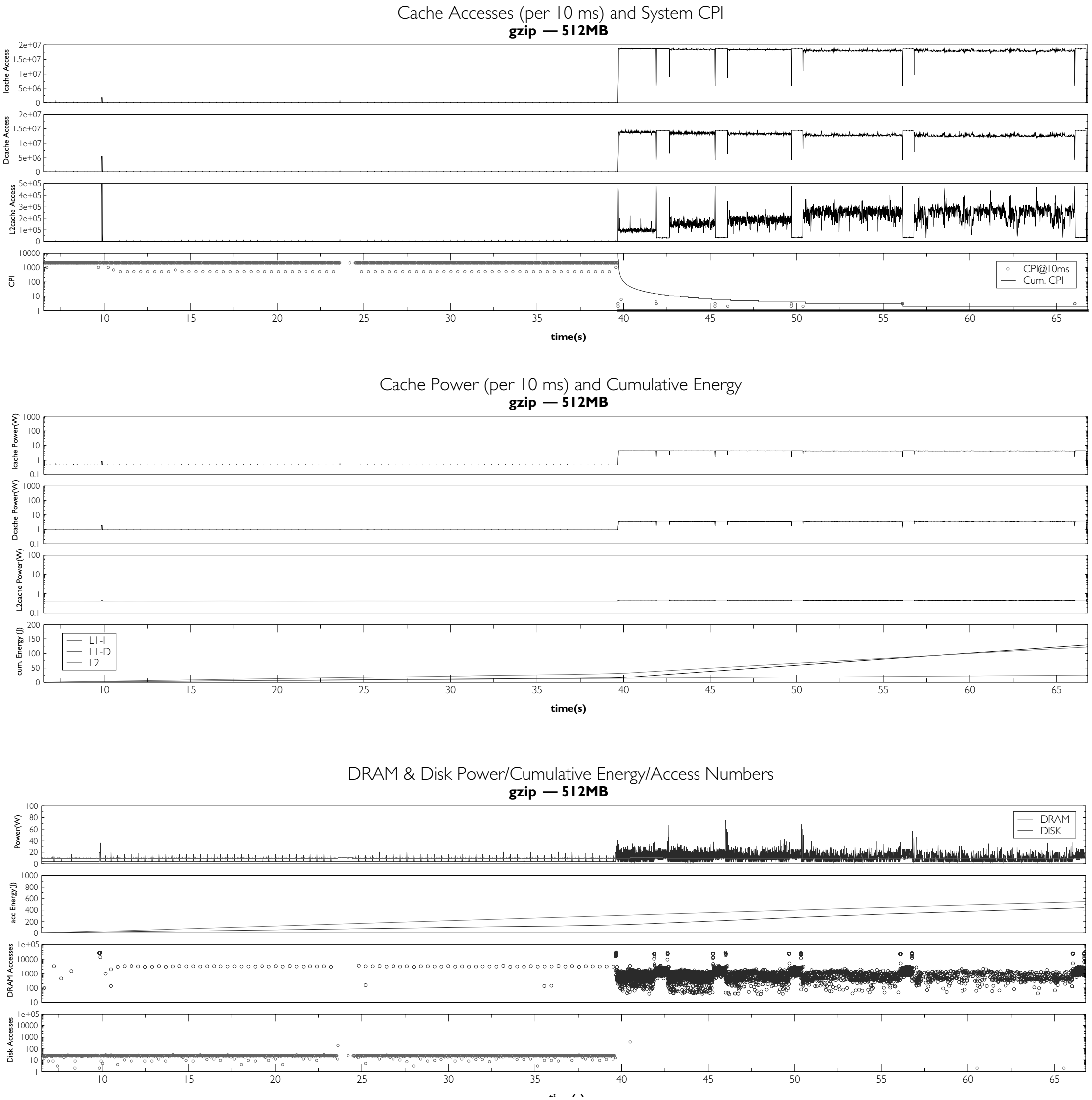


FIGURE Ov.27: Full execution of Gzip. The figure shows the entire run of gzip. System configuration is a 2-GHz Pentium processor with 512 MB of DDR2-533 FB-DIMM main memory and a 12k-RPM disk drive with built-in disk cache. The figure shows the interaction between all components of the memory system, including the L1 instruction and data caches, the unified L2 cache, the DRAM system, and the disk drive. All graphs use the same x-axis, which represents execution time in seconds. The x-axis does not start at zero; the measurements exclude system boot time, invocation of the shell, etc. Each data point represents aggregated (not sampled) activity within a 10-ms epoch. The CPI graph shows two system CPI values: one is the average CPI for each 10-ms epoch, and the other is the cumulative average CPI. A duration with no CPI data point indicates that no instructions were executed due to I/O latency. During such a window the CPI is essentially infinite, and thus, it is possible for the cumulative average to range higher than the displayed instantaneous CPI. Note that the CPI, the DRAM accesses, and the disk accesses are plotted on log scales.

more compute-intensive phase in which the CPI asymptotes down to the theoretical sustainable performance, the single-digit values that architecture research typically reports.

- By the end of execution, the total energy consumed in the FB-DIMM DRAM system (a half a kilojoule) almost equals that of the energy consumed by the disk, and it is twice that of the L1 data cache, L1 instruction cache, and unified L2 cache combined.

Currently, there is substantial work happening in both industry and academia to address the latter issue, with much of the work focusing on access scheduling, architecture improvements, and data migration. To complement this work, we look at a wide range of organizational approaches, i.e., attacking the problem from a parameter point of view rather than a system-redesign, component-redesign, or new-proposed-mechanism point of view, and find significant synergy between the disk cache and the memory system. Choices in the disk-side cache affect both system-level performance and system-level (in particular, DRAM-subsystem-level) energy consumption. Though disk-side caches have been proposed and studied, their effect upon the total system behavior, namely execution time or CPI or total memory-system power including the effects of the operating system, is as yet unreported. For example, Zhu and Hu [2002] evaluate disk built-in cache using both real and synthetic workloads and report the results in terms of average response time. Smith [1985a and b] evaluates a disk cache mechanism with real traces collected in real IBM mainframes on a disk cache simulator and reports the results in terms of miss rate. Huh and Chang [2003] evaluate their RAID controller cache organization with a synthetic trace. Varma and Jacobson [1998] and Solworth and Orji [1990] evaluate destaging algorithms and write caches, respectively, with synthetic workloads. This study represents the first time that the effects of the disk-side cache can be viewed at a system level (considering both application and operating-system effects) and compared directly to all the other components of the memory system.

