International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2019 IJSRCSEIT | Volume 5 | Issue 3 | ISSN : 2456-3307
DOI : 10.32628/CSEIT1953166

Evaluation of Embedded System Behaviour Using Full-System Software

Swati Singh
Department of Computer Science and Engineering, Saraswati Higher Education and Technical College of Engineering, Varanasi, Uttar Pradesh, India

ABSTRACT

With embedded processor technology moving towards faster and smaller processors and systems on a chip, it becomes increasingly difficult to accurately evaluate real time performance. This research describes an evaluation method using an embedded architecture software emulator that models the Motorola M-CORE processor architecture. This emulator is used to evaluate and compare the real-time performance of a public-domain experimental Real-Time Operating System (RTOS) against a bare-bones multi-rate task scheduler. The results of the experiment, as shown in arrival time JITTER, response-time DELAY, and CPU BREAKDOWN figures, show the trade-offs between job load, job frequency, and kernel overhead. This research suggests full-system software emulation to be a valid method of evaluating embedded systems' behavior and real-time performance.

Keywords : Embedded System, Real-Time Operating System , JITTER

I. INTRODUCTION

With embedded processor technology moving towards faster and smaller processors and systems on a chip, it becomes increasingly difficult to accurately evaluate real time performance. Probing a piece of silicon, or accurately measuring values approaching less than one nanosecond becomes more expensive and more difficult, if not impossible. It becomes necessary to find additional methods to evaluate and debug embedded system.

Embedded Systems

Embedded systems has become a buzz word in the last five years, but embedded systems and processors have been around for much longer than that. One only needs to look around to see embedded systems everywhere: cell phones, alarm clocks, personal data assistants (PDAs), automobile subsystems such as ABS and cruise control, etc. This section takes a look at embedded systems, the issues and tools involved in their design, current trends, and how they can benefit from the research performed for this report.

Hardware/Software Codesign

One of the methodologies gaining wide acceptance in both the embedded world and the general purpose world is that of Hardware/Software codesign. This section first defines the concept and then the methodology of Hardware/Software codesign. Then a slightly different method of codesign is described. This section is concluded with how

Hardware/Software Codesign can benefit from the emulator developed in this research.

Hardware/Software Codesign: The Concept

For years, designers have partitioned systems into hardware and software components that were developed separately. When this is done, the hardware designers usually make architectural choices early in the design process. These decisions are based on their knowledge of the hardware requirements and their limited knowledge and understanding of the software requirements. And they are usually hard pressed to go back and make changes to these choices. The result is that often the software designers are forced to make up for problems in the hardware through additional work of the software, often leading to a less than optimal overall design of the system.

The concept of Hardware/Software Codesign is that of both hardware and software designers work together to develop a system, whether that system be an embedded one, a general purpose one, or high performance one. From specification of the requirements to exploration of the design space, and from development of the physical design to the simulation and test of the final product, hardware and software designers work cooperatively, concurrently, and most importantly, they communicate.

Hardware/Software Codesign: The Methodology

In response to these problems listed above, designers as well as EDA tool manufacturers are moving towards a design methodology that has hardware and software engineers working together from the beginning of the specification phase all the way through simulation and test. In hardware/software codesign, designers from both disciplines integrate their work. The process begins with a functional exploration of the project that they are undertaking. The designers define requirements and create a working specification. Then the hardware and software designers work together to map this specification on hardware and software architectures. The designers then implement these architectures onto silicon and code and come back together to simulate and test. The entire process benefits from open communication from both sides

Real-Time Operating Systems

Real-Time Operating Systems (RTOS) are commonly used in the development, productizing, and deployment of embedded systems. Unlike the world of general purpose computing, real-time systems are usually developed for a limited number of tasks and have different requirements of their operating systems. This section first gives the requirements of real-time operating systems, then breaks down the internals of RTOSs and explains them in detail. This section concludes with how the emulator developed in this research would aid in the evaluation of RTOSs.

.

