Measuring Performance for Real-Time Systems

Freescale Semiconductor
Author, Nat Hillary

freescale™
semiconductor

CONTENTS

1. About This Document

1.1 Introduction

Real-time computer systems are all around us, in large and small applications. The control and monitoring system of a nuclear power plant is an example of a real-time system. So is a pacemaker or a real-time first person video game. What makes these systems unique is the fact that time is used to dictate whether they operate correctly, or not.

In the examples above, these timings are critical. If the nuclear control and monitoring system does not respond to meltdown conditions quickly enough, then the impact is catastrophic. If a pacemaker does not respond to changing conditions fast enough, then the impact is catastrophic for the patient. For the gamer, a system that does not respond in a timely manner normally leads to a lost life in the virtual world.

Okay, so the gamer losing a virtual life is not so critical. Or is it? If this system does not meet the performance criteria of the customer base, then they will choose not to use it. Therefore, the real-time characteristics of the game, which are essential for customer satisfaction, become business critical.

So time is an essential part of the definition of a real-time computer system, and software execution performance becomes all-important. Performance refers to the response time or throughput as seen by the users.

But how do you build performance into an application? For that matter, how do you test the performance of an application? What about when your application works correctly, but not quickly enough?

Real-time systems present unique challenges to the definition, development and testing of software. This is the focus of this paper.

There are many issues surrounding the manufacture of real-time systems, and this paper aims to identify them and to discuss possible solutions. The solutions discussed are based primarily on software performance measurement techniques. For multi-tasking applications, however, there are no metrics that can be used to guarantee performance. In this realm, the paper discusses the use of algebraic prediction methods such as Rate Monotonic Analysis and investigates how the accuracy of these techniques may be affected by using the metrics obtained using the performance measurement techniques discussed.

Starting from the perspective of the developer, the principles of Software Performance Engineering and how these may be employed to build performance into an application are described. For those applications that have already passed this stage, some of the aspects of software design and implementation that affect performance, such as multi-tasking and dynamic memory usage are then described, and how these may be resolved.

Throughout this paper, the hardware assisted analysis methods utilized by the Freescale Semiconductor CodeTEST® product are used as an example of how measurements pertaining to software performance may be made.

1.2 Scope

1.3 Definitions and Acronyms

1.4 Related Documents

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

[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, Fifth 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

1.5 Author

Any comments about this document should be directed to:

Nat Hillary

Applied Microsystems Corporation

5020 148th AVE. N.E.

P.O. Box 97002

Redmond, WA 98073-9702

Voice: (425) 882-5259

Fax: (425) 883-3049

e-Mail: nath@amc.com

2. So Exactly What Are Real-Time Systems?

Simply put, a real-time system is one that fails if its performance criteria are not met. Within this class of system, there are two broad categories: hard real-time systems and soft real-time systems. There is a third category, firm real-time systems, whose definition falls between those of hard and soft real-time systems. For the purposes of making this White Paper as clear as possible, this category of system will not be discussed.

A hard real-time system is one that must meet its performance objectives every time and all the time. As soon as one of these systems does not meet one of its performance criteria, it fails. An example of a hard real-time system is a fly-by-wire flight control system, where if the system does not respond to a pilot's commands within microseconds, then the system fails with potentially catastrophic circumstances.

A soft real-time system is one that must meet its performance objectives on average only. This means that if every now and then a performance deadline is missed, the system does not fail. If, however, the system repeatedly misses its performance deadlines, then it fails. An example of a soft real-time system is a streaming media player, where if the system does not meet its performance objectives in a single instance, then the buffered information ensures that there is no loss of information. Should this loss continue over time, however, the quality of the connection becomes reduced and may eventually be lost.

2.1 Real-Time—That Means Real Fast, Right?

Not necessarily. A pacemaker is an example of a real-time system, and it does not need to operate at the speed of light. It simply needs to operate fast enough to guarantee the health and safety of the patient. A real-time system, then, is simply one in which performance criteria are a critical part of its definition, so much so that when these performance criteria are not being met, the system is considered to have failed.

It's important to note here that systems whose definition include such performance criteria as "shall work twice as fast as its predecessor" or "will handle incoming messages as fast as possible" are NOT real-time systems. The performance criteria here are too subjective to have any credence as real-time objectives. For real-time systems, a definition would more likely appear as "response time between input from the pilot and full flight control surface movement shall be no less than 16 ms, and shall not exceed 20 ms." In this definition, the criteria against which real-time performance successes are measured are clearly stated. A system definition containing this type of real-time system performance criteria will normally also include a clear definition of what type of fault correction is required if these objectives are not being met.