We use a full-system, execution-based simulator combining Bochs [Bochs 2006], Wattch [Brooks et al. 2000], CACTI [Wilton & Jouppi 1994], DRAMsim [Wang et al. 2005, September], and DiskSim [Ganger et al. 2006]. It boots the RedHat Linux 6.0 kernel and therefore can capture all application behavior, and all operating-system behavior, including I/O activity, disk-block buffering, system-call overhead, and virtual memory overhead such as translation, table walking, and page swapping. We investigate the disk-side cache in both single-disk and RAID-5 organizations. Cache parameters include size, organization, whether the cache supports write caching or not, and whether it prefetches read blocks or not. Additional parameters include disk rotational speed and DRAM-system capacity.

We find a complex trade-off between the disk cache, the DRAM system, and disk parameters like rotational speed. The disk cache, particularly its write-buffering feature, represents a very powerful tool enabling significant savings in both energy and execution time. This is important because, though the cache's support for write buffering is often enabled in desktop operating systems (e.g., Windows and some but not all flavors of Unix/Linux [Ng 2006]), it is typically disabled in enterprise computing applications [Ng 2006], and these are the applications most likely to use FB-DIMMs [Haas & Vogt 2005]. We find substantial improvement between existing implementations and an ideal write buffer (i.e., this is a limit study). In particular, the disk cache's write-buffering ability can offset the total energy consumption of the memory system (including caches, DRAMs, and disks) by nearly a factor of two, while sacrificing a small amount of performance.

Ov.4.2 Fully Buffered DIMM: Basics

The relation between a traditional organization and a FB-DIMM organization is shown in Figure Ov.28, which motivates the design in terms of a graphics-card organization. The first two drawings show a multi-drop DRAM bus next to a DRAM bus organization typical of graphics cards, which use point-to-point soldered connections between the DRAM and memory controller to achieve higher speeds. This arrangement is used in FB-DIMM.

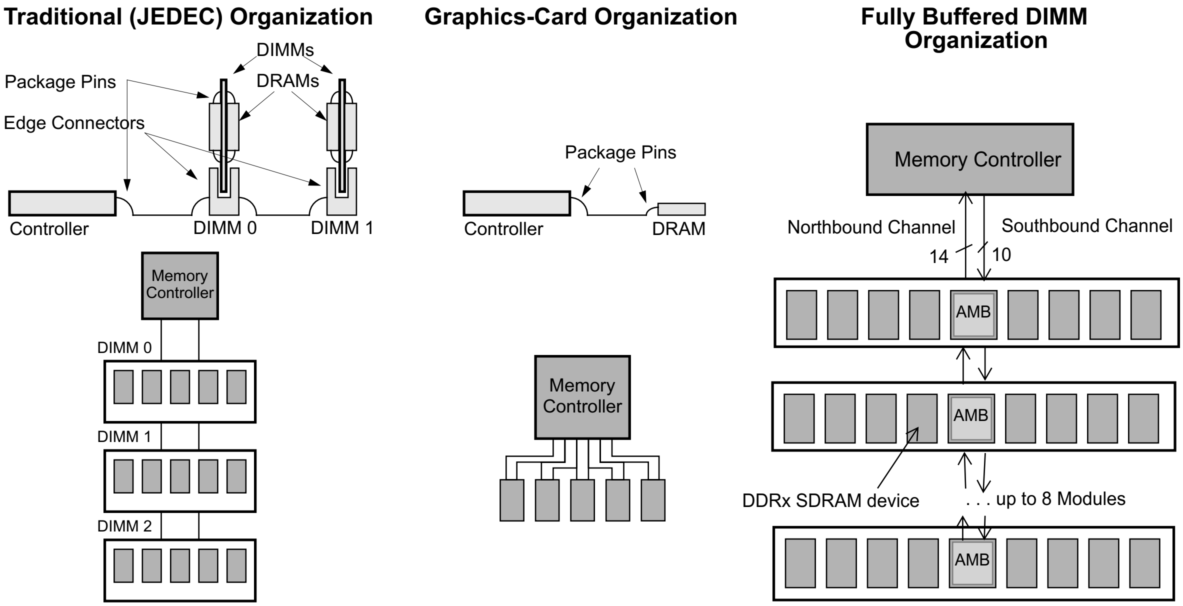


FIGURE Ov.28: FB-DIMM and its motivation. The first two pictures compare the memory organizations of a JEDEC SDRAM system and a graphics card. Above each design is its side-profile, indicating potential impedance mismatches (sources of reflections). The organization on the far right shows how the FB-DIMM takes the graphics-card organization as its *de facto* DIMM. In the FB-DIMM organization, there are no multi-drop busses; DIMM-to-DIMM connections are point to point. The memory controller is connected to the nearest AMB via two unidirectional links. The AMB is, in turn, connected to its southern neighbor via the same two links.

A slave memory controller has been added onto each DIMM, and all connections in the system are point to point. A narrow, high-speed channel connects the master memory controller to the DIMM-level memory controllers (called *Advanced Memory Buffers* or AMBs). Since each DIMM-to-DIMM connection is a point-to-point connection, a channel becomes a *de facto* multi-hop store and forward network. The FB-DIMM architecture limits the channel length to eight DIMMs, and the narrower inter-module bus requires roughly one-third as many pins as a traditional organization. As a result, an FB-DIMM organization can handle roughly 24 times the storage capacity of a single-DIMM DDR3-based system, without sacrificing any bandwidth and even leaving headroom for increased intra-module bandwidth.

The AMB acts like a pass-through switch, directly forwarding the requests it receives from the controller to successive DIMMs and forwarding frames from southerly DIMMs to northerly DIMMs or the memory controller. All frames are processed to determine whether the data and commands are for the local DIMM. The FB-DIMM system uses a serial packet-based protocol to communicate between the memory controller and the DIMMs. Frames may contain data and/or commands. Commands include DRAM commands such as row activate (RAS), column read (CAS), refresh (REF) and so on, as well as channel commands such as write to configuration registers, synchronization commands, etc. Frame scheduling is performed exclusively by the memory controller. The AMB only converts the serial protocol to DDRx-based commands without implementing any scheduling functionality.

The AMB is connected to the memory controller and/or adjacent DIMMs via unidirectional links: the southbound channel which transmits both data and commands and the northbound channel which transmits data and status information. The southbound and northbound datapaths are 10 bits and 14 bits wide, respectively. The FB-DIMM channel clock operates at six times the speed of the DIMM clock; i.e., the link speed is 4 Gbps for a 667-Mbps DDRx system. Frames on the north- and southbound channel require 12 transfers (6 FB-DIMM channel clock cycles) for transmission. This 6:1 ratio ensures that the FB-DIMM frame rate matches the DRAM command clock rate.

