Random-Access Data Storage Components in Customized Architectures

Lode Nachtergaele

Francky Catthoor

Chidamber Kulkarni
IMEC

This tutorial covers the basic design choices involved in customized data storage, including those for register files, local memory, caches, and main memory.

STORAGE TECHNOLOGY TAKES center stage in increasingly more systems because of the eternal push for more complex applications, especially those with larger and more complicated data types.[1] In addition, the access speed, size, and power consumption associated with storage form a severe bottleneck in these systems, especially in the embedded context. In this article, we will investigate several building blocks and overall organizations for memory storage, with a focus on customized memory architectures. The main emphasis will be on the generic storage components that are used in modern multimedia- and telecom-oriented processors, of both the digital-signal-processor (DSP) and application-specific types. We will not address the specific memory organizations (using instantiations of these components) that are part of complete system designs.

The content of this article is at the tutorial level; we intend it mainly as an introduction to other articles that explore customized memory organizations. For more details, see the references cited. An article by Kozyrakis et al.,[2] for example, discusses the impact of embedded DRAM technology on the global memory organization of more general-purpose programmable multiprocessors.

General principles and storage classification

The goal of a storage device is to store a number of *n*-bit data words for a short or long term. Under control of the address unit(s), these data words are transferred to the custom processing units (which we'll call processors, for simplicity) at the appropriate point in time (cycle), and the results of the operations are then written back in the storage device for future use. Because of the different characteristics of storage and access, different styles of devices have been developed.

Among memories for frequent and repetitive use, we can see a very important difference between short- and long-term storage. Short-term memories are normally located very close to the operators and require a very short access time so that they can be accessed in the same cycle as the arithmetic or logic operation applied to the resulting data. Consequently, they should be limited to a relatively small capacity (typically less than 128 words), and are usually taken up in feedback loops over the operators (for example, the loop RegfA-BufA-

0740-7475/01/$10.00 © 2001 IEEE

BusA-EXU-RegfA in Figure 1) or at the input of execution units. This kind of short-term memory is typically called foreground memory.

Devices for longer-term storage generally cater to far larger capacities (from 256 to 32 million 32-bit words) and take a separate cycle for read or write access. They are connected through a data bus (which can be partly off-chip) with the execution units.

We can make six other important distinctions using the tree-like "genealogy" of storage devices presented in Figure 2. The tree on the left focuses on the external usage behavior of the storage approach. It contains three layers.

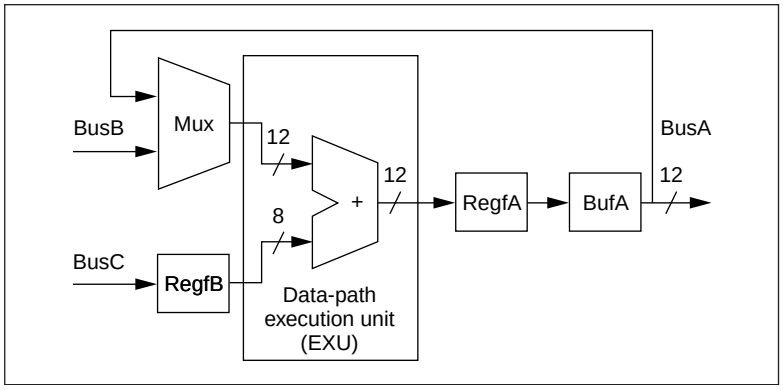


Figure1. Register file illustration in a feedback loop of a simple customized data path.

The first layer addresses read-only versus read/write (R/W) access. Some memories (ROMs, for example) are used only to store constant data. When the addresses are sparsely spread or there are multilevel logic circuits, especially when the amount of data is relatively small, good storage choices are devices such as programmable logic arrays (PLAs). In most cases, however, data must be overwritable at high speed, meaning read and write are treated with the same priority (R/W access), such as in random-access memories (RAMs). In some cases, ROMs can be made electrically alterable (EAROM) or erasable—"write-few"—or programmable (PROM) using means such as fuses. This article covers only R/W memories.

The next layer of Figure 2a concerns whether or not the memory is volatile. For R/W memories, the data is usually removed once the power goes down. In some cases, this problem can be avoided, but these nonvolatile options are expensive and slow. Examples are magnetic media and tapes, which are intended for slow access of mass data, or flash RAMs, which also allow higher speeds. We will restrict ourselves to what is common on most chips, namely volatile memory.

The last layer of Figure 2a concerns the address mechanism. Some devices require only sequential addressing, such as first-in, first-out (FIFO) queues or first-in, last-out (FILO) stack structures. Sequential addressing puts a tight restriction on the order in which the data are read out. However, for many applications, this restriction is acceptable. A more general, but still sequential access order is available in pointer-addressed memory (PAM). The main limitation

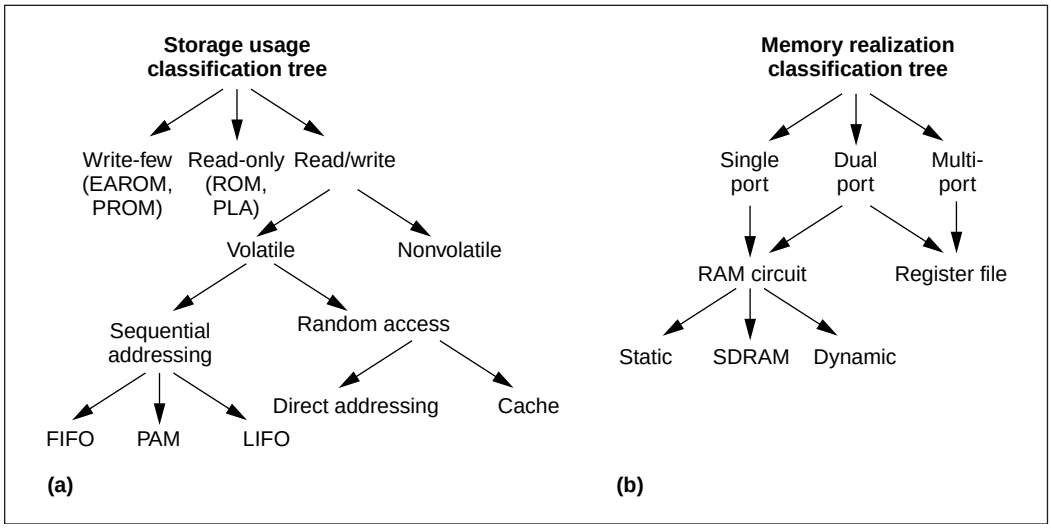


Figure 2. Storage classification trees: usage (a) and memory realization (b).

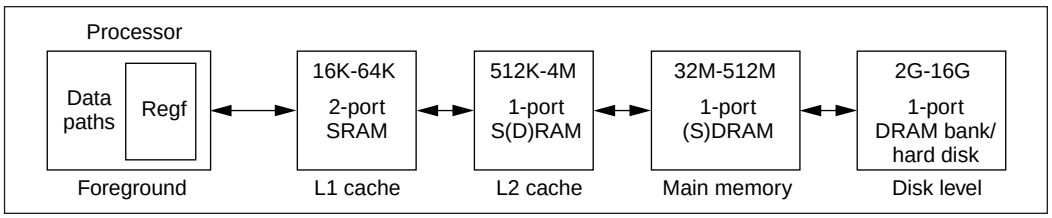


Figure 3. Hierarchical memory pipeline.

of PAM is that each data value is written and read once in any statically predefined order.

