The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems

Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh,
Paul Kohout, Christine Smit, Tiebing Zhang, and Bruce Jacob

Dept. of Electrical & Computer Engineering
University of Maryland at College Park
College Park, MD 20742
http://www.ece.umd.edu/~blj/embedded/

{ktbaynes,chriscol,ericf,brinda,pkohout,csmit,zhangtb,blj}@eng.umd.edu

ABSTRACT

This paper presents the modeling of embedded systems with SimBed, an execution-driven simulation testbed that measures the execution behavior and power consumption of embedded applications and RTOSs by executing them on an accurate architectural model of a microcontroller with simulated real-time stimuli. We briefly describe the simulation environment and present a study that compares three RTOSs: µC/OS-II, a popular public-domain embedded real-time operating system; Echidna, a sophisticated, industrial-strength (commercial) RTOS; and NOS, a bare-bones multi-rate task scheduler reminiscent of typical "roll-your-own" RTOSs found in many commercial embedded systems. The microcontroller simulated in this study is the Motorola M-CORE processor: a low-power, 32-bit CPU core with 16-bit instructions, running at 20MHz.

1. INTRODUCTION

With embedded systems moving toward faster and smaller processors and systems on a chip, it becomes increasingly difficult to accurately quantify embedded-system behavior. Probing a piece of silicon, or accurately measuring timing values down to a nanosecond or less becomes more expensive and more difficult—in some cases impossible [21, 28]. This poses a serious obstacle for future systems design.

There are three recent trends relevant to this observation. First is the increasing popularity of hardware/software cosimulation or codesign [17, 1]. As opposed to developing the hardware and software for a system separately, the hardware/software codesign methodology realizes the advantages of designing the two together. Doing so provides benefits in performance, reliability, and time to market, due to the observation that when hardware and software designers communicate during the design process, there is less chance of problems arising due to ignorance [23].

Another trend is the use of real-time operating systems [20, 8, 35]. Their benefits are well known: they provide numerous facilities including cooperative and preemptive multitasking, support for both periodic and aperiodic tasks, semaphores, inter-process communication, etc. In doing so they can dramatically reduce the time to design, develop, and test a product [13, 2, 14].

The third trend is the rapidly growing demand for computing devices that are both compute-intensive and battery operated, for example PDAs, wearable computers, and laptops [4, 26].

These three trends meet at a simple, clear conclusion: It is prudent to have a simulation-based experimental environment for real-time embedded systems, but, if the model is to be truly useful for developing modern embedded systems, it must be accurate enough to run unmodified real-time operating systems, and it must accurately characterize the energy consumption of the system. High-level language modeling of applications and their operating systems has been performed by the SimOS group [27], and there has been a large number of recent studies modeling the power consumption of microprocessors and applications [32, 18, 24, 15, 16, 11, 5, 36, 31, 25, 10], but this is the first study of which we are aware that performs both.

1.1. SimBed

Our group has developed *SimBed*, a high-level-language model of an embedded hardware system that runs unmodified real-time operating systems (i.e., the binary that runs on the simulator is the same binary that runs on real hardware). In this study, we present a processor model written in C that emulates the M-CORE microcontroller, a low-power, 32-bit CPU core with 16-bit instructions [33, 34]. All devices, interrupts, and interrupt handlers used by the operating systems and applications are accurately simulated. The model has been verified as cycle-accurate to within 100 cycles per million compared to actual hardware (the difference is due to a handful of variable-latency hardware instructions such as multiplication that, for simplicity, we model as having constant latencies).

We have also instrumented the processor simulator to measure energy consumption, using existing instruction-based techniques [31]. We have verified the simulator's output to measurements of actual hardware, and our results are within 10–15% of real measurements. This level of accuracy for modeling power at the processor level is about where current research stands (e.g. [5, 31]).

An interesting side note is that some of SimBed's measurements represent quantities that cannot be obtained via traditional means (e.g., probes and logic analyzers) on current M-CORE chips without perturbing the observed system, as M-CORE offerings all use on-chip memories. For example, the division of time and energy into kernel, user, idle, and interrupt-handler components could be

CASES'01, November 16-17, 2001, Atlanta, Georgia, USA.

obtained by either instrumenting code or using off-chip memory and a logic analyzer, but both schemes would change the system's execution time and energy consumption.

1.2. Experiments & Results

This paper presents an experimental study using *SimBed* in which the performance and energy consumption of three different RTOSs are compared: a public-domain preemptive multitasking kernel, an industrial-strength cooperative multitasking kernel, and a bare-bones task scheduler (which represents the limiting case of a light-weight cooperatively-scheduled RTOS).

The benchmarks executed on each operating system are periodic apps, for which an absolute deadline is less important than relative deadline—i.e. these are applications for which a 500Hz task requires its $i+1^{th}$ invocation to run exactly 2ms after its i^{th} invocation and could care less whether the very first invocation started at time t_0 or t_0 plus some small delta, provided no dependence relations are violated. These applications have slightly different goals than traditional real-time applications; for instance if the RTOS schedules a 500Hz task to run every 2ms, but the task is executed exactly 1ms "late" on every invocation, then—as far as the outside world is concerned—it is a 500MHz task that is on-time every invocation. Thus, the measure of an RTOS's effectiveness in executing these applications can be determined by external observation; one does not need to know the contents of the scheduler's data structures to determine whether a periodic application is invoked on-time or not.

