Beyond Profiling

Gaining Control of Software Performance

Freescale Semiconductor
Author, Nat Hillary

Document Number: BYNDPROFILNGWP
Rev. 0
11/2005

CONTENTS

1. About this Document 1
1.1 Introduction 1
1.2 Related Documents 1
1.3 Author 2
2. Inclusive and Exclusive Profiling 3
3. Monitoring Critical Sections of Code 4
3.1 Monitoring Critical Sections of Code 5
3.2 Multi-Tasking Applications and Rate Monotonic Analysis 5
3.3 A-B Timers in Code-Test 6
4. Understanding the Impact of Interrupt Service Routines 6
5. Conclusion 7

1. About this Document

1.1 Introduction

There is an unspoken credo in the world of software development that says, "Make it run, make it pretty, make it fast." Developers have learned that functionality comes first, with reliability, performance, etc. falling a distant second place[1]. By the time performance becomes an issue, it is too late. The only way of guaranteeing performance is to build it in. More often than not, when attempting to improve the performance of a piece of software, designers find that the only real performance gains to be had can only be realized through architectural changes. The best that can be hoped for at this stage is incremental performance improvements.

Tom DeMarco stated that "You can not control what you can not measure,"[2] and building performance into software very much proves this rule. In order to guarantee that the software being written will meet its performance objectives, software execution speed must be monitored at every step of the way, with any trade off decisions being based on objective data rather than subjective reasoning.

Software performance profiling is the most popular means of determining where a software application spends its time. Available in a range of tools, performance profiling enables developers to see how much time their software is spending in each function in the system, enabling them to focus on those areas of code that would offer the most benefits from optimization efforts.

As profiling data is gathered over a period of time (rather than a one-off measurement), it provides vital information on the application. For instance, a function may perform to spec when the system starts, but due to degradation of resources may become less and less efficient as time goes on. This degradation is revealed with profiling information. As a result, profiling offers significant benefits to developers interested in improving the performance of their software, but it is incomplete.

Performance profiling tools measure the time that a software application spends in each function, and presents this performance information on a function by function basis as either inclusive (time spent in parent function and all called functions) or exclusive (time spent in parent function only) timing data. From this information, it is possible to identify those functions in which the system spends most of its time. Profiling measurements do not allow developers to measure the execution time of specific sections of code, nor do they normally provide performance information on interrupt service routines (ISRs), both of which are essential to understanding and controlling the performance of an application.

Each of these measurements is crucial to understanding software performance. This paper describes each of these timing measurements in detail and paints a clear picture of why these measurements are important. It will also describe why the recent addition of these timing capabilities to the CodeTEST® tool suite Performance Analysis tool helps to differentiate it from the slew of performance profiling tools on the market.

1.2 Related Documents

[Dijkstra71] E.W. Dijkstra, Hierarchical Ordering of Sequential Processes, Acta Informatica 1, pp. 115-138, 1971

1 Erik Wettersen, Implementing Performance Engineering, Presentation from InterWorks Technical Users Forum of Interex Conference April 21-24, 1996 2 Tom DeMarco, Controlling Software Projects, Management Measurement and Estimation, Prentice Hall, 1982

2 Technical Marketing Group Last Modified 11/27/2001 3:38 PM

[Douglass98] Bruce Powel Douglass, Real-Time UML : developing efficient objects for embedded systems, Addison Wesley, 1998

[Grehan98] Rick Grehan, Robert Moote, Ingo Cyliax, Real-time Programming: a guide to 32-bit embedded development, Addison Wesley, 1998

[Heath98] Steve Heath, Embedded Systems Design, Newnes, 1998

[Hillary98] Nat Hillary, Bridging the Gap between Requirements and Design with Use Cases and Scenarios, Paper presented at InDOORS 98, 5th Annual DOORS User Group Meeting, 1998

[Kamer93] Cem Kaner, Jack Falk, Hung Quoc Nguyen., Testing Computer Software, International Thomson Computer Press, 1993

[Smith98] Connis U. Smith, Lloyd G. Williams, Software Performance Engineering for Object Oriented Systems: A Use Case Approach, Performance Engineering Services, 1998

[XANALYS00] The Memory Management Reference, http://www.xanalys.com/software_tools/mm/index.html

[SEI01] Software Technology Overview; Rate Monotonic Analysis, http://www.sei.cmu.edu/activities/str/descriptions/rma.html

[Kit95] Edward Kit , Software Testing in the Real World: Improving the Process, Addison-Wesley, 1995

[Leveson93] Nancy Leveson, Clark S. Turner, An Investigation of the Therac-25 Accidents, IEEE Computer, Vol. 26, No. 7, July 1993, pp. 18-41

