**Big iron lessons, Part 3: Performance monitoring and tuning**

*Considerations for the system architect*

Sam Siewert (Sam.Siewert@Colorado.edu), Adjunct Professor, University of Colorado

**Summary:** For many developers and engineers, performance is often an afterthought. But when a product functions as designed and has proven stability and the right feature mix, success in the marketplace often depends upon performance. Architectural decisions define the ultimate feasible performance of any product. In this article, learn how performance-monitoring technology initially developed for mainframes can help you improve your own code's performance.

**Date:** 19 Apr 2005  
**Level:** Intermediate  
**Activity:** 282 views  
**Comments:** 0 (Add comments)

【Average rating (based on 17 votes)】

If you're concerned with performance -- and almost all system architects claim to be -- then you should include performance modeling and prediction in the early phases of your system design. But how much attention should performance receive relative to feature and functionality analysis and design? Isn't performance prediction a bit like weather or stock market prediction, anyway? What most often happens with computing systems is that the first product in a roadmap does not perform so well. But, given a good feature set, stability, and usability, the product becomes successful. For continued success, the next-generation product is often expected to provide the same features with better performance and lower cost. Lower cost here means that it takes up fewer resources, so performance modeling and tuning of software applications becomes much more critical to success. Furthermore, it is very hard to predict the performance of a brand-new product with never-before-implemented features. However, second- and third-generation products have a much better chance of being accurately modeled based upon measurements and analysis of the first generation.

This article is a continuation of the *Big iron lessons* series. Mainframes, datacenters, and HPC (high-performance computing) and transaction processing systems have a rich tradition of performance measurement and capacity planning. These tools and methods are now also available for PC and embedded system architects. Historically, embedded systems have also provided services that must meet demanding performance (and sometimes scalability) requirements. For example, datacom and telecom systems are embedded, but, much like mainframe systems, they require performance and scalability with minimal cost for services added. It's initially difficult to perform performance modeling and prediction for these systems, and the results are often inaccurate, to the point that the work involved almost seems like a waste of time. But anyone who has worked on a product line knows that eventually, with continued effort, the models become more accurate, and that, with persistence, performance can be designed into the system.

**Early performance models and prediction**

Early, first-generation product, performance models, and prediction often include significant errors. While there is no magic solution to this problem, the best approach is to leverage the experience of others. For the most part, you can view almost any system in terms of its resources, including CPU, memory, I/O, power consumption, heat capacity, size, and mass. Engineering power, thermal, and sizing issues are outside the scope of this article, which focuses on CPU, memory, and I/O. The potential to make tradeoffs between physical and logical performance is an interesting issue, but much more advanced than the current topic.
Given constraints on power consumption, heat generation, and size, how does the architect predict performance based upon allocation of hardware resources to CPU, memory, and I/O in a system? More importantly, how can an architect predict a resource bottleneck? Will the system ultimately be CPU-bound, I/O-bound, memory-bound, or some combination of the three? In large part, the answers to these questions not only depend upon the system design, but also upon the workload from real applications and services. Numerous sources of information and modeling methods exist and even focus on specific applications and architectures that may be similar to a new architecture being designed (see the Resources section below for links to recent books by Raj Jain and Fortier and Michel on analysis and modeling methods). Some investment in these tools and methods is worthwhile, especially as a product roadmap is further developed. By investing early, the accuracy and payoff from these tools and methods for modeling and predicting performance will be realized sooner.

System bottlenecks and resource sizing

Application code and services are rate-limited by resource limits. The limiting resource is typically called the system bottleneck. The bottleneck could be due to memory, CPU, or I/O bandwidth and latency limitations. Most systems today include not only a traditional CPU, but chipset processors or microsequencers that run firmware as well. The first challenge when tuning a system and optimizing performance is finding the bottleneck. The second challenge is to figure out if the bottleneck can be eliminated by adjusting resources or making better use of them. Adjusting resources will only help if the bottleneck has been correctly identified; ultimately, doing so will only move the bottleneck, unless a perfect match of resources is achieved end-to-end. Adjusting hardware resources may often be impossible. If you need more single-cycle on-chip memory, a huge cost and time investment is necessary to adjust this resource. Instead, if you can modify firmware and software to minimize the loss of performance through a bottleneck, you might reach improvement without hardware changes.

System architects should ideally determine what their bottleneck will be and then plan for scalability when product roadmaps are defined. Firmware and software engineers with tuning tools and some hardware support built into their systems can ensure that maximum performance is achieved for any given set of hardware resource constraints. The rest of this article discusses common built-in CPU performance measurement hardware features and how you can use them to tune firmware and software for optimal performance.

Building performance monitoring into hardware

Some basic methods for building performance monitoring capability into the hardware include:

- Performance event counters
- Execution trace port (branch encoded)
- Trace register buffer (branch encoded)
- Cycle and interval counters for time stamping

This article focuses on built-in event counters and how they can be used to profile firmware and software. However, it is important for any engineer who is designing a system for performance tunability or tuning such a system to consider support for tracing as well as profiling.

In general, tracing provides shorter-duration visibility into the function of the CPU and firmware, but can be cycle-accurate and provide a view of the exact order of events within a CPU pipeline. Trace is often invaluable for dealing with esoteric timing issues and hardware or software interaction bugs, and you can also use it to better understand performance bottlenecks.

