

simg5

Mark Szymczyk
Me and Mark Publishing
mark@meandmark.com

Abstract

Some types of Mac OS X programs, most notably scientific and graphical applications, require maximum performance. These programs need to know where the code is slow and why. Use Shark to discover where the code is slow. Use simg5 to discover why it's slow on G5 Macs.

Introduction

As its name suggests simg5 is a simulator of the G5 processor, the PowerPC 970. Feed it a trace file containing machine instructions that were recorded as your program ran, and simg5 generates statistics about how the instructions run on a G5 processor. You do not need a G5 processor to run simg5; you can run it on a G4 Mac.

Simg5 is not the first tool you should use to profile your program. Run Shark to find the slow areas of your code. Run the slow areas of your code through simg5 to learn why they're slow.

Generating the Data simg5 Needs

Before you can run simg5 you must create a trace file that contains the data simg5 needs. Amber is the tool that creates trace files. Amber is a command-line tool so you must launch the Terminal application first. Use the cd command to move to the directory where you want amber to write the trace files. The easiest way to run amber is to launch your program before running amber. Use the -a option and supply your program's process ID.

amber -a processID

The process ID is a number the operating system assigns to each running program. Run the Activity Monitor program or the command-line tool top to find your program's process ID.

After running amber you must tell it to start tracing. Press Control+Esc to start a trace, and press Control+Esc a second time to stop the trace. If you quit amber without stopping the trace, amber doesn't save any information. When you stop a trace, amber reports the number of instructions traced and the amount of time spent tracing. Write down the number of traced instructions. You'll need the information when running simg5. Press Option+Esc to quit amber.

Running amber with the -a option traces the entire program, recording every machine instruction. Even small applications have tens of thousands of machine instructions, creating too much information for you to digest. You're normally interested in only a few functions, not the entire program. This is where the -B and -E options come in. The -B option tells amber to start tracing when it reaches the function you specify. The -E option tells amber to stop tracing when it reaches the function you specify.

amber -a processID -B StartFunction -E StopFunction

C++ functions are more complicated to trace with amber. Multiple C++ functions can share the same name. The compiler mangles the function names so each function's name is unique. You must supply the mangled function names to amber. To find the mangled function names, open the object file where the function resides. You can find the options in the following directory in your project folder:

build/ProjectName.build/ProjectName.build/Objects-normal/ppc

Yes, there is a second ProjectName.build directory. There is one object file for each source code file in your object. Open the object file in a text editor. Search for a function name to find its mangled name. The gcc compiler normally mangles C++ function names by giving each function name the prefix ZN and the suffix Ev.

To find the trace files amber created, go to the directory you were in when you ran amber. There will be one folder for each trace you ran. The folder names will be trace_001, trace_002, and so on. Inside a trace folder is one trace file for each thread in your program. The trace files have names thread_001.tt6e, thread_002.tt6e, and so on. The trace files are what you will pass to simg5.

Running simg5

Like amber, you must run simg5 from the Terminal application. Go to the directory where the trace file is. You must supply five arguments to simg5: input file, instruction count, CPI interval, reset point, and output file. The input file is a trace file amber created for you. The instruction count is the number of instructions to run through simg5. That's why you should write down the number of traced instructions when you run amber. The number of traced instructions depends on how much code you traced, ranging from hundreds of instructions for a single function to millions of instructions for an entire program.

The CPI interval is the number of instructions that must pass before simg5 displays the cycles per instruction (CPI) statistics. If you trace 500 instructions and specify a CPI interval of 50, simg5 displays the CPI ten times, once every 50 instructions. If you don't care about the CPI interval, specify an interval equal to the instruction count.

The reset point is the number of instruction that pass before simg5 resets its statistics. If you don't want to reset the statistics, pass a reset point of 1.

The output file is the file where simg5 writes its results. Simg5 adds the extension .results to the file. The following command:

```
simg5 thread_001.tt6e 300 75 1 trace
```

tells simg5 to use the trace file thread_001.tt6e and simulate 300 instructions. It displays the CPI every 75 instructions. The statistics do not reset, and simg5 writes the output to a file called trace.results.