The SimOS Approach

SimOS is a full-system simulation environment that is capable of modeling computer hardware in enough detail to run a complete operating system, and all of the applications running on that operating system, on top of it. The SimOS project started in 1992, and was built to study the execution behavior of modern workloads. It is capable of studying both uniprocessors and multiprocessor systems and is used to study and evaluate the performance of high-performance and general purpose computers.

The SimOS environment is a simulation layer that runs on top of general-purpose Unix multiprocessors such as the Silicon Graphics Inc. Challenge series [40]. On top of that general purpose multiprocessor

system is the operating system running on that hardware, and in this case, it is IRIX version 5.x. On top of this software is run the SimOS environment. The SimOS environment takes in a hardware description file and is capable of modeling uniprocessors, multiprocessors, RAM, Ethernet, hard disk, and other pieces of hardware associated with today's hardware platforms. On top of the SimOS is run an operating system that has been ported to the hardware platform that the SimOS environment is currently modeling. Finally, on top of that operating system is run the unaltered applications programs. All of this can be seen in Figure 1.

One of the advantages of the SimOS operating systems is that it allows the user to choose which level of output detail in which to simulate. The system offers a simple trade-off of speed versus detail of simulation. If the user is interested in obtaining detail simulation results of a particular program, SimOS employs slower, more detailed simulation. And when the user wishes to run an application for long periods of time

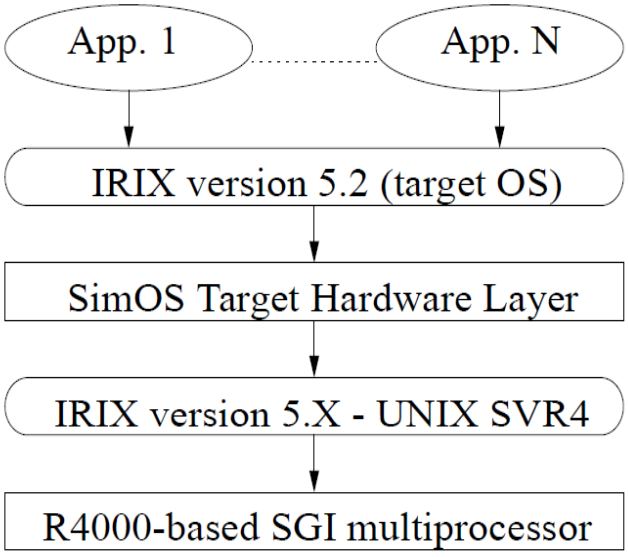

Figure 1: The SimOS Environment

This figure shows the layout of the SimOS development environment. The SimOS target hardware layer runs on top of a Unix Operating System running on a R4000-based SGI multiprocessor workstation. On top of the SimOS environment is run the target Operating System and any applications that are run on top of that Operating System.

Instead of for detailed simulation results, SimOS can scan over unimportant parts of the workload. Also, SimOS allows the user to modify this choice on-the-fly. The user can choose certain sections of code that he is interested in seeing the simulation results for, and scan over the rest of the code as unimportant.

Emulator

For this project, Motorola's M-CORE architecture was used as the model architecture for our emulator. This architecture was chosen because the M-CORE architecture is one of the cutting edge embedded processors on the market today, and the M-CORE was designed for high performance and low power operation. In this chapter, first the M-CORE architecture is described, followed by the specifics of the emulator, how it works, what information it takes as an input, how it processes that information, and what information it outputs during the emulation. Finally, the method used to validate the emulator is described. Figure 2 shows a system view of the emulator and both the hardware and operating system that it is running on and the Real-Time Operating System and applications that are running on it.

M-CORE Architecture

The Motorola M-CORE architecture is a 32-bit Load/Store architecture with a fixed 16-bit instruction length and 32-bit data length. Figure 3 shows all of the available instruction formats in the M-CORE architecture. It has a 16 entry 32-bit general register file, a 16 entry 32-bit alternate register file to allow fast interrupt support, and a 13

entry control register file accessible only by the supervisor mode. Its execution pipeline's four stages are completely hidden from the application software. Most instructions execute in a single cycle with two cycle execution for loads, stores, and taken branches and jumps.