Our performance measurements yield both predictable and surprising results. Predictably, as system load is increased, the RTOSs hit their job deadlines consistently until a critical system load is reached, beyond which point the RTOSs miss deadlines with increasing frequency and by increasing amounts of time. The surprising results include situations where the industrial RTOSs miss deadlines with predictable regularity and with probability 1, even when the system is under light load. In general, to ensure on-time task invocations in the face of unpredictable events (e.g., external device interrupts), an RTOS must maintain significant CPU headroom: 10–20% idle CPU cycles is not too much.

The energy-consumption measurements demonstrate a trade-off that the more complex RTOSs seem to have taken: while the bare-bones scheduler has the lowest energy consumption, that consumption scales with the workload. The more complex RTOSs have a higher initial energy consumption, but this consumption does not increase as quickly as the user-level computational load. Therefore, the energy consumption and CPU requirements of these systems are likely to be much more predictable than a simpler RTOS.

2. METHODOLOGY

This study characterizes the real-time behavior and power-consumption break-down of two industrial-strength RTOSs and a simple scheduler:

uC/OS-II: A preemptive multitasking RTOS that is in the public domain [19]. It is ROMable and scalable (only modules that are needed are compiled into the executable). Execution times of all kernel functions and services are deterministic. Despite its small size (1700 lines of code), it offers such services as mailboxes, queues, semaphores, time-related functions, etc. It is chosen to represent sophisticated preemptive multitasking RTOSs with footprints small enough for microcontroller systems.

ECHIDNA: A cooperative multitasking RTOS based on Chimera [29] that swaps Chimera's POSIX-like threads in the microkernel for port-based objects [30]; it supports reconfigurable component-based software for microcontrollers and digital signal processors [9]. This is chosen to be representative of sophisticated dynamic-priority cooperative RTOSs with footprints small enough for microcontroller systems (Echidna has a footprint of ~6KB).

NOS: A bare-bones, fixed-priority, multi-rate executive based on descriptions of "roll-your-own" RTOSs given by embedded-systems designers in industry [12]. Though it is just a task scheduler and not a full OS, we refer to it in this paper as an "RTOS" for convenience. It is chosen to represent the attainable energy and performance limit of non-preemptive RTOSs.

On each of these, we execute several application kernels that exploit multitasking to the extent possible in the given OS (µC/OS provides preemptive multitasking, Echidna provides cooperative multitasking, and NOS schedules work on function boundaries) and use for all data transfer whatever inter-process communication mechanism is supplied by the RTOS. Within a task, we stress the RTOS's communication mechanism by having different independently scheduled jobs read the input and write the output; i.e., the same job does not perform both reads and writes to the I/O system. Therefore, the minimum workload for any application is a task of two independently scheduled jobs (terminology from Liu [22]).

The application kernels include *raw IPC* (both periodic and aperiodic), *up-sampling*, *down-sampling*, and a 128-tap *FIR filter*. Background load in the form of aperiodic interrupt-driven tasks and a control loop performing administrative work makes the system less predictable and thus makes life more difficult for each scheduler.

Control Loop: This task runs in the background at a period of 32ms to simulate the background load that many embedded systems have running while they are performing other tasks, such as a cell phone that has a task that runs periodically to refresh its LCD display. This control loop performs several memory lookups with an index that is randomly generated.

Aperiodic IPC: An I/O interrupt is generated by hardware, and a high-priority user-level job is scheduled in response that writes to the I/O space. This is the mechanism used to determine system response time under load. The interrupt inter-arrival times obey a geometric distribution: the emulator generates an interrupt every 100µs with a probability of 0.01, giving an average of 100 interrupts a second.

Note that the same application code is executed on all three operating systems (with minor RTOS-specific modifications). The experiments keep track of real-time jitter, response-time delay, and total CPU energy consumption divided into *user*, *kernel*, *handler*, *semaphore*, and *idle* components.

Jitter: Jitter is measured by keeping track of inter-arrival times of periodic output. For example, if a task is scheduled to generate an output every ten milliseconds, its average inter-arrival time should be ten milliseconds. Any variation in the inter-arrival time represents output that fails to arrive on time.

Note that this differs slightly from the traditional definition because if a scheduler happens to execute a task consistently *late*, it will nonetheless appear *on-time* to the external world.

Delay: Delay is measured by keeping track of the time between actions in aperiodic stimulus-response pairs. In the *aperiodic-IPC* workload, we keep track of the delay between the I/O interrupt that signals the input and the time that the application output is received at the I/O system (as opposed to