Southbound frames comprise both data and commands and are 120 bits long; northbound frames are data only and are 168 bits long. In addition to the data and command information, the frames also carry header information and a frame CRC (cyclic redundancy check) checksum that is used to check for transmission errors. A northbound read-data frame transports 18 bytes of data in 6 FB-DIMM clocks or 1 DIMM clock. A DDRx system can burst back the same amount of data to the memory controller in two successive beats lasting an entire DRAM clock cycle. Thus, the read bandwidth of an FB-DIMM system is the same as that of a single channel of a DDRx system. Due to the narrower southbound channel, the write bandwidth in FB-DIMM systems is one-half that available in a DDRx system. However, this makes the *total* bandwidth available in an FB-DIMM system 1.5 times that of a DDRx system.

Figure Ov.29 shows the processing of a read transaction in an FB-DIMM system. Initially, a command frame is used to transmit a command that will perform row activation. The AMB translates the request and relays it to the DIMM. The memory controller schedules the CAS command in a following frame. The AMB relays the CAS command to the DRAM devices which burst the data back to the AMB. The AMB bundles two consecutive bursts of data into a single northbound frame and transmits it to the memory controller. In this example, we assume a burst length of four corresponding to two FB-DIMM data frames. Note that although the figures do not identify parameters like t_CAS, t_RCD, and t_CWD, the memory controller must ensure that these constraints are met.

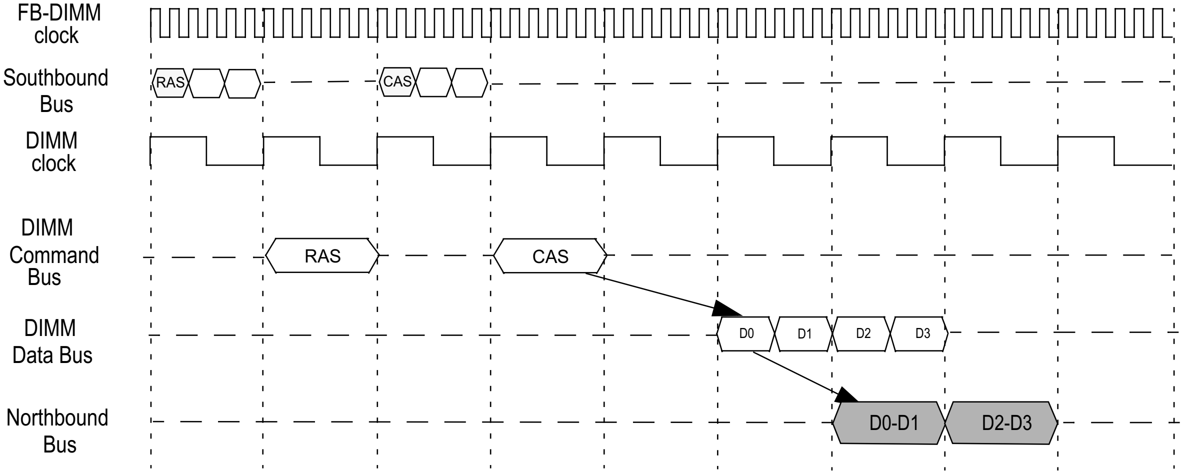


FIGURE Ov.29: Read transaction in an FB-DIMM system. The figure shows how a read transaction is performed in an FB-DIMM system. The FB-DIMM serial busses are clocked at six times the DIMM busses. Each FB-DIMM frame on the southbound bus takes six FB-DIMM clock periods to transmit. On the northbound bus a frame comprises two DDRx data bursts.

The primary dissipater of power in an FB-DIMM channel is the AMB, and its power depends on its position within the channel. The AMB nearest to the memory controller must handle its own traffic and repeat all packets to and from all downstream AMBs, and this dissipates the most power. The AMB in DDR2-533 FB-DIMM dissipates 6 W, and it is currently 10 W for 800 Mbps DDR2 [Staktek 2006]. Even if one averages out the activity on the AMB in a long channel, the eight AMBs in a single 800-Mbps channel can easily dissipate 50 W. Note that this number is for the AMBs only; it does not include power dissipated by the DRAM devices.

Ov.4.3 Disk Caches: Basics

Today's disk drives all come with a built-in cache as part of the drive controller electronics, ranging in size from 512 KB for the micro-drive to 16 MB for the largest server drives. Figure Ov.30 shows the cache and its place within a system. The earliest drives had no cache memory, as they had little control electronics. As the control of data transfer migrated from the host-side control logic to the drive's own controller, a small amount of memory was needed to act as a speed-matching buffer, because the disk's media data rate is different from that of the interface. Buffering is also needed because when the head is at a position ready to do data transfer, the host or the interface may be busy and not ready to receive read data. DRAM is usually used as this buffer memory.

In a system, the host typically has some memory dedicated for caching disk data, and if a drive is attached to the host via some external controller, that controller also typically has a cache. Both the system cache and the external cache are much larger than the disk drive's internal cache. Hence, for most workloads, the drive's cache is not likely to see too many reuse cache hits. However, the disk-side cache is very effective in opportunistically prefetching data, as only the controller inside the drive knows the state the drive is in and when and how it can prefetch without adding any cost in time. Finally, the drive needs cache memory if it is to support write caching/buffering.

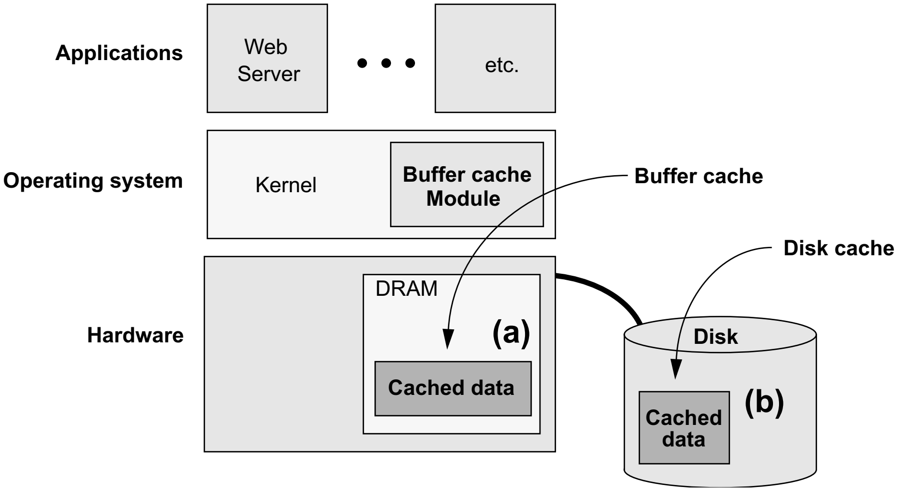


FIGURE Ov.30: Buffer caches and disk caches. Disk blocks are cached in several places, including (a) the operating system's *buffer cache* in main memory and (b), on the disk, in another DRAM buffer, called a *disk cache*.