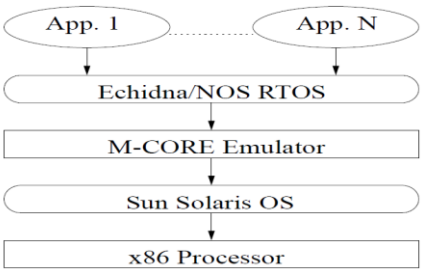

Figure 2: The Emulation Environment

This figure shows the emulation environment developed in this work for real-time system evaluation. The emulator runs on top of the Sun Solaris Operating System, running on top of a x86 Processor. On top of the emulator is run the target Real-Time Operating System, either Echidna or NOS. On top of the RTOS, the benchmark applications are run.

The address space is byte, half word, and word addressable, and allows both fast and normal interrupts, allowing those interrupts to be either vectored and auto vectored interrupts. The pipeline for the M-CORE consists of four stages: instruction fetch, instruction decode/register file read, execute, and write back. All of these stages operate simultaneously, making single cycle instructions possible. All sixteen general purposes registers can be used as source operands and instruction results (i.e. it is an orthogonal register file).

Echidna RTOS

Echidna is a cooperative multitasking Real-Time Operating System that is based on the Chimera [48] operating system developed at the Advanced Manipulators Laboratory at Carnegie Mellon University. A smaller version of Chimera (˜6KB footprint), Echidna swaps Chimera's POSIX-like threads in the microkernel for port-based objects and supports reconfigurable component-based software for microcontrollers and digital signal processors [10].

The traditional coding method used by most of today's real-time operating systems is that processes are created, each with their own main(). Each of these processes executes their own user code and controls the flow of the program. This process calls upon the operating system whenever an operating system service is needed. These services include communication, time control, the creation of new processes, and synchronization.
The port-based object method, on the other hand, gives a consistent structure for every process, and thus operating system services as listed above are performed in a predictable manner. Only when necessary, the operating system calls a port-based object's method to perform user-defined functions.

In this port-based object model, each independent object does not need to explicitly communicate or synchronize with any other component in the system, making integration very easy. When an object needs information, it obtains that information from its input ports. When that object generates information that needs to be passed on to either another process, or to a future invocation of itself, it sends that information to its output ports. The information on these ports is stored in shared memory so data can be sent between objects.

Echidna was designed to support dynamically reconfigurable real-time software and was targeted to run on 8 to 32 bit microcontrollers as well as DSPs [49]. Like Chimera, Echidna provides cooperative multitasking, but unlike Chimera, it offers a good deal of functionality in a relatively small footprint,

and therefore it is a good candidate for this study in real-time performance in embedded systems.

Non-Operating System

The Non-Operating System (NOS) is a bare-bones, fixed priority, multi-rate executive similar to a real-time operating system that an embedded-systems designer might create on the fly. Although not a full operating system, just a task scheduler, NOS represents the attainable performance limit of a non-preemptive RTOS.

A multi-rate executive scheduler was chosen over something simpler, such as a basic cyclic executive or a time-driven cyclic executive scheduler [27], because we needed the ability to have several jobs that had different frequencies, and the timing of the multirole executive is less independent on the code size of the jobs run upon it, which is important because each of the benchmarks to be run is contains a different amount of code.

JITTER

JITTER measurements represent the time deltas between successive outputs seen at the I/O ports for a given task. When more than one task is running, each task is assigned a separate I/O port to write to, enabling the distinction between tasks. On those runs, the jitter information for each of the tasks is combined into a single set of data points.

All of the graphs shown are probability density graphs, centered on the desired period. Negative numbers along the x-axis represent tasks that have run early, and positive numbers represent tasks that have run late, in relation to the previous task. To maintain readable graphs, only non-zero y-values have been shown, and all of the values have been grouped into 100ms intervals.