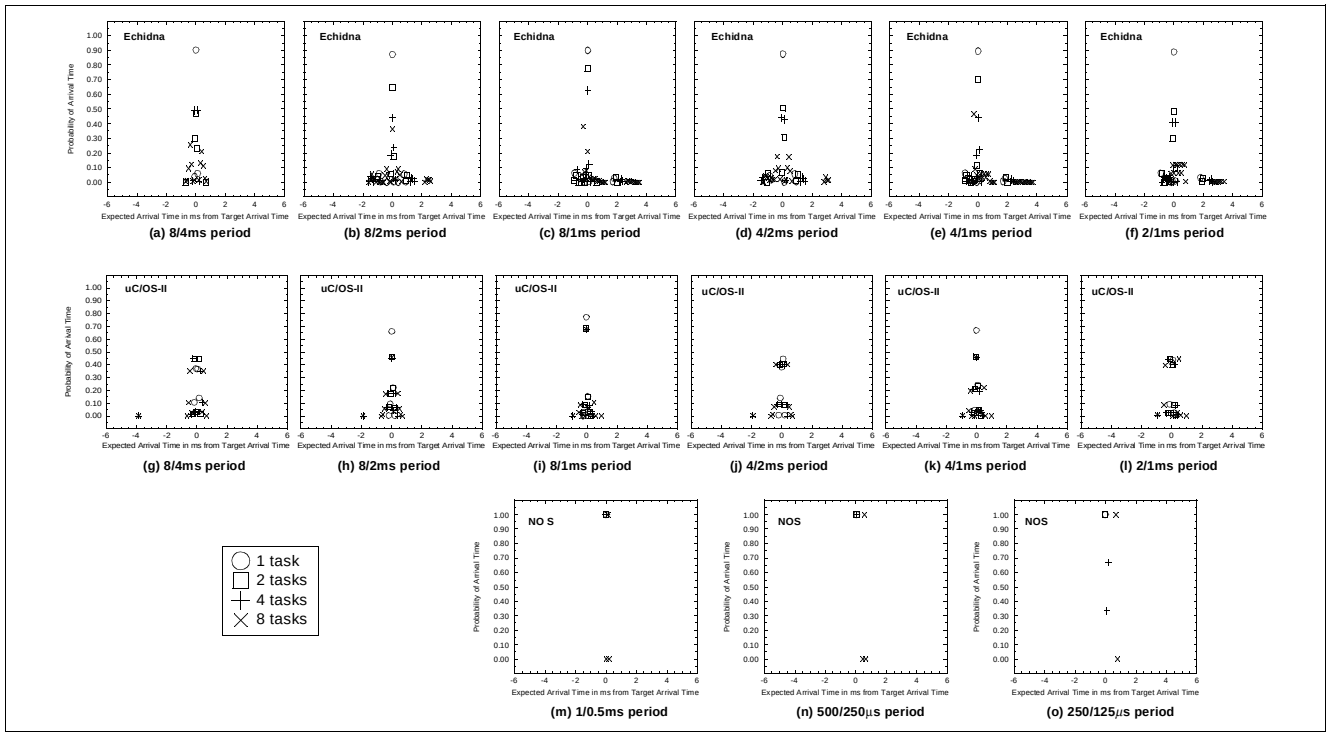

Figure 1: JITTER probability density graphs for UP. The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks.

the time that the handler is invoked or the moment that the output to I/O system is initiated). This represents the response time of the system as a function of system load.

Note that this differs significantly from traditional definitions of interrupt latency, which characterize a system by the time interval from raising the interrupt to executing the handler for that interrupt. Moreover, traditional measurements of delay give a single number, whereas we present a distribution.

Energy consumption: Energy consumed is tagged with the currently executing instruction's program counter, indicating what function in the system is being executed. The execution time for NOS and Echidna is divided into *user*, *kernel*, *handler, semaphore*, and *idle* components. The execution time for µC/OS is divided more finely, including *idle, user, event handling, semaphore management, time management, context switching, interrupt handling, interrupt disabling and enabling, thread scheduling, task management (creation/deletion/etc.)* and *initialization*.

More detail on the MCORE processor, SimBed's internals, applications, and RTOS models can be found elsewhere [37, 6, 3].

3. EXPERIMENTAL RESULTS

3.1. Real-Time Jitter

Our jitter data is shown in probability density graphs centered on the expected period. Data points at positive x-coordinates indicate late execution; data at negative x-coordinates indicate early execution. To keep the graphs readable, only non-zero y-values are shown. Note that the probability density graphs do not smooth out as more data is collected; for example, there are only minimal differences in graphs generated from 50 million data points as compared to graphs generated from 1 billion data points. When multiple tasks are executing simultaneously, each writes to a different I/O port, enabling the distinction between tasks, and each task contributes equally to the data in the graphs.

Figure 1 presents the jitter measurements for the up-sampler running at 2:1, 4:1, and 8:1 ratios; the results are fairly representative of all benchmarks. The graphs show spikes of data points, usually centered at zero (indicating an on-time arrival of output I/O), with any number of data points on either side of the spike. The height of a data point indicates the probability of seeing that time delta—for instance, Figure 1(a) shows that when Echidna is running one task (consisting of an 8ms job and a 4ms job), output arrives on-time roughly 90% of the time; 5% of the time output is a little late (off by about 0.1 millisecond), and 5% of the time it is a little early. With two tasks, output is on-time 45% of the time; 28% of the output is a little early, and 22% is a little late, and a small amount of output is early/late by thee-quarters of a millisecond. With four tasks, no output is exactly on time; roughly 50% of the output is a little early and 50% is a little late. A very small percentage is early/late by 0.2 ms. With eight tasks running, there is a clear bimodal distribution that shows roughly symmetric data peaks centered at about 0.2ms early and 0.2ms late.