However, in most cases the address sequence should be allowed to be random (including repetition). Usually these devices are implemented with a direct-addressing scheme, typically called a RAM. An important requirement is that the access time be independent of the selected address. Many programmable processors use a special case of random-access-based buffering, exploiting comparisons of tags and usually also including associativity (in what is called a cache buffer).

The tree on the right in Figure 2 focuses on the way the memory is realized as a circuit or architecture component. It also contains three layers:

1. *Number of independent addresses and corresponding gateways (buses) for access.* This parameter can be one (single-port device), two (dual-port device), or even more (multiport device). Any of these ports can be for reading only, writing only, or R/W. Of course, the area occupied increases considerably with the number of ports.
2. *Internal organization.* The memory can be meant for capacities that remain small or that can become large. Here, designers usually must trade off between speed and area efficiency. The register files we describe in the next section constitute an example of fast, small-capacity organizations, which are usually also dual- or even multiported. The queues and stacks are meant for medium-sized capacities. The RAMs can become extremely large (up to 1 Gbit in the state-of-the-art devices), but they are also far slower in random access.
3. *Static or dynamic.* For R/W memories, the data can remain valid as long as V_{DD} is on (static cell in SRAM), or the data should be regularly refreshed (dynamic cell in DRAM). Within the dynamic class, high-throughput synchronous DRAMs (SDRAMs) have recently become available. For circuit-level issues, see overview articles such as Evans and Franzon[3] (for SRAMs), Itoh[4] and Prince[5] (for DRAMs).

Custom processors for data-dominated applications are driven from a customized memory organization. In programmable instruction set processors, this organization is usually constructed on a rigorous hierarchy almost like a bidirectional pipeline, from the disk (possibly with a disk cache to hide the long latency) over the main memory and the L2/L1 cache to the multiport register file (see Figure 3). Still, the pipeline becomes increasingly saturated and blocked because of the large latencies introduced compared to the CPU clock cycles in current process technologies.

Register files and local memory organization

Figure 4 shows an illustrative organization for a dual-port register file with two address buses, where the separate read and write addresses are generated from an address calculation unit (ACU). This organization uses two data buses (A and B), but only in one direction, so the write and read addresses directly control the port access. In general, the number of different address words can be smaller than the number of port(s) when they are shared (for example, for either read or write), and the buses can be bidirectional. Additional control signals decide whether to write or read and for which port the address applies. The number of address bits per word is $\log_2(N)$.

The register file of Figure 4 can be used very efficiently in the feedback loop of a data path, as already illustrated in Figure 1. In general, it is used only for the storage of temporary variables in the

application running on the data path (sometimes also referred to as the execution unit). Such register files are also used heavily in most modern general-purpose RISC chips and especially for modern multimedia-oriented signal processors, which have register files at up to 128 locations. (In this case, it becomes doubtful whether a register file is really a good option, because of the very high area and power penalty.)

Multimedia-oriented VLIW processors or recent superscalar processors give register files very large access bandwidth and many ports—for example, the 17-port iWarp register file and the 15-port TriMedia register file.[6] Application-specific instruction set processors (ASIPs) and custom processors make heavy use of register files for the same purpose. Although these register files have the clear advantage of very fast access, the number of data words to be stored should be minimized as much as possible because of their power- and area-intensive structure, because of both the decoder and the cell overhead.[7]

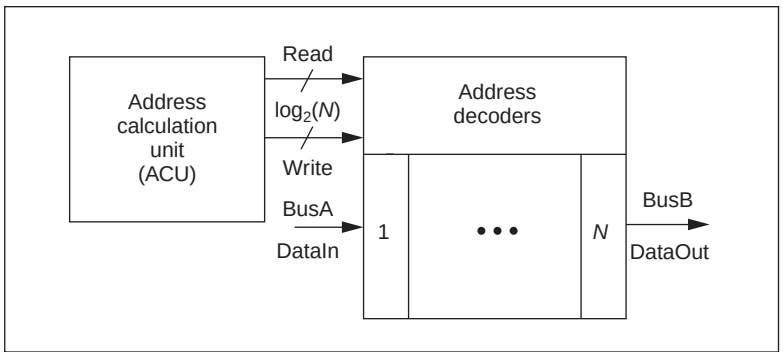


Figure 4. Two-port register file with a capacity of *N* words, with both read and write address supplied in parallel from an address calculation unit.

Cache memory organization

Nearly every modern programmable processor, especially those intended as general-purpose devices, uses caches.

Basic cache architecture

Caches are mainly intended to exploit the temporal and spatial locality of memory accesses.[8] Here, we summarize the different steps involved in cache operation. For our explanation, we'll use a direct-mapped cache as shown in Figure 5 (see also the "Cache architecture design choices" sidebar), but the basic principles remain the same for other types of cache memories.[8] The following steps occur whenever there is a read/write from or to a cache:

1. address decoding;
2. selection based on index, block offset, or both;
3. tag comparison; and
4. data transfer.

In Figure 5 and Figure 6 (page 45), these steps are highlighted in gray circles.

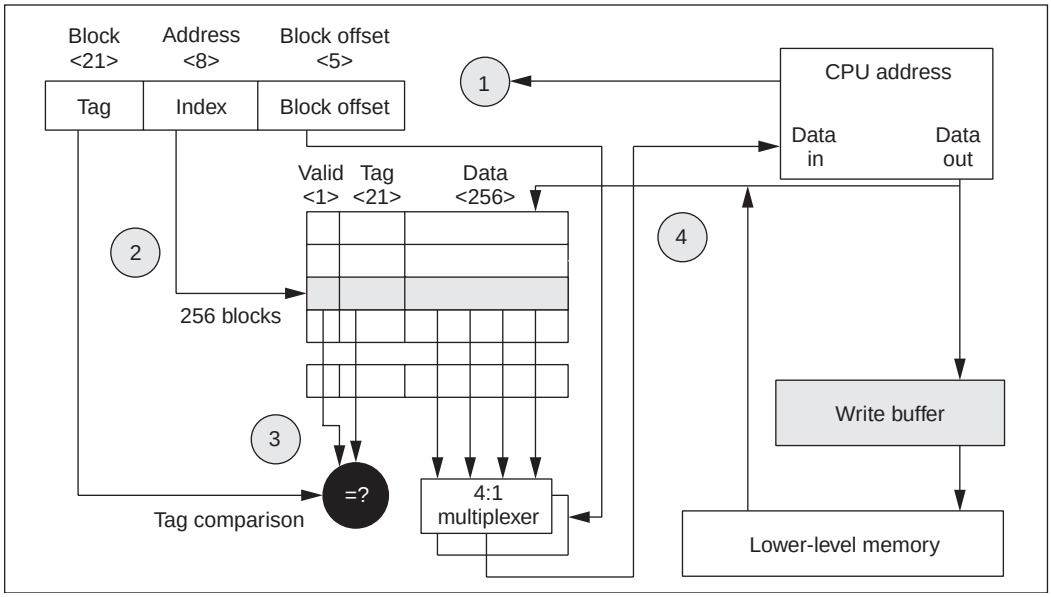


Figure 5. Eight-Kbyte, direct-mapped cache with 32-byte blocks.

