

# Cache Consciousness: coding for machines with deeper pipelines and shallow caches

By Chris Russ  
jcr6@reindeergraphics.com

Imagine, if you will, a processor that runs at 40GHz, using 0.070 $\mu$ m design rules and a 60-stage pipeline. With current trends, (thanks to Gordon Moore) this will be the state-of-the-art in 2010.



Code that is written for such a machine will have some profound differences from the code we write today; the most expensive operation will be a failed IF-test.

In the past, different instructions have held the award for the most expensive operation: (expensive in the sense of processor cycles)

- Integer multiplication
- Floating point multiplication
- Division
- Inversion
- Square roots and other transcendental
- Non-cached Memory Access
- Branch prediction misses

Each generation of programmer has had to come up with ways to avoid or minimize the costs of some of these functions, by using other coding "tricks." These tricks include the subtle use of addition, shifts & adds, integer vs. floating point math, look-up tables, refactoring, compression, prefetching data, data alignment, etc. Some of these tricks have even migrated

into the optimizing compilers that we use, but it is generally true that code that is written for a specific platform will run faster than generic code. Code that is hand-assembled for a specific platform will grossly outperform that same generic code.

## Recent Processor History - How did we get here?

Starting with the first monolithic CPU chips (4004, 8008, 6502, etc.) and going up to the current generation of microprocessors there have been a number of trends that have helped processor architecture and performance, in addition to the simple scalar improvements from geometry shrinks.

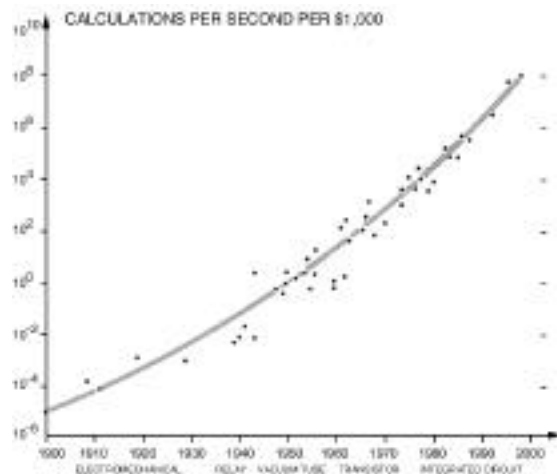


Figure 1a. Moore's Law: CPU,  
Calculations Per Second Per \$1,000

These trends include:

- Increased number of registers
- Integrated L1 Cache
- Harvard Architecture (split Data/Instruction caches)
- Pipelined instructions
- Stack Cache
- On-Chip FPU
- On-Chip addressing for FFT Decimation
- Branch prediction
- RISC vs. CISC design
- Conditional Execution
- Integrated L2 Cache
- Increased Cache Associativity
- Multiprocessor bus-snooping/cache coherency hardware
- DMA
- Streaming data instructions
- Microcode/Micro-OPS
- Multiple Decoders
- Cached decoded instructions
- Deep Pipelines
- SIMD instructions (Vector processing)
- Embedded reconfigurable FPGA blocks
- Integrated L3 Cache
- VLIW and EPIC designs

These trends become possible as a direct result of increased real-estate on the chip. Moore's Law states that the rate of increase of density in a chip will continue, so processor speeds will double in approximately 18 months. (When he coined the law it was 24 months. This doubling rate appears to be accelerating.) Costs per MFLOP (million Floating Point

Operations Per Second) are dropping at the same rate.

