Memory*

*From Chapter 3 of High Performance Computing

Memory

Even if CPU is infinitely fast, still need to read/write data to memory.

Speed of memory increasing much slower than processor speed.

As computers become faster, people want to handle problems of ever increasing size.

Requires a large, yet fast memory system. Hard to get, and won't be cheap!

Memory Cont.

Possible approaches:

- The entire memory can be made fast to handle every memory access.
- Slow memory can be accessed using a “round-robin” approach to give appearance of faster system.
- Make memory system wide, so each transfer provides many bytes of memory.
- Divide memory system into faster and slower sections. Ensure that the faster section used more often.

Cost dominating factor. A cheap system that performs well most of the time will be preferred to an extremely fast, but extremely expensive system.

A combination of the above are usually used.

Memory Technology

Two types of memory used: *dynamic random access memory* (DRAM), and *static random access memory* (SRAM).

- With DRAMs, each bit is an electrical charge stored in a tiny capacitor. The charge leaks away over time, so must be refreshed.

  Reading a bit causes the charge to be lost, and thus must be refreshed. Can't access while being refreshed.

  DRAM has higher number of memory cells per chip, and thus cheaper, uses less space and power, and produces less heat.

- SRAM is created using logic gates. A bit is stored using 4-6 transistors.

SRAM is faster.

In addition to technology, performance is limited by wiring layout and the external pins for the chips.

Access Time

Memory access time: The time required to read or write a location in memory.

Memory cycle time: States how often you can repeat a memory reference.

Clock period of home computers went from 210ns for IBM XT, to 3ns for a 300 MHz Pentium II.

Access time for commodity DRAM went from 200ns to 50ns in same time.

Processor performance was doubling every 18 months, but memory every 7 years!

Memory Hierarchy

Cray supercomputers created main memory out of SRAM, but required liquid cooling. Can't manufacture inexpensive systems like this.

To solve the problem, a memory hierarchy is used.

Hierarchy consists of CPU registers. Next comes 1-3 levels of high-speed SRAM cache. Then DRAM for main memory, followed by virtual memory typically stored on disk drives.

Which level of hierarchy we access greatly affects speed of program.

Programmer must carefully manage access to hierarchy.

Registers

CPU registers operate at same speed of processor thus at top of hierarchy.

When doing calculations, goal to keep operands and temporary values in registers where possible.

Handled by optimizing compiler.

Only practical to add so many registers to a processor. Most problems won't fit, requiring accessing lower levels of hierarchy.

Caches

Caches are the next level of hierarchy.

Consist of small amounts of SRAM. Stores a subset of the memory.

Hopefully, the cache will contain the subset that you need, when you need it.

Usually have 1-2 on-chip caches, and perhaps one off-chip.

Access time for cache include moving data between layers of hierarchy, and cost to keep cache consistent with main memory.

When data always in cache, we say we have a 100% hit rate. Want 90% for level 1 cache, and better than 50% for L2 cache.

Can characterize the performance of memory hierarchy by considering the probability that a memory reference would be satisfied by a particular level of a hierarchy.

Cache Lines

Important to keep track of which areas of main memory are stored in the cache at the moment.

To make this easier, and to reduce the space required to keep track of this, the cache is partitioned into several slots of the same size called *lines.*

Every line contains an equal number of sequential memory locations from main memory. Often 4-16 integers/real numbers.

Cache Lines Cont.

Data from a given line come from the same location, other lines may have data from a different part of memory. See Figure 3-1.