[Myers79] G.J. Myers, The Art of Software Testing, John Wiley, 1979

[Wettersten96] Erik Wettersen, Implementing Performance Engineering, Presentation from InterWorks Technical Users Forum of Interex Conference April 21-24, 1996

1.3 Author

Any comments about this document should be directed to:
Nat Hillary
Applied Microsystems Corporation
5020 148th AVE. N.E. Redmond, WA 98052 Voice: (425) 882-5259 FAX: (425) 883-3049
email: nath@amc.com

2. Inclusive and Exclusive Profiling

Software performance profiling tools provide data on how much time the system is spending in each function in the system. Normally presented in tabular form, the data presented is statistical, such as the data in the following table.

Function Name	Min (mS)	Max (mS)	Average (mS)	Cumulative (S)
`Foo`	13	341	211	612
`Bar`	9	512	339	408
`DoThis`	54	84	62	71
`DoThat`	16	73	45	11
`DoTheOther`	22	99	67	8

From this data, it can be seen that the system spends the majority of its time in the `Foo` function, even though it does not have the largest maximum execution time. The interpretation of this data differs depending on whether the collected data reflects exclusive or inclusive timing measurements.

If this data were exclusive, then the time recorded for function `foo()` would be the time spent in that function only, and not any of the functions that it calls. If the data were inclusive, then the time recorded for function `foo()` would be the time spent in that function and all of the functions called from within the context of `foo`(), as explained by the following figure.

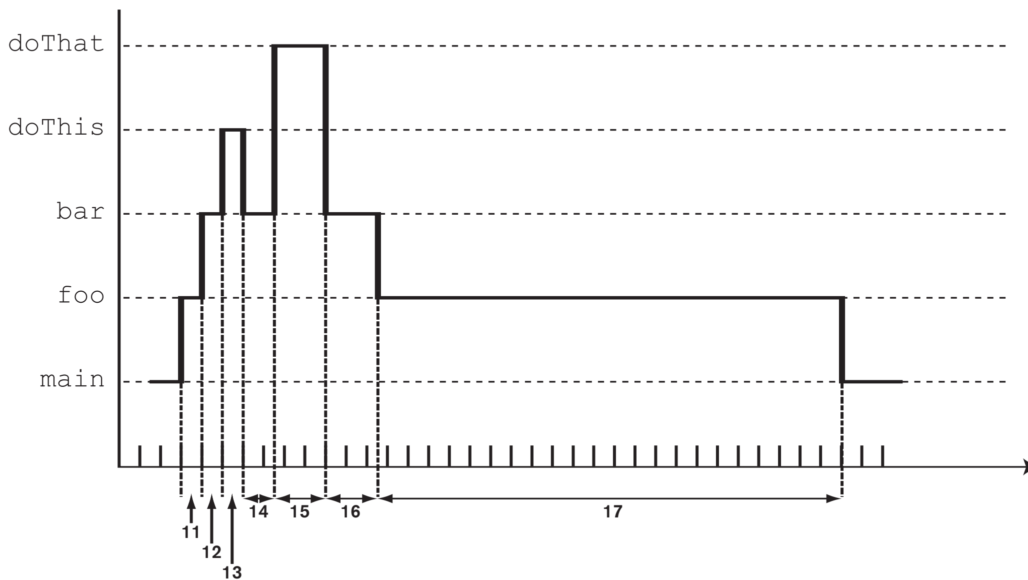

FIGURE 1-FUNCTION TIMING DIAGRAM

When viewed as a timing diagram (Figure 1), the exclusive timing reported for `foo()` would be t1+t7. With inclusive timing, the time reported for `foo()` would be t1+t2+t3+t4+t5+t6+t7.

For performance critical systems, it is common for functions to be assigned a performance budget. If the function exceeds its budget, then it must be optimized to fulfill its budget requirements. To understand whether a function meets its performance budget, only inclusive profiling information may be used. This is because the worst-case execution time for a particular function will be determined by taking a specific path through the function, including all of the sub-functions that are called.

At the same time, understanding whether the poor performance of a function is due to the way that it has been written, or due to the functions that it calls, relies on using exclusive profiling data. This will reveal whether time is spent in the parent function only. If this time is insignificant, then the problem lies within a sub-function.

Through the use of exclusive and inclusive profiling information, it is therefore possible to quickly drill down into the performance of the application and understand which sections of code would benefit most from optimization work.