Cache architecture design choices

Several important decisions are involved in the architecture design of a cache. Here, we discuss the choice of line size, degree of associativity, updating policy, and replacement policy.

Line size

The line size is the unit of data transfer between the cache and the main memory. In many publications, line size is also called block size. As the line size increases from very small to very large, the miss ratio initially decreases, because a miss fetches more data at a time. Further increases then cause the miss ratio to increase, as the probability of using the newly fetched (additional) information becomes less than the probability of reusing the data that is replaced, and also as fewer lines fit into the cache.

The optimal line size is completely algorithm dependent. In general, very large line sizes are not desirable, because they contribute to larger load latencies and increased cache pollution. This is true for both general-purpose and embedded applications.

Mapping and associativity

The process of transferring data elements from main memory to the cache is called mapping. Associativity refers to the process of retrieving several cache lines and then determining if any of them is the target. The degree of associativity and the type of mapping significantly affect cache performance. Most caches are set associative; an address maps to a set, and then an associative search is made of that set (see Figures 5 and 6 in the main text).

Empirically, and as one would expect, increasing the degree of associativity decreases the miss ratio. The highest miss ratios are observed for direct-mapped caches; two-way associativity is significantly better, and four-way is slightly better still.

Further increases in associativity only slowly decrease the misses. Nevertheless, a cache with a greater associativity requires more tag comparisons, and these comparators constitute a significant amount of total power consumption in the memories. Thus, for embedded applications, where power consumption is an important consideration, associativities greater than four or eight are not commonly observed.

Articles by Su and Despain[1] and Ko, Balsara, and Nanda[2] include some architectural techniques for low-power caches.

Updating policy

The process of maintaining coherency between two consecutive memory levels is termed the updating policy. There are two basic approaches to updating the main memory: write through and write back. With write through, all the writes are immediately transmitted to the main memory (apart from writing to the cache); with write back, most writes are written to the cache and are then copied back to the main memory as those lines are replaced.

Initially, write through was the preferred updating policy, because it is very easy to implement and solves the problems of data coherence. But it also generates lot of traffic between various levels of memories. Hence, most current state-of-the art media and DSP processors use a write-back policy. Lately, we have seen a trend toward giving control of the updating policy to the user (or compiler). This can be effectively exploited to reduce power, because of reduced write backs, by compile-time analysis.

Replacement policy

A cache's replacement policy refers to the type of protocol used to replace a partial or complete line in the cache on a cache miss. Typically, a least recently used (LRU) policy is preferable, because it has acceptable results.

Current state-of-the-art general-purpose and DSP processors have hardware-controlled replacement policies; hardware counters monitor the least recently used data in the cache. In general, we have observed that a policy based on compile-time analysis always has significantly better results than fixed statistical decisions.

References

1. C. Su and A. Despain, "Cache Design Trade-Offs for Power and Performance Optimization: A Case Study," *Proc. IEEE Int'l Symp. Low-Power Design,* IEEE CS Press, Los Alamitos, Calif., 1995, pp. 63-68.
2. U. Ko, P. Balsara, and A. Nanda, "Energy Optimization of Multi-Level Processor Cache Architectures," *Proc. IEEE Int'l Workshop on Low Power Design,* IEEE CS Press, Los Alamitos, Calif., 1995, pp. 45-50.

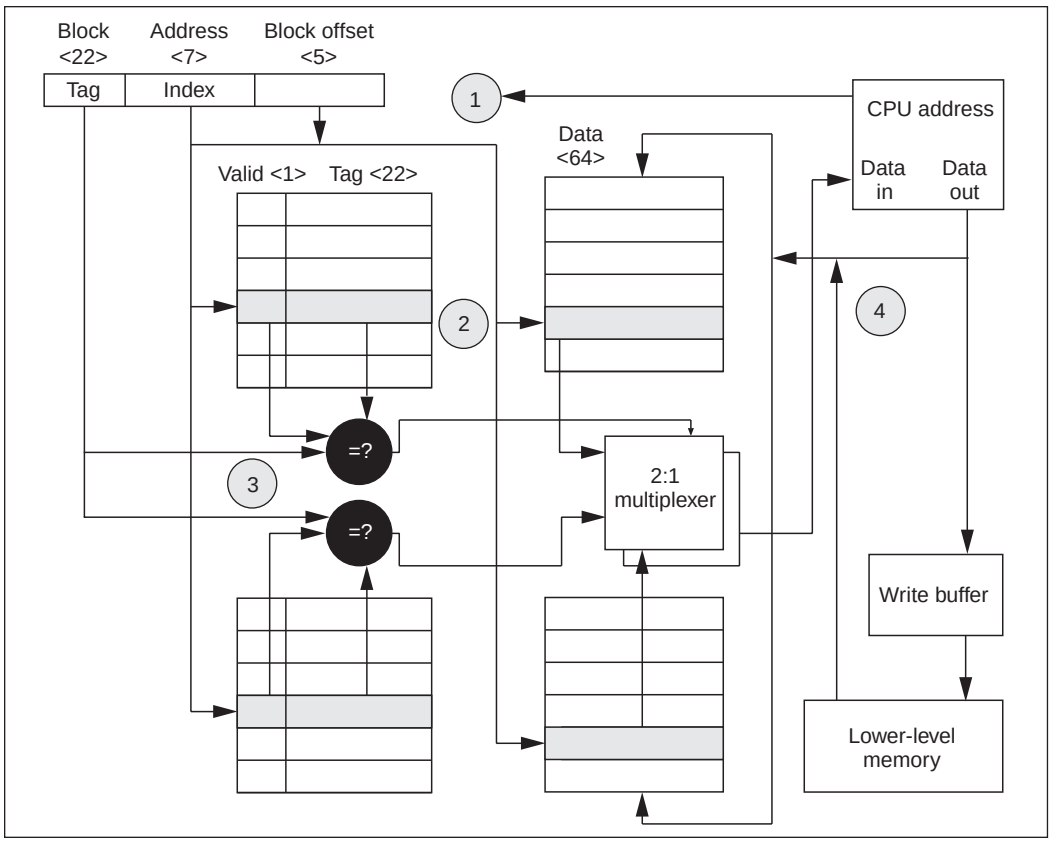


Figure 6. Eight-Kbyte, two-way associative cache with 32-byte blocks.

The first step performs the task of decoding the address, supplied by the CPU, into the block address and the block offset. The block address is further divided into the cache's tag and index addresses. Once the address decoding is complete, the index part of the block address obtains the particular cache line demanded by the CPU. This line is chosen only if the valid bit is set and the tag portion of the data cache and the block address match each other. This involves comparing all the individual bits. For a direct-mapped cache, we have only one tag comparison. After this process, the block offset obtains the particular data element in the chosen cache line. This data element is now transferred to or from the CPU.