With write caching, the drive controller services a write request by transferring the write data from the host to the drive's cache memory and then reports back to the host that the write is "done," even though the data has not yet been written to the disk media (data not yet written out to disk is referred to as *dirty*). Thus, the service time for a cached write is about the same as that for a read cache hit, involving only some drive controller overhead and electronic data transfer time but no mechanical time. Clearly, write caching does not need to depend on having the right content in the cache memory for it to work, unlike read caching. Write caching will always work, i.e., a write command will always be a cache hit, as long as there is available space in the cache memory. When the cache becomes full, some or all of the dirty data are written out to the disk media to free up space. This process is commonly referred to as *destage*.

Ideally, destage should be done while the drive is idle so that it does not affect the servicing of read requests. However, this may not be always possible. The drive may be operating in a high-usage system with little idle time ever, or the writes often arrive in bursts which quickly fill up the limited memory space of the cache. When destage must take place while the drive is busy, such activity adds to the load of drive at that time, and a user will notice a longer response time for his requests. Instead of providing the full benefit of cache hits, write caching in this case merely delays the disk writes.

Zhu and Hu [2002] have suggested that large disk built-in caches will not significantly benefit the overall system performance because all modern operating systems already use large file system caches to cache reads and writes. As suggested by Przybylski [1990], the reference stream missing a first-level cache and being handled by a second-level cache tends to exhibit relatively low locality. In a real system, the reference stream to the disk system has missed the operating system's buffer cache, and the locality in the stream tends to be low. Thus, our simulation captures all of this activity. In our experiments, we investigate the disk cache, including the full effects of the operating system's file-system caching.

Ov.4.4 Experimental Results

Figure Ov.27 showed the execution of the GZIP benchmark with a moderate-sized FB-DIMM DRAM system: half a gigabyte of storage. At 512 MB, there is no page swapping for this application. When the storage size is cut in half to 256 MB, page swapping begins but does not affect the execution time significatly. When the storage size is cut to one-quarter of its original size (128 MB), the page swapping is significant enough to slow the application down by an order of magnitude. This represents the hard type of decision that a memory-systems designer would have to face: if one can reduce power dissipation by cutting the amount of storage and feel negligible impact on performance, then one has too much storage to begin with.

Figure Ov.31 shows the behavior of the system when storage is cut to 128 MB. Note that all aspects of system behavior have degraded; execution time is longer, *and* the system consumes more energy. Though the DRAM system's energy has decreased from 440 J to just under 410 J, the execution time has increased from 67 to 170 seconds, the total cache energy has increased from 275 to 450 J, the disk energy has increased from 540 to 1635 J, and the total energy has doubled from 1260 to 2515 J. This is the result of swapping activity—not enough to bring the system to its knees, but enough to be relatively painful.

We noticed that there exists in the disk subsystem the same sort of activity observed in a microprocessor's load/store queue: reads are often stalled waiting for writes to finish, despite the fact that the disk has a 4-MB read/write cache on board. The disk's cache is typically organized to prioritize prefetch activity over write activity because this tends to give the best performance results and because the write buffering is often disabled by the operating system. The solution to the write-stall problem in microprocessors has been to use write buffers; we therefore modified DiskSim to implement an ideal write buffer on the disk side that would not interfere with the disk cache. Figure Ov.32 indicates that the size of the cache seems to make little difference to the behavior of the system. The important thing is that a cache is present. Thus, we should not expect read performance to suddenly increase as a result of moving writes into a separate write buffer.