Profiling, in which samples of CPU state are taken on a periodic basis, provides a statistical view of how well
resources are being used over much longer periods of time. Profiling requires that a workload be presented to the DUT (device under test) for a lengthy period of time while statistical counts are gathered in the background. A profile provides insight into the areas where software is spending most of its time and where it is efficiently or inefficiently executing, and illustrates how frequently functions are being called. By comparison, while a profile can't tell the tuner anything about latency, a trace will provide a precise measure of latency.

Understanding latency for I/O and memory access can enable better overlap of processing with I/O. One of the most common optimizations is to queue work and start I/O early so that the CPU is kept as busy as possible. So, while profiling is the most popular approach to tuning, tracing features should still be designed into the hardware for debug and performance tuning.

How trace ports work

Many (though not all) common CPU architectures include trace ports or the option to include a trace port. The IBM® and AMCC PowerPC® 4xx CPUs, Xilinx Virtex II Pro, ARM Embedded Trace Macrocell, and Tensilica are among the chips that fall into this category. The 4xx series of processors has included a branch-encoded trace port for many years. The branch-encoded trace port makes a nice compromise between visibility into the core and pin-count coming off-chip to enable tracing. Because the trace is encoded, it is not cycle-accurate, but is accurate enough for hardware or software debugging and for performance optimization. Given modern EDA (electronic design automation) tools, ASIC design verification has advanced to the point where all significant hardware issues can be discovered during simulation and synthesis. SoC (System on Chip) designs make post-silicon verification difficult if not impossible, since buses and CPU cores may have few or no signals that come off-chip. For hardware issues that might arise with internal on-chip interfaces, the only option is to use built-in logic analyzer functions, such as those provided by the Virtex II Pro Chip-Scope. This type of cycle-accurate trace is most often far more accurate than is necessary for tuning the performance of firmware or software.

Encoded-branch trace, such as that provided by the IBM and AMCC PowerPC 4xx, typically requires only eight I/O pins per traceable core. The encoding is based upon trace output that on a core-cycle basis provides information on interrupt vectors, relative branches taken, and the occasional absolute branch taken. Most branches and vectors that the PC (program counter) follows are easily encoded into 8 bits. Typical processors might have at most 16 interrupt vectors, requiring only 4 bits to encode the interrupt source. Furthermore, relative branches for loops, C switch blocks, and \texttt{if} blocks, typically span short address ranges that might require 1 or 2 bytes to encode. Finally, the occasional long branch to an absolute 32-bit address will require 4 bytes. So, overall, encoded output is typically 1 to 5 bytes for each branch point in code. Given that most code has a branch density of 1 in 10 instructions or less, the encoded output, which can take up to five cycles to output an absolute branch, is still very accurate from a software execution order and timing perspective.

Drilling and mining

The term \textit{drill-down} as a description for the data mining of performance counter and trace data analysis was coined by Intel® and describes the profile mining process VTune uses. The concept is sometimes referred to as \textit{data mining} for the Apple Macintosh PowerPC analysis tools called CHUD (Computer Hardware Understanding Development). The images of drilling down into the layers of software, firmware, and hardware to understand performance, or of mining understanding from the samples, are key concepts for using performance profile data.

The program counter is assumed to linearly progress between branch points. So, the branch trace is easily correlated to C or assembly source code after it is acquired through a logic analyzer or acquisition tool such as RISCTrace (see \textit{Resources} for more information on this tool). Due to the high rate of acquisition, which is nearly the cycle rate of the CPU, the trace window for a trace port will be limited. For example, even a 64MB trace buffer would capture approximately 16 million branch points, or about 200 million instructions. At a clock rate of 1GHz, that's only one fifth of a second of code execution. This information is invaluable for hardware
and software debugging and timing issues, as well as direct measurement of latencies. However, most applications and services provide a large number of software operations per second over long periods of time. For visibility into higher-level software performance, profiling provides information that is much easier to use than the information tracing provides. After a profile is understood, trace can provide an invaluable level of drill-down to understand poorly performing sections of code.

How performance counters work

Performance counters first appeared in the IBM PowerPC architecture in a patent approved in 1994 (see Resources). Since then, the manufacturers of almost all other processor architecture have licensed or invented similar features. The Intel PMU (performance monitoring unit) is a well-known example in wide use, perhaps most often used by PC game developers. The basic idea is simple. Instead of directly tracing code execution on a CPU, a built-in state machine is programmed to assert an interrupt periodically so that an ISR (interrupt service routine) can sample the state of the CPU, including the current address of the PC (program counter). Sampling the PC address is the simplest form of performance monitoring and produces a histogram of the addresses where the PC was found when sampled. This histogram can be mapped to C functions and therefore provides a profile indicating the functions in which the PC is found most often.

What does this really tell you? It's an indication of calling frequency, of the size of a function (larger functions have larger address bins), and of the number of cycles that are spent in each function. With 32-bit-word-sized address bins, the profile can provide this information down to the instruction level and therefore by line of C code. Most performance counter state machines also include event detection for CPU core events, including cache misses, data dependency pipeline stalls, branch mispredictions, write-back queue stalls, and instruction and cycle counts.

These events, like periodic cycle-based sampling, can also be programmed to assert an interrupt for ISR sampling of current PC and related event counts. This event profile can indicate address ranges (modules, functions, lines of code) that have hot spots. A hot spot is a code location where significant time is spent or where code is executing poorly. For example, if a particular function causes a cache miss counter to fire the sampling interrupt frequently, this indicates that the function should be examined for data and instruction cache inefficiencies. Finally, from these same event counts it is also possible to compute metrics such as CPI (clocks per instruction) for each function with simple instrumentation added to function entry and exit points. This use of counters with inline code to sample those counters (instrumentation) is a hybrid approach between tracing and profiling, often referred to as event tracing. The performance counters require a hardware cell built into the CPU core, but also require some firmware and software support to produce a profile or event trace. If the cost, complexity, or schedule prevents inclusion of hardware support for performance monitoring, pure software methods can be used instead, as you'll see next.