As an example of a real-time system that does not need to be real fast, consider the pacemaker example from earlier. In this example, the pacemaker fails if it does not respond to a change in pulse rate of the patient within sufficient time to guarantee that neither discomfort nor cardiac damage occur. With healthy heart rates in the region of 60-70 beats per minute at rest, this means that the pacemaker does not have to be based on cutting edge silicon to meet its real-time objectives. It is, nevertheless, a real-time system, as its performance criteria determine the success (i.e., healthy patient), or failure (i.e., injured patient, possibly resulting in death).

2.2 So Where Do Real-Time Criteria Come From?

Most real-time systems exist in a world where their environment imposes performance criteria on them. These criteria may be determined by physical or logical constraints. An example of a physical constraint is the response of a flight control system, where the system must pass flight control information from the pilot to the flight controls in a timely manner in order to control the aircraft in flight. For logical constraint, an example is an industrial robot; that is expected to handle work at a piece rate that enables a manufacturing line to meet its expected production margins.

Both of these performance criteria are critical. The performance of the flight control system ensures the safety of the aircraft, while the performance of the industrial robot is critical to meeting the business objectives of the customer for whom it has been installed.

As these two examples demonstrate, the environments in which these systems operate and the rates at which they interact are all-important. It is these environmental interactions that determine the real-time criteria of a system.

In the case of the industrial robot, the environmental interaction is the number of steps that must be completed on each work piece, and the number of work pieces that must be processed per unit time. These provide hard and fast performance criteria for system designers to work with. For flight control systems, the environmental interactions are much more difficult to characterize, as the response of the system determines many things, such as aircraft maneuverability, or the ergonomic characteristics of the design.

3. The Design of Real-Time Systems

For the most part, the development of real-time systems is a black art; there are no consistently used development approaches throughout the embedded industry. This section looks at the issues that must be considered early in the development of a real-time system, such as characterizing a system's real-time environment, and breaking the system design into hardware and software components. At this point, there are also decisions to be made, such as whether to use COTS hardware and/or software products and in choosing hardware products, selecting a CPU that has the right MIPS capability.

This section takes a high level look at the issues surrounding the early stages of real-time systems design, up to the point where the hardware and software responsibilities have been defined and assigned.

3.1 Characterizing the Real-Time Environment

Before a real-time system can be built, it is necessary to understand the environment in which it must operate, and how the system must interact with it. This is an essential step, as this is where the performance criteria for the system are identified and defined.

The first step in characterizing a real-time system is to define the system context. This is simply a map of the world of interest to the system. This map includes all of the people or other systems with which the system must interact.

Once the system context has been defined, the events and messages that are used by the system to interact with its environment must then be identified. External events and messages are the signals that the system uses to understand and interact with its external environment. These signals include actuation, feedback and monitoring signals. Timing information is then added to this signal information when the external events and messages are characterized.

Events and messages may be either predictable or random. In either case, the response of the system must be well defined. The timing characteristics of the system are included in these responses. For example, for an Antilock Braking System (ABS), there is a maximum time between the system detecting that a wheel has stopped spinning and the brakes being released. This time will be determined by stringent research and testing, but is an essential element to the safety of the system. The response time of an ABS system forms a crucial part of the real-time constraints of the system.

In the case of the ABS system, the correct interaction of the system with its environment relies on the experience of domain experts. In this instance, the research scientists are the domain experts. It is they who define what the safe operating criteria for the system are. For systems such as this, the interaction between the system and the external environment are often documented using such notations as Use Cases and Scenarios, where Message Sequence Charts feature strongly. For some systems, however, the time constraints of the environment in which they operate are dictated by a communications standard that they must use to interact with external systems.

Many telecommunications systems manufacturers in Europe use the Specification and Design Language (SDL) to define particular communications protocols. Any system designed to support a particular protocol must conform to the protocol definition and provide an interface that conforms to the interface specification defined in SDL. As speed is critical to the transfer of information in telecommunications, timing information forms an essential part of protocol definition. The timings required by these interfaces and the throughput of the system then contribute to the real-time criteria that the system is required to meet.

After a system has been defined and characterized, it is then necessary to make hardware and software platform choices that will enable the system to be built.

3.2 System Design: Assigning Hardware and Software Responsibilities

Choosing the right platform for a particular system is never a straightforward decision, and is normally influenced by factors such as development time and recurring costs, in addition to whether the system will meet its real-time objectives.