Figure 5 shows a direct-mapped cache; an *n*-way set-associative cache would result in the following differences:

- Rather than just one, *n* tag comparisons are used.
- Fewer index bits and more tag bits are present. Figure 6, which shows a two-way set-associative cache, illustrates this circumstance.

In the sidebar, we briefly discuss some of the issues that must be considered during cache memory design.

Caches are prime candidates for a realization with embedded-memory technology. The L1 cache, which supplies data directly to the processor data paths, requires an access speed that is still too high to realize with DRAM components, so SRAM is preferred for that level. For the much larger L2 cache, where the access requirements are less stringent, eDRAM is a very good candidate technology, especially when based on the synchronous DRAM principles. (For a discussion of these issues in the context of programmable processors, see also Kozyrakis et al.[2])

Designers use three main types of architectural enhancement techniques to reduce compulsory misses, conflict misses, and write backs.

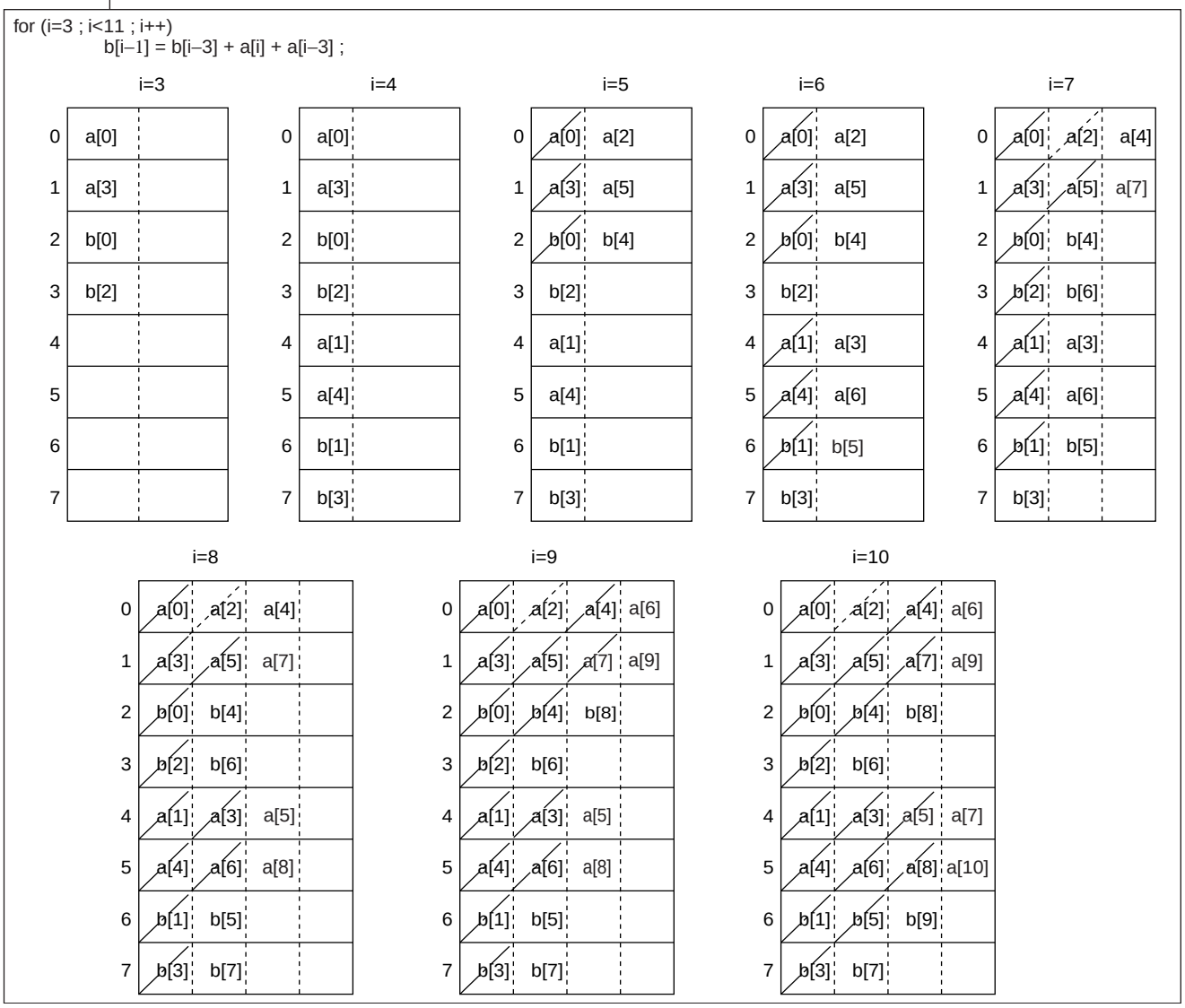


Figure 7. Initial algorithm and cache states corresponding to different iterations *i* of that algorithm for a fully associative cache. Diagonal bars represent cache misses. This example has 26 cache misses: 26/32, an 81% miss rate.

They are victim buffers (or similar buffers that store a limited number of words evicted from the cache at its different ports),[9] hardware or software prefetching,[10] and cache locking.[6] Cache locking used for reduced write backs and software prefetching need extensive compiler support (see "Data Memory Organization and Optimizations in Application-Specific Systems," by P. Panda et al., in this issue). Other researchers have written about interesting techniques related to cache and memory hierarchies in the context of low power and reduced memory bandwidth.[11]

Classification of cache misses

Whenever data requested by the CPU is not found in the cache, a cache miss has occurred. Essentially, three types of cache misses exist:

- *Compulsory cache misses.* The first access to the block is not in the cache, so the block must be brought into the cache.
- *Capacity cache misses.* If the cache cannot contain all the blocks needed during execution of a program (even when it is fully associative), capacity misses will occur