Firmware and software tuning requires measurement

Characteristics of code segments that most affect performance include:

- Segment path length (instruction count)
- Segment execution efficiency (cycle count for a given path)
- Segment calling frequency (in hertz)
- Execution context (critical path, or error path)

The critical path includes code executed at high frequency for performance evaluation workloads and benchmarks. It is often a small portion of the overall code implemented and can also often fit into Level 1 instruction or L2 unified cache.
Building performance monitoring into software

Most of the basic methods for building performance monitoring capability into a system require firmware and software support:

- Performance counter API (hardware supported)
- Direct code instrumentation to sample cycle and event counters for trace (hardware supported)
- Steady-state asynchronous sampling of counters through interrupts (hardware supported)
- Software event logging to memory buffer (software only, but hardware time-stamp improves quality)
- Function or block entry/exit tracing to a memory buffer (software only, with post-build modification of binary executables)

Software event logging is a pure software in-circuit approach for performance monitoring that requires no special hardware support. Most embedded operating system kernels provide an event logging method and analysis tools, such as the Wind View and Linux Trace Toolkit event analyzers (see Resources). At first, this approach might seem really intrusive compared to the hardware-supported methods. However, modern architectures such as the PowerPC provide features that make event logging traces very efficient. Architectures such as the PowerPC include posted-write buffers for memory writes, so that occasional trace instructions writing bit-encoded event codes and time-stamps to a memory buffer take no more than a single instruction. Given that most functions are on the order of hundreds to thousands of instructions in length (typically a line of C code generates multiple assembly instructions), a couple of additional instructions added at function entry and exit will contribute little overhead to normal operation.

Event trace logs are invaluable for understanding operating system kernel and application code event timing. While almost no hardware support is needed, an accurate time-stamp clock will make the trace timing more useful. Without a hardware clock, it is still possible to provide an event trace showing only the order of events, but the inclusion of microsecond or better accuracy timestamps vastly improves the trace. System architects should carefully consider hardware support for these well-known and well-tested tracing and profiling methods, as well as scheduling time for software development.

**Understanding hardware-software interaction**

*Figure 1* depicts the process of mapping an address profile back to more meaningful levels such as the code module or function where most time is spent during workload testing. Numerous tools provide this type of trace or profile data mining. For example, the Agilent logic analyzers have historically provided mapping from addresses traced or profiled on a memory bus to source code and code symbols. The mapping and ability to correlate the hardware-collected address data to various levels of software is fundamental for using the data to turn around and optimize code.

Integrating performance monitoring into software, firmware, and hardware integration and test

When you implement performance-monitoring hardware and software on an embedded system, the ability to collect huge amounts of data is almost overwhelming. Efforts put into data collection is of questionable value in the absence of a plan to analyze that data. Trace information is typically the easiest to analyze and is taken as needed for sequences of interest and mapped to code or to a timeline of operating system events. When traces are mapped to code, engineers can replay and step through code as it runs at full speed, noting branches taken and overall execution flow. For event traces mapped to operating system events, engineers can analyze multi-thread context switches made by the scheduler, thread synchronizing events such as semaphore takes and gives, and application-specific events.

Profiles that collect event data and PC location down to the level of a 32-bit address take more analysis than simple mapping to be useful. Performance tuning effort can be focused well through the process of drilling
Drilling down provides the engineer analyst with hot spots where more detailed information, such as traces, should be acquired. The drill-down process ensures that you won't spend time optimizing code that will have little impact to bottom-line performance improvement.

The tuning process

Using performance counters to produce a cycle-based and event-based profile is often the ideal way to start the performance tuning process for firmware and software optimization. The profile data sampled in 32-bit address bins can be mapped first to code modules (object code or libraries). The analyst first can drill into the module where the most time is spent and where execution efficiency is lowest. After identifying this module, the analyst should next drill down to a function level within the module. Following that, he or she can drill down to the level of individual lines of code (perhaps using the utility addr2line found in Cygwin and Linux™ binutils). Most often, drilling down to a line of code is far enough. (This method works well for languages like C, where a PC address sample can be correlated back to a function and line of code. For interpreted languages such as Python or the Java™ language, the same principles apply, but hardware performance counters will only help profile the code executing on the CPU. Profiles of interpreted code must come from sampling integrated into the bytecode interpreter instead.)

After an address profile has been mapped to functions and specific lines of code, it may be useful to view the source in mixed assembly or source format to review the code the compiler generated. Often, this shows that specific compiler optimizations may be advisable. Also, if a particular line of C code has a high frequency of cache miss events, examining the code might reveal that fields of a data structure are updated in a structure that straddles two cache lines. Simply aligning this structure better to a single cache line would likely improve execution efficiency. The profile might also reveal that a section of code causes a high frequency of data dependency pipeline stalls. In this case, it might be worth tracing the code to determine cycle distances from MMR (memory mapped register) or other MMIO (memory mapped I/O) reads to usage of that data. Increasing the cycle distance will minimize the stall time. After making the optimizations, you can re-take the profile to determine whether the optimizations actually helped. Figure 1 depicts this concept of drilling down from modules to individual lines of code.