Consider, for instance, a new MP3 player design. The development team has a choice of whether to implement the MP3 protocol in software, or to use an MP3 converter chip. Implementing the protocol in software will yield significantly reduced recurring costs, but will result in higher development costs. Choosing a hardware MP3 converter, however, may minimize recurring costs, but may also impose real-estate and functionality restrictions. As mentioned before, an MP3 player is a good example of a soft real-time system.

For some unique environments, the choices of hardware and software responsibility are obvious. For example, the high data rates required in many telecommunications switching applications are such that it is impossible for these systems to be implemented in software. For many of these applications, the high-speed portion of the design must be implemented in hardware, with the software responsibilities being relegated to system management and maintenance functions.

Another choice that must be made at this stage is whether to use custom hardware and software, or to use Commercial-off-the-Shelf (COTS) components. Once again, this is a trade-off between function, time and money.

COTS choices for hardware can be anything from individual components to complete systems that may be deployed in production systems. At the heart of hardware decisions are choices of: CPU, peripherals, memory, form factor and communications capabilities (VME or CompactPCI systems are good examples where form factor and communications capabilities are combined). Many modern systems use standard hardware platforms (such as a CompactPCI card), with specialized hardware components connected to a standard bus (such as PCI).

Software components are an area where many trade-offs occur for real-time systems.

Traditionally, many real-time systems either do not use a multi-tasking executive or real-time operating system (RTOS), so that the developers have complete control of the performance of every aspect of their system. With CPUs becoming more and more powerful, and with each new generation of a system requiring more and more features, this approach is becoming less common. Also, by its very nature, writing highly optimized code for a given hardware platform also makes this type of software inherently difficult to port to a new platform.

To realize significant improvements in development time, portability and maintainability, many real-time systems are now being developed to use one of the many RTOSes on the market.

Associated benefits that are gained from this approach are the third party libraries that are available for these RTOSes, such as a plethora of communications protocols, math libraries or class libraries.

For some new systems, the designers may want to include code written for a previous system. This is done for many reasons, most frequently to take advantage of and to extend a previous feature set. The problem with this scenario is that legacy code is seldom written for an RTOS, and so to take advantage of the code base, the developers are constrained to using a previously defined environment, or to facing the overhead of porting the legacy code to a new one.

Once the hardware and software responsibilities are defined, and hardware and software environments are chosen, it is then possible to start building the real-time system. In building a real-time system, there are commonly two approaches to achieving software performance. The first is to build performance into the software from the outset, and the other is to measure the performance of the code once it is functionally correct, and make any improvements that are required.

The problem with the second approach is that performance improvements can often only be gained through a significant redesign of the architecture of the software. There is rarely enough time left in a development cycle to accommodate this type of effort, so building performance into the application is the only means of guaranteeing that the real-time constraints of the system will be met.

Whichever method is chosen, the success of both approaches relies on being able to measure software performance. In sections 4, 5 and 6, these two approaches will be discussed in a bit more detail, and a method for measuring software performance will be described. Finally, in section 7, for those instances where software performance absolutely MUST be improved downstream, some of the issues that affect software performance and how these may be addressed to improve overall performance are discussed.

4. Building Software Performance into a Real-Time System

There is an unspoken credo in the general world of software development that says, "Make it run, make it pretty, make it fast," in that order. Within this general arena, real-time embedded software systems are unique; they have their performance objectives dictated by the environment in which the system will operate, i.e., it is fixed before any code has been written. A real-time system that does not meet the performance objectives imposed on it by its operating environment is considered to have failed. Developing software for real-time embedded systems must therefore involve disciplines that are not common to other types of software engineering.

Software Performance Engineering (SPE) is a method for constructing software systems to meet the performance objectives of the system. Performance refers to the response time or throughput as seen by the users.

The SPE process begins early in the development lifecycle at the System Requirements Analysis stage, where the performance objectives imposed by the environment are documented. Then, throughout the design and implementation phases, quantitative methods are used to identify a satisfactory architecture and to eliminate those that are likely to have unacceptable performance.

SPE continues throughout the development process to: predict and manage the performance of the evolving software, monitor actual performance against specifications and report problems as they are identified. SPE begins with deliberately simple performance models that are matched to the current level of knowledge about the emerging software. These models become progressively more detailed and sophisticated as more details about the software are known. The SPE process also includes performance measurement, quantitative analysis techniques and performance design principles.