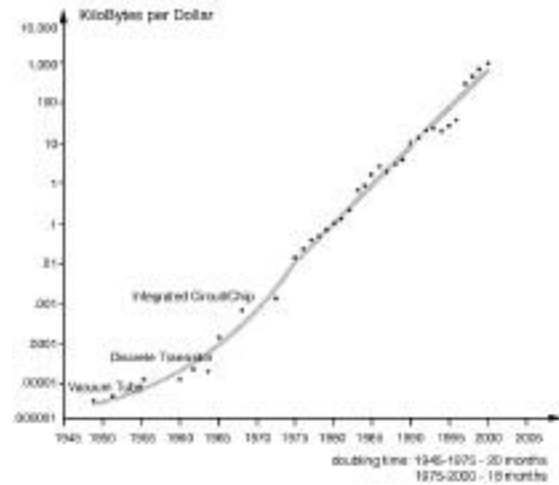


Figure 1b. Moore's Law: RAM, Kilobytes per Dollar

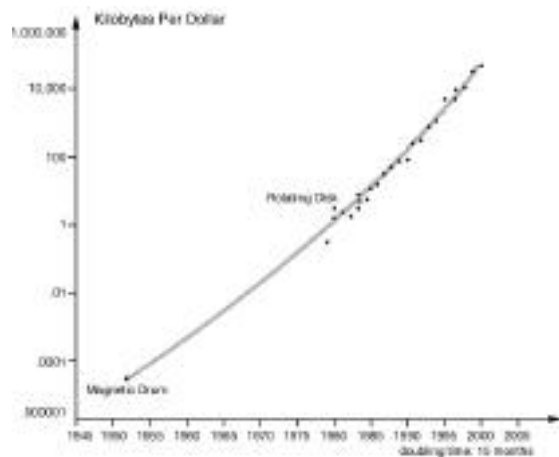


Figure 1c. Moore's Law: Storage, Kilobytes per Dollar

Because of the architectural trends, the number of cycles per operation have dropped. Back from the days of the 68000 where some instructions could take hundreds of cycles, we look at the number of operations that finish per cycle and use that in conjunction with the processor speed to get some measure of actual performance. Today, operations per cycle in the 1-3 range are quite reasonable.

MAC counts (Multiply ACcumulate instructions:  $\text{reg} += a * x$ ) per clock are used as a measure of performance with DSP chips. Using vector operations like AltiVec, it is possible to take a Gigahertz processor and get more than 4 GigaFLOPS as a peak measure of performance. This doesn't necessarily mean that the ACTUAL number of cycles to complete an instruction is less than one -- just that one or more complete per cycle. Several instructions are executing at once. More on this when we talk about pipelines.

Naturally, as there is more space on the chip, it becomes possible to implement more types of hardware support to improve performance. However, we are running out of kinds of parallelism to exploit.

Some of these trends will have to continue to permit the higher clock rates and higher performance of the future.

- **Deeper pipelines**
- **More on-chip caching**
- **More exotic SIMD instructions**

The first two of these are easy to predict and have profound implications on the coding styles that we will be forced to use to get the most performance out of them.

## The Pipeline

The basic execution cycle of an instruction is broken into a number of stages. The most primitive of these is the three-stage cycle with the following stages:

- 1 - Fetch
- 2 - Execute
- 3 - Store

Where the op-code is fetched from memory, it is executed, and the results are

stored back. If the processor were running at 1 MHz, this would permit an instruction to complete every 3rd clock cycle. (Most complex instructions will take more than one clock cycle to execute.)

If we could fetch the next instruction while the current one is executing, barring complications from complex instructions, it would be possible to use the fetch hardware essentially for free. That is the concept of a pipeline. In this case, a three-stage pipeline could complete one instruction every clock cycle and would be three instructions deep.

There are some complications to this notion:

- Not all instructions can execute in one cycle
- If one instruction in the pipeline is dependent upon the result of another that is executing before it, the pipeline will stall while the result is pending.
- If there is a conditional operation, then what do you fill the pipeline with while you're waiting for the result of the conditional? (Did the code branch or not?)

And there are work-arounds for each of these complications:

- **RISC instructions** (Reduced Instruction Set) were designed to better fit a pipeline approach so that it would be easier to build a deeper pipeline and execute code more efficiently. (At one time it was thought that this would kill CISC -- Complex Instruction Set designs.)
- **Out-of-order execution and register renaming** permit non-dependent code to execute at full speed. Instead of having the entire pipeline stall, it is often possible to execute something else to fill the time while waiting for

results to become available for a dependent instruction

- **Branch prediction and conditional execution** make it possible to better predict which way a branch will go and potentially execute part of the code down both paths while waiting to see which way the branch actually

went. As long as memory doesn't get altered incorrectly, this can be a powerful tool. (One major problem comes in if there are too many branch operations close to each other -- there is a small finite limit to how much conditional execution can help in a cascaded if-then-else structure.)