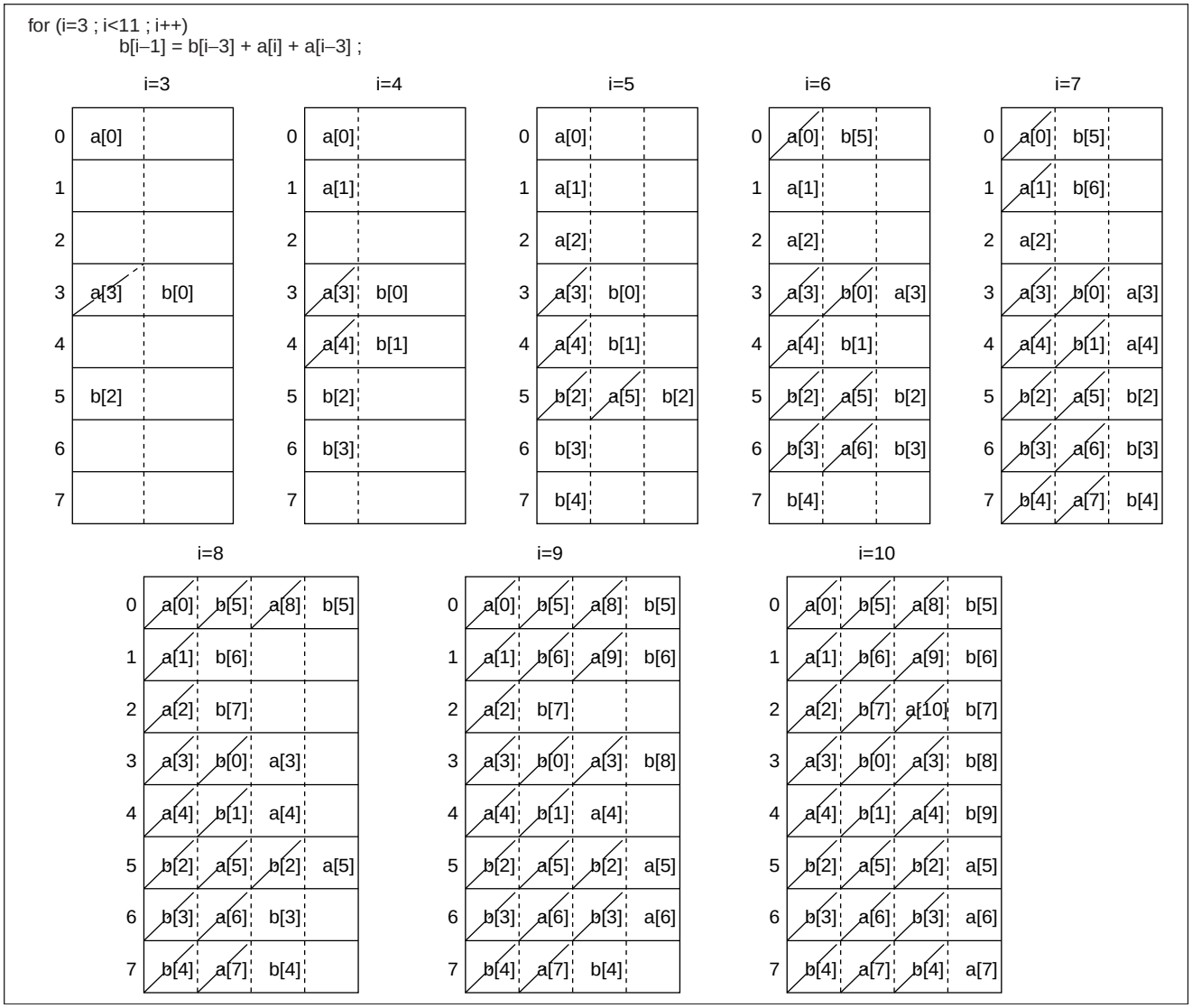


Figure 8. Initial algorithm and cache states corresponding to different iterations *i* of that algorithm for a direct-mapped cache. Diagonal bars represent cache misses. This example has a 100% miss rate: 32/32.

because of blocks being discarded and later retrieved. (An alternative definition for capacity misses is the total number of misses in a fully associative cache.)

- *Conflict misses.* If the block placement strategy is set-associative or direct-mapped, conflict misses occur (in addition to compulsory and capacity misses), because a block can be discarded and later retrieved if too many blocks map to its set. Lower associativity results in more conflicts. The difference between the total number of cache misses in a direct-mapped or set-associative cache and that of a fully associative cache is defined as the number of conflict misses.

In most real-life applications, capacity and conflict misses are dominant. Hence, reducing these misses is vital to achieving better performance and reducing power consumption.

Now let's look at an example that illustrates these three types of cache misses. Figure 7 and Figure 8 show the detailed cache states for a fully associative cache and a direct-mapped cache. A diagonal bar on an element indicates that the particular element is replaced by the next element—in the next column of the same row without a bar—because of the (hardware)

Table 1. Differences between hardware and software caches for current state-of-the-art multimedia and DSP processors.

Characteristic	Hardware control	Software control
Basic concepts	Lines and sets	Lines
Data transfer	Hardware	Partly software
Updating policy	Hardware	Software
Replacement policy	Hardware	Software

cache-mapping policy. Hence, every diagonal bar represents a cache miss. The main memory layout is assumed to be single and contiguous. That is, array $a[]$ resides in locations 0 to 10, and array $b[]$ in locations 11 to 21. For this illustration, we use a cache of size 8 bytes and a line size of 1 byte.

We observe from Figure 7 that the algorithm needs 32 data accesses. To complete these 32 data accesses, the fully associative cache requires 26 cache misses. Out of these 26 misses, 21 are compulsory and the remaining five are capacity misses. The direct-mapped cache requires 32 misses, as seen in Figure 8. This means that out of 32 data accesses, all of them result in misses, and the on-chip direct-mapped cache is not exploiting the available data reuse at all (for the given access order). Thus, for the algorithm in our example, we have 21 compulsory misses, five capacity misses, and six conflict misses.

Hardware- and software-controlled caches

Until recently, the embedded hardware cache controller steered most cache behavior—a reasonable setup if little can be analyzed about the behavior of the application at compile time. This is the case for programs that are heavily dependent on user interaction, such as office automation tools, databases, or debuggers. Over the years, however, the cache controller became more and more costly in terms of both area and power.

For many modern applications such as multimedia and telecommunications, cache behavior can be analyzed at compile time far more easily, even when irregular but manifest access patterns are present. Still, the code that is initially written by designers usually lacks locality (temporal, spatial, or both); in that case, the hardware controller no longer does a good job. However, recently developed, advanced compiler technology is simplifying analysis of such applications (see "Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors," by F. Catthoor et al., and "Data Memory Organization and Optimizations in Application-Specific Systems," by P. Panda et al., both in this issue). Therefore, the newer cache architectures make more software control available. An example is the software-lockable cache in the Philips TriMedia.[6]

Table 1 lists the major differences between hardware- and software-controlled caches for state-of-the-art multimedia and DSP processors. First, the hardware-controlled caches rely on the hardware to do the data cache management. Hence, to perform this task, the hardware uses basic concepts of cache lines and sets. On the other hand, for software-controlled caches, the compiler (or the user) performs cache management. The complexity of designing a software cache is far lower and requires only a basic concept of unit data transfer equivalent to a cache line.

Second, the hardware performs the data transfer based on the execution order of the algorithm at runtime, using fixed statistical measures. In contrast, for the software-controlled cache, either the compiler or the user performs this task. This is currently possible using high-level compile-time directives such as ALLOCATE() and link-time options such as LOCK to lock certain data in part of the cache, through the compiler or linker.[6]

Third, the most important difference between hardware- and software-controlled caches is in the way the next higher level of memory is updated—namely, the way the caches maintain coherence of data. For a hardware-controlled cache, the hardware writes data to the next higher level of memory either every time a write occurs or when the particular cache line is evicted. In contrast, for the software-controlled cache, the compiler decides when and whether to write back a particular data element.[6] This results in a large reduction in the amount of data transfers between different levels of memory hierarchy, contributing to lower power and reduced band-

width usage by the algorithm.