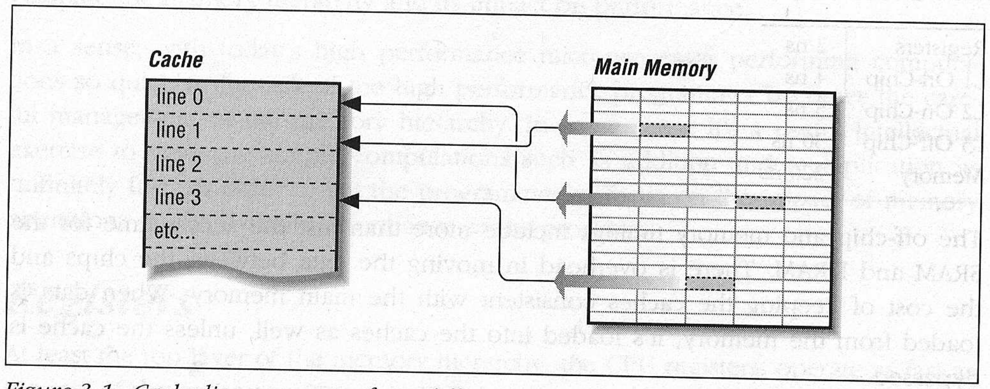

Figure 3-1: Cache lines can come from different parts of memory *

When data requested, computer checks if a cache line contains it. If not, a new line is retrieved from main memory.

This means another is deleted. Hopefully not the next one you'll need!

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Cache Consistency

If new data is stored in cache, there must be a means to also modify the corresponding location in main memory.

Two main types of cache:

Writeback cache: Data written to the cache stays there until the cache line is replaced. The data is then written to main memory.

Performs well when CPU is writing to successive locations in a cache line.

Write-through cache: Date is written to main memory and cache right away. Takes up a lot of main memory bandwidth. A problem with multiple processors.

When a line is replaced, a writeback is not needed.

Cache Coherency

What if writeback caches were used on a multiprocessor system?

In a multiprocessor system, we have two choices:

- Data must be written back to main memory so it will be visible to other processors.

- All processors must be made aware of cache activity on all other processors.

  They need to be told to invalidate cache lines containing the modified data.

  This is called maintaining *cache coherency.*

  Can cause significant traffic with many processors.

Cache Effectiveness

Caches perform well because programs exhibit the characteristics called: *spatial* and temporal locality of reference.

In other words, programs usually access instructions and data that are near to other instructions and data in space and time.

When a cache miss occurs, the new line contains not only the required data, but neighboring data.

It's likely that the next bit of data needed is in this line or one loaded recently.

Caches work best when memory is accessed sequentially.

Effect of a subroutine call?

Unit Stride

Say we have a program reading 32 bit integers. Assume a cache line size of 256 bits.

When program reads 1st word in line, it waits for line to be loaded. But, next 7 integers will be in that cache line as well.

Called *unit stride* as address of each element separated by one, and all data in line used.

For example:

```
  DO I  = 1, 1000000
     SUM = SUM + A(I)
  ENDDO
```

Unit Stride Cont.

In FORTRAN, a 2-dimensional array is stored as follows: The first column stored sequentially, followed by the second and so on.

A unit stride memory reference pattern:

```
REAL*4 A(200,200)
DO J = 1, 200
  DO I  = 1, 200
   SUM = SUM + A(I,J)
  ENDDO
ENDDO
```

However, if we swapped the loops we'd get a non-unit stride pattern.

```
REAL*4 A(200,200)
DO I = 1, 200
  DO J  = 1, 200
   SUM = SUM + A(I,J)
  ENDDO
ENDDO
```

Cache Organization

Pairing cache lines with memory locations is referred to as *mapping.*

Cache is smaller than memory means multiple memory locations must map to same cache line.

Each line stores info about the memory location it holds (referred to as the *tag*), and sometimes when it was last used.

How a location is mapped to a cache line can greatly alter program speed.

Three methods:

- Direct mapped
- Fully associative
- Set associative.

Direct-Mapped Cache

Simplest approach. See Figure 3-2.

Uses fixed pattern. If processor has 4K cache, then memory location 0, 4K, 8K, 12K, etc all map onto cache line 0.