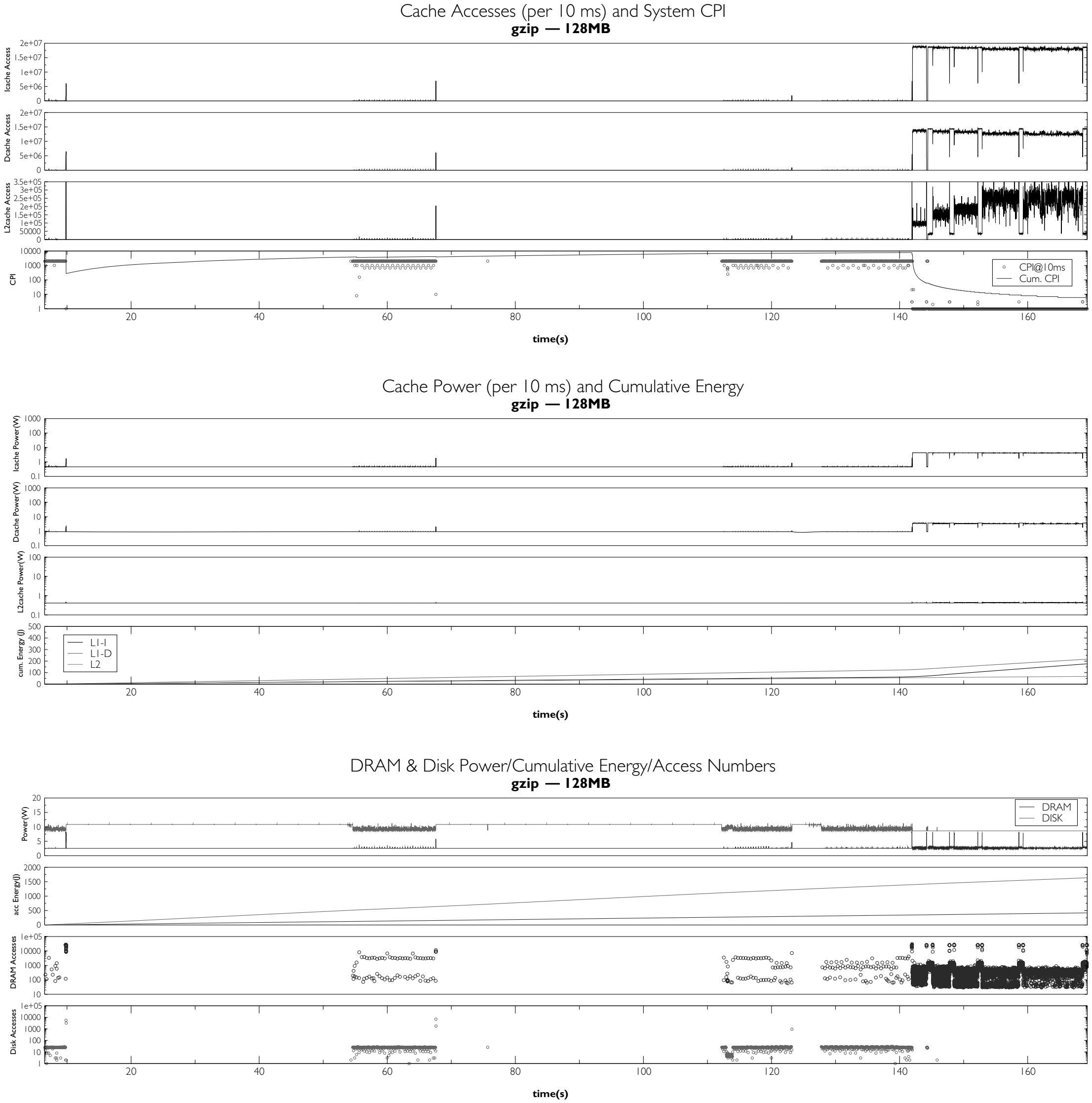


FIGURE Ov.31: Full execution of GZIP, 128 MB DRAM. The figure shows the entire run of GZIP. System configuration is a 2 GHz Pentium processor with 128 MB of FB-DIMM main memory and a 12 K-RPM disk drive with built-in disk cache. The figure shows the interaction between all components of the memory system, including the L1 instruction cache, the L1 data cache, the unified L2 cache, the DRAM system, and the disk drive. All graphs use the same *x*-axis, which represents the execution time in seconds. The *x*-axis does not start at zero; the measurements exclude system boot time, invocation of the shell, etc. Each data point represents aggregated (not sampled) activity within a 10-ms epoch. The CPI graph shows 2 system CPI values: one is the average CPI for each 10-ms epoch, the other is the cumulative average CPI. A duration with no CPI data point indicates that no instructions were executed due to I/O latency. The application is run in single-user mode, as is common for SPEC measurements; therefore, disk delay shows up as stall time. Note that the CPI, the DRAM accesses, and the Disk accesses are plotted on log scales.

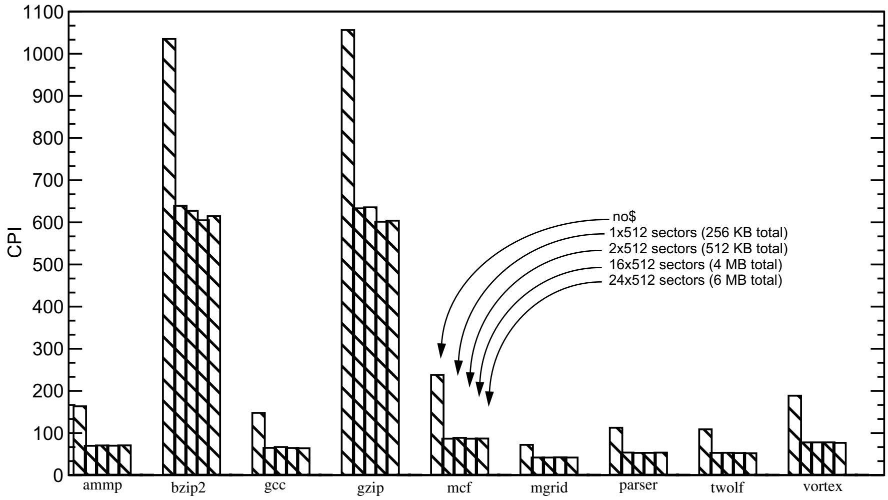


FIGURE Ov.32: The effects of disk cache size by varying the number of segments. The figure shows the effects of a different number of segments with the same segment size in the disk cache. The system configuration is 128 MB of DDR SDRAM with a 12k-RPM disk. There are five bars for each benchmark, which are (1) no cache, (2) 1 segment of 512 sectors each, (3) 2 segments of 512 sectors each, (4) 16 segment of 512 sectors each, and (5) 24 segment of 512 sectors each. Note that the CPI values are for the disk-intensive portion of application execution, not the CPU-intensive portion of application execution (which could otherwise blur distinctions).

Figure Ov.33 shows the behavior of the system with 128 MB and an ideal write buffer. As mentioned, the performance increase and energy decrease is due to the writes being buffered, allowing read requests to progress. Execution time is 75 seconds (compared to 67 seconds for a 512 MB system); and total energy is 1100 J (compared to 1260 J for a 512-MB system). For comparison, to show the effect of faster read and write throughput, Figure Ov.34 shows the behavior of the system with 128 MB and an 8-disk RAID-5 system. Execution time is 115 seconds, and energy consumption is 8.5 KJ. This achieves part of the performance effect as write buffering by improving write time, thereby freeing up read bandwidth sooner. However, the benefit comes at a significant cost in energy.

Table Ov.5 gives breakdowns for **gzip** in tabular form, and the graphs beneath the table give the breakdowns for **gzip**, **bzip2**, and **ammp** in graphical form and for a wider range of parameters (different disk RPMs). The applications all demonstrate the same trends: to cut down the energy of a 512-MB system by reducing the memory to 128 MB which causes both the performance and the energy to get worse. Performance degrades by a factor of 5–10; energy increases by 1.5× to 10×. Ideal write buffering can give the best of both worlds (performance of a large memory system and energy consumption of a small memory system), and its benefit is independent of the disk's RPM. Using a RAID system does not gain significant performance improvement, but it consumes energy proportionally to the number of disks. Note, however, that this is a uniprocessor model running in single-user mode, so RAID is not expected to shine.