Delay

The delay measurements represent the time between an external interrupt generating an aperiodic IPC and the corresponding output to an I/O port from the responding thread. Therefore, this delay measures the response time of the system in terms of when the first reaction to an interrupt could take place.
Neither Echidna nor NOS handles interrupts preemptively; both use a polling technique. The difference is that Echidna has a periodic thread that is scheduled to run every 1ms to check for an interrupt, and if one is found, respond to it; NOS checks to see if an interrupt has occurred only when the system is idle: If the system is either busy or overloaded, an interrupt will be ignored, perhaps indefinitely, unless the system returns to an idle state and checks to see if an interrupt is waiting.

An important difference to note between the values obtained for the Jitter graphs and the values obtained for the Delay graphs is that the values on the delay graphs are grouped into intervals of 10ms, instead of 100ms like in the Jitter graphs. This is done because in many of the Delay graphs, all of the values would fit into the first 100ms, but would give several points in a 10ms interval graph.

CPU Breakdown

The CPU breakdown graphs show the amount of time spent by the system in kernel, application, interrupt handling, and idle portions of the code. Two different types of graphs are those with a constant task period, while varying the number of tasks that are being run; those with a constant number of tasks running, while varying the frequency that those tasks are running at. On each graph, there are three distinct groups of data. The first group is the calculated theoretical limit of a system running the application code. This group only contains application and idle segments, and the

values are calculated by multiplying the number of tasks to be run at that frequency by the time it takes to run a single task. The second group of bar graphs shows the CPU breakdowns for the runs on NOS. The final group of bar graphs is the CPU breakdowns for those runs on the Echidna RTOS. As with the Delay graphs, only the results from simulations of IPC and FIR are shown, as the results from UP and DOWN fall in between them.

Analysis Summary

This experiment has evaluated three aspects of real-time behavior: jitter, delay, and CPU breakdown. With Jitter, it was observed that as the number of tasks is increased, the amount of scheduling overhead incurred is increased, more so with Echidna than NOS. With IPC, UP, and DOWN, the limit for Echidna is reached when 8 tasks are running at periods of 1ms (or 2/1ms, or 1/2ms), while NOS can continue to run on time for periods lower than that of Echidna's limit. For FIR, scheduling overhead is only one factor in calculating the limit, and both NOS and Echidna reach a limit of 8 tasks running at 2ms, or 4 tasks running at 1ms. For Echidna runs, if there is any background load, the data points start to move away from the origin, but the average run is still on time. For NOS, the background has very little affected.

With delay, when a system has a light load, both Echidna and NOS are able to service the interrupt immediately (within 1ms is as fast as Echidna can check the interrupt). However, if the system is running with a significant load, Echidna can take up to four times as long to service the interrupt, and NOS has the possibility of dropping the interrupt entirely. Addition of the control loop has very little affected on these characteristics.

With CPU breakdown, several things were seen. Interrupt handling overhead was insignificant because both RTOSs use polling. On the systems where applications are not computationally intensive it is cheaper to run fewer applications at a faster period than to run more applications at a slower period. And once the system is overloaded it gravitates to an optimal ratio of kernel versus user time.

Conclusions

This report has presented a method of using full-system emulation to evaluate the real-time performance of an embedded system. An embedded architecture emulator was created, using the C programming language, that emulates the Motorola M-CORE embedded processor down to the register level and is accurate to within 100 cycles per million as compared to actual hardware. With tests and experiments run on this emulator, the goal of this report was to show that this method can be successfully used in the evaluation of embedded systems.

A study of non-preemptive real-time operating systems was presented, focusing on Echidna, a small, public domain RTOS, and comparing it to NOS, a bare-bones scheduler that represents the performance limit for non-preemptive RTOSs. Three different real-time performance characteristics were measured: JITTER, DELAY, and CPU USAGE.