Figure 1. Performance profile data mining (hot spot drill-down) concept
Counting instructions

If you're looking for a less tedious approach than counting instructions by hand from assembly source, you can use a single-step debugger and walk through a code segment in disassembled format. An interesting side effect of this approach for counting instructions is that the counter often learns quite a bit about the flow of the code in the process. Many single-step debugging tools, including JTAG (Joint Test Applications Group) hardware debuggers, can be extended or automated so that they can automatically step from one address to another, keeping count of instructions in between. Watching the debugger auto-step and count instructions between an entry point and exit point for code can even be an instructive experience that may spark ideas for path optimization.

Figure 1 shows the concept of drilling down from modules where most time is spent and where execution efficiency is low to a specific line of code. Drill-down is a key concept in analyzing performance counter profile data. By drilling down in this way, the performance analyst can focus attention on the segments of code most in need of optimization. You can find the concept and methods for drill-down in the Intel VTune tools for analyzing performance counter data. On the PowerPC, the CHUD tools provide a similar approach for mining the large amounts of profile and trace data that can be collected with these tools. (See Resources for more information on CHUD and VTune.) Clearly, both profiling with counters and collecting data with trace methods can leave the analyst swimming in information that must be organized to help guide the process of optimizing code that will lead to the biggest improvements. The CHUD tools documentation for the Apple Macintosh G4/G5 has excellent examples of how to analyze the data collected using a suite of analysis tools that includes MONster and Shark.

Counting instructions by hand

Armed with nothing but hardware support to time stamp events, it is still possible to determine code segment path length and execution efficiency. Ideally, performance counters would be used to automate the acquisition of these metrics. However, when performance counters aren't available, you can measure path length by hand in two different ways. First, by having the C compiler generate assembly code, you can then count the instructions
by hand or by a word count utility. Second, if a single-step debugger is available (for example, a cross-debug
agent or JTAG), then you can count instructions by stepping through assembly by hand. Though this is
laborious, it is possible, as you'll see by looking at some example code.

The code in Listing 1 generates numbers in the Fibonacci sequence.

**Listing 1. Simple C code to compute the Fibonacci sequence**

```c
typedef unsigned int UINT32;
#define FIB_LIMIT_FOR_32_BIT 47

UINT32 idx = 0, jdx = 1;
UINT32 seqCnt = FIB_LIMIT_FOR_32_BIT, iterCnt = 1;
UINT32 fib = 0, fib0 = 0, fib1 = 1;

void fib_wrapper(void)
{
    for(idx=0; idx < iterCnt; idx++)
    {
        fib = fib0 + fib1;
        while(jdx %lt; seqCnt)
        {
            fib0 = fib1; fib1 = fib; fib = fib0 + fib1;
            jdx++;
        }
    }
}
```

The easiest way to get an instruction count for a block of code such as the Fibonacci sequence generating
function in Listing 1 is to compile the C code into assembly. With the GCC C compiler, this is easily done with
the following command line:

```bash
$ gcc fib.c -S -o fib.s
```

The resulting assembly is illustrated in Listing 2. Even with the automatic generation of the assembly from C,
it's still no easy task to count instructions by hand. For a simple code block such as this example, hand counting
can work, but the approach becomes time-consuming for real-world code blocks.

**Listing 2. GCC-generated ASM for Fibonacci C code**

```
.globl _fib_wrapper
.section _TEXT,__text,regular,pure_instructions
.align 2
_fib_wrapper:
    stmw r30,-8(r1)
    stwu r1,-48(r1)
    mr r30,r1
    mflr r0
    bcl 20,31,"L00000000001$pb"
"L00000000001$pb":
    mflr r10
    mtlr r0
    addis r2,r10,ha16(_idx="L00000000001$pb")
    la r2,lo16(_idx="L00000000001$pb") (r2)
    li r0,0
    stw r0,0(r2)
L2:
```
addis r9, r10, ha16(_idx-"L00000000001$pb")
la r9, lo16(_idx-"L00000000001$pb") (r9)
addis r2, r10, ha16(_Iterations-"L00000000001$pb")
la r2, lo16(_Iterations-"L00000000001$pb") (r2)
lwz r9, 0(r9)
lwz r0, 0(r2)
cmplw cr7, r9, r0
blt cr7, L5
b L1

L5:
addis r11, r10, ha16(_fib-"L00000000001$pb")
la r11, lo16(_fib-"L00000000001$pb") (r11)
addis r9, r10, ha16(_fib0-"L00000000001$pb")
la r9, lo16(_fib0-"L00000000001$pb") (r9)
addis r2, r10, ha16(_fib1-"L00000000001$pb")
la r2, lo16(_fib1-"L00000000001$pb") (r2)
lwz r9, 0(r9)
lwz r0, 0(r2)
add r0, r9, r0
stw r0, 0(r11)

L6:
addis r9, r10, ha16(_idx-"L00000000001$pb")
la r9, lo16(_idx-"L00000000001$pb") (r9)
addis r2, r10, ha16(_seqIterations-"L00000000001$pb")
la r2, lo16(_seqIterations-"L00000000001$pb") (r2)
lwz r9, 0(r9)
lwz r0, 0(r2)
cmplw cr7, r9, r0
blt cr7, L8
b L4