An interesting result seen in the graphs is that, even at light workloads (e.g. tasks running with 8ms periods, also seen in other benchmarks with tasks running at 16ms), Echidna and µC/OS execute a number of jobs too late—and usually an equal number of tasks too early. Moreover, the number of early/late job invocations does not seem to scale with the workload (for example, µC/OS at task periods of 8/4ms cannot get more than 50% of the tasks to execute on-time, but more than 50% do execute on-time when the second task is running four times as often (compare Figures 1(g) and 1(i)). This behavior is caused in both RTOSs by task self-interference. This is specific to tasks with jobs that run with different

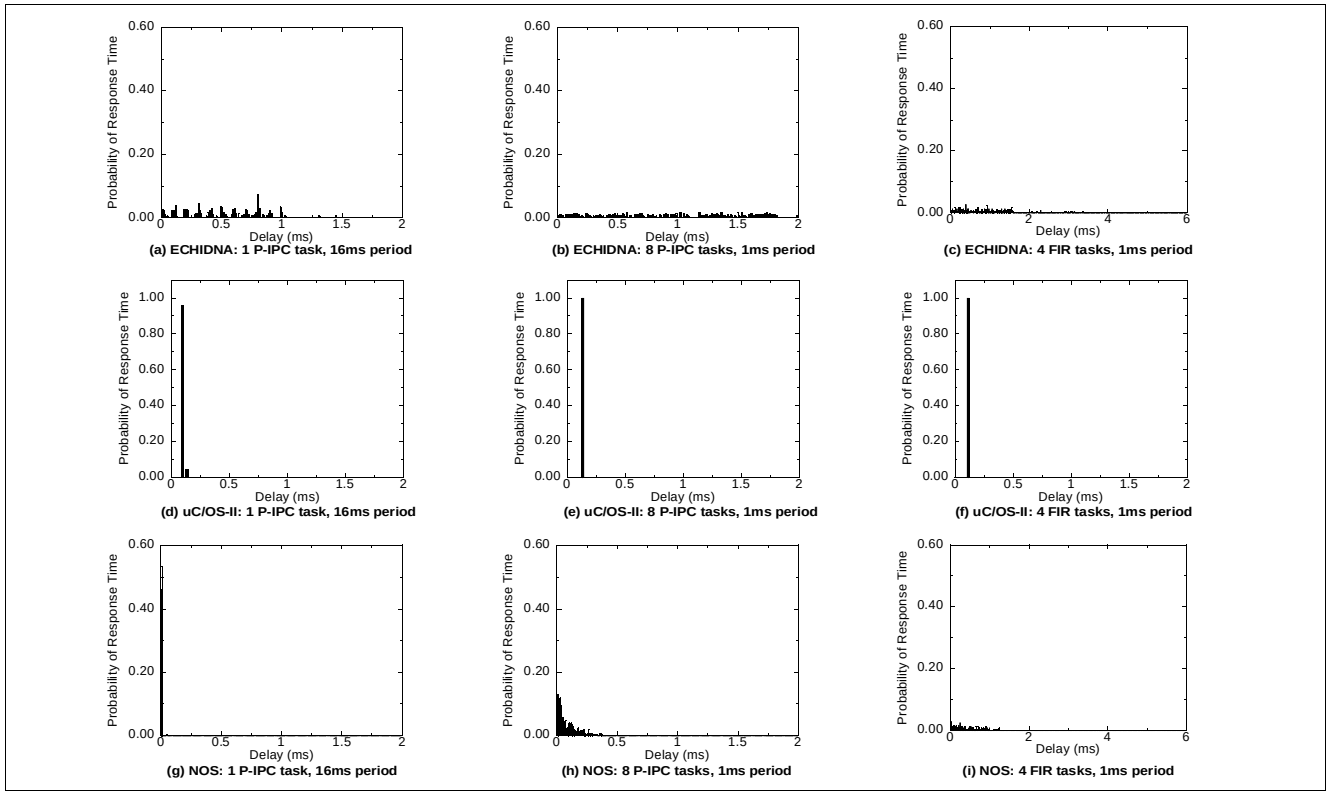

Figure 2: DELAY probability density graphs for IPC and FIR. The x-axis represents time between an interrupt being generated by an I/O device and the corresponding output to an I/O port of the responding thread. The y-axis indicates the probability of each delta. All measurements are for configurations with both types of background load (32Hz periodic control loop and aperiodic interrupt-driven IPC)—these delay measurements are for the interrupt-driven IPC that is the background load. Results range from little foreground load (1 IPC task) to heavy foreground load (4 FIR tasks). Note that the y-axis scale is different for the uC/OS graphs and that the x-axis scales are different in figures (c) and (i).

periods; when the periods are not relatively prime, job invocations coincide in time every Nth invocation. If the RTOS fails to distribute the workload appropriately, the system experiences a traffic jam every Nth invocation, resulting in late executions for many of the jobs. Thus, we see that the larger the ratio between the two periods, the fewer instances of traffic jams, even if the total workload increases. This also means that when different jobs periodically all want the same invocation time, the traffic jams will happen with probability 1, even if the workload is light.