CodeTEST tool suite uses a combination of source code instrumentation and hardware data collection for measuring function performance. Instrumentation tags are placed at the entry and exit of each function, and this data is then sent to the CodeTEST tool suite data collection agent during code execution. This data is time-stamped by the data collection agent, and the resulting function performance data is then presented in inclusive or exclusive form.

This exclusive and inclusive function performance data therefore enables users to understand and optimize the performance of their applications. In addition, CodeTEST tool suite also provides a Software Execution Trace capability that shows in detail the software execution path that was executed during the measurement.

Both of these capabilities can be used to clearly understand where the system is spending its time, and therefore where to focus optimization efforts. Inclusive and exclusive profiling information may be used to identify hot spots in code, and then the software execution trace mechanism may be used to understand the detailed code flow, helping to identify inefficiencies in code execution.

3. Monitoring Critical Sections of Code

There are times when monitoring the performance of an application on a function-by-function basis is not enough. Examples of this include when attempting to improve the performance of a particular algorithm, or when trying to understand whether a particular section of code is meeting its performance objectives, or even when trying to capture some deterministic data that may be used to calculate whether a multitasking system will meet its performance objectives.

In these instances, a mechanism for placing a start marker at a particular location in code, and a stop marker at another, so that the performance of the marked section of code can be monitored would be ideal (this method is generically referred to as Point-Point timers). Point-Point measurements of this type are possible with Logic Analyzers, but what is missing from a Logic Analyzer solution is the ability to monitor the performance of the section of code over an extended period of time. The ideal then would be to gather Point-Point timing data over time and present it as profile data. In this

way, an accurate picture of how the performance of the code varies with time may be obtained.

This type of Point-Point timing measurement has some specific applications. Data gathered in this way may be used to focus on a particular section of code so that optimization may be performed on it. In addition, monitoring the performance of critical sections of code that execute as tasks produces data that may be used in methods such as Rate Monotonic Analysis to predict whether a multitasking system will always be able to meets its performance objectives.

3.1 Monitoring Critical Sections of Code

When developing performance critical applications, there are always sections of code for which timing objectives are specified. If these sections of code do not perform to spec, then the system is considered to have failed. A good example of this would be a pacemaker, which must perform its dispensing duties within a fixed time period after detecting anomalous heart activity.

These critical sections of code are rarely encapsulated as a single function, so function profiling data cannot be used to verify whether the performance objectives of the critical section of code have been met by the chosen implementation, or not. A Point-Point timing mechanism such as that described above may be used to verify whether critical sections of code are meeting their performance objectives.

Using this mechanism, a marker is placed at the beginning of the critical section of code, and another at the end. The start point normally coincides with the point in code where a particular event is being received, and the stop point normally coincides with where the response is generated.

3.2 Multitasking Applications and Rate Monotonic Analysis

For multitasking applications, such as those using real-time operating systems (RTOSs), the timing of critical sections of code take on particular significance. Due to the pre-emptive nature of an RTOS, it is not possible to guarantee that a particular section of code will run to completion without the task in which it is executing being pre-empted by another task. This makes guaranteeing the determinism of a multitasking system extremely difficult. There is no single metric that may be gathered that indicates whether a multitasking application will always meet all of its performance deadlines.

Instead, an arithmetic method for predicting whether a multitasking system will consistently meet its performance objectives has been created. Known as Rate Monotonic Analysis (RMA), this method estimates the CPU loading from a particular multitasking application. If the loading is below a pre-determined threshold, then the system is considered to be deterministic. The loading is calculated from a number of factors, including the number of tasks executing, the priority of those tasks and the time that it takes for each task to run to completion.

The main problem with this method is the difficulty associated with estimating the execution time of each task. The more accurate the data on task execution times is, then the higher the CPU loading threshold can be. Getting the most accurate timing data is therefore essential.

The Point-Point timing mechanism described above may be used to gather accurate timing data for each task in the system. The critical sections of code for each task, which is normally a portion of code that loops forever, are identified, and then the start marker is added to the beginning of the loop, and the stop marker added to the end. Measurements of task execution times are then made with the system running normally (i.e. all of the tasks in the system are active). Statistical profiling over a significant period of time is essential at this point to ensure that all possible task interactions have the opportunity to occur. Additional stress tests or error conditions injected during this profiling period are a great help. This means that the maximum time measured for each task is an accurate reflection on what would occur in the real world.

3.3 Point-Point Timers in CodeTEST Tool Suite

CodeTEST tool suite provides a Point-Point profiling capability. The user is able to select start and stop markers in code, and the CodeTEST tool suite data collection agent then measures the time that the system spends in this section of code. As with all profiling, the performance data may be collected over a significant period, ensuring that an accurate picture is obtained of the how the execution time of the marked section of code varies over a significant period. Users may select automatic instrumentation points as start and stop markers (such as function entry and exit points inserted by performance instrumentation) or they may manually insert instrumentation points as start and stop markers.