Simply put, the principles of SPE can be outlined by the following flow of events:

1. Set quantifiable Architectural, Unit and Detailed design performance goals, as dictated by the environment.

2. Design and develop code to meet these objectives.

3. As code is written, measure the code against its performance goals, making corrections where necessary.

4. During system integration, measure performance of each Unit against performance goals, making corrections where necessary.

5. During system test, measure performance of system against performance goals, making corrections where necessary.

Key to the success of SPE is the ability to accurately measure software performance. Section 5 looks at several different software performance measurement methods, and considers the pros and cons of each technique.

5. Measuring Software Performance

There are several techniques used for measuring software performance, from hardware-assisted techniques through software-only methods. The hardware-assisted techniques include the use of such tools as Logic Analyzers and In-Circuit Emulators (ICEs), while software-only methods include approaches such as stack sampling. Incorporating technologies from both of these solutions, there are also source code instrumentation techniques that may be used with a variety of data collection agents.

5.1 Software Performance Metrics

Before considering the ways in which software performance may be measured, it is important to consider the types of metrics that may be obtained and what they mean. The following is a list of metrics that may be used to understand and analyze the performance of a piece of software. Each of the metrics is described in detail, including a brief summary of why this information may be important.

Performance profiling: This is a means of determining where a system is spending its time on a function-by-function basis. This allows developers to identify problem areas and to focus on those sections of an application where the maximum benefit would be gained from performance optimization efforts.

A–B timing: This is a means of measuring the time that it takes to get from one specified point in code to another. This is probably the most crucial timing measurement for real-time systems, as it allows developers to verify whether the timing objectives for a piece of code are being satisfied.

Response to external events: This is a means of measuring the time between an external event occurring and the software responding to that event (e.g., interrupt latency periods). This measurement is especially valuable to real-time system developers for whom hardware components are used to guarantee software performance.

RTOS task performance: These measurements fall into two categories—task deadline performance measurements and task profiling performance measurements. Task deadline performance measurements are related to the time that it takes

for each task in a multi-tasking application to reach its deadline after a triggering event has occurred. Task profiling performance measurements are much like the performance profiling described above, save for the fact that the information is presented on a task level and not a function level. Task profiling measurements record where the system is spending its time on a task-by-task basis.

5.2 Logic Analyzers and In-Circuit Emulators

Natural enhancements of the basic capabilities of a storage oscilloscope, Logic Analyzers are commonly used tools that are used in the design and debug of digital circuitry. By connecting to the bus that components use to communicate, a designer may use a Logic Analyzer to verify that bus traffic obeys the correct protocol, and to debug the circuitry and software when it is does not.

Connecting directly to a CPU bus also allows Logic Analyzers to capture software execution trace information by capturing the sequence of instruction fetch cycles that a processor uses to load program information from RAM. Logic Analyzers provide high-resolution timing information to accompany the software execution trace information. It is therefore possible to make accurate point-to-point timing measurements between points in code. It is also possible to measure the time lag between an external event occurring and the system's response.

For In-Circuit Emulators (ICEs), one of their most powerful capabilities is the bus trace capability. Similar to the trace capabilities provided by Logic Analyzers, Bus Traces allow users to witness what their code is doing on an instruction-by-instruction basis. Combined with powerful triggering mechanisms, ICE Bus Trace has proven to be a very powerful tool for software debugging. As an ICE is able to mimic the execution of a CPU running full speed, on an instruction-by-instruction basis, extremely accurate timing information can be added to the Bus Trace information, making point-to-point timing measurements between points in code possible.

Both Logic Analyzers and ICEs have two main drawbacks as software performance analysis tools. First of all, both are extremely expensive pieces of equipment. The next is that they are both rendered useless in the face of cache-based CPUs.

In order to reduce the performance limitations imposed by communicating with external memory, modern CPUs will copy both data and instructions into high-speed processor-resident memory. By accessing this memory directly, significant improvements in performance may be realized. The problem for Logic Analyzers and ICEs is that when a CPU is executing from cache, there is no external bus traffic. As a result, the activity of the CPU needs to be inferred from what little bus traffic there actually is. With time, CPU cache sizes are getting larger and larger, making any inference techniques more and more difficult (and expensive) to implement.

Due to the fact that both Logic Analyzers and ICEs rely on being able to see external bus traffic, they are both useless as software performance analysis tools with cache based CPUs. This is especially exacerbated as the best software performance can only be obtained when cache use is enabled. In addition, these tools have a limitation in that they are good for point-to-point timing measurements, but they are next to useless when it comes to performance profiling (i.e., determining where the system is spending it's time on a function by function basis). When RTOSes are used, this problem is exacerbated even further, as the task context of each function is critical to the accuracy of timing measurements as well.