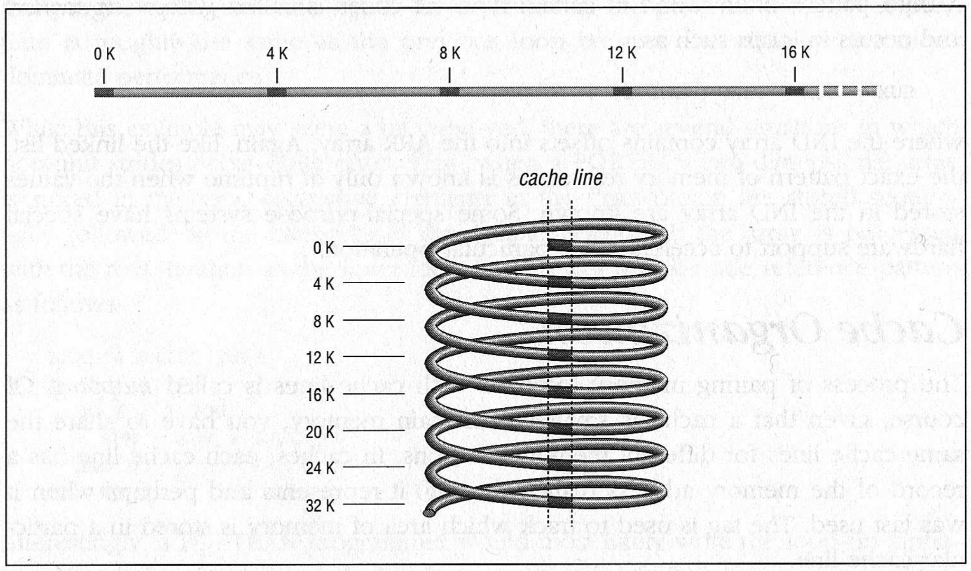

Figure 3-2: Many memory addresses map to the same cache line *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Direct-Mapped Cache Cont.

Problem when alternating memory accesses all resolve to same cache line.

Each causes cache miss, and replaces the previous memory location. Called *thrashing.*

Makes cache a liability.

Pathological case:

```
REAL*4 A(1024), B(1024)
  DO I  = 1, 1024
   A(I) = A(I) * B(I)
  ENDDO
```

To fix, adjust size of arrays (avoid powers of 2!), or insert other variables in between them.

Fully Associative Cache

Most complex algorithm. Any location can be mapped to any cache line.

Name comes from memory type used: *associative memory.* Contains not only data, but information about the data such as where it comes from.

Processor asks all cache lines for memory reference at same time. If one has it, it signals processor. Else, cache miss.

Different algorithms used to decide which line to replace.

Common method is to replace least recent accessed line.

Set-Associative Cache

In between the two. A two-way set-associative cache has two direct-mapped caches side by side. See Figure 3-3.

Each location corresponds to a cache line in each.

Can choose which line to replace. Usually least recently accessed.

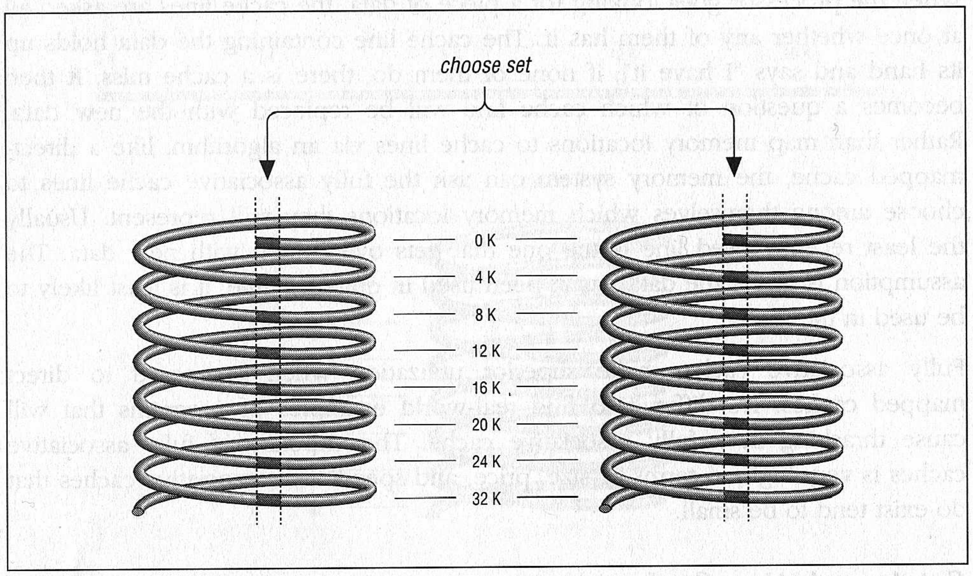

Figure 3-3: Two-way set-associative cache *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Set-Associative Cache Cont.