L8:
addis r9, r10, ha16(_fib0-"L00000000001$pb")
la r9, lo16(_fib0-"L00000000001$pb") (r9)
addis r2, r10, ha16(_fib1-"L00000000001$pb")
la r2, lo16(_fib1-"L00000000001$pb") (r2)
lwz r0, 0(r2)
stw r0, 0(r9)
addis r9, r10, ha16(_fib0-"L00000000001$pb")
la r9, lo16(_fib0-"L00000000001$pb") (r9)
addis r2, r10, ha16(_fib1-"L00000000001$pb")
la r2, lo16(_fib1-"L00000000001$pb") (r2)
lwz r0, 0(r2)
stw r0, 0(r9)
addis r11, r10, ha16(_fib-"L00000000001$pb")
la r11, lo16(_fib-"L00000000001$pb") (r11)
addis r9, r10, ha16(_fib0-"L00000000001$pb")
la r9, lo16(_fib0-"L00000000001$pb") (r9)
addis r2, r10, ha16(_fib1-"L00000000001$pb")
la r2, lo16(_fib1-"L00000000001$pb") (r2)
lwz r2, 0(r2)
addi r0, r2, 1
stw r0, 0(r9)
b L6

L4:
addis r9, r10, ha16(_idx-"L00000000001$pb")
la r9, lo16(_idx-"L00000000001$pb") (r9)
Listing 3 provides a main program that can be compiled for G4- or G5-based Macintosh computers running Mac OS X. You can download the CHUD toolset for the Mac and use it with the MONster instrumentation included in Listing 3 to measure the cycle and instruction counts using the hardware performance counters. See Resources for more information on using CHUD tools. The example code in Listing 3 takes the Fibonacci code from Listing 1 and adds sampling of the PowerPC G4 performance counters through the CHUD interface to the MONster analysis tool. The same type of cycle and event counters are found in the PowerPC performance counters in embedded IBM PowerPC cores.

**Listing 3. Simple C code for PowerPC to compute CPI for Fibonacci code block**

```c
#include "stdio.h"
#include "unistd.h"

// This code will work on the Macintosh G4 PowerPC with the Monster
// PowerPC Performance Counter acquisition and analysis tool.
// Simply pass in -DMONSTER_ANALYSIS when compiling the example.
// MONSTER provides a fully featured set of PMAPI counters and
// analysis along with the full suite of CHUD tools for the Macintosh
// G series of PowerPC processors.
//
// Alternatively on x86 Pentium machines which implement the Time Stamp
// Counter in the x86 version of Performance Counters called the PMU,
// pass in -DPMU_ANALYSIS. For the Pentium, only CPU cycles will be
// measured and CPI estimated based upon known instruction count.
//
// For the Macintosh G4, simply launch the main program from a Mac OS
// shell with the MONSTER analyzer set up for remote monitoring to follow
// along with the examples in the article.
//
// Leave the #define LONG_LONG_OK if your compiler and architecture
// support 64-bit unsigned integers, declared as unsigned long long in
// ANSI C.
//
// If not, please remove the #define below for 32-bit unsigned
// long declarations.
//
#define LONG_LONG_OK
#define FIB_LIMIT_FOR_32_BIT 47

typedef unsigned int UINT32;

#endif MONSTER_ANALYSIS
#endif LONG_LONG_OK

typedef unsigned long int UINT64;

#ifdef MONSTER_ANALYSIS
#include "CHUD/chud.h"
#include "mach/boolean.h"

#else
#endif

#endif
```

Listing 3 provides a main program that can be compiled for G4- or G5-based Macintosh computers running Mac OS X. You can download the CHUD toolset for the Mac and use it with the MONster instrumentation included in Listing 3 to measure the cycle and instruction counts using the hardware performance counters. See Resources for more information on using CHUD tools. The example code in Listing 3 takes the Fibonacci code from Listing 1 and adds sampling of the PowerPC G4 performance counters through the CHUD interface to the MONster analysis tool. The same type of cycle and event counters are found in the PowerPC performance counters in embedded IBM PowerPC cores.
UINT64 startTSC = 0;
UINT64 stopTSC = 0;
UINT64 cycleCnt = 0;

UINT64 readTSC(void)
{
    UINT64 ts;

    __asm__ volatile(".byte 0x0f,0x31" : "=A" (ts));
    return ts;
}

UINT64 cyclesElapsed(UINT64 stopTS, UINT64 startTS)
{
    return (stopTS - startTS);
}

#else
typedef struct
{
    UINT32 low;
    UINT32 high;
} TS64;

TS64 startTSC = {0, 0};
TS64 stopTSC = {0, 0};
TS64 cycleCnt = 0;

TS64 readTSC(void)
{
    TS64 ts;
    __asm__ volatile(".byte 0x0f,0x31" : "=a" (ts.low), "=d" (ts.high));
    return ts;
}

TS64 cyclesElapsed(TS64 stopTS, TS64 startTS)
{
    UINT32 overFlowCnt;
    UINT32 cycleCnt;
    TS64 elapsedT;

    overFlowCnt = (stopTSC.high - startTSC.high);

    if(overFlowCnt && (stopTSC.low < startTSC.low))
    {
        overFlowCnt--;
        cycleCnt = (0xffffffff - startTSC.low) + stopTSC.low;
    }
    else
    {
        cycleCnt = stopTSC.low - startTSC.low;
    }

    elapsedT.low = cycleCnt;
    elapsedT.high = overFlowCnt;

    return elapsedT;
}
#endif
#endif

UINT32 idx = 0, jdx = 1;
UINT32 seqIterations = FIB_LIMIT_FOR_32_BIT;
UINT32 reqIterations = 1, Iterations = 1;
UINT32 fib = 0, fib0 = 0, fib1 = 1;