Finally, the hardware-controlled cache needs an extra bit for every cache line to determine the least recently used data, which will be replaced on a cache miss. For the software-controlled cache, the compiler manages the data transfer, so no need exists for additional bits or a particular replacement policy.

Main and global-hierarchical memory organization

As mentioned earlier, the access bottleneck is especially severe for main memories in the hierarchical pipeline (see Figure 3). This bottleneck is due to the large sizes of these main memories, and partly to their location (until now) off chip, which causes a highly capacitive on/off chip bus. Because of the use of embedded (S)DRAM technology, the latter problem can be solved for several new applications.

However, in many cases, this new solution will work at the L2 cache level but, because of the large size requirements, not yet for the main memory. Kozyrakis et al.[2] discuss this issue, indicating that modern multiprocessors still rely on a (large) central shared main memory (at least in terms of the address space). Even with eDRAM-based main memory solutions, the central shared memory would still impose heavy restrictions on the bandwidth; nor would eDRAM sufficiently solve the large energy consumption issues. So the customization benefits available with an on-chip eDRAM solution are not really suited for the main memory stage in the hierarchy. Hence, the bottlenecks still largely remain.

SRAMs are excluded for the large sizes required for main memories, because of their significantly lower density—more than an order of magnitude lower than (S)DRAMs.

The literature on storage includes proposals for a large variety of possible types of RAMs for use as main memories, and research on RAM technology is still very active, as demonstrated by interest in meetings such as the International Solid-State Circuits (ISSCC) and Custom Integrated Circuit (CICC) conferences. Several summary articles on this research are available.[1,5,12] The general organization depends on the number of ports, but usually single-port structures are present in high-density memories. This is also the restriction here.

Most other distinguishing characteristics relate to the circuit design (and the technological issues) of the RAM cell, the decoder, and the auxiliary R/W devices. One main conclusion of a literature study on the recent evolution of main memory is that the main emphasis—in almost all avenues of research—is on the access structure and not on the way the actual storage takes place in the "cell." This access structure, which includes the selection part and the input/output part, starts to dominate the overall cost in terms of power, speed, and throughput (assuming, of course, that leakage power in the memory matrix is kept under control). When power-sensitive memories are constructed, the area contribution also becomes more balanced between the heavily distributed memory matrices and the access structures.

The external data access bottleneck

In most cases, a processor requires one or more large external memories to store the long-term data (mostly of the DRAM type). For data-dominated applications in the past, total system power cost was for a large part due to the presence of these external memories on the board. Until recently, most DRAMs were of the pipelined page-mode type. Pipelined memory implementations improve a memory's throughput, but they don't improve the latency. For instance, in an extended data-out (EDO) RAM, the address decoding and the actual access to the storage matrix (of the previous memory access) occur in parallel. In that case, the access sequence to the DRAM is also important.

Figure 9 (next page) shows a DRAM's principal blocks. The address bus is divided into a column address and a row address, and these two parts are latched. This allows the device to pipeline the transfer of the address in two stages. To make this work, two control pins are necessary: The row address strobe (RAS) pin controls the row address latch, and the column address strobe (CAS) pin controls the column address latch.

To read data from DRAM, the following steps must take place (see also Figure 9):

1. The row address is placed on the address

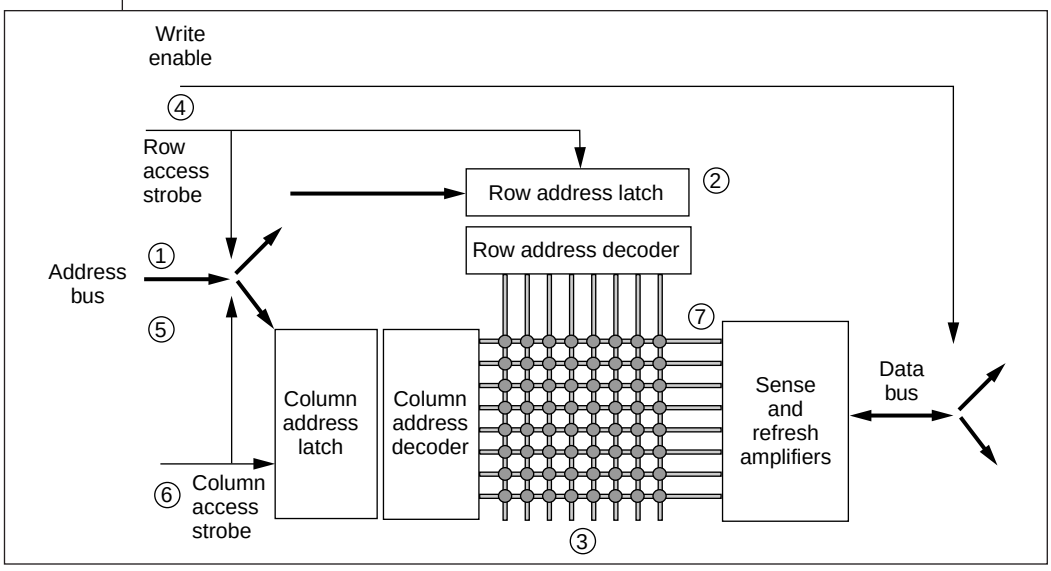


Figure 9. Illustration of DRAM operation.

pins via the address bus.
2. The RAS pin is activated, placing the row address onto the row address latch.
3. The row address decoder selects the proper row to be sent to the sense amplifiers.
4. The write-enable is deactivated so that the DRAM knows that it's not being written to.
5. The column address is placed on the address pins via the address bus.
6. The CAS pin is activated, placing the column address on the column address latch.
7. The CAS pin also serves as the output-enable, so once the CAS signal is stable, the sense amplifiers place the data from the addressed row and column on the data-output pins, and the data can be transferred to the processor.
8. The RAS and CAS are both deactivated so that the cycle can begin again.

Fast page-mode DRAMs. FPM DRAMs are currently the most used category of DRAMs; they are faster than previous-generation DRAMs because of their ability to work within a page (the section of memory available within a row address). Within one specific row are several columns of memory bits. With FPM DRAMs, you need only specify the row address once for accesses within the same page addresses. Successive accesses to the same page of memory require selecting only a column address, thus saving time in accessing the memory. The electronic standards agency, JEDEC, has specified standards for FPM DRAM.

Extended data-out DRAMs. EDO DRAMs work very similarly to FPM DRAMs, also having the ability to work within a page. The primary advantage of EDO over FPM is that EDO DRAMs hold the data valid even after the signal that strobes the column address goes inactive. Faster microprocessors can thus manage time more efficiently, performing many tasks without having to attend to slower memory. That is, while the EDO DRAM is retrieving an instruction for the microprocessor, the microprocessor can perform other tasks without worrying that the data will become invalid. JEDEC has also specified standards for the EDO DRAM.

Example. Consider the following C program:

```
const int N=100;
int A[N][N];
int r,c;
for (c=0; c < N; c++) {
  for (r=0; r < N; r++) {
       ... = foo(A[r][c]);
  }
}
```

The physical address of array A, assuming a row-wise organization, is baseaddress(A) + $100 \times r + c$. Hence, every time the inner iterator r increments, the DRAM address is increased by 100. This means the row address must be reapplied to the DRAM and latched into its suited latch; hence several cycles are lost.