Figure 2. RISC vs. CISC

One of the big architectural battles of the 1990's was between the CISC and RISC camps. The disadvantages of RISC code are that it is larger (every instruction was 4 bytes), the number of instructions (despite the word "reduced" in the name) increased significantly, and programming it in assembly became really painful. The advantages include a rich register set and a large set of "reduced" instructions that a good coder or compiler could use to produce code to fit a pipeline really well, enabling high-speed multi-stage pipelines and use of superscalar designs.

CISC, on the other hand, had some disadvantages as well. There are a number of really quirky slow instructions and a dearth of registers, making some of the inherent parallelism in an algorithm difficult to exploit. But there were advantages. Because memory speed was a serious issue, the time to load the CISC instructions (many of which were 2 bytes) was less, the code was smaller.

The ultimate solution was both. Underneath the current generation of CISC processors runs a decoder that essentially decompiles the CISC instructions into RISC instructions and then executes them through the pipeline. The RISC engines do not need this extra level of decoding, but the CISC engines have survived by becoming the competition under-the-hood. On-chip instruction caches rendered the code bloat problems moot, since most time-critical code gets re-executed in inner loops.

The current generation is termed "Post-RISC" because the new efforts are on pipeline depth and number of execution engines available to service the pipeline.

As the pipelines get deeper, the cost of a branch misprediction gets much higher. On a 20-stage pipeline (Pentium 4), a

misprediction can cost 19 cycles. If this is happening very often, a 2 GHz processor

is effectively acting like a 100 MHz processor.

So, how do you write code to take advantage of the hardware. The first rule remains the best rule:

## The 90/10 Rule

Ten percent of the code will be executed Ninety percent of the time. We don't have to optimize the entire program, just the part that is doing the bulk of the work. The piece of code that reads the preferences from disk or writes the output results away probably isn't the part that is slowing your computer down (unless you do it a lot).

So where, generally, do we want to look for the important 10%?

## Inner Loops

Loops that are nested 3, 4, 5 or more deep are the prime candidates for investigation. There are several properties and characteristics of an efficient inner loop that we want to achieve:

- 1) Few or no conditionals *inside* the loop: Branch prediction misses are really expensive.
- 2) Don't use switch statements *inside* the loop: These are usually implemented as a cascaded set of if-then-else's, which is bad. On the 68K, they can be implemented as jump tables, which is worse.
- 3) Loop unrolling: Try to execute multiple iterations at the same time through the loop. This enables multiple copies of an expression to be evaluated at the same time and also reduces the amount of loop overhead
- 4) Tandem loops: If you've got several loops all the same, covering the same section of data, it is more efficient (especially if you've got a machine with a lot of registers) to combine the

loops together, unless it adds a conditional to the loop. Handing the data (meaning load/store) fewer times (especially if the data is bigger than L1) will improve performance quite a bit. There are many things in the operating system that will cause the L1 cache to flush, so if you were depending upon L1 to still hold useful data, it may not. (This is especially true in a multitasking environment.)

- 5) Data Prefetching: If the loop operates over a section of memory that you haven't been in recently (probably not in L1 or even L2), using one of the BlockMove or memcpy instructions (or one of the streaming data instructions) to load it into L1 (assuming it is small enough!) can make a profound difference. A stall while DRAM is accessed will again slow the machine down to the memory latency, which is much slower than even the main bus speed. (Can you say Rambus?)
- 6) Vector Processing: If there is a SIMD unit, it is possible to take advantage of it (sometime a great deal of advantage from it), but if it is starved for data (because you'll probably spend most of your time waiting for data to be loaded from main memory), you will achieve less real-world improvements.
- 7) Streaming Data: This can be very useful when working sequentially though a large dataset (larger than L2).
- 8) Try not to use Look Up Tables unless they are very small: In the old days, the processors were slow, so it was worth precomputing things. Now, often the time spent to look up data (and push other meaningful data out of the L1 and L2 caches) is longer than the time spent recomputing the data. This is not always true, but you'd be surprised. However, anything that reduces the apparent size of L1 can have a *big* impact on the execution speed of your processor.