#define FIB_TEST(seqCnt, iterCnt) \
  for(idx=0; idx $lt; iterCnt; idx++) \
  { \
    fib = fib0 + fib1; \
    while(jdx < seqCnt) \
    { \
      fib0 = fib1; \
      fib1 = fib; \
      fib = fib0 + fib1; \
      jdx++; \
    } \
  } \

void fib_wrapper(void)
{ 
  FIB_TEST(seqIterations, Iterations);
}

#ifdef MONSTER_ANALYSIS
char label[]="Fibonacci Series";

int main( int argc, char *argv[])
{ 
  double tbegin, telapse;
  
  if(argc == 2)
  { 
    sscanf(argv[1], "%ld", &reqIterations);
    seqIterations = reqIterations % FIB_LIMIT_FOR_32_BIT;
    Iterations = reqIterations / seqIterations;
  }
  else if(argc == 1)
  { 
    printf("Using defaults\n");
  }
  else
  { 
    printf("Usage: fibtest [Num iterations]\n");
  }
  
  chudInitialize();
  chudUmarkPID(getpid(), TRUE);
  chudAcquireRemoteAccess();
  tbegin=chudReadTimeBase(chudMicroSeconds);
  
  chudStartRemotePerfMonitor(label);
  FIB_TEST(seqIterations, Iterations);
  chudStopRemotePerfMonitor();
  telapse=chudReadTimeBase(chudMicroSeconds);
  
  printf("\nFibonacci(%lu)=%lu (0x%08lx) for %f usec", 
    seqIterations, fib, fib, (telapse-tbegin));
  
  chudReleaseRemoteAccess();
  return 0;
}
#else
#define INST_CNT_FIB_INNER 15
#define INST_CNT_FIB_OUTTER 6

int main( int argc, char *argv[])
{

double clkRate = 0.0, fibCPI = 0.0;
UINT32 instCnt = 0;

if(argc == 2)
{    
  sscanf(argv[1], "%ld", &reqIterations);

  seqIterations = reqIterations % FIB_LIMIT_FOR_32_BIT;
  Iterations = reqIterations / seqIterations;
}
else if(argc == 1)
  printf("Using defaults\n");
else
  printf("Usage: fibtest [Num iterations]\n");

instCnt = (INST_CNT_FIB_INNER * seqIterations) +
(INST_CNT_FIB_OUTTER * Iterations) + 1;

// Estimate CPU clock rate
startTSC = readTSC();
usleep(1000000);
stopTSC = readTSC();
cycleCnt = cyclesElapsed(stopTSC, startTSC);

#ifdef LONG_LONG_OK
  printf("Cycle Count=%llu\n", cycleCnt);
  clkRate = ((double)cycleCnt)/1000000.0;
  printf("Based on usleep accuracy, CPU clk rate = %lu clks/sec,\",
         cycleCnt);
  printf(" %7.1f Mhz\n", clkRate);
#else
  printf("Cycle Count=%lu\n", cycleCnt.low);
  printf("OverFlow Count=%lu\n", cycleCnt.high);
  clkRate = ((double)cycleCnt.low)/1000000.0;
  printf("Based on usleep accuracy, CPU clk rate = %lu clks/sec,\",
         cycleCnt.low);
  printf(" %7.1f Mhz\n", clkRate);
#endif

printf("\nRunning Fibonacci(%d) Test for %ld iterations\n", seqIterations, Iterations);

// START Timed Fibonacci Test
startTSC = readTSC();
FIB_TEST(seqIterations, Iterations);
stopTSC = readTSC();
// END Timed Fibonacci Test

#ifdef LONG_LONG_OK
  printf("startTSC =0x%016x\n", startTSC);
  printf("stopTSC =0x%016x\n", stopTSC);
  printf("nFibonacci(%lu)=%lu (0x%08lx)\n", seqIterations, fib, fib);
  printf("nCycle Count=%lu\n", cycleCnt);
  printf("nInst Count=%lu\n", instCnt);
  fibCPI = ((double)cycleCnt) / ((double)instCnt);
  printf("\nCPI=%4.2f\n", fibCPI);
#else
  printf("startTSC high=0x%08x, startTSC low=0x%08x\n", startTSC.high, startTSC.low);
  printf("stopTSC high=0x%08x, stopTSC low=0x%08x\n", stopTSC.high, stopTSC.low);
  cycleCnt = cyclesElapsed(stopTSC, startTSC);
#endif
Running the code from Listing 3 built for the Macintosh G4, the MONster analysis tool determines the path length and execution efficiency for the Fibonacci sequence code block, as summarized in Listing 4. The example analysis in Listing 4 was collected using the MONster configuration and source code easily compiled with GCC and downloadable from the Resources below.

Listing 4. Sample Macintosh G4 MONster analysis for example code

```
printf("nFibonacci(%lu)=%lu (0x%08lx)\n", seqIterations, fib, fib);
printf("nCycle Count=%lu\n", cycleCnt.low);
printf("nOverflow Count=%lu\n", cycleCnt.high);
fibCPI = ((double)cycleCnt.low) / ((double)instCnt);
printf("nCPI=%4.2f\n", fibCPI);
#endif
}
#endif
```

Note that in the example runs, for larger numbers of iterations, the Fibonacci sequence code becomes cached and the CPI drops dramatically. If the Fibonacci code were considered to be critical path code for performance, you might want to consider locking this code block into Level 1 instruction cache.