This type of early/late behavior is not confined to self-interference, however. We saw the behavior in all applications studied; the presence of background load that occasionally (but not always) intrudes on execution time also causes regular traffic jams. In Echidna, the background control loop is a periodic task with period 32ms. The control loop is executed every other job invocation when run against 16ms tasks, every fourth job invocation when run against the 8ms tasks, etc. Whenever the control loop runs, it pushes the actual invocation times of other jobs out slightly so that they run late and then early on the next invocation. Therefore, 16ms tasks are upset by the disturbance more than 1ms tasks, even though they represent a higher system workload.

The disturbance in µC/OS is the aperiodic IPC interrupt that happens on average every 10ms. Because µC/OS is preemptive, the task invoked by the interrupt handler has a higher priority than any of the periodic application tasks, so it preempts application threads whenever it runs. 16ms tasks are upset most by this (in other benchmarks not shown), because the interrupt displaces a user thread on roughly every other job invocation (thus, only 50% of the job invocations are on-time). As the user threads execute more frequently, the interrupt preempts user threads with decreasing frequency, and we see that more job invocations are on-time, even though the system load has increased.

The systems would clearly benefit from better load distribution. For example, if the future job invocations were scheduled relative to the *actual* job invocation time rather than the *intended* invocation time, the system would naturally spread out the jobs, and it would only have late invocations during the first round of invocations. Neither µC/OS nor Echidna manages to spread the tasks out in time. In contrast, NOS schedules tasks relative to their actual invocation time. Thus even if a task runs late the first time, the following invocations will be on time. However, this is not a panacea: as the workload increases this relative scheduling cannot prevent late invocations, as NOS is non-preemptive, and therefore low-priority tasks can delay high-priority tasks.

3.2. Response-Time Delay

Our delay numbers represent the time between an Aperiodic-IPC interrupt and the moment that the I/O system sees the corresponding output from the user-level task invoked as a result of the interrupt. Thus, the delay measures the response time of the system in terms of when the first physical reaction to an external stimulus could take place.

The µC/OS-II kernel handles interrupts preemptively; both Echidna and NOS use polling. The difference between Echidna and NOS is that Echidna supports only periodic tasks and will not spawn a new task as a result of an interrupt; this must be done by a periodic application task. Therefore, our Echidna interrupt-handler

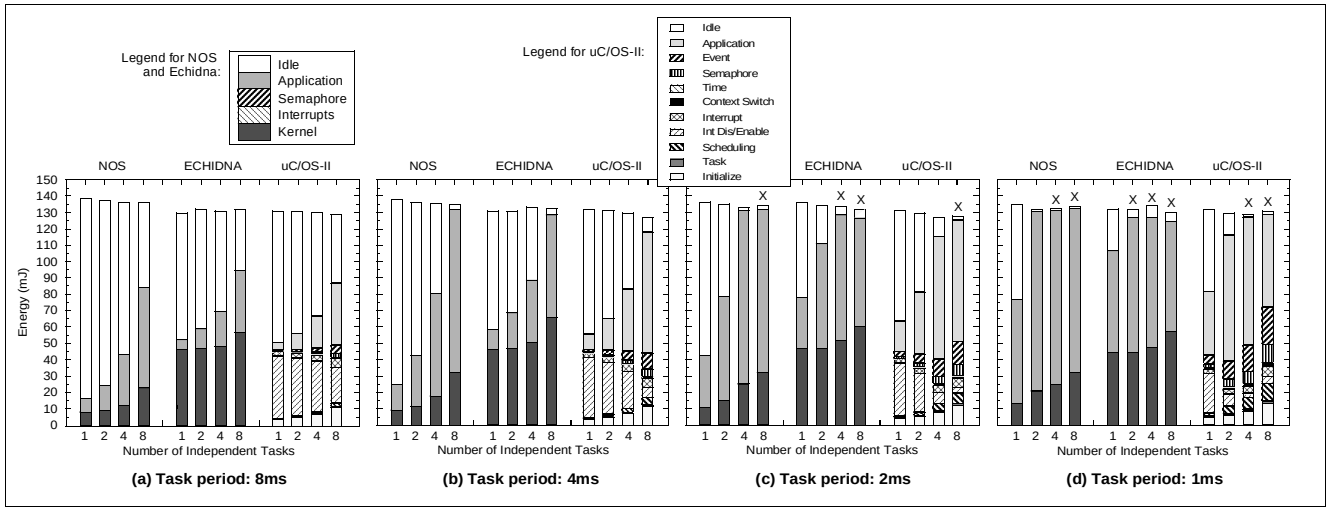

Figure 3: ENERGY CONSUMPTION graphs for FIR. The x-axis represents increasing workloads, as a result of increasing the number of executing tasks or. The y-axis represents the total CPU energy consumption and breakdowns for how much energy is consumed by executing kernel code, executing user application code, handling interrupts, performing semaphore handling, and sitting idle. "X" at the top of a bar represents a configuration that missed a significant portion of deadlines. Note that "idle" includes both time sleeping as well as some loop overhead in the main loop and parts of the timekeeping code for Echidna.

task is periodic with the shortest period supported by Echidna, 1ms, and it simply checks for IPC-related interrupts whenever it executes, sending output to an I/O port whenever it finds that such an interrupt has happened. NOS treats interrupts as tasks with a fixed priority (LOW). When an interrupt occurs NOS first finishes the currently active task, if any, and then looks at the ready queue. If there are no ready tasks with a higher priority than the interrupt handler, NOS services the interrupt. Thus, at light workloads an interrupt gets serviced almost at once. With a heavier workload, this response time can vary from very low to very high depending on what the instantaneous workload is when the interrupt occurs.