The start and stop markers equate to 32-bit writes to a fixed memory location (where a hardware assisted data collection agent is listening). Using manual Point-Point instrumentation only, this means that for each pass through the critical section of code, the only overhead associated with monitoring the performance of the code is the two 32-bit writes. This monitoring method is therefore minimally intrusive, with the overhead associated with the start and stop markers being easy to calculate. This is therefore the perfect means of making code performance measurements of real-time applications; accurate measurements made with minimum intrusiveness, and with a fixed overhead that is easy to determine.

4. Understanding the Impact of Interrupt Service Routines

Interrupts are the most efficient mechanism possible for interfacing software with its external environment. With interrupts, a software application is informed of an environment change by the assertion of a hardware signal that in turn triggers the execution of an Interrupt Service Routine (ISR). The Interrupt Service Routine, normally an extremely efficient run-to-completion piece of code, will then translate the environment change information into data that the application understands. The efficiency of this mechanism means that it is used for all performance critical aspects of a system, such as providing software with a real-time clock mechanism.

ISR code is executed whenever an interrupt occurs, and does not form part of the main application code. Instead, ISRs are small pieces of code located at very specific memory address locations. The jobs that ISRs fulfill are very simple, normally involving reading data from a hardware device, interpreting the results into a form that the application understands and then passing the interpreted data to the application for processing as part of its normal operation.

The main application is never aware when an interrupt occurs—it simply processes the data that is passed to it. Normally this does not cause a problem, but ISRs do take up time on the CPU. For performance critical applications, it is important to understand how much time the system is spending handling interrupts, and how this is affecting the ability of the application to meet its performance objectives. This may occur when either an ISR is being called too frequently, or it takes too long to execute.

CodeTEST tool suite measures the performance of ISRs in much the same manner as for function performance information. The entry and exit points of each ISR are instrumented, with the tags being written to the same CodeTEST tool suite data collection agent. However, ISR data is held separately from function performance data so that ISR performance information can be explicitly presented independently of the function performance data.

Once again, a combination of CodeTEST tool suite profiling and software execution trace measurements may be used to monitor the performance of ISR code, allowing users to understand whether their ISR implementations are efficient enough, and whether they are interfering with the main application.

5. Conclusion

When it comes to controlling software performance, developers need a means of accurately recording software execution speeds. This is normally provided by software profiling tools. However, most profiling tools have serious limitations. Either they use software data collection based on sampling, or the data that they provide does not give the user all of the information that they need to understand and control the performance of their application. Ideally, a software performance profiling tool will provide both inclusive and exclusive function performance data, in addition to the ability to monitor critical sections of code using Point-Point markers plus the ability to monitor the performance of ISRs.

Function profiling information is essential to understanding where the system is spending its time, and therefore where to focus optimization efforts. The combination of inclusive and exclusive profiling data helps to streamline this optimization process.

Performance-critical applications contain critical sections of code that have strict performance objectives set for them. The only way of ensuring that these objectives are met is to measure the performance of these critical sections of code only. Point-Point measurements provide a minimally intrusive means of measuring the performance of critical sections of code. In addition, measuring the performance of critical sections of code in a multitasking application is essential to being able to predict whether the application will always be able to meet its performance deadlines, or not.

ISRs are designed to be highly efficient code fragments that link hardware events with the main application. Even though they are extremely efficient (often being written in assembly), they still take up CPU time. Understanding the execution time of ISRs and how they affect the main application is essential to controlling the performance of a real-time system.

CodeTEST tool suite is a software performance measurement tool that uses a combination of source code instrumentation and hardware data collection to provide all of these measurements with the highest possible timing resolution. Based on this technology, CodeTEST tool suite provides profiling data on function performance, Point-Point timing and ISR performance. These on their own provide an extremely powerful suite of measurements that enable developers to understand and control the performance of their applications. In addition, CodeTEST tool suite also provides a software execution trace tool that provides developers with detailed visibility of what is going on in their code.

CodeTEST tool suite is the only tool that provides this combination of hardware data collection, comprehensive profiling data and software execution trace visibility, providing developers with the software measurement capability that they need to control the performance of their application.

How to Reach Us:

Home Page:
www.freescale.com

e-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
1-800-521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright license granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© Freescale Semiconductor, Inc., 2005.

Document Number: BYNDPROFILNGWP
Rev. 0
11/2005