5.3 Software-Only Performance Profiling

Several software-only techniques exist for performance profiling, ranging from stack sampling tools to code instrumentation and software data capture technologies.

Stack sampling tools work by taking a periodic snapshot of the CPU stack. From this information, the designer can then determine where the system is spending its time on a function-by-function basis. The main drawback of this approach is that as the stack is only sampled, it is not possible to see all of the function calls that occur in a given measurement period. This means that some functions, especially those with an extremely short execution time, are not ever likely to appear in the profiling results. In the presence of an RTOS, stack sampling becomes exponentially more complex, as each task maintains its own stack context. This means that the context of the stack must be monitored in addition to the contents.

To counter the problem of missing some function activity during profiling measurements, a range of code instrumentation and software assisted data collection methods have been developed. There are essentially two approaches in this category: one that adds instrumentation to object code (i.e., after compilation) and another that adds instrumentation at the source code level.

Object code instrumentation involves parsing the object files produced by a compiler and adding additional instructions at salient points in the code (such as function entries and exits). As with all instrumentation techniques, during code execution, these additional instructions send data to a data collection agent that then processes them to obtain profiling information.

In addition to the intrusive nature of software data collection methods discussed below, Object code instrumentation

methods are inherently limited in their scope by being processor specific. It is therefore common for tools that use this technology to be available on only a limited number of processor platforms.

Source code instrumentation is a similar process, save for the fact that the instrumentation process occurs prior to compilation.

The data produced by either of these software only methods of data collection are restricted in their accuracy and are inherently intrusive.

Unlike Logic Analyzers and ICEs, software data collection agents do not have access to high-resolution timers, so the resolution of software-only performance analysis techniques are much larger grained than any hardware-assisted methods. This resolution is exacerbated considerably by the intrusive nature of software data collection agents that introduce further inaccuracies.

Whichever instrumentation method is chosen, software data collection agents need to do a significant amount of processing on the data that is sent to them. This processing includes time stamping, buffering and transmission of the data off-target. As this processing is in addition to the normal operation of the application under development, software data collection agents are inherently intrusive.

Normally implemented as an independent task within a multi-tasking environment, a software data collection agent becomes intrusive by imposing extra demands for processing time on the system. This can severely affect the timing behaviors of a system, especially when there is a task priority conflict between the data collection agent and one of the other tasks in the system.

Due to the low resolution and intrusive nature of these techniques, they cannot be used for verifying the performance of real-time systems. At best, the information produced using these methods must be considered to be representative only.

5.4 Source Code Instrumentation and Hardware-Assisted Performance Measurements

Probably the best method for making software performance measurements is a technique that combines the best of the hardware-assisted and software-only methods described above. The technique and technology that are described in this section are those that are used by the CodeTEST product from Freescale Semiconductor.

In this method, source code instrumentation is used in conjunction with a hardware data collection agent. Using this technique, highly accurate measurements (comparable with that of either a Logic Analyzer or ICE) may be made with most CPUs, including cache-based processors, with the minimum of intrusion.

This method of instrumentation occurs during code compilation, between the traditional four-stage compile pre-processor and front-end stages. During the instrumentation process, statements are added to salient points in the code, such as function entries and exits.

At run-time, these instrumentation statements pass unique tag values to a fixed memory address. The hardware data collection agent then detects each of these writes, and after time stamping and processing, the data is presented to the user for display.

As the memory locations that the tags are written to are required to be off-chip locations, this approach for measuring software performance allows timing measurements to be made on a cache based CPU. This enables performance measurements to be made when an application is running at the maximum possible rate with a given CPU.

Unlike the measurements that are possible with a Logic Analyzer or ICE, there is no limit to the period of time over which performance metrics for an application may be gathered. This means that performance statistics covering extended periods of time may be gathered, allowing the true performance characteristics of the system over time to be understood. Also, whereas the timing measurements available from Logic Analyzers and ICEs are restricted to A–B timing measurements, the source code instrumentation and hardware data collection method of measuring performance also may be used to conduct performance profiling measurements.

The instrumentation is considered to be intrusive by some, but the overhead imposed is minimal and manageable. For instance, the most basic instrumentation required to make a performance measurement is to have a single start tag and a single end tag. With each tag equating to a handful of assembly instructions each (the actual number is dependent on the CPU being used), the level of intrusion is very low.

In addition to software Performance Analysis, this measurement schema makes many types of software analysis possible, including Coverage Analysis, Memory Analysis and Software Execution Trace.

