DAA Ames

11 P.

IN-16840

Is Random Access Memory Random?

Peter J. Denning

January 30, 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-86.1

Is Random Access Memory Random?

Peter J. Denning

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS Technical Report TR-86.1
January 30, 1986

Most software is constructed on the assumption that the programs and data are stored in random-access memory (RAM). Physical limitations on the relative speeds of processor and memory elements lead to a variety of memory organizations that match processor addressing rate with memory service rate. These include interleaved and cached memory. A very high fraction of a processor's address requests can be satisfied from the cache without reference to the main memory. The cache requests information from main memory in blocks that can be transferred at the full memory speed. Programmers who organize algorithms for locality can realize the highest performance from these computers.

Work reported herein was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) to the Universities Space Research Association (USRA).

This is a preprint of the column *The Science of Computing* for *American Scientist 74*, No 2, March-April, 1986.

Is Random Access Memory Random?

Peter J. Denning
Research Institute for Advanced Computer Science

January 30, 1986

The familiar model of a program consists of a processor following a sequence of instructions stored in random-access memory (RAM). The random-access memory is so named because its addressing hardware can read or write any memory location in a fixed amount of time, usually called the memory cycle time and denoted here by T. The total time to complete any sequence of r references to a RAM is rT. Because the length of the longest electrical path on a memory board or chip is proportional to the square root of the area in which the bits are stored, the cycle time has a lower bound that is proportional to the square root of the number of bits the RAM contains. Thus, the larger a RAM of given technology, the longer its cycle time.

In contrast to RAM is the SAM, or sequential-access memory. Access to a location consists of moving the requested information and the read-write heads into juxtaposition. This is accomplished by moving the storage medium itself, such as with tape or disk, or by electronically moving the information, such as in a delay line or bubble memory. A SAM runs fastest when information is accessed in the order of the storage locations that hold it. For example, a rotating disk with revolution time 8 milliseconds and 4000 bytes per circumferential track would take 8 milliseconds to access all those bytes in sequence and approximately 16 seconds to access them via a random permutation. RAM is generally 10^4 to 10^6 times faster for single-byte access and 10 times more expensive than SAM.

Programmers often use RAM and SAM models as guides to organizing algorithms for minimum running time. For example, a programmer who needs information from a tape or file will call for it by reading large segments sequentially into a buffer area of a RAM. A programmer who believes that instructions and data are held in RAM will pay no attention to the order in which the information is stored or accessed.

Computer architects, on the other hand, are called upon to design memory systems that are as cheap as SAM but as fast as RAM. This challenge results in a memory hierarchy consisting of at least three levels: cache, main store, and secondary store. Level 1, the cache, is the smallest, fastest, and most expensive. Level 3, the secondary store is the largest, slowest, and least expensive. Level 2, the main store, is intermediate in size, speed, and cost. The average access time for the hierarchy is a sum of the form $\sum p_i a_i$, where p_i is the probability that the requested information is in Level i and a_i is the access time given that it is in Level i. The hardware and operating system attempt to move code and data around automatically in this hierarchy so that the average access time is as small as possible. A common approach is to move a copy of an information-item into the cache when it is requested but not present there, and move it gradually down the hierarchy as other items move up. The result is that the most recently used items will often be found in the cache.

To isolate programs from the actual technology and capacity of each level, it is common to hide the memory hierarchy behind a RAM interface. Thus, the programmer sees only an address register, a data register, a read-initiate signal line, and a write-initiate signal line. The size of an address is sufficient to span a space as large as the main store and sometimes as large as the secondary store. (In the latter case, the address space is called virtual memory.) Although used like RAMs, memory hierarchies do not behave like RAMs; their access times are better for some reference patterns than for others. What model is useful for helping programmers organize algorithms for best performance in such systems? The rest of this column answers this question for the cache and main store part of the hierarchy.

Let us consider first a simple system without cache; the highest level of the hierarchy is the main store. The main store usually consists of a bank of M identical RAMs, rather than one large RAM of the required capacity. The individual RAMs are called modules and are numbered 0, 1, ... $M-1$; M is usually a power of 2. The primary reason for modularity is to match the memory service rate to the aggregate addressing rate of processors: M modules in parallel can potentially run M times faster than one module. A secondary reaso for modularity is that the system can be reconfigured to operate with a smaller number of modules after a memory failure.

The hardware maps addresses into modules by interleaving or by partitioning. Under interleaving, consecutive addresses are assigned cyclically among the modules; addresses $0, M, 2M, \cdots$ go in module 0, addresses $1, M+1, 2M+1, \cdots$ go in module 1, and so on through module $M-1$. The starting times of the modules are staggered so that one address request can be accepted every T/M seconds if the address are consecutive. Under partitioning, groups of consecutive addresses are assigned to separate modules: if C is the capacity of a module, addresses $0, \ldots, C-1$ go in module 0, addresses $C, \ldots, 2C-1$ go in module 1, and so on through module $M-1$. Interleaving is usually more efficient because it spreads the addressing load among the modules. For example, a processor that generates a run of r consecutive addresses would experience a delay of approximately rT in the partitioned memory but only rT/M in the interleaved memory. Processors executing shared code are much more likely to queue up on the same module of a partitioned memory than they are on the same module of an interleaved memory.