Reading simg5's Output

Open the output file in a text editor to examine simg5's output. The more you know about the G5 processor, the easier time you'll have understanding simg5's output.

CPI

The number of CPI listings depends on the number of instructions and the CPI interval you specified when you ran simg5. The example I used earlier to run simg5 would generate four CPI listings. Each listing builds upon the previous listing.

Simg5 tells you the following information in each CPI listing: the total number of architected instructions, the total number of clock cycles, the number of internal instructions, and the CPI. On the G5 processor each architected instruction expands into one or more internal instructions. The CPI equals the number of clock cycles divided by the number of architected instructions.

Branch Prediction Statistics

Computer instructions normally execute sequentially. Branches are the exception. Calling functions, running loops, and conditional statements cause branches. Modern processors, including the G5, predict branches. When the processor correctly predicts the branch, the code executes faster.

The G5 processor has two items to help with branch prediction: the branch history table (BHT) and the count cache. The BHT contains the results of recent branches. The G5 uses the BHT to predict future branches. The count cache contains the target addresses of recent branch instructions. The target address is where the branch instruction moves program execution.

Simg5 provides statistics about the BHT's accuracy at predicting various types of branches. It also tells you the percentage of count cache hits and the percentage the count cache was correct. Finally, simg5 reports the number of branch mispredict flushes. When the G5 processor predicts a branch, it loads the instructions that make up the predicted path. If the CPU predicts the branch incorrectly, it must flush the loaded instructions out of the pipeline.

Instruction Fetch and Translation Statistics

When the G5 processor fetches an instruction, the first place it looks for the instruction is the effective to real translation table for the instruction cache (IERAT). The effective address is the logical memory address; each 64-bit application has (2^{64}) bytes of logical memory space. The real address is the physical memory address. If an instruction isn't in the IERAT, the CPU looks for the instruction in the translation lookaside buffer (TLB), which contains recent address translations. If the instruction isn't in the TLB, the CPU looks in the L2 cache for the instruction, then memory.

Simg5 reports the number of IERAT hits and misses, the number of TLB hits and misses, the number of instructions found in the L2 cache, and the number of instructions found in memory. The less the CPU has to go to the L2 cache and memory for instructions, the faster your code runs.

Instruction Cache Statistics

The instruction cache statistics deal with the Level 1 (L1) cache. The L1 instruction cache contains the most recently loaded instructions. Your code runs faster when the CPU finds instructions in the L1 cache. When the CPU can't find an instruction in the L1 cache, a cache miss occurs. In the event of a cache miss, the CPU looks for the instruction in the prefetch buffer. The prefetch buffer contains instructions that are fetched early to keep the pipeline full. If the instruction is not in the prefetch buffer, the CPU looks in the L2 cache. If the instruction isn't in the L2 cache, the CPU loads the instructions from memory.

Simg5 reports the percentage of cache hits, which occur when the CPU finds the instruction in the L1 cache. For cache misses simg5 reports the percentage the instruction was in the prefetch buffer, the percentage the instruction was in the L2 cache, and the average time the CPU had to wait for the instruction to be loaded. Simg5 takes the cache misses and instruction prefetches and tells you the number of times the instruction was loaded from the L2 cache and the number of times the instruction was loaded from memory.

Data Side Translation Statistics

When the G5 processor fetches data, the first place it looks is the effective to real translation table for the data cache (DERAT). If an instruction isn't in the DERAT, the CPU looks for the instruction in the translation lookaside buffer (TLB), which contains recent address translations. If the instruction isn't in the TLB, the CPU looks in the L2 cache for the data, then memory.

Simg5 reports the same kinds of information for data translation as it does for instruction translation: DERAT hits, TLB hits, data found in the L2 cache, and data found in memory. The data translation statistics are broken down into loads and stores. Simg5 also breaks down the DERAT hits into hits on the first try and hits anytime.