Whether the developers of a real-time system chose to build performance into their system, or to improve the performance of their software once the functionality is correct, software performance measurements are essential. If a development team is faced with the latter situation, what are their immediate options for improving the performance of the system? If these performance optimization efforts do not work, what other issues can be addressed to help maximize system performance?

The next two sections address each of these issues in turn. Section 6 considers what may be done to improve the performance of a system that is already built, but does not yet meet its performance specs. Section 7 then considers the type of issues that affect the performance of all real-time systems.

6. Improving the Software Performance of a Real-Time System

Improving software performance once a system has been implemented is, at best, an activity in incremental improvements. Most performance bottlenecks are caused by architectural and design limitations that require significant amounts of recoding to overcome. The following, however, are a couple of techniques that have been used successfully to improve software performance.

6.1 Performance Profiling and Algorithm Improvements

Performance profiling is where performance measurements are made on a function-by-function basis. Using the information produced from these measurements, it is possible to tell where the system is spending its time. The objective of using this information and approach is not to improve the performance of every function in the system. Instead, the objective is to determine which functions to focus effort on to achieve the biggest improvements in performance.

Once a performance profiling measurement has been made, each of the functions that consume the most system time must then be considered candidates for optimization. This optimization may take several forms, from algorithm improvements to rewriting and replacing standard library calls with a more efficient, custom version.

The main benefit of performance profiling is that the performance of the system over time may be considered. For instance, during system startup, there is often a heavy workload associated with the initial setup of the system. If performance measurements are made only at this time, they will probably indicate that the setup functions occupy a high proportion of the CPU time. Most of these functions, however, will not be used heavily after setup is complete, and so will not significantly affect the normal system performance. By monitoring performance over time, performance profiling can help to identify how the system uses functions differently with time. This can help developers to identify which areas of code to focus on to improve the performance of particular phases of system operation.

6.2 Code Butting

With modern cache-based CPUs, significant gains in CPU power may be realized when the processor executes instructions from cache. At any time, however, the CPU may be required to access an address that is not resident in cache. In this case, the contents of cache are flushed, and a new section of memory is copied in. Once this operation is complete, execution once again continues from cache.

If an application causes a CPU to flush and re-load its cache too often, then performance slowdown occurs as the processor accesses external memory over the relatively slow system bus. To maintain a consistently high level of performance, then, it is necessary to ensure that the CPU spends as much time executing from cache as possible and the minimal amount of time is spent on re-populating cache.

Code butting is a process whereby those functions that are highly interdependent are located as close as possible to one another in the address map of the application executable. By doing this, it is possible to improve the probability that both functions will be located in cache at the same time, ensuring that a costly cache flush is not required when a function calls another on which it is highly dependent.

Identifying which functions are highly independent is done using call-pair data. Call-pair data is simply a measure of which function calls what other functions during code execution and how many times. For a pair of functions where the caller calls the callee a high number of times, these are interdependent functions.

Simply changing the link order of the object files during the link stage of the build process positions interdependent functions consecutively in the memory map of the application executable. Those object files that contain interdependent functions are simply linked one after the other in the link stage of the build process. As much as a 30 percent improvement in performance has been realized using this method, without a single line of code or their design being changed!

Although these methods do lead to improved software performance, they are not always enough to help to get the system performance where it needs to be. Where does a developer turn at this point? There are a number of issues that affect almost all real-time systems that can be considered. These issues, and their impacts, are described in the next section. In addition to considering the performance of an application during one-off measurements, the issues below will also impact the ability of a real-time system to sustain its execution speed or even its health.

7. Issues Affecting Software Performance

7.1 Dynamic Memory Usage

Each computer system has a finite amount of memory. During operation, this memory is separated into a number of regions, including:

> The code area that contains the executable code.

> A stack area that holds local variables and function return pointers.

> A data area where the global variables are stored.

> A heap area that holds dynamically allocated memory.

> A system area that contains information maintained by the operating system.

Dynamic memory allocations occur when the software makes a specific request for memory, such as when the `malloc()` 'C' library call is used. The memory that is allocated as a result of this request is held within the heap memory region. Similarly a deallocation, such as might result from the `free ()` call, will release memory held within the heap. The amount of memory consumed through dynamic memory allocations is determined during run-time, and is generally difficult to predict unless coding rules are used that prohibit the use of dynamic memory to a single allocation operation during system startup (common with Real-Time Embedded Systems and Flight/Mission Critical Systems).