This problem can easily be alleviated by a loop interchange:

```
const int N=100;
int A[N][N];
int r,c;
for (r=0; r < N; r++) {
  for (c=0; c < N; c++) {
       ... = foo(A[r][c]);
  }
}
```

This is very obvious in the preceding example, but in general it takes a systematic global approach to trade off all possibilities. "Data Memory Organization and Optimizations in Application-Specific Systems," by P. Panda et al. addresses this topic. The "EDO DRAM access" sidebar provides a more elaborate example.

Synchronous DRAMs. SDRAMs pipeline further and can provide a huge theoretical bandwidth. Basically, the internal state machine enables the enlargement of the pipeline. An SDRAM can sustain this high throughput rate by data-access interleaving over several banks. However, the address must be known several cycles before the data is actually needed. Otherwise the data path will stall, canceling the advantage of the pipelined memory.

Rambus DRAM. RDRAMs exploit a high-speed interface technology to provide transfer speeds of 1 Gbyte/s or more. RDRAMs are connected via a single 16-bit-wide data bus. Hence, each RDRAM chip can drive the 16-bit data bus by itself. This data bus is part of a larger shared Rambus channel, to which every RDRAM chip in the system is attached. This interconnectivity allows multiple memory controllers, theoretically providing multiple times the Rambus channel bandwidth. For example, Sony's PlayStation 2 uses two RDRAM channels, each with a single RDRAM, to achieve a total of 3.2 Gbytes/s memory bandwidth.

The negative side of this single-bus system is that the system latency can be as great as the latency to the RDRAM farthest away from the memory controller. An advantage that RDRAM has over SDRAM is that it has separate row and column control, allowing pipelining of the addressing with the data transfers. This reduces latency when data is needed from the same bank but in another row.

As with SDRAMs, bank conflicts are still possible. However the high bank count, typical for RDRAMs, reduces the probability of conflicts. For example, a RIMM of eight devices, 16 banks each, has 128 banks.

The cost to pay for RDRAM's high bandwidth is the high power consumption. Power management techniques keep RDRAM chip dissipation under control. Six states of activity are defined, ranging from a power-down mode to the highest attention state. The higher the activity state, the

EDO DRAM access example

Let's work out a brief example, directly based on the material in Coelho and Hawash.[1] The data sheet of an enhanced data-out (EDO) memory chip specifies a sequence such as {10-2-2-2}{3-2-2-2}, where the numbers represent clock cycles. Each curly bracket indicates four bus cycles of 64 bits each—that is, one cache line. The first sequence, {10-2-2-2}, specifies the timing if the page is first opened and accessed four times. The second sequence, {3-2-2-2}, specifies the timing if the page was already open and accessed four additional times—that means you did not already open and access any other memory page in the meantime. The last sequence repeats as long as you access memory within the same page.

The used data sheet relates to a memory bus running at 66 MHz. For another processing speed, such as a 233-MHz bus, the timing becomes {35-7-7-7}{11-7-7-7} in processor clocks. Table A provides the timing in the different parts of the overall memory organization.

Reference

1. R. Coelho and M. Hawash, *DirectX, RDX, RSX, and MMX Technology: A Jumpstart Guide to High Performance APIs,* Addison-Wesley, Reading, Mass., 1997.

Table A. Memory architecture and timing for a system using the Pentium II processor (at 233 MHz) and EDO memory.

Bus	Bus clock cycles	CPU clock cycles	Total clock cycles	Bandwidth (Mbytes/s)
L1 cache	{1-1-1-1}	{1-1-1-1}	4	1,864
L2 cache	{5-1-1-1}	{10-2-2-2}	16	466
EDO	{10-2-2-2}	{3-2-2-2}	56	133
memory	{35-7-7-7}	{11-7-7-7}	32	233
SDRAM	{11-1-1-1}	{2-1-1-1}	51	146
	{39-4-4-4}	{7-4-4-4}	19	392

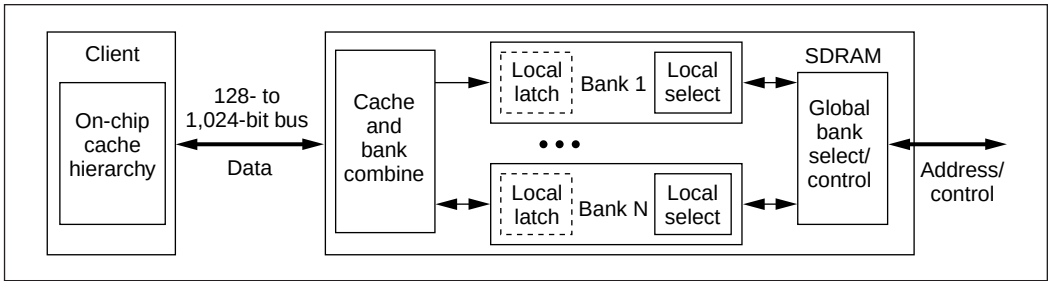


Figure 10. External data access bottleneck illustration with SDRAM.

smaller the latency for access but the greater the power consumption. For power consumption control, usually only a few chips can be active at the same time. Hence, when data is distributed across several RDRAMs, high latencies can be observed when accessing the data sequentially. This is because each chip has to alternate between a low-power mode and a high active state.

Data-memory power bottleneck

Because of the heavy push toward lower-power solutions to keep package costs low, and recently also for mobile applications or to address reliability issues, the power consumption of such external DRAMs has been reduced significantly. Apart from circuit and internal organization techniques for achieving this,[4,13] technology modifications such as switching to a SOI (silicon-on-insulator) approach have been considered. With all these techniques for distributing the power consumption from a few hot spots to all parts of the architecture, the end result is indeed a very optimized design for power, where every part of the memory organization consumes a similar amount.[4,14]

However, little more can be gained in this area, because the bag of tricks now contains only the more complex solutions with a smaller return on investment. Nevertheless, the combination of all these approaches indicates a very advanced circuit technology that still outperforms the current state of the art in data path and logic circuits for low-power design (at least in industry). Hence, the relative power in the non-storage parts can be reduced still more drastically, if this goal receives similar investments. Combined with the advance in process technology, all this has led to a remarkable reduction of DRAM-related power—from several watts for the 16- to 32-Mbit generation to about 100 mW for 100-MHz operation in a 256-Mbit DRAM.

Hence, modern stand-alone DRAM chips, which are often of this SDRAM type, also offer low-power solutions, but at a price: Internally, they contain banks and a small cache with a very wide width connected to the external high-speed bus (see Figure 10).[15] So the low-power operation per bit is only feasible when they operate in burst mode with entire, or parts of, memory columns transferred over the external bus. This is not directly compatible with the actual use of the data in the processor data paths; so without a buffer to the processors, most of the data words exchanged would be useless and discarded. Obviously, in this case the effective energy consumption per useful bit becomes very high, so the effective bandwidth is low.