Figure Ov.35 shows the effects of disk caching and prefetching on both single-disk and RAID systems. In RAID systems, disk caching has only marginal effects to both the CPI and the disk average response time. However, disk caching with prefetching has significant benefits. In a slow disk system (i.e., 5400 RPM), RAID has more tangible benefits over a non-RAID system. Nevertheless, the combination of using RAID, disk cache, and fast disks can improve the overall performance up to a factor of 10. For the

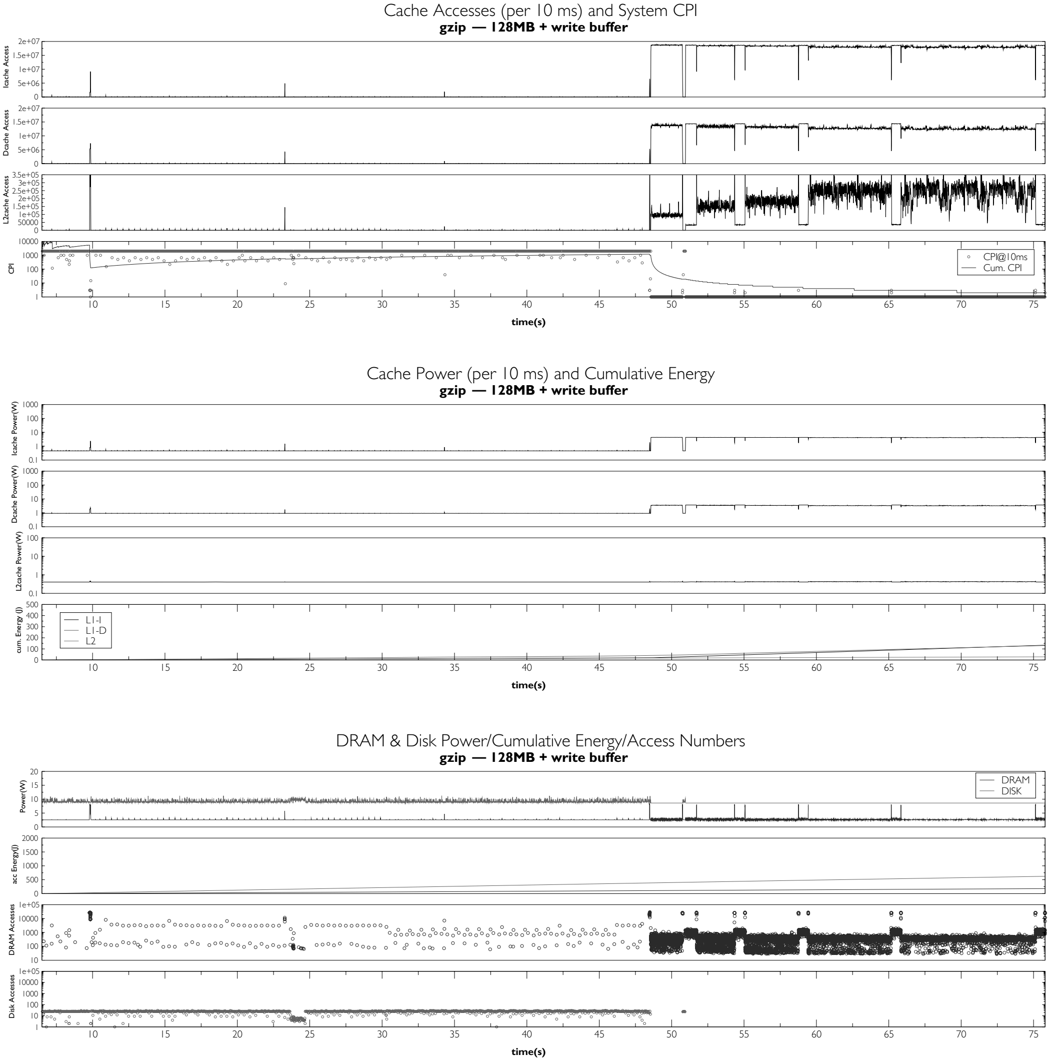


FIGURE Ov.33: Full execution of GZIP, 128 MB DRAM and ideal write buffer. The figure shows the entire run of GZIP. System configuration is a 2 GHz Pentium processor with 128 MB of FB-DIMM main memory and a 12 K-RPM disk drive with built-in disk cache. The figure shows the interaction between all components of the memory system, including the L1 instruction cache, the L1 data cache, the unified L2 cache, the DRAM system, and the disk drive. All graphs use the same *x*-axis, which represents the execution time in seconds. The *x*-axis does not start at zero; the measurements exclude system boot time, invocation of the shell, etc. Each data point represents aggregated (not sampled) activity within a 10-ms epoch. The CPI graph shows two system CPI values: one is the average CPI for each 10-ms epoch, the other is the cumulative average CPI. A duration with no CPI data point indicates that no instructions were executed due to I/O latency. The application is run in single-user mode, as is common for SPEC measurements; therefore, disk delay shows up as stall time. Note that the CPI, the DRAM accesses, and the Disk accesses are plotted on log scales.

