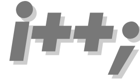
By Greg Thain,
Lead Software Engineer,
NewMonics, Inc.



As timing is very variable on different types of modern CPU's, is becomes very difficult for real-time Java programmers to determine how many cycles a given piece of compiled Java consumes. Therefore we take a closer look at the tools that can help you solve this problem.

INTRODUCTION

This programming statement, unarguably familiar to any C, C++ or Java programmer, is most often found at the heart of critical inner loops in real time programs. It would seem obvious that since real time programs are unusable if they function too slowly, predicting the worst case speed of this critical building block should be one of the simple, daily tasks of a real-time programmer. Unfortunately, this prediction is nearly impossible on the high-speed, modern CPUs required by today's demanding real-time applications.

Let us assume that this statement is written in Java, compiled to a single Intel architecture machine instruction (as it would be in any Java environment), and run on a 500-MHz Pentium III CPU. Even within this reasonable scenario it remains impossible to determine how many cycles this one simple machine instruction might take. In fact, the issue of timing is so variable that even the official Intel architecture use manuals don't list clock cycles for each instruction.

WIZARDRY

The source of these difficulties stems from a variety of components in the modern Intel architecture CPU. Components include instruction caches, data caches, branch prediction buffers, write buffers, a deep pipeline, multiple execution units, and other kinds of untold wizardry, all of which increase dramatically the speed at which instructions usually execute. However, in rare cases, they cause the same instructions to execute very slowly. Unfortunately, real-time programmers must concern themselves with these rare cases.

A programmer unaware of the possible slowdowns these circumstances cause might be tempted to write a simple loop executing this instruction millions of times, then measure the total amount of time and divide the loop by the number of iterations. After accounting for loop overhead, this process yields the average time. However, the worst case can be commonly dozens of times slower than the average case time. Consequently, this method has the potential to produce answers that are catastrophically wrong.

VTUNE ™

The way to solve these problems, and achieve consistently accurate answers, is to use the Intel VTune™ Performance Analyzer, supporting all Intel Pentium, Pentium II, and Celeron™ processors. This sophisticated profiling package, alongside the NewMonics PERC real-time Java environment, allows Java developers to determine exactly how many cycles a given piece of compiled Java consumes. Intel has incorporated their unique micro-architectural CPU knowledge to provide cycle-accurate profiling and simulation of Java code for the Intel Celeron™, Pentium®, and Pentium II processors.

Problems

When applied to a Java programming environment such as PERC the VTune analyzer's performance analysis of real-time Java programs provides solutions for a legion of problems. First among these problems is the lack of timer precision. The only method of timing an operation, when using vanilla Java code, is the System.currentTimeMillis() method. As its name suggests, this method returns the number of milliseconds from an epoch. Most real-time systems require timer precision at least in microseconds, if not nanoseconds. Exacerbating this situation, the timing resolution is system-dependent, and the reference Java implementation returns time with a ten-millisecond granularity. Clearly, this is insufficient for the needs of most real-time programs.

A somewhat more workable approach to profiling real-time Java code is to use one of the many profilers specifically designed to time desktop Java programs. Many of these profilers work in the same way like the desktop Java programs, using specially instrumented Java interpreters. This is problematic, as the special Java virtual machine is only available for desktop operating systems. Therefore, desktop profilers cannot be used for profiling code, if that code must run on specific embedded hardware. Moreover, the timing will probably be significantly different for the profiled VM running on the desktop than it is for the one running on the embedded board.

Solutions

By using the VTune analyzer with a VM, such as PERC, all of these problems can be solved. This is because the VTune analyzer is not one, but three profilers. To conduct a high level overview of the performance characteristics of a PERC Java program, begin with the call-graph view. As one might expect, VTune analyzer runs the select Java program with the PERC virtual machine and displays a graphical representation of the program's call graph.