Therefore, a hierarchical and typically far more power-hungry intermediate memory organization is needed to match the central DRAM to the data-ordering and bandwidth requirements of the processor data paths. The reduction of power consumption in fast random-access memories is not as advanced yet as in DRAMs, but this is also an area that is becoming saturated; many circuit- and technology-level tricks have already been applied in SRAMs. As a result, fast SRAMs continue to consume on the order of watts for high-speed operation around 500 MHz. Thus, the memory-related system power bottleneck remains a very critical issue for data-dominated applications.

FROM THE PROCESS technology perspective, the importance of the memory-related system power bottleneck is not surprising, especially for submicron technologies. The relative power cost of interconnections is increasing rapidly

compared with the transistor-related (active circuit) components. Clearly, local data paths and controllers themselves contribute little to this overall interconnect compared to the major data/instruction buses and the internal connections in the large memories. Hence, if all other parameters remain constant, the energy consumption, as well as the delay or area, in the storage and transfer organization will become even more dominant, especially for deep-submicron technologies. The remaining basic limitations lie in transporting the data and the control (such as addresses and internal signals) over large on-chip distances, and in storing them.

One last technological recourse for alleviating the energy-delay bottleneck is to embed the memories as much as possible on chip. This has been the focus of several recent activities, such as the Mitsubishi announcement of an SIMD processor with a large distributed DRAM in 1996[16] (followed by the offering of embedded DRAM technology by several other vendors) and the Intelligent RAM (IRAM) initiative of Dave Patterson's group at UC Berkeley.[17] The results show that the option of embedding logic on a DRAM process leads to a reduced power cost and an increased bandwidth between the central DRAM and the rest of the system. This is indeed true for applications where the increased processing cost is acceptable.[18] However, it is a one-time drop, after which the widening energy-delay gap between the storage and the logic will continue to progress, because of the unavoidable evolution of the relative interconnect contributions.

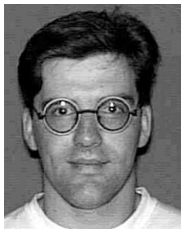
Thus, it is clear that for midterm (and even short-term) projects, the access and power bottlenecks should be broken by other, nontechnological means. This is feasible with quite spectacular effects through a more optimal design of the memory organization and system-level code transformations applied to the initial application specification. (See "Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors," by F. Catthoor et al., and "Data Memory Organization and Optimizations in Application-Specific Systems," by P. Panda et al.)

The price paid for these solutions will be increased system design complexity, which can be offset with appropriate design methodology support tools. This requires looking at platform architecture design and mapping methodologies from a new perspective. In addition, architectures will have to become more compiler friendly.[19] ■

References

1. G. Lawton, "Storage Technology Takes the Center Stage," *Computer,* vol. 32, no.11, Nov. 1999, pp.10-13.
2. C. Kozyrakis et al., "Scalable Processors in the Billion-Transistor Era: IRAM," *Computer,* vol. 30, no. 9, Sept. 1997, pp. 75-78.
3. R. Evans and P. Franzon, "Energy Consumption Modeling and Optimization for SRAMs," *IEEE J. Solid-State Circuits*, vol. SC-30, no. 5, May 1995, pp. 571-579.
4. K. Itoh, "Low Voltage Memory Design," in tutorial on "Low Voltage Technologies and Circuits," *Proc. IEEE Int'l Symp. Low-Power Design,* IEEE CS Press, Los Alamitos, Calif., 1997.
5. B. Prince, "Memory in the Fast Lane," *IEEE Spectrum,* Feb. 1994, pp. 38-41.
6. G. Slavenburg, S. Rathnam, and H. Dijkstra, "The Trimedia TM1 PCI VLIW Media Processor," *8th Hot Chips Symp.,* 1996.
7. N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design,* Addison-Wesley, Reading, Mass., 1993.
8. D. Patterson and J. Hennessey, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, San Francisco, 1996.
9. N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. Int'l Symp. Computer Architecture,* ACM Press, New York, 1990, pp. 364-373.
10. T.F. Chen and J.L. Baer, "Effective Hardware-Based Prefetching for High Performance Microprocessors," *IEEE Trans. Computers,* May 1995, pp. 609-623.
11. T. Johnson and W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. Int'l Symp. Computer Architecture,* ACM Press, New York, 1997, pp. 315-326.
12. R. Comerford and G. Watson, eds., "Memory Catches Up," *IEEE Spectrum,* Oct. 1992, pp. 34-57.

13. T. Yamagata et al., "Circuit Design Techniques for Low-Voltage Operating and/or Giga-Scale DRAMs," *Proc. IEEE Int'l Solid-State Circuits Conf.,* IEEE CS Press, Los Alamitos, Calif., 1995, pp. 248-249.
14. T. Seki et al., "A 6-ns 1-Mb CMOS SRAM with Latched Sense Amplifier," *IEEE J. Solid-State Circuits,* vol. SC-28, no. 4, Apr. 1993, pp. 478-483.
15. T. Kirihata et al., "A 220 mm^2, Four- and Eight-Bank, 256 Mb SDRAM with Single-Sided Stitched WL Architecture," *IEEE J. Solid-State Circuits,* vol. SC-33, Nov. 1998, pp. 1711-1719.
16. T. Tsuruda et al., "High-Speed, High Bandwidth Design Methodologies for On-Chip DRAM Core Multi-Media System LSIs," *Proc. IEEE Custom Integrated Circuits Conf.,* IEEE CS Press, Los Alamitos, Calif., 1996, pp. 265-268.
17. D.A. Patterson et al., "Intelligent RAM (IRAM): Chips that Remember and Compute," *Proc. IEEE Int'l Solid-State Circuits Conf.,* IEEE CS Press, Los Alamitos, Calif., 1997, pp. 224-225.
18. N. When and S. Hein, "Embedded DRAM Architectural Trade-Offs," *Proc. First Design and Test in Europe Conf.,* IEEE Press, Piscataway, N.J., 1998, pp. 704-708.
19. N. Mitchell, L. Carter, and J. Ferrante, "A Compiler Perspective on Architectural Evolutions," *IEEE TC on Computer Architecture Newsletter,* June 1997, pp. 7-9.



Lode Nachtergaele is a senior research engineer at IMEC, responsible for the technical coordination of IMEC's Multimedia Image Compression Systems (MICS) group. His research interests include distilling an operational system design methodology that improves design times of embedded multimedia systems, using the resulting design flow as a stepping stone to future application challenges. Nachtergaele has a BS in industrial engineering from the Katholieke Industriele Hogeschool, Oostende, Belgium.

Chidamber Kulkarni is a PhD student in DESICS at IMEC. His research interests include design methods and tools for cost-efficient embedded implementation of multimedia applications and architectural support for software systems. Kulkarni has a BE in electronics and communication engineering from Karnataka University, India, and an ME and PhD in electrical engineering from the Katholieke Universiteit, Leuven.

The biography of **Francky Catthoor** appears on page 4 in this issue.

■ Direct questions and comments about this article to Francky Catthoor, IMEC, Kapeldreef 75, B3001 Leuven, Belgium; catthoor@imec.be.