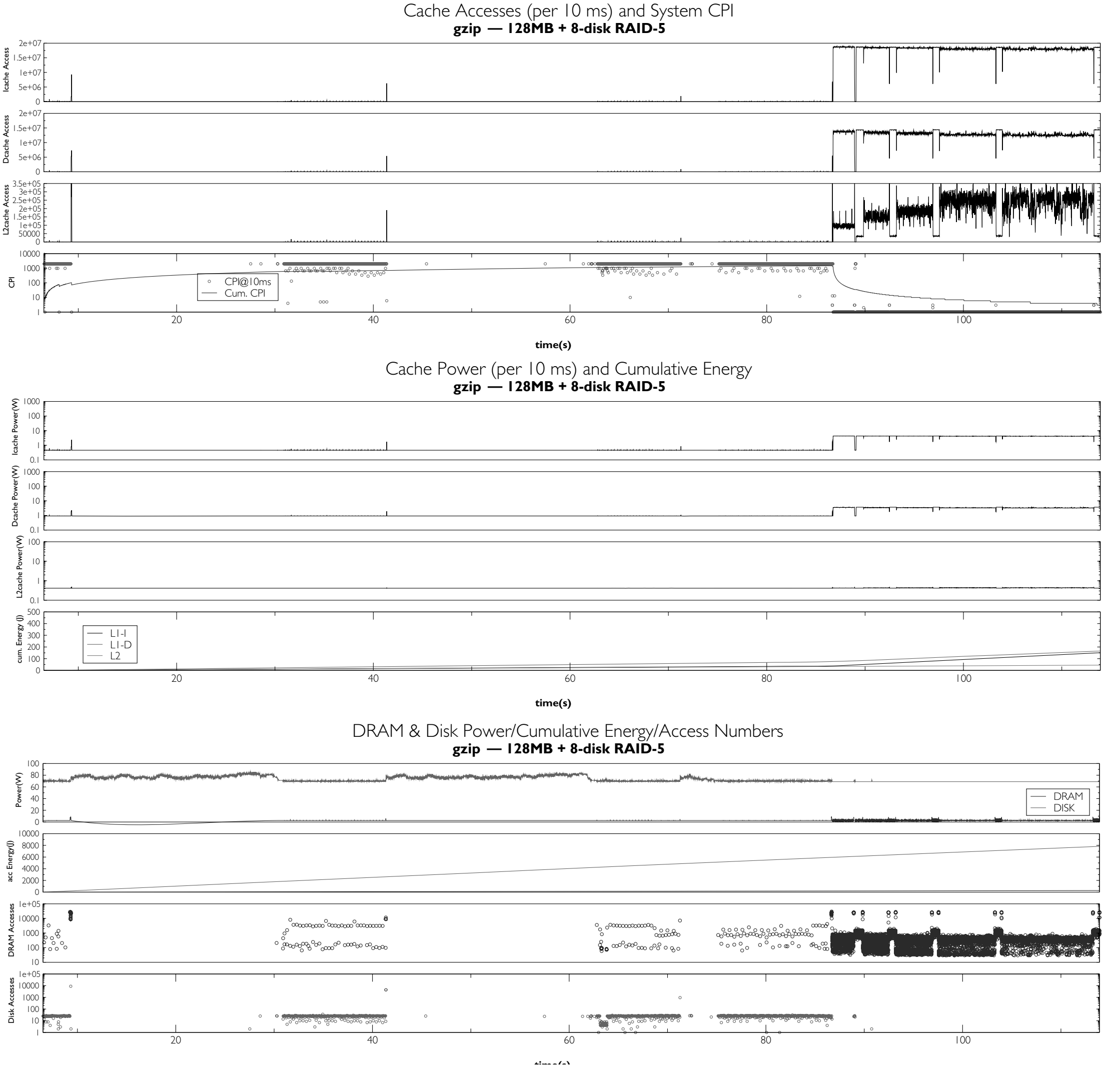
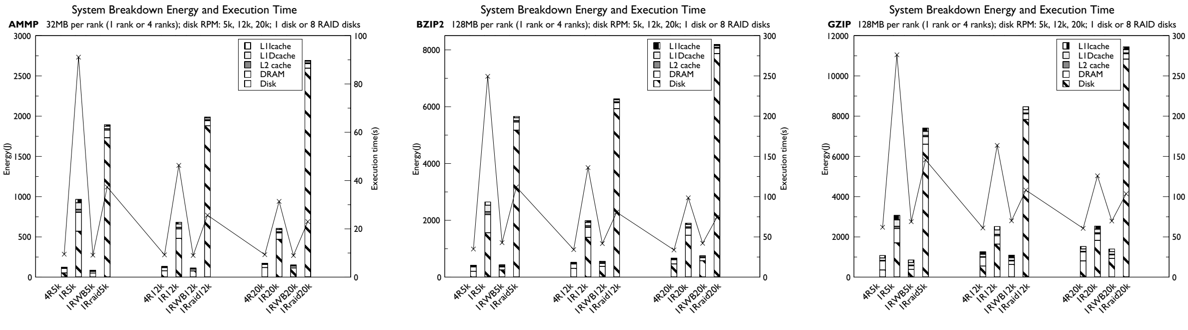


FIGURE Ov.34: Full execution of GZIP, 128 MB DRAM and RAID-5 disk system. The figure shows the entire run of GZIP. System configuration is a 2 GHz Pentium processor with 128 MB of FB-DIMM main memory and a RAID-5 system of eight 12-K-RPM disk drives with built-in disk cache. The figure shows the interaction between all components of the memory system, including the L1 instruction cache, the L1 data cache, the unified L2 cache, the DRAM system, and the disk drive. All graphs use the same *x*-axis, which represents the execution time in seconds. The *x*-axis does not start at zero; the measurements exclude system boot time, invocation of the shell, etc. Each data point represents aggregated (not sampled) activity within a 10-ms epoch. The CPI graph shows two system CPI values: one is the average CPI for each 10-ms epoch, the other is the cumulative average CPI. A duration with no CPI data point indicates that no instructions were executed due to I/O latency. The application is run in single-user mode, as is common for SPEC measurements; therefore, disk delay shows up as stall time. Note that the CPI, the DRAM accesses, and the Disk accesses are plotted on log scales.

Table Ov.5 Execution time and energy breakdowns for GZIP and BZIP2

System Configuration (DRAM Size - Disk RPM - Option)	Ex. Time (sec)	L1-I Energy (J)	L1-D Energy (J)	L2 Energy (J)	DRAM Energy (J)	Disk Energy (J)	Total Energy (J)
GZIP							
512 MB–12 K	66.8	129.4	122.1	25.4	440.8	544.1	1261.8
128 MB–12 K	169.3	176.5	216.4	67.7	419.6	1635.4	2515.6
128 MB–12 K–WB	75.8	133.4	130.2	28.7	179.9	622.5	1094.7
128 MB–12 K–RAID	113.9	151	165.5	44.8	277.8	7830	8469.1



average response time, even though the write response time in a RAID system is much higher than the write response time in a single-disk system, this trend does not translate directly into the overall performance. The write response time in a RAID system is higher due to parity calculations, especially the benchmarks with small writes. Despite the improvement in performance, care must be taken in applying RAID because RAID increases the energy proportionally to the number of the disks.

Perhaps the most interesting result in Figure Ov.35 is that the CPI values (top graph) track the disk's *average **read** response time* (bottom graph) and not the disk's *average response time* (which includes both reads and writes, also bottom graph). This observation holds true for both read-dominated applications and applications with significant write activity (as are **gzip** and **bzip2**). The reason this is interesting is that the disk community tends to report performance numbers in terms of average response time and not average *read* response time, presumably believing the former to be a better indicator of system-level performance than the latter. Our results suggest that the disk community would be better served by continuing to model the effects of write traffic (as it affects read latency) by reporting performance as the average *read* response time.

Ov.4.5 Conclusions

We find that the disk cache can be an effective tool for improving performance at the system level. There is a significant interplay between the DRAM system and the disk's ability to buffer writes and prefetch reads. An ideal write buffer homed within the disk has the potential to move write traffic out of the way and begin working on read requests far sooner, with the result that a system can be made to perform nearly as well as one with four times the amount of main memory, but with roughly half the energy consumption of the configuration with more main memory.