The delay times are shown in Figure 2. These represent the range of CPU load from very light (1 IPC task, 16ms period) to very heavy (4 FIR tasks, 1ms period). As expected of a cooperatively multitasked RTOS, Echidna's response time is more-or-less evenly distributed over a 1ms interval, until the system becomes heavily loaded, at which point the execution time of the periodic interrupt-handler task can vary by a significant amount (up to several milliseconds). The NOS system has the simplest interrupt handling mechanism of all, and its response time is extremely good when the system is lightly loaded—in fact, even faster than the preemptive µC/OS-II kernel, because its cooperatively scheduled nature means that no state needs to be saved on task switch. As the system load increases, the average response time of NOS increases, and it obeys a geometric distribution corresponding to the average execution time of the application's jobs.

The preemptive µC/OS-II kernel handles interrupts with relative precision. Yet the figures shows that its overhead varies slightly from benchmark to benchmark. The variation is due to the RTOS's implementation of preemptive scheduling: A task is made ready when the current task blocks or a hardware timer tick occurs. Both events cause the RTOS to scan the Task Control Block (TCB) list and mark all appropriate tasks ready to run. The ready task with the highest priority is then made the current running task. This is a common design for preemptive schedulers, but because on a task switch the scheduler traverses the entire TCB list, the time to complete a task switch is dependent on the number of tasks. This explains the observed behavior that the response time scales with the number of tasks in the configuration.

In addition, Figure 2(d) shows that µC/OS occasionally takes longer to service interrupts than average, even though the interrupt handler has the highest priority in the system. The instances of longer interrupt-service times are due to the interrupt arriving during a moment when interrupts are disabled, thereby delaying the invocation of the handler. Non-intuitively, this seems to happen most often when the CPU is in low use. The idle task in µC/OS increments a protected counter that determines system load. The RTOS treats the update of this counter as a critical section of code and protects it by disabling interrupts before the increment and restoring interrupts after. Therefore, when the system is largely idle, the chance of an external interrupt arriving during a critical section is actually higher than when the system is busy.

3.3. Energy Consumption

To measure energy consumption, we ran each configuration for the same number of application iterations. The results are shown in Figure 3, which shows the energy overhead one pays for an RTOS. This closely mirrors the overhead one pays in terms of execution time as well [7]. Results are only shown for FIR, the application with the greatest overhead (~233µs per job invocation).

The results show that RTOS kernel overhead is reasonable, given the benefits provided by that RTOS. The use of the NOS scheduler increases energy consumption by less than a factor of two, and the Echidna and µC/OS-II kernels increase energy consumption by less than a factor of three. Several behaviors can be seen in the data, from the obvious to the not-so-obvious:

- Interrupt handling overhead is significant in systems that are interrupt-driven and insignificant in the cooperative systems. The latter makes sense, because in the polled systems, no state is saved or restored during interrupt handling. The former is interesting; the µC/OS-II kernel demonstrates that in heavily loaded systems, it can use interrupts to off-load some of Echidna's overhead.
- The user components for the more sophisticated RTOSs (Echidna and µC/OS-II) tend to be less than the user components for NOS. This simply represents the trade-off of being able to move some of the functionality from the application into the kernel. However, in less computationally intensive benchmarks, the user components are higher than NOS—low computational requirements in the application can

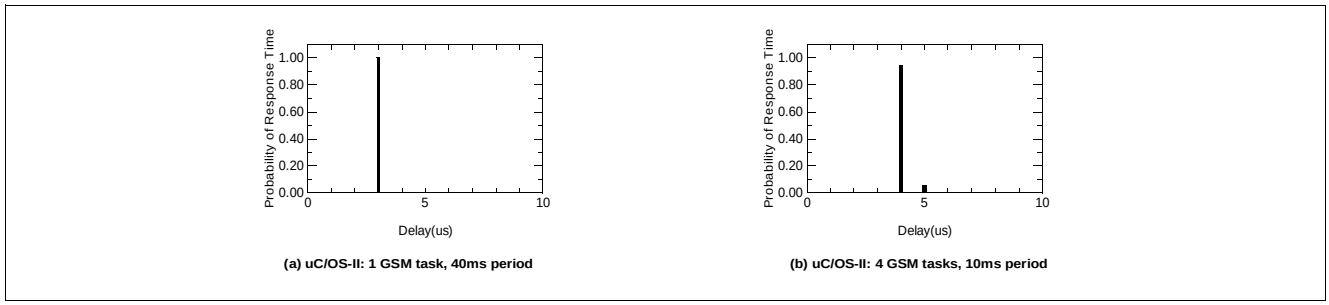

Figure 4: DELAY probability density graphs for GSM on TMS320C6201. The x-axis represents time between an interrupt being generated by an I/O device and the corresponding output to an I/O port of the responding thread. The y-axis indicates the probability of each delta. All measurements are for configurations with both types of background load (32Hz periodic control loop and aperiodic interrupt-driven IPC)—these delay measurements are for the interrupt-driven IPC that is the background load.