If you don't have access to a PowerPC platform, an alternative Pentium TSC (time stamp counter) build, which counts CPU core cycles and uses a hand-counted instruction count to derive CPI, is included in Resources. You can download and analyze this code on any Macintosh Mac OS X PC or Windows® PC running Cygwin tools. On the Macintosh, using MONster with the "Remote" feature, the application will automatically upload counter

```
printf("nFibonacci(6)=13 (0x0000000d) for 7545.213589 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 100
Fibonacci(6)=13 (0x0000000d) for 7545.213589 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 1000
Fibonacci(36)=24157817 (0x01709e79) for 5883.610250 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 10000
Fibonacci(28)=514229 (0x0007d8b5) for 6008.113638 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 100000
Fibonacci(27)=317811 (0x0004d973) for 36554.381943 usec
Sam-Siewerts-Computer:~/TSC samsiewert$ ./perfmon 1000000
Fibonacci(17)=2584 (0x00000a18) for 1720178.701376 usec
Sam-Siewerts-Computer:~/TSC samsiewert$
```

Note that in the example runs, for larger numbers of iterations, the Fibonacci sequence code becomes cached and the CPI drops dramatically. If the Fibonacci code were considered to be critical path code for performance, you might want to consider locking this code block into Level 1 instruction cache.

If you don't have access to a PowerPC platform, an alternative Pentium TSC (time stamp counter) build, which counts CPU core cycles and uses a hand-counted instruction count to derive CPI, is included in Resources. You can download and analyze this code on any Macintosh Mac OS X PC or Windows® PC running Cygwin tools. On the Macintosh, using MONster with the "Remote" feature, the application will automatically upload counter
data to the MONster analysis tool, including the cycle and instruction counts for the Fibonacci code block. Find links to all the necessary code in the Resources below.

**Using MONster to count instructions with performance counter hardware**

As shown in the G4 MONster example, the best approach for counting cycles, instructions, and events related to execution (for example, cache misses) is to use built-in hardware counters that can be programmed to take multiple counts while code runs at full speed. Almost all major CPU architectures now include some form of performance monitor counters and a simple register set that allows the user to control them, monitor them, and configure the types of events that will be counted. From an instruction and cycle count, important metrics such as CPI for a code block can be computed.

Basic code optimizations

Given analysis of code such as the example Fibonacci sequence code block, how do you take the information and start optimizing the code? The best approach is to make gains by following the "low-hanging fruit" model of optimization -- apply optimizations that require the least effort and change to the code, but provide the biggest improvements first. Most often this means simply ensuring that compiler optimizations that can be used are being used. After compiler optimizations are exhausted, then, simple code re-structuring might be considered to eliminate cache miss hot spots. The optimization process is often architecture-specific. Most architectures include application programming notes with ideas for code optimization. While the methods for optimizing go beyond the scope of this introductory paper, some of the most common are summarized here.

Some basic methods for optimizing code segments:

- Use compiler basic block optimizations (inline, loop unrolling, and so on).
- Simplify algorithm complexity and unnecessary instructions in code.
- Precompute commonly used expressions up front.
- Lock critical path code into L1 instruction cache.
- Lock critical-path, high-frequency reference data into L1 data cache.
- Take advantage of memory prefetch -- prefetch data to be read and modified into L1 or L2 cache.
- Use MMIO prefetch -- start I/O for data used in algorithms early to avoid data dependency pipeline stalls.

After you’ve exhausted some of the optimization methods above using profiling to support identification of the blocks most in need of optimization, you need to consider more complex optimization methods for additional improvement. Good functional regression testing should be included in the optimization process to ensure that functionality is not broken by optimizing code changes. This is especially true for more complex architecture-specific optimizations. It should also be noted that architecture-specific optimizations make code much less portable.

Some more advanced methods for optimizing code segments:

- Use compiler feedback optimization -- profile provided as input to compiler.
- Hand-tune assembly to exploit features such as conditional execution.

Importance of workloads and drivers for performance process

The performance of firmware and software must be tuned to a workload. A *workload* is a sequence of service requests, commands, I/Os, or other quantifiable transactions that exercise the software. Workloads most often are produced by workload generators rather than real-world service provision. Good workloads capture the
essential features of real-world workloads and allow for replay or generation of the service requests at a
maximum rate so that bottlenecks can be identified in systems (see Resources on the TPC benchmarks and
IOMeter workload generator for examples). Most workload-generation tools are application or service domain-
specific, as the TPC benchmarks and workloads are.

Performance tuning regression

Ideally, after a workload set is identified and performance optimizations are being considered for inclusion into
a code base, ongoing performance regression testing should be in place. Performance regression testing should
provide simple high-level metrics to evaluate current performance in terms of transactions or I/Os per second.
Also, some level of profiling should ideally be included, perhaps supported by performance counters. The
generation of this data should be automatic and should be able to be correlated to specific code changes over
time. In the worst case, this can allow for backing out code optimizations that did not work and for quick
identification of code changes or features added that adversely affect performance.

A roadmap for performance

After the current marriage of software, firmware, and hardware is optimized, how does the architect improve the
performance of the next generation of the system? At some point, for every product, the cost of further code
optimization will outweigh the gain of improved performance and marketability of the product. However, you
can still use performance analysis features such as performance counters to characterize potential modifications
that should go into next-generation products. For example, if a function is highly optimized, but remains as a
high consumer of CPU resources, and the system is CPU-bound such that the CPU is the bottleneck, then
hardware re-architecting to address this would be beneficial. That particular function might be considered for
implementation in a hardware-accelerating state machine. Or perhaps a second processor core could be
dedicated to that one function. The point is that performance counters are not only useful for coaxing maximum
performance out of an existing hardware design, but can also be useful for guiding future hardware design.