Memory utilization is an important consideration in the design of any embedded system, with unique implications for real-time systems. Each memory allocation and deallocation request requires a finite amount of execution time to complete, which may be detrimental to performance if not managed properly. This situation is further exacerbated when memory fragmentation or memory leaks occur.

While poor use of memory can have a detrimental effect on the performance of a system, careful memory management during run-time and compile time can result in significant performance improvements.

7.1.1 Memory Fragmentation

Fragmentation of memory occurs in much the same way that file fragmentation occurs on a hard disk. When memory is allocated, the memory manager assigns the next block of available contiguous memory that is large enough to satisfy the allocation request. When the same block of memory is deallocated, then it is freed as a contiguous block. In a system that performs multiple allocations and deallocations of varying block sizes, a situation may result over time where the blocks of contiguous memory that remain are only large enough to satisfy an allocation request for a small block of memory. This means that an allocation request for a large block of memory will fail, even though summing all of the free memory space might indicate that there is enough available memory in total. If the free memory is not in one contiguous block, then it cannot be used to fulfill a single allocation request.

For real-time systems, memory fragmentation has serious performance implications long before reaching this critical stage. This comes about because memory allocations in a system with fragmented memory take longer than with those in a system without them. When memory fragmentation occurs, the MMU takes an inordinately long time to find a contiguous block of memory of the correct size to satisfy an allocation request. This can have an adverse effect on the ability of a system to satisfy its real-time objectives.

In many systems this fragmentation is hard to predict, and even harder to test for. There are, however, many standard designs that may be applied to protect against it. These designs include restricting dynamic memory usage to a single block of memory allocated during application initialization and allocating and deallocating memory in fixed block sizes. The latter schema ensures that whenever a block of memory is deallocated, enough memory is freed for any subsequent allocation to succeed.

7.1.2 Memory Leaks

Memory leaks are the cancer of software systems. They occur when an error in memory allocation/deallocation arithmetic causes a steady increase in memory consumption by an application over time. A good example of this is when a pointer is set to NULL before the memory block that it points to is freed, resulting in an allocated block of memory that cannot be freed, and hence not re-used. When an undetected memory leak occurs, this will ultimately cripple any application when all of the available memory is consumed.

The impact of memory leaks is so severe that modern software languages such as Java™ and ADA95 have introduced "Garbage Collection" mechanisms to protect against them. Garbage collection is the automatic recycling of dynamically allocated memory. A garbage collector recycles memory that it can prove will never be used again.

For real-time systems, Garbage Collection introduces a unique set of problems. Normally implemented at the operating system level, Garbage Collection algorithms must run to completion once started, during which time all

other activities are suspended. This can have a detrimental effect on the ability of a real-time system to meet its performance objectives. What makes this even more troublesome to real-time systems is that Garbage Collection algorithms cannot be scheduled using the normal tasking mechanisms. Its operation is therefore non-deterministic. If a programming language such as Java or ADA95 is used for a real-time system, then Garbage Collection is typically disabled.

Restricting dynamic memory usage to a single block of memory allocated during application initialization is not only a good method for preventing memory fragmentation, but also a good mechanism for preventing memory leaks. There is, however, a significant amount of development overhead in determining how much memory to allocate during initialization and in developing a memory manager to control access to this local "heap." Also, adding a run-time memory manager to an application will affect its performance and the memory space that it requires.

Whether designing a solution to prevent memory fragmentation, or to protect against memory leaks, the success of both relies on having data on how an application uses memory. Information on the allocation block sizes required by an application is essential to designing a mechanism to prevent memory fragmentation. Monitoring allocations and deallocations also reveals how much memory an application requires. Also, this type of analysis has been successfully used to detect memory leaks.

For real-time systems, memory analysis may also result from the same measurement technique used to monitor performance.

7.1.3 Measuring Memory Usage in Real-Time Applications

Memory analysis of a Real-Time application may be achieved using the same instrumentation and data collection mechanism described in section 5.4. During the same instrumentation process that is used for performance analysis, each memory allocation and deallocation statement is detected, and instrumentation is added to monitor the pointer to which memory is allocated to or deallocated from, in addition to the amount of memory consumed or released. This information is then sent to the data collection agent.

Using this method, each allocation and deallocation to and from the heap may be monitored. The data collected by this method provides information on the amount of memory that an application consumes, and the block sizes in which the allocations occur. This information may then be used as reference data to design against fragmentation, and to implement the 'allocate once and manage locally' type of schema described in the previous two sections. In addition, when this information is mapped onto source code, it may be used to detect memory leaks in executing code.