- 9) Bigger code is not always a problem: Once Harvard Architectures (split L1 Data and Instruction caches) were introduced, it became much less expensive to make the inner loops large. Your data does not push out the code, nor does the code push out the data. By the time everything is big enough to affect L2, you've got other problems.
- 10) Carve up your problem set into L1 (or smaller) sized chunks: This will enable the processor to run at full speed most of the time, but also allow data streaming to refill/empty the L1 cache. This is more efficient than fetching data onesy-twosy from main memory.
- 11) Linked Lists: There was a time that the linked list was the greatest thing since sliced bread. Unfortunately, any time a new link (something that wasn't already cached) is followed, another slow memory hit occurs. There are some strategies for dealing with these problems including periodic re-sorting, chunky loading, node size reduction (to increase the number of nodes that fit within L1/L2), etc.
- 12) Memory Access Patterns: Sequential memory access is much faster than random. Horizontal access through images is much faster than vertical. Cache Associativity is another big problem, especially with data sets that have records a multiple of 8192 bytes apart. Having a record that is multiple of the Vector Size works well for fetching/storing efficiently, but being exactly 8192 (or a multiple) can reduce memory accesses to DRAM speeds because they always use the same cache lines. Also, anything that can cause VM paging is BAD. (This is one reason that Photoshop uses a tile-based storage system.)
- 13) Invalidate the cache behind you: After you're done with some data, invalidating the section of the cache that holds this stale data makes more of the L1 available for data that you do care about. This is especially

important in a multi-tasking environment because as these other processes get swapped in and out they have a big impact on L1 and L2. If you've flushed out data that you don't care about, the data that you do care about is more likely to still be there.  
(I call this "Low Impact Camping.")

- 14) Bitwise operators vs. logical operators: Boolean operations `&&` and `||` are much, much slower than `&` and `|` ... as long as your logic can handle bit-wise operations instead of logical operations. The overhead of the C short-circuiting mechanism can be expensive. Also, `&` and `|` will not cause a branch to take place.

These are something to think about when writing your inner loop. One fabulous trick is to evaluate *both* conditions at the same time, but use multiplication to select the one that you want:

```

if (a > b)    zr = (first equation);
else         zr = (second equation);

// - - replaced with - -

cc1 = (a > b);
cc2 = (a <= b);
w = (first equation);
v = (second equation);
zr = cc1 * w + cc2 * v;

```

Figure 3a. Evaluate BOTH conditions.  
This code contains no IF-tests.

In the above example we assume that this code snippet is in the middle of a loop. The variable `zr` cannot be assigned until the result of the comparison (`a > b`) has been evaluated. By changing the code to use condition variables and multiplication, it is possible to evaluate BOTH the first and second equation while the condition code registers are being evaluated. If the cost of computing both equations is small compared to the cost of invalidating the pipeline (increasing with

every processor generation), then it is worth it to evaluate all conditions.

```
for (y = 1; y < height-1; y++)
  for (x = 1; x < width-1; x++)
  { //Photoshop's Sharpen function
    long value = array[y][x];

    value = (8 * value + 2 -
             (array[y-1][x] +
              array[y+1][x] +
              array[y][x-1] +
              array[y][x+1])) / 4;
    if (value < 0)
      value = 0;
    else if (value > 255)
      value = 255;

    newarray[y][x] = value;
  }
```

Figure 3b. Limit test inside of an inner loop. This is **Bad**. (There are faster ways to do the array access, too.)

In this example, we need to limit the result of computation before stuffing the value back or else there will be *\*ugly\** consequences in the resulting image. The classic way to do this is with a small Look Up Table that covers the entire numerical output range. In this case the largest number is 510 and the smallest number is -255, so it would fit comfortably in the L1 cache.

```
do
{
  //some function
} while (a < a_limit) && (found);

// - - replaced with - -

do
{
  //some function
} while (a < a_limit) & (found);
```

Figure 3c - replace **&&** with **&**

Finally, we look at a simple characteristic of the C language that once-upon-a-time was a great thing. Short-circuited IF

statements. C was supposed to automatically bypass the other condition comparisons if there was series of logical ANDs and the first one was false. The theory was that the time for the comparison would be large and best avoided. No longer. The extra IF-THEN-ELSE constructs that are generated by the compiler can be worse than if the short circuit code was not there.