With Jitter, it was observed that as the number of tasks was increased, the amount of scheduling overhead incurred was increased, more so with Echidna than NOS. With IPC, UP, and DOWN, the limit for Echidna is reached when 8 tasks are running at periods of 1ms (or 2/1ms, or 1/2ms), while NOS can continue to run on time for periods lower than that of Echidna's limit. For FIR, scheduling overhead is only one factor in calculating the limit, and both NOS and Echidna reach a limit of 8 tasks running at 2ms, or 4 tasks running at 1ms. For Echidna runs, if

there is any background load, the data points start to move away from the origin, but the average run is still on time. For NOS, the background has very little affected. With delay, when a system has a light load, both Echidna and NOS are able to service the interrupt immediately (within 1ms is as fast as Echidna can check the interrupt). However, if the system is running with a significant load, Echidna can take up to four times as long to service the interrupt, and NOS has the possibility of dropping the interrupt entirely. The addition of background load has very little affected on these characteristics. With CPU breakdown, several things were seen. Interrupt handling overhead was insignificant because both RTOSs use polling. On the systems where applications are not computational-intensive, it is cheaper to run fewer applications at a faster period than to run more applications at a slower period. Once the system is overloaded it gravitates to an optimal ratio of kernel versus user time.

II. CONCLUSION

All of the results obtained in this report could have been obtained using other methods, such as using a logic analyzer to obtain those signals that leave the chip (i/o signals) or using breakpoint instructions to bring off-chip those signals that do not normally leave the chip (register contents). However, those signals that could be obtained with the logic analyzer can only be obtained in this particular instance because an evaluation board of the M-CORE was used in which the components were discrete parts on a printed circuit board, rather than logic blocks on an integrated circuit. The M-CORE processors used in industry are systems on a chip, and therefore those signals would not leave the chip. For those signals that are brought off-chip using the breakpoint instruction, this incurs its own penalty, both slowing the system down, as well as modifying some of the register values. This report is a tool thesis. It presents the emulator, describes how it works, and then provides an experiment to validate it.

With the tests and experiments run on this emulator, the report and the research that has lead up to it has shown that this method can be successfully used as an additional method in the evaluation of embedded systems.

III. ACKNOWLEDMENT

The author is grateful to Department of Computer Science and Engineering, Saraswati Higher Education and Technical College of Engineering, Varanasi for rendering their support and help for completing the work.

I. REFERENCES

[1] L. Abeni and G. Buttazzo. "Integrating multimedia applications into hard real-time systems." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1998.

[2] J. H. Anderson, et al. "Efficient object sharing in quantum-based real-time systems." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1998.

[3] T. Anderson. "System-on-chip design with virtual components." Circuit Cellar, No. 109, pp. 12-19, August 1999.

[4] M. J. Bach. The Design of the Unix Operating System. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[5] S. R. Ball. Embedded Microprocessor Systems: Real-World Design. Newnes, Butterworth-Heinemann, Boston MA, 1996.

[6] L. A. Barroso, et al. "Memory system characterization of commercial workloads." In Proc. 25th Annual International Symposium on Computer Architecture (ISCA '98), Barcelona, Spain, June 1998, pp. 3-14.

[7] A. Bestavros and S. Nagy. "Value-cognizant admission control for RTDB systems." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1996.

[8] M. Brockmeyer, et al. "A flexible, extensible simulation environment for testing real-time specifications." In Proc. IEEE Real-Time Systems Symposium (RTSS), 1997.

[9] D&T Roundtable. "Hardware-Software codesign." IEEE Design and Test of Computers, Vol. 14, No. 1, pp 75-83, 1997.

[10] Echidna. Echidna: A Real-Time Operating System to Support Reconfigurable Software on Microcontrollers and Digital Signal Processors. Software Engineering for Real-Time Systems Laboratory, University of Maryland, http://www.ece.umd.edu/serts/research/echidna/index.shtml, 2000

Cite this article as :

Swati Singh, "Evaluation of Embedded System Behaviour Using Full-System Software", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 5 Issue 3, pp. 521-528, May-June 2019.
Journal URL : http://ijsrcseit.com/CSEIT1953166