7.2 Multi-Tasking Designs

"(Analyzing and) debugging a multitasking application is more challenging than debugging a single-tasking application for the simple reason that, in a multitasking application, more that one thing is happening at a time. Tasks interact with one another—reading and writing global memory, acquiring and releasing synchronization objects—and sometimes, in those places where the activity of one task affects the activity of another, the two tasks can trip one another up. Add more tasks to the mixture, and you have more opportunity for trip-ups." [Grehan98]

Unlike function performance metrics within an application with a single thread of execution, the number of other tasks executing concurrently affect task performance metrics. For applications with a single execution thread, building test cases to measure the maximum execution time of a given single function is relatively straightforward. In a multitasking system, however, it is not appropriate to only measure the performance of any given task in isolation, as this is not enough to guarantee that the task will meet its performance deadlines when additional tasks are present.

In practice, there is no single metric that may be used to prove that a multitasked application will meet all of its performance objectives in all circumstances. There are, however, techniques such as Rate Monotonic Analysis (RMA) that can be used to understand and predict the timing behavior of real-time software systems.

Rate Monotonic Analysis (RMA) is a collection of quantitative methods and algorithms that allow engineers to specify, understand, analyze and predict the timing behavior of real-time software systems.

RMA provides the analytical foundation for understanding the timing behavior of real-time systems that must manage many concurrent threads of control. Real-time systems often have stringent latency requirements associated with each thread that are derived from the environmental processes with which the system is interacting. RMA provides the basis for predicting whether such latency requirements can be satisfied. Some of the important factors that are used in RMA calculations include:

> The worst-case execution time of each thread of control.

> The minimum amount of time between successive invocations of each thread.

> The priority levels associated with the execution of each thread.

> Sources of overhead such as those due to an operating system.

> Delays due to interprocess communication and synchronization.

> Allocation of threads of control to physical resources such as CPUs, buses and networks.

These factors and other aspects of the system design are used to calculate worst-case latencies for each thread of control. These worst-case latencies are then compared to each thread's timing requirements to determine if the requirement can be satisfied.

A problem commonly revealed as a result of rate monotonic analysis is priority inversion. Priority inversion is a state in which the execution of a higher priority thread is forced to wait for a resource while a lower priority thread is using the resource. Not all priority inversion can be avoided but proper priority management can reduce priority inversion. For example, priority inheritance is a useful technique for reducing priority inversion in cases where threads must synchronize.

Since RMA is an analytic approach that can be used before system integration to determine if latency requirements will be met, it can result in significant savings in both system resources and development time.

The accuracy of any RMA calculation is dependent on the fidelity of measuring the worst-case execution time of each thread of control. Using the same instrumentation and data collection mechanism described in section 5.4, highly accurate performance measurements may be made for each task. This then ensures that the results of any RMA calculation are as accurate as possible. With accurate RMA results, a designer may then make best use of the processing power available in the chosen CPU.

8. Conclusion

Whether they are soft or hard real-time, guaranteeing the performance of a real-time software system is not something that can be done in a single step. For most systems, meeting the required performance criteria is not something that can be achieved by some magical process that is applied once the code is complete. Careful consideration of the interaction between the system and its environment needs to be undertaken as early as possible, and captured in the Requirements and Design documentation. This environmental interaction information leads to the system's performance criteria that must then be fed into the design specs for each system component.

As real-time software is developed, performance must be measured at each step of the way. The execution speed of each component needs to be measured to ensure that they meet their performance criteria. During integration, the performance of the system needs to be continually measured for the same reason.

Whether a system is meeting its performance objectives, or not, there are many issues that need to be considered in order to ensure that the system continues to fulfill its requirements throughout its operating life. Dynamic memory usage and multitasking behaviors need to be addressed to ensure that the performance of a real-time system does not degrade with time.

The ability to measure software execution is essential to the successful development of real-time systems. Although Logic Analyzers and In-Circuit Emulators have been successfully used to measure the execution time of sections of code, they are restrictive when it comes to performance profiling, or for measuring multitasking applications. Software-only measurement methods, such as stack sampling or software data collection from instrumented code, can do performance profiling and monitor multitasking behaviors, but they are intrusive, and do not provide highly accurate performance measurements.

High fidelity software performance measurements may be achieved by using a combination of source code instrumentation and hardware data collection. In addition to performance analysis, this technology may also be used to monitor dynamic memory usage and multitasking behavior. Available with the CodeTEST product from Freescale, this technology takes software execution performance for real-time systems from the realms of possibility and places it firmly in the realm of reality.

freescale™
semiconductor