Easy to implement. If large enough can perform on par with fully associative caches.

However, can still be made to thrash.

Hard to detect, other than a certain sense of slowness...

Instruction Cache

Instructions and data treated different. Could execute an instruction with a cache miss, side by side instructions that don't need data.

Why make them wait?

Instructions and data often come from separate locations. Don't want an instruction cache miss to bump useful data.

Makes sense to separate the two. Known as *Harvard Memory Architecture.*

Typically, such processors have separate L1 caches for instructions and data, and a shared L2 cache.

Virtual Memory

Separates the addresses used by program (virtual addresses) from actual addresses where data is stored (physical address).

Program sees address space as 0 to some large number. Physical address might be quite different.

Virtual memory systems divide program's memory into pages.

Sizes vary from 512MB to 1MB and greater - depends on the computer.

Pages are not necessarily physically contiguously allocated - the program sees them as contiguous, though.

Makes programs easier to arrange in memory, and to be swapped to disk.

Page Table

When a program access a variable, say at location 1000 in its address space, the physical location must be looked up. This requires a translation.

The map with this information is called a *page table.*

Each process has several associated with it. They correspond to different areas. ie. instructions and data sections.

Page Table Cont.

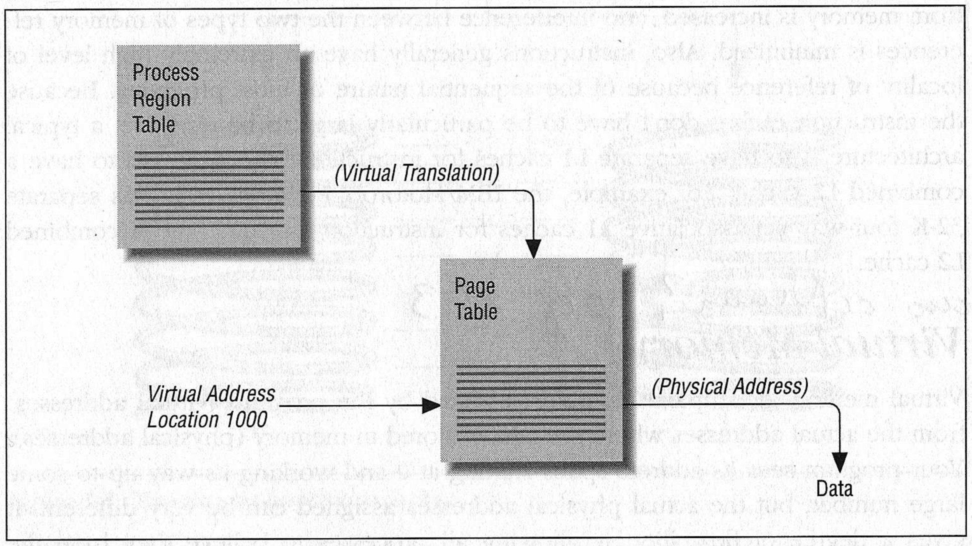

Figure 3-4: Virtual-to-physical address mapping *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Translation Lookaside Buffer

By itself, virtual memory would be slow. However, virtual addresses tend to be grouped together. May do the same page mapping many times per second.

Can use special cache referred to as *translation lookaside buffer* (TLB) to speed up the translation.

Input to TLB: identifier of program making memory reference and virtual page requested.

Output: pointer to physical page number.

Translation Lookaside Buffer Cont.

TLB of finite size. If info not in TLB, then have a *TLB miss.*

A new page may need to be created, or info retrieved from page table in main memory.

Also has pathological case. Assume TLB page size is < 40KB.

```
REAL*4 X(10000000)
DO I  = 0,9999
  DO J=1, 10000000, 10000
    SUM = SUM + X(J+I)
  ENDDO
ENDDO
```

Page Faults

Page table entry contains additional information about page: Flags about validity of translation, if page can be modified, how a new page should be initialized etc.