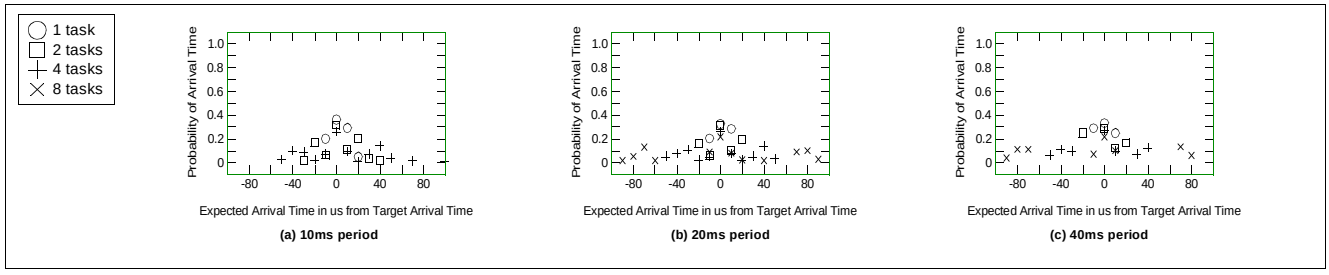

Figure 5: JITTER probability density graphs for GSM on TMS320C6201. The x-axis represents time deltas in microseconds between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks.

expose commonly used (but otherwise overshadowed) RTOS mechanisms as significant consumers of power, such as the clock-tick handler in µC/OS-II that runs every clock tick interrupt and wakes up sleeping threads when it determines that their periods have expired.

- The kernel overhead in NOS scales with the application workload, while the kernel components in the other RTOSs are more constant. The more sophisticated RTOSs do a better job of ensuring that all computations are deterministic in the time and energy it takes to perform them, which gives more predictable system behavior. The cost is obviously a higher starting point for energy consumption.
- It is cheaper to run tasks faster than to add tasks to the system. For instance, compare NOS:8 in Figure 3(a), NOS:4 in Figure 3(b), NOS:2 in Figure 3(c), and NOS:1 in Figure 3(d), which represent different trade-offs of speed and number of tasks. The user components is the same for these configurations, as the configurations all represent the same amount of work: 2000 job invocations per second, broken down as (respectively) 16 jobs, each scheduled every 8ms; 8 jobs, each scheduled every 4ms; 4 jobs, each scheduled every 2ms; and 2 jobs, each scheduled every millisecond. Though the work is the same, the kernel energy is not; this is seen in other configurations as well as in NOS. The reason is simple: the RTOSs maintain queues of tasks, typically as linked lists, which grow with the number of tasks.

Please note that “idle” time is both time spent sleeping and time in certain inactive loops. Just because Echidna still has idle time after the system is overloaded with work does not mean that any more useful work can be done.

4. FUTURE WORK

This paper characterizes the performance and power consumption of a few sample applications, running with various realistic RTOSs, on a low-power embedded processor. Future papers will be focused on updating, validating, and extending this research.

Because the Motorola MCORE processor is being phased out, a more modern processor is being modeled - the Texas Instruments TMS320C6201. This common DSP appears in several modern products, including devices for wireless communication, broadband communication, audio/video processing, encryption, and medical equipment. Not only is this processor newer, it is also quite different. The processor is a high performance 8-way VLIW 32-bit fixed-point DSP, operating at up to 1600 MIPS. Its performance is much higher, but so is its power consumption.

The sample applications used in this paper are realistic, however, more widely used benchmarks would allow the results to be related to existing research more easily. Therefore, standard benchmarks, including the MediaBench benchmark suite [38], are being utilized.

Some preliminary results have been obtained - specifically the jitter and delay measurements of the DSP running the GSM encode benchmark, from the MediaBench suite. The timescale for these measurements are in the microsecond range, because of the faster processor. The data points for the jitter measurements have been placed into 10 microsecond wide groups, for readability.

These results will significantly contribute to the research of RTOS and microprocessor performance and power optimization.

5. CONCLUSION

SimBed is a simulation-based environment for evaluating the performance and energy consumption of embedded real-time operating systems. The simulator's performance measurement is accurate to within 100 cycles per million compared to identical software executing on reference hardware. Its energy measurement is accurate to within 10–15%.

We presented a study of preemptive and non-preemptive real-time operating systems, focusing on two industrial-strength RTOSs. We compared these to a raw scheduler that should represent the realistic performance and energy-consumption limit for non-preemptive RTOSs, since it has none of the overhead that would be found in a real RTOS, such as support for semaphores, message-passing, etc. Among other things, we find an interesting trade-off that the more complex RTOSs seem to have taken: while the bare-bones scheduler has the lowest energy consumption, that consumption scales with the workload. The more complex RTOSs have a higher initial energy consumption, but this consumption does not increase quickly as the user-level computational load grows. Therefore, the energy consumption and CPU requirements of these systems are likely to be much more predictable than a simpler RTOS.

ACKNOWLEDGMENTS

The work of Kathleen Baynes and Christine Smit was supported in part by NSF's sponsorship of undergraduate research through grant NSF-9912218. The work of Chris Collins, Brinda Ganesh, Paul Kohout, and Tiebing Zhang was supported in part by NSF grant EIA-9806645 and NSF grant EIA-0000439. The work of Bruce Jacob was supported in part by NSF CAREER Award CCR-9983618, NSF grant EIA-9806645, NSF grant EIA-0000439, DOD award AFOSR-F496200110374, and by Compaq and IBM.