Notice that the VTune analyzer does not employ or require the Java source code in generating the call graph. Thus, third party, or system class libraries can

Copyright 2000 by Dedicated Systems Magazine - 2000 Q2 (http://www.dedicated-systems.com)

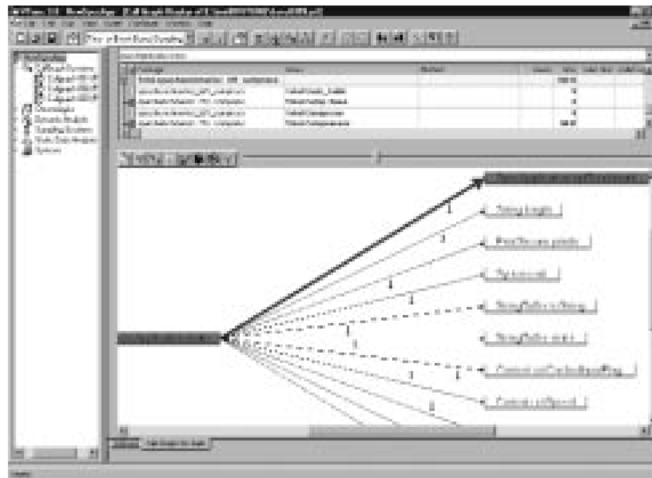


Figure 1. VTune Analyzer Call Graph Display

be profiled with ease. More importantly, because the code is running with the PERC virtual machine-potentially on the actual target machine-all board and device specific Java code can be profiled. Both Just-In-Time (JIT) compiled Java code and native methods implemented in C show up in the call graph.

Multi-threaded Java applications fail to confuse the VTune analyzer, as all multi-threaded code is appropriately accounted for. This profiling method is implemented within the PERC virtual machine by inserting calls to the profiler into user code. Because of this, there is a small, but measurable overhead in using this mode. The overhead is small enough that complete Java programs can be run at near-full speed, but the actual real-time performance may be effected.

Modes

Call graph mode is useful in beginning to analyze the performance of a program running under PERC, and useful for honing in on method(s) of particular interest. However, call graph mode is similar to what is provided by desk profiling tools. The real power in using the VTune analyzer and PERC comes into play with the two remaining profiling modes: sampling and dynamic execution.

Sampling mode interrupts the running program and samples the program counter. When the program is finished, a chart of sampled program counters is displayed for the user. Sampling can occur either at a predetermined interval or when any one of the dozens of "processor events" transpires. It's useful to start with time-based sampling. Because both sampling modes are interrupt driven, they add little overhead to a running program and can be used with minimal disruption of the timing patterns for the tested system.

After time-based sampling, a more detailed examination will utilize event-based sampling, including micro-architecture specific events defined by the CPU performance monitoring counters. Examples of these events include branch prediction failures, L1 instruction cache misses, and other stall conditions. Sampling specific processor events may seem esoteric to the casual developer, sampling of this kind can quickly and definitively expose the effect of caches, branch prediction, and other features of modern CPUs on program execution. Using this mode, programmers can immediately pinpoint instructions that are causing cache misses, or other difficulties.

Dynamic execution

No other Java language tool provides this level of detail about program execution. Because these events all occur inside the CPU, they are invisible to the external bus, so that not even the venerable logic probe can detect them.

The final profiling mechanism the analyzer provides is more precise and sophisticated than the sampling methods. This precision is accomplished by simulating the execution this Java program at the machine instruction level.

Obviously, a simulation runs the program several orders of magnitude slower than the real CPU. Therefore, it can only be used to examine small fragments of programs. The two previous profiling mechanisms can be used to identify the program fragments that require a higher level of detailed inspection.

The VTune analyzer refers to this simulation as Dynamic Execution, and provides simulators for each major subtype of the Pentium processor family. Dynamic Execution provides cycle-accurate measurements of every profiled instruction, including all stall conditions, as well as detailed descriptions of the source of each stall. Like the other profiling modes, no access to Java source code is required, so one can simulate Java systems or third-party class libraries.

TO SUMMARIZE:

The answer to the question: "How many cycles does a

common Java increment statement take?" is complex. NewMonics' PERC is the only real-time Java environment with support for the Intel VTune performance analyzer, which can answer this question completely and thoroughly. The answer can be revealed by looking at three modes of profiling, which provide progressively more detail: call-graph mode, sampling mode, and dynamic execution. Real-time programmers use this best-of-class tool to achieve precise measurements of the timing behavior of their Java programs, rarely available even to C programmers.

The Intel VTune Performance Analyzer is currently supported on all Intel Pentium, Pentium II, and Celeron(tm) processors: Please contact NewMonics for a list of supported operating systems.

*All other brands and names are property of their respective owners ■

Greg Thain is a Lead Software Engineer at NewMonics, Inc., the world leader in real-time Java solutions, where he is responsible for compiler design and implementation. He can be reached at thain@newmonics.com

Copyright 2000 by Dedicated Systems Magazine - 2000 Q2 (http://www.dedicated-systems.com)