References to invalid pages are called *page faults.* This occurs when page hasn't been created yet, or has been swapped to disk.

Time consuming, but not errors. Perhaps caused by first access to a variable, or call to a subroutine.

Pool of physical pages limited as main memory limited. More programs running, more likely to cause page faults.

Paging space (swap space) on hard drive is slowest form of memory. Want to avoid as much as possible.

Improving Memory Performance

Two main attributes of memory system performance:

Bandwidth: Deals with best steady-state transfer rate for memory.

Measured by running lengthy unit-stride loop reading/writing memory.

Latency: Measures worst-case performance when moving small amount of data (ie. 32 or 64 bit word) from processor to memory and vice versa.

As memory systems composed of components, each has own bandwidth and latency values. See Figure 3-5.

We next look at ways to improve these values.

Improving Memory Performance Cont.

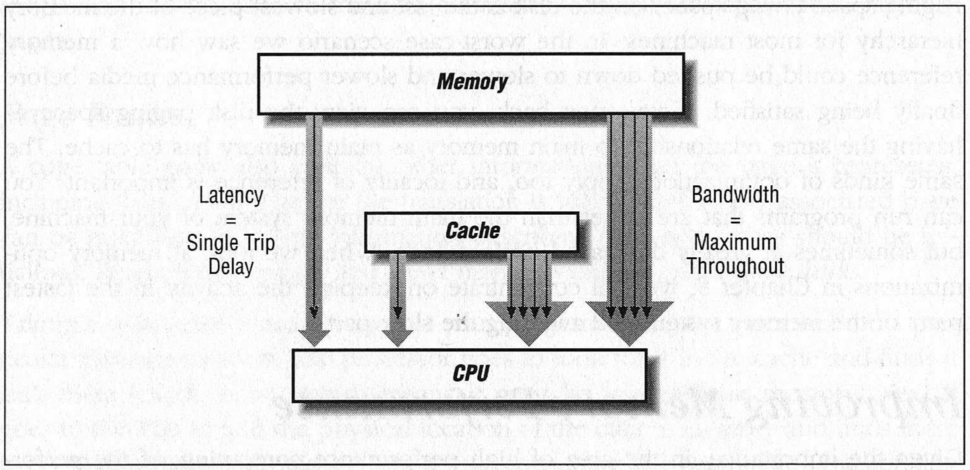

Figure 3-5: Simple memory system *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Large Caches

Cache sizes are increasing. Some high end systems with 8MB! More memory than many PCs had a few years ago.

Expensive approach, but effective. Small/medium problems may fit entirely in cache!

Beware when testing systems. If application grows beyond cache, could see a factor of 10 slowdown!

Beware of benchmarks!

Wider Memory Systems

When cache line refilled: contiguous memory locations are read from main memory.

Maybe 16, 256 bytes or more. Want operation to be as fast as possible.

Can increase fill speed by “widening” the data bus as shown in Figure 3-7.

Wider Memory Systems Cont.