Resources

- You can download all of the code in this article, plus some bonus code. The code available includes:
  - fib.c, equivalent to Listing 1 in this article
  - fib.s, equivalent to Listing 2 in this article
  - perfmon.c, equivalent to Listing 3 in this article
  - MonsterResults.txt, equivalent to Listing 4 in this article
  - Monster.session.mtr, the config file you'll need to run the MONster session described in this article
  - A version of fib.s illustrating fib.c's assembly when compiled for an x86-based processor
  - pmu.txt, illustrating the output from perfmon.c on an x86-based processor
  - Makefiles for perfmon.c for both PPC and x86 architectures

- Read the other parts of Sam Siewert's Big iron lessons series on developerWorks.

- For a look at how you can optimize code for performance on AltiVec-based processors, check out Peter
  Seebach's Unrolling AltiVec series at developerWorks.

- You can convert addresses into filenames and line numbers with addr2line, available on Linux or under
  Cygwin. Learn more at the addr2line man page.
• If you're using Windows, you'll need the Cygwin tools to run some of the code described here.

• Raj Jain's book on The Art of Computer Systems Performance Analysis (Wiley- Interscience, 1991) is the classic text on performance modeling, prediction, and analysis.


• Although Leonardo da Pisa (aka Fibonacci) is best known for his preoccupation with rabbits, he is also the person responsible for introducing the decimal number system to Europe -- this saving many generations of students from the agony of having to do multiplication and long division with Roman numerals.

• The CHUD performance counter tools, including MONster and Shark, provide an excellent way to profile code on the Macintosh G4/G5 PowerPC platform. You can download the latest CHUD 4.1.1 tools.

• The TPC benchmarks and IOmeter workload generator are two popular tools often used for comparative performance evaluation.

• You can also download the IOZone filesystem workload generation and performance analysis tool.

• System benchmarking tools can be useful for finding application bottlenecks. Fresh Diagnose is a free system analysis tool for the PC. These types of tools can help with identification of system bottlenecks so that users can upgrade and better match a system's I/O, CPU, and memory for typical applications.

• IBM and AMCC 4xx series processors include built-in tracing that can be acquired and decoded for full branch-level trace. "Software optimization techniques for PowerPC 405 and 440," Neil Leeder (developerWorks, February 2004) provides an overview of PowerPC instruction set optimizations and examples of RISCTrace usage to acquire and decode the PowerPC 4xx branch trace output on the trace port.

• AMCC PowerPC 4xx CPUs, IBM cores (some of which are based on PowerPC 4xx), Xilinx Virtex II Pro, and Tensilica processors all use trace ports (and are all, incidentally, IBM partners). ARM Embedded Trace Macrocell also uses 'em.

• The Wind River Systems Wind View trace tool provides RTOS event and application task context switch event tracing. The tool can collect data on any embedded architecture that can run an instrumented version of the Wind kernel, including PowerPC.

• The Linux Trace Toolkit (LTT) provides a nice way to view Linux system context switches and operating system events. Most UNIX® systems also support strace or ktrace, which provide a textual log of all system calls made during the run of an application.

• The Intel VTune performance counter analysis tool for x86 and Xscale can be used to perform drill-down analysis of code execution performance. It is comparable to the CHUD toolset for the Apple Macintosh. Like MONster and Shark, VTune provides performance counter data analysis for profiles.

• The PAPI (Performance API) project has open source performance counter analysis tools for almost any architecture that supports such counters and more information can be found on the PAPI home page.
- Check out Phillip Mucci's PAPI project paper, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," from the University of Tennessee (in PDF format).

- This PowerPC PMAPI on-line tutorial is a great place to start learning more about how PowerPC performance counters work and how to interface to them for embedded systems.

- Read IBM research's first discussion of PowerPC Performance counters. Since the introduction of performance counters and trace ports on PowerPC architecture, many other architectures have developed similar features.

- Check out the original patent filed by IBM for the performance counter concept built into the PowerPC.

- The concept of performance counters can be found not only on the embeddable PowerPC, but also on mainframes. Big iron PMAPI using CICS provides a similar API for the Z series as is provided for the PowerPC.

- IBM has a nice overview of big iron performance measurement and capacity planning for z/OS. Performance profiling methods have often been pioneered on big iron and HPC systems such as the Z series.

- The indepth PMAPI (Performance Monitor API) tutorial from IBM provides an excellent overview of interfacing to the hardware performance counters.

- Learn how to read the TSC (Time Stamp Cycle) counter on the Pentium PC, which was used as a reference for the example on CPI calculation for those without access to a PowerPC platform.

- The Brink and Abyss Pentium 4 PMU tools provide a method to profile code performance for Linux.

- Check out this overview of the use of logic analyzer tools for tracing. The Agilent analyzers include statistical performance analysis and correlation from profile and trace addresses to source code for the PowerPC.

- Have experience you'd be willing to share with Power Architecture zone readers? Article submissions on all aspects of Power Architecture technology from authors inside and outside IBM are welcomed. Check out the Power Architecture author FAQ to learn more.

- The Power Architecture Community Newsletter includes full-length articles as well as recent news about members of the Power Architecture community and upcoming events of interest. Learn more about the Power Architecture Community Newsletter and how to contribute to it. Subscription is free.

About the author

Dr. Sam Siewert is an embedded system design and firmware engineer who has worked in the aerospace, telecommunications, and storage industries. He also teaches at the University of Colorado at Boulder part-time in the Embedded Systems Certification Program, which he co-founded. His research interests include autonomic computing, firmware/hardware co-design, microprocessor/SoC architecture, and embedded real-time systems.

Trademarks