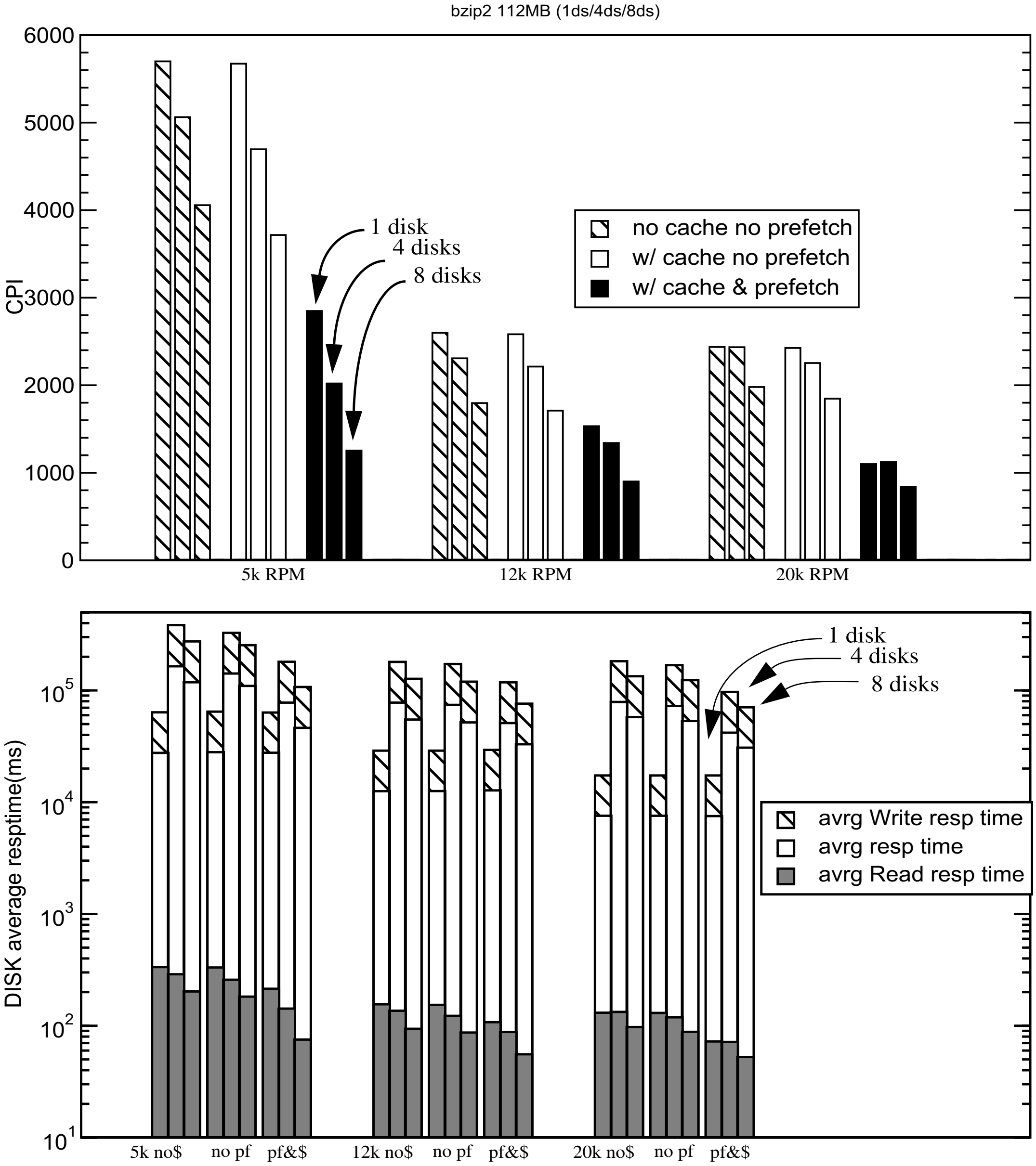


FIGURE Ov.35: The effects of disk prefetching. The experiment tries to identify the effects of prefetching and caching in the disk cache. The configuration is 112 MB of DDR SDRAM running bzip2. The three bars in each group represent a single-disk system, 4-disk RAID-5 system, and 8-disk RAID-5 system. The figure above shows the CPI of each configuration, and the figure below shows the average response time of the disk requests. Note that the CPI axis is in linear scale, but the disk average response time axis is in log scale. The height of the each bar in the average response time graph is the absolute value.

This is extremely important, because FB-DIMM systems are likely to have significant power-dissipation problems, and because of this they will run at the cutting edge of the storage-performance trade-off. Administrators will configure these systems to use the least amount of storage available to achieve the desired performance, and thus a simple reduction in FB-DIMM storage will result in an unacceptable hit to performance. We have shown that an ideal write buffer in the disk system will solve this problem, transparently to the operating system.

Ov.5 What to Expect

What are the more important architecture-level issues in store for these technologies? On what problems should a designer concentrate?

For caches and SRAMs in particular, power dissipation and reliability are primary issues. A rule of thumb is that SRAMs typically account for at least one-third of the power dissipated by microprocessors, and the reliability for SRAM is the worst of the three technologies.

For DRAMs, power dissipation is becoming an issue with the high I/O speeds expected of future systems. The FB-DIMM, the only proposed architecture seriously being considered for adoption that would solve the capacity-scaling problem facing DRAM systems, dissipates roughly two orders of magnitude more power than a traditional organization (due to an order of magnitude higher per DIMM power dissipation and the ability to put an order of magnitude more DIMMs into a system).

For disks, miniaturization and development of heuristics for control are the primary considerations, but a related issue is the reduction of power dissipation in the drive's electronics and mechanisms. Another point is that some time this year, the industry will be seeing the first generation of hybrid disk drives: those with flash memory to do write caching. Initially, hybrid drives will be available only for mobile applications. One reason for a hybrid drive is to be able to have a disk drive in spin-down mode longer (no need to spin up to do a write). This will save more power and make the battery of a laptop last longer.

For memory systems as a whole, a primary issue is optimization in the face of subsytems that have unanticipated interactions in their design parameters.

From this book, a reader should expect to learn the details of operation and tools of analysis that are necessary for understanding the intricacies and optimizing the behavior of modern memory systems. The designer should expect of the future a memory-system design space that will become increasingly difficult to analyze simply and in which alternative figures of merit (e.g., energy consumption, cost, reliability) will become increasingly important. Future designers of memory systems will have to perform design-space explorations that consider the effects of design parameters in all subsystems of the memory hierarchy, and they will have to consider multiple dimensions of design criteria (e.g., performance, energy consumption, cost, reliability, and real-time behavior).

In short, a holistic approach to design that considers the whole hierarchy is warranted, but this is very hard to do. Among other things, it requires in-depth understanding at all the levels of the hierarchy. It is our goal that this book will enable just such an approach.