A memory controller mediates between the processors and modules. It transmits read-write requests to modules and returns their responses to the requesting processors. It arbitrates between near-simultaneous requests for the same module. Two configurations are common, the crossbar and the buss. A crossbar is a multi-path switch capable of connecting any processor directly to any module; it gets its name by analogy with telephone switches. A buss is a single unit that is seized by one processor at a time long enough to exchange a message between a processor and a module; it gets its name by analogy with power-conducting strips in electronic equipment. The most sophisticated busses do not hold processor-memory connections continuously while the memory is serving a processor's request; instead, they pass a single request from a processor to a module, or a single response from a module to a processor. When a buss's holding time is small compared to T/M, the buss behaves approximately the same as a crossbar. For N processors and M memories, the hardware complexity of a crossbar is proportional to the product NM, whereas hardware complexity of a buss is proportional to the sum $N+M$. Neither unit scales well: for large N and M, the number of components of the crossbar and the waiting time for access to the buss become too large.

How busy can the N processors keep the M modules? This is a difficult question because of possible interference among processors seeking access to the same module. Individual programs tend to generate many runs of consecutive addresses. Thus, if $N=1$, it would be possible to match processor and memory throughput simply by choosing T/M somewhat shorter than the processor's interaddress delay. When $N>1$, however, the address patterns of the processors will be randomly intermixed; the effect on the memory will be very close to that of random address patterns.

In a study of this case *(1)*, Forest Baskett and Alan Smith showed that the fraction of modules kept busy per memory cycle is approximately

$$U = 1 + \frac{N}{M} - \sqrt{1+(\frac{N}{M})^2} .$$

When there are as many processors as modules ($N = M$), the utilization (U) is approximately 60%; when there are twice as many processors ($N = 2M$), the utilization is approximately 76%. Memory service rate will be balanced with processor addressing rate when the parameters are chosen so that a processor's interaddress delay is TN / UM.

These utilizations are not bad, but with cache memory they can be much better. A cache is a small, very fast RAM placed between a processor and the main store. In all, there are N caches, one for each processor. One of the earliest reports of this idea was published in 1965 by Maurice Wilkes of the University of Cambridge *(2)*. A recent comprehensive survey has been given by Alan Smith *(3)*.

The cache is organized as a set of frames all of the same size; the main store is divided into blocks of the same size as the frames. The block is the unit of information transfer between cache and main store. Typical block sizes range from 4 to 128 bytes, with 32 being common. The cache hardware keeps track of which block is stored in each frame.

A processor first presents a memory request to its cache; if a cache frame contains the block in which the requested address is located, the request can be satisfied without reference to the main store. Otherwise, the processor's request is deferred until the cache can transfer a copy of the missing block into an unused frame. Because each cache request is a full block transfer, the address trace processed by the main store will be highly sequential. This means that higher utilizations can be obtained than are predicted by the random-reference model described earlier. For example, a B-byte block with M-module interleaving will require about $(M-1)/2$ module cycles of length T/M for access to the first byte and B additional module cycles to transfer the bytes; this is a utilization of $U = 2B/(2B+M-1)$. For 32-byte blocks in an 8-module memory, utilization is 90%. Caches allow high module utilization independent of the number of processors.

There are, however, advantages to cache memory of even greater significance. Some numbers will clarify them. Typical main memories range from 1000 to 64000 Kbytes; typical caches range from 4 to 32 Kbytes. A typical access time for main memory is 500 nanoseconds; for cache it is 50 nanoseconds. Miss ratios, the proportion of requests that cannot be satisfied from the cache, are generally below 5% for caches on the order of 20 Kbytes. (In specific cases, miss ratios may be well under 1%.) This means that a typical average access time is less than $0.95\times50+0.05\times500 = 72.5$ nanoseconds. The cost of the cache is a few percent of the total cost of the memory, and yet the average access time in the memory system is very close to that of the cache!

This line of argument leads to the following model of the active (computational) memory of a computer. Hiding behind a RAM addressing interface are combinations of interleaved modules and caches. The cache size is easily chosen so that at least 95% of the time a processor quickly finds the information it needs in the cache. When cache-memory transfers do occur, they are full blocks, and can be obtained at high utilization from interleaved memory.

The contents of the cache, sometimes called the working set of the program using the processor, will be a copy of the most recently used blocks of the program's instruction and data. To minimize miss ratio, the programmer must organize instructions and data to minimize sudden changes in the working set – jumps to groups of blocks not overlapping the current region of execution. This idea is called "programming for locality." It is very simple to apply and yet powerful enough to help programmers realize the capacity of their computers.

In a later column, I will discuss the more general form of memory hierarchy called virtual memory. The principle of locality figures prominently in that discussion.

References:

1. F. Baskett and A. J. Smith. 1976. "Interference in multiprocessor computer systems with interleaved memory." ACM *Communications.* June.

2. M. V. Wilkes. 1965. "Slave memories and dynamic storage allocation." IEEE *Transactions on Computers.* April.

4. A. J. Smith. 1982. "Cache memories." ACM *Computing Surveys*, September.