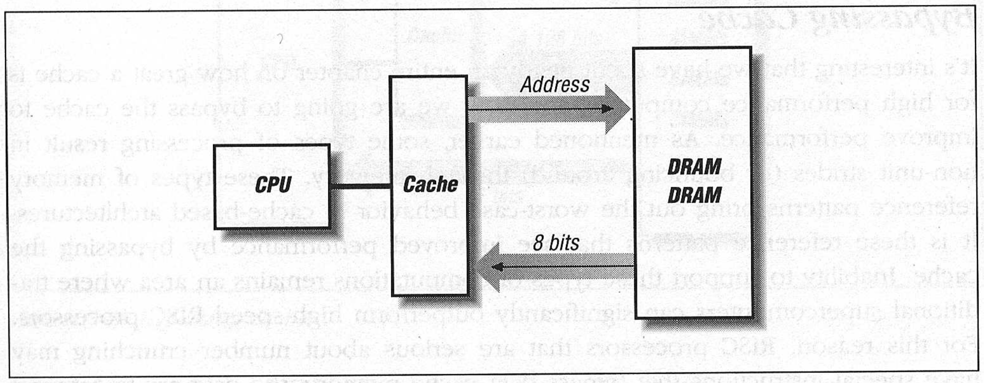

Figure 3-6: Narrow memory system *

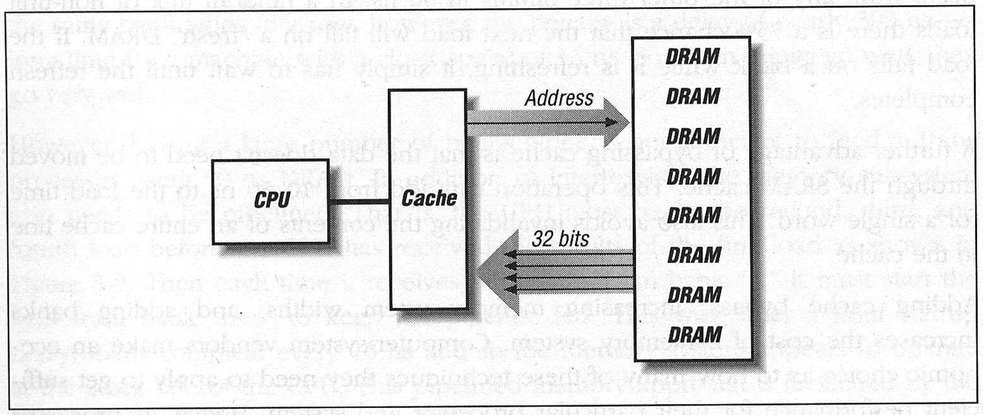

Figure 3-7: Wide memory system

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Bypassing Cache

Spent so much time talking about how great caches are, now we want to bypass it to speed things up?

Some memory access patterns are non-unit stride. Bounces all around memory.

Causes worst case behaviour for caches.

In these cases, we want to bypass cache.

Some RISC processors have special instructions that bypass the cache.

Data transferred directly between processor and main memory. See Figure 3-8.

Bypassing Cache Cont.

Have good chance that each access is different RAM. Only have to wait for access time (ie 50ns) instead of cycle time (100ns)

Don't have to move data through cache. Could add 10-50ns to load time per word. Also, won't invalidate a cache line.

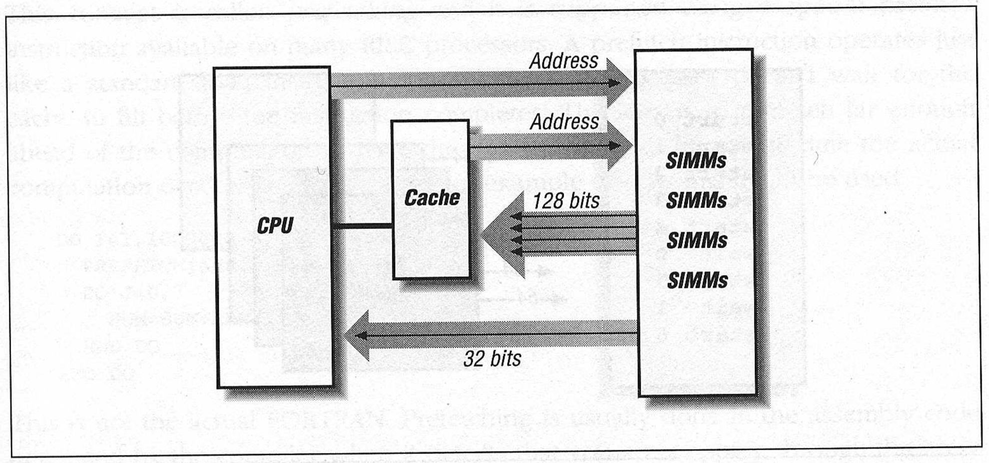

Figure 3-8: Bypassing cache *

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Interleaved and Pipelined Memory Systems

Let's talk high end!

The Cray Y/MP and Convex C3 use multi-banked memory systems.

C3 has up to 256 way interleaving - each bank 64 bits wide. See Figure 3-9.

Reduces chance of using same bank twice in a row.