Data Prefetch Statistics

The G5 processor prefetches data to keep the pipeline full. The prefetched data comes from data streams in the AltiVec unit. Up to four data streams can be active at one time. Simg5 reports the percentage of clock cycles that 0, 1, 2, 3, and 4 data streams were active.

After the data stream percentages comes prefetch behavior statistics. There are too many statistics to list here, but simg5 breaks the statistics down into good and bad behavior for you to easily find problems in your program.

Data Cache Statistics

The data cache statistics deal with the Level 1 (L1) data cache, which contains recently used data. When the CPU finds the data it needs in the L1 cache, your code runs faster. If the CPU can't find data in the L1 cache, a cache miss occurs. When a cache miss occurs, the CPU looks for the data in the load miss queue (LMQ). The LMQ holds recent load misses. If the CPU can't find the data in the LMQ, it looks in the L2 cache. If it can't find the data in the L2 cache, the CPU loads the data from memory.

Simg5 reports the following data cache statistics: the cache hit rate, the chances of a cache miss hitting in the LMQ, the chances of a cache miss hitting in the L2 cache, the average wait time for a cache miss, and the average wait time for a cache reload. For cache misses simg5 reports the number of times the data was found in the L2 cache and the number of times it was found in memory.

Execution Unit Statistics

The execution unit statistics report the percentage of clock cycles an instruction was issued to each execution unit. Simg5 reports statistics for the following execution units: two integer units, two load/store units, two floating-point units, branch register unit, condition register unit, vector permute unit, vector simple integer unit, vector complex integer unit, vector float unit, and vector store unit.

Queue Resource Usage Statistics

The queue resource usage statistics section covers two types of queues: information queues and issue queues. Information queues help the G5 processor keep track of all the instructions in the pipeline. There can be over 200 instructions in the pipeline. Simg5 reports the average number of entries in each information queue as well as the average number of architected and internal instructions in the pipeline.

Issue queues move dispatched instructions to the G5's execution units. Simg5 reports the average number of instructions in each issue queue.

Rename Resource Usage Statistics

Instructions in compiled code frequently write to a register after reading or writing to the same register in the previous instruction. Without rename registers the instruction would have to wait for the previous

instruction to execute. By renaming the register the instruction can execute without waiting for the previous instruction to execute.

Rename registers also store the results of instructions when they are dispatched to execution units. Executing one instruction at a time would be inefficient. Rename registers let the CPU work on instructions early, storing the results in rename registers until they're needed.

Simg5 reports the average number of rename registers used. The number of rename registers ranges from 16 in the Link/Count registers to 80 in the integer, floating-point, and vector units. The higher the average, the more likely the CPU is running out of rename registers, which slows your program.

Instruction Frequency

The instruction frequency section contains two tables of information. The first table is for architected instructions, also known as ops. The architected instruction table has one listing for each architected instruction. The easiest way to explain the information in an architected instruction table listing is with an example.

lmw | 34 | 1.3465 | 142 | 4.1740 | 4.176

This listing says that the instruction `lmw` executed 34 times during this run, making up 1.3465 percent of all architected instructions. The 34 `lmw` instructions expanded into 142 internal instructions, making up 4.174 percent of all internal instructions. The ratio of internal instructions to architected instructions is 4.176.

The second table is for internal instructions, also known as iops. There is one listing in the table for each internal instruction. The easiest way to explain the information in an internal instruction table listing is with an example.

rlwinm | 36 | 1.0582 | 5 | 0.2535 | 31 | 2.1678

This listing says the internal instruction `rlwinm` executed 36 times during this run, making up 1.0582 percent of all internal instructions. 5 of the 36 `rlwinm` instructions were not expanded from architected instructions. These 5 instructions make up 0.2535 percent of all non-expanded internal instructions. 31 of the 36 `rlwinm` instructions were expanded from architected instructions, making up 2.1678 percent of all expanded instructions.