Furthermore, if the two values being logically ANDed are Boolean (0/1) then **&** and **&&** will give exactly the same result, except that **&&** generates several instructions while **&** generates only one.

## Branch Prediction

Modern processors use a variety of branch prediction methods:

- Some simply have none
- Some use the rule: backward branches are mostly taken, forward branches are mostly not. (This works since for-loops mostly branch back, and if-tests mostly succeed.) Writing your code with this assumption can make a big difference on older platforms.
- Some processors cache each branch instruction and keep statistics on whether that branch is likely to succeed or not. The size of the table and the quantity of predictive execution that takes place can make a huge difference (AMD K6 vs. K7) on the performance of the code.
- Some compilers can provide "hints" in the instructions themselves as to whether the branch is likely to be taken or not.
- Processors like the Transmeta actually learn what the high-percentage use sections of code are and recompile them dynamically. Theoretically, Dynamic Recompiling Engines like Dynamo could have a significant impact on code and coding styles.

Better strategies come along all the time. As the processors can store more run-time statistics about the code it becomes possible to figure out what kinds of short cuts become possible. At this time, there is really no good way to get run-time data back to the compiler so that it can recompile the code again. A profiler application that could watch and learn and then feedback to the compiler would be an amazing tool.

Current CPU designs will flush the pipeline if a misprediction occurs. Some mainframe architectures would execute along both paths until the condition evaluated. It is only a matter of time before that strategy is implemented in microprocessors.

## Legacy Code

One benchmark for performance will be how fast legacy code (code that is *not* optimized for the new architectures) can run. In most cases, it was either *never* optimized or optimized for a different target where different instructions were expensive. There are still people running programs that run under DOS out there. There are still people running programs compiled in COBOL or FORTRAN 4.

This antique code has one tremendous characteristic:

***It works. It has been tested.***

The software running on the space shuttle is the same software that ran on Apollo, albeit with some patches. There is no source code -- one contractor will send the **object code** to the next contractor and a set of new patches will be added to it.

This is one argument why both emulation and dynamic optimization are so critical, and why Y2K happened at all... If it works, don't screw with it. As code becomes larger and larger, the older stuff that works becomes more and more valuable, even if it is quirky.

## Conclusion

As the speed (both latency and streaming) of different kinds of memory becomes more diverse (from hard disk to on-chip registers), the need for multiple levels of caching will continue to increase. L3 will migrate on chip, L4 will become tens or hundreds of megabytes, and main memory will rival the size of current hard disks. However, most caching will be optimized for small data sets, even though problem sets are becoming larger and larger.

As we continue to ramp up to higher processor speeds, especially with deeper pipelines and more cache stages, it will be more important to structure our inner loops to take this into consideration. Otherwise, new processor generations will not appear significantly faster, just as they do not run legacy code with the same performance that new code or even reoptimized old code.

Fine-grain multitasking (including the multiple core/ multiple thread work going on at IBM) will help processor utilization, but ultimately it will remain a coding problem to get the most out of a single thread. We have to learn to *code differently*.

## Bibliography

[Booth] Booth, Rick *Inner Loops: A Sourcebook for Fast 32-Bit Software Development* Addison-Wesley, 1996.

[DeMone1] DeMone, Paul. *RISC vs. CISC Still Matters* RealWorldTech.com, 2/13/2000.

[DeMone2] DeMone, Paul. *Alpha EV8: Simultaneous Multi-Threat Part 1-3* RealWorldTech.com, 12/13/2000, 12/26/2000, 1/16/2001.

[Every] Every, David K., *What is RISC?* MacKiDo.com, 1999.

[Howland] Howland, John, *Intel's Pentium 4 - A Worthy Successor?*  
RealWorldTech.com, 9/9/2001.

[Jannotti] Jannotti, John *HP's Dynamo*  
arstechnica.com, 3/2000.

[Kurzweil] Kurzweil, Raymond *The law of accelerating returns* kurzweilai.net,  
3/7/2001.

[Stokes] Stokes, John "Hannibal", *The Pentium 4 and G4e: an architectural Comparison* arstechnica.com, 7/2001.

[Stokes2] Stokes, John "Hannibal", *RISC vs. CISC: the Post-RISC Era*  
arstechnica.com, 4/1999.