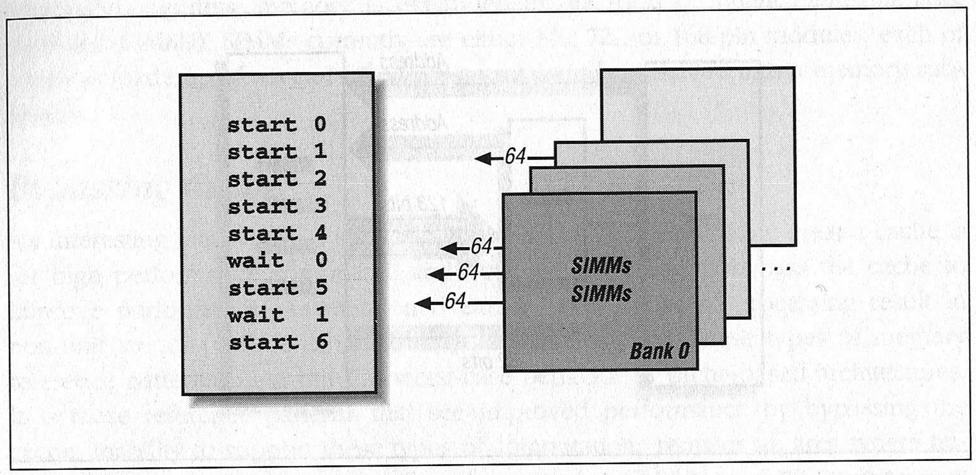

Figure 3-9: Multibanked memory system [*]

*K. Dowd and C. Severance, *High Performance Computing, 2nd Ed.*, O'reilly, 1998.

Interleaved and Pipelined Memory Systems Cont.

Not enough! Also need memory pipelining.

If have 16 ns processor and 50 ns memory, CPU must start 2nd, 3rd, and 4th memory access before receiving the first one.

When receive results from bank n, must start access to bank $n+4$ to keep pipeline moving.

Memory Prefetching

"Poor man's pipeline."

Special *prefetch instructions* added to RISC processors.

Idea to load data into cache before it's needed.

Usually added by compiler when notices specific memory access patterns.

Need superscalar processors to be effective.

Subroutine Calls

Each call to a subroutine has lot of overhead.

If task is small, may be better to do in calling routine.

Subroutine calls make compilers inflexible.

- Side effects of subroutine unknown at time of call. Everything must be saved.
- Subroutine may be compiled separately. Compilers aren't free to intermix, and possibly reorder instructions to optimize.

Consider alternatives:

- Macros
- Procedure inlining

Macros

Macros are small procedures that inserted in-line at compile time.

Functions are only added once during program linking.

Macros are inserted wherever they occur by the compiler.

First stage, compiler looks for macros and inserts the defined code.

Later stages, compiler sees macros as normal source code.

Can be optimized as normal. No subroutine overhead.

Procedure Inlining

Macros usually used for small bits of code, generally one line.

For longer (but not too long) bits of code, use procedure inlining.

Function details copied inline into the functions that call them.

Can define modular function, then inline to avoid overhead and expose parallelism.

Supported by C++.

Problem: If overdone, you get a bloated binary.

Cause more (instruction) cache misses and perhaps swapping.

Branches Within Loops

Numerical programs generally spend the largest portion of their time in loops.

Want to remove unnecessary code from loops.

Loop Invariant Conditionals

These are loops that contain a test that doesn't depend on the loop. ie. we know the results before the loop

```
DO I  = 1, K
  IF (N .EQ. 0) THEN
     A(I) =A(I) + B(I) *C
  ELSE
     A(I) =0
  ENDIF
ENDDO
```

Loop Index Dependent Conditionals

For this case, the test is true for a certain range of loop index variables.

As this has a predictable pattern, we can restructure.

```
DO I=1,N
  DO J  = 1,N
    IF (J .LT. I)
      A(J,I) = A(J,I) + B(J,I) *C
    ELSE
      A(J,I) = 0.0
    ENDIF
  ENDDO
 ENDDO
```

Moving Code

Can get big savings by moving unnecessary or repeated (invariant) operations out of loop.

```
DO I=1,N
  A(I) = A(I) / SQRT(X*X + Y*Y)
ENDDO
```