CPI Stack

The CPI stack divides the cycles per instruction information into categories that `simg5` calls reasons. There are too many reasons to explain in this paper, but `simg5`'s report provides a brief description of each reason. `Simg5` breaks down the reasons into three major categories. Pipeline contention reasons occur when a group of instructions is waiting to complete and some of the instructions have stalled. Pipeline empty reasons occur when there are no groups of instructions waiting to complete. Pipeline refill reasons occur when a group of instructions is waiting to complete and there are no stall conditions.

For each reason in the CPI stack, `simg5` tells you the reason, the number of clock cycles spent in the reason, and the CPI for the reason. In the following example:

Wait_load_miss 2.12624 5373

the CPU spent 5373 clock cycles waiting because of loads missing in the data caches. The waiting accounted for a CPI of 2.12624.

Examining an Instruction's Path Down the Instruction Pipeline

For those of you interested, `simg5` provides an option to examine individual machine instructions and the path each instruction takes down the instruction pipeline. If you want to examine the instructions, you must tell `simg5` to generate pipe output. Generating pipe output requires you to run `simg5` with three options: `-p`, `-b`, and `-e`. The `-p` option specifies the scrolling method for the pipe output. There are three scrolling methods: by architected instruction count (`-p 1`), by internal instruction count (`-p 2`), and by clock cycle count (`-p 3`). Architected instructions expand into one or more internal instructions.

The `-b` option tells `simg5` when to start scrolling. If you want to start scrolling from the beginning of the trace, use the value 1. The `-e` option tells `simg5` when to stop scrolling. If you choose to scroll by clock cycle count, remember that there are more clock cycles than instructions in a trace. The following command shows how you would run the earlier example to generate pipe output:

```
simg5 thread_001.tt6e 300 75 1 trace -p 1 -b 1 -e 300
```

When you run `simg5` to create pipe output it creates two additional files. One file has the extension `.pipe`, and the other file has the extension `.config`.

To examine the pipe output run the program Scroll Pipe Viewer. Scroll Pipe Viewer has a GUI, but you must launch it from the command line by typing `scrollpv`.

Choose File > Open > Open File 1 to open a pipe file. If the pipe file has a lot of instructions, loading the file can take a long time. That's why running `simg5` on a couple of functions is better than running it on an entire program. An entire program takes too long to load, especially if you have a G4 Mac.

To make navigation in Scroll Pipe Viewer easier, choose Scroll Mode > Symbol Tracking, which opens a dialog box. Select the Track on checkbox. Tell Scroll Pipe Viewer to track on F with an offset of 0. F is the symbol to fetch an instruction, which initiates the instruction's path down the pipeline. When you scroll down to see later instructions, Scroll Pipe Viewer scrolls to the right automatically so you see the clock cycles where the later instructions were fetched.

Scroll Pipe Viewer shows you the following information about each instruction: the instruction number, the instruction, the instruction's memory address, and the memory address of the data the instruction uses. This information appears on the right side of the window.

The left side of the window contains a grid, which you can see in Figure 1. Each column in the grid represents a clock cycle, and each row represents an instruction. The grid consists of one-character symbols that tell you what the instruction was doing during this clock cycle. Most grid symbols are a period, which means the instruction wasn't in the pipeline during that clock cycle. Scroll Pipe Viewer uses colored text for symbols that are in the pipeline. Red symbols indicate performance problems.

<Insert figure1.jpg>

Figure 1. Scroll Pipe Viewer grid.

Moving the mouse over a symbol in the grid tells you what the instruction was doing during that clock cycle. The information appears in the control panel above the grid. Clicking a symbol tells you the following information about the instruction: the internal instruction ID, the architected instruction ID, the instruction, the instruction's memory address, and the read and write registers the instruction used.

Conclusion

For most of you who are interested in profiling code, Shark is sufficient. It tells you the functions where your program spends the most time. When you want to learn why those functions are consuming so much time, run the functions through `simg5`. It generates a lot of low-level statistics about your code. You can also examine each machine instruction's journey down the G5's instruction pipeline.