REFERENCES

[1] A. Allara, C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. "System-level performance estimation strategy for SW and HW." In *International Conference on Computer Design*, Austin TX, October 1998.

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

[3] K. Baynes, C. Collins, E. Fiterman, C. Smit, T. Zhang, and B. Jacob. "The performance and energy consumption of embedded real-time operating systems." Tech. Rep. UMD-SCA-TR-2000-04, University of Maryland Systems & Computer Architecture Group, November 2000.

[4] L. Benini and G. D. Micheli. "System-level power optimization: Techniques and tools." In *International Symposium on Low Power Electronics and Design*, August 1999, pp. 288–293.

[5] D. Brooks, V. Tiwari, and M. Martonosi. "Wattch: A framework for architectural-level power analysis and optimizations." In *Proc. 27th Annual International Symposium on Computer Architecture (ISCA'00)*, Vancouver BC, June 2000, pp. 83–94.

[6] C. M. Collins. "An evaluation of embedded system behavior using full-system software emulation." Master's Thesis, University of Maryland at College Park, May 2000.

[7] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. "Power analysis of embedded operating systems." In *37th Design Automation Conference*, Los Angeles CA, June 2000, pp. 312–315.

[8] C. Ellis. "The case for higher-level power management." In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, 1999.

[9] Embedded Research Solutions. Embedded Zone — Publications. http://www.embedded-zone.com, 2000.

[10] J. Flinn and M. Satyanarayanan. "Powerscope: A tool for profiling the energy usage of mobile applications." In *Workshop on Mobile Computing Systems and Applications (WMCSA)*, February 1999, pp. 2–10.

[11] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick. "The energy efficiency of IRAM architectures." In *Proc. 24th Annual International Symposium on Computer Architecture (ISCA'97)*, Denver CO, June 1997, pp. 327–337.

[12] J. Ganssle. "Conspiracy theory, take 2." *The Embedded Muse newsletter*, no. 47, March 22, 2000.

[13] J. G. Ganssle. "An OS in a can." *Embedded Systems Programming*, January 1994.

[14] J. G. Ganssle. "The challenges of real-time programming." *Embedded Systems Programming*, vol. 11, no. 7, pp. 20–26, July 1997.

[15] R. Gonzalez and M. Horowitz. "Energy dissipation in general purpose microprocessors." *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, pp. 1277–1284, September 1996.

[16] J. K. M. Gupta and W. Mangione-Smith. "The Filter Cache: An energy efficient memory structure." In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO'97)*, Research Triangle Park NC, December 1997, pp. 184–193.

[17] J. Hennessy and M. Heinrich. "Hardware/software codesign of *processors: Concepts and examples." In Hardware*/Software Co-Design, G. De Micheli and M. Sami, Eds. 1996, pp. 29–44, Kluwer Academic Publishers.

[18] M. Horowitz, T. Indermaur, and R. Gonzalez. "Low-power digital design." In *IEEE Symposium on Low Power Electronics*, October 1994, pp. 8–11.

[19] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. R&D Books (Miller Freeman, Inc.), Lawrence KS, 1999.

[20] Y. Li, M. Potkonjak, and W. Wolf. "Real-time operating systems for embedded computing." In *International Conference on Computer Design*, Austin TX, October 1997.

[21] C. Liema, F. Nacabal, C. Valderrama, P. Paulin, and A. Jerraya. "System-on-a-chip cosimulation and compilation." *IEEE Design and Test* of Computers, vol. 14, no. 2, pp. 16–25, April–June 1997.

[22] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, Upper Saddle River NJ, 2000.

[23] D. Roundtable. "Hardware-software codesign." *IEEE Design and Test of Computers*, vol. 14, no. 1, pp. 75–83, January–March 1997.

[24] K. Roy and M. C. Johnson. "Software design for low power." In *Software Design for Low Power*, Nato ASI series, August 1996.

[25] J. Russell and M. Jacome. "Software power estimation and optimization for high performance, 32-bit embedded processors." In *International Conference on Computer Design*, Austin TX, October 1998.

[26] J. Scott, L. Lee, A. Chin, J. Arends, and B. Moyer. "Designing the m.core m3 cpu architecture." In *International Conference on Computer Design*, Austin TX, October 1999.

[27] SimOS. SimOS: The Complete Machine Simulator. Stanford University, http://simos.stanford.edu/, 1998.

[28] M. J. Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, Reading MA, 1997.

[29] D. B. Stewart, D. E. Schmitz, and P. K. Khosla. "The Chimera II real-time operating system for advanced sensor-based applications." *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1282–1295, November/December 1992.

[30] D. B. Stewart, R. A. Volpe, and P. K. Khosla. "Design of dynamically reconfigurable real-time software using port-based objects." *IEEE Transactions on Software Engineering*, vol. 23, no. 12, pp. 759–776, December 1997.

[31] V. Tiwari and M. T.-C. Lee. "Power analysis of a 32-bit embedded microcontroller." *VLSI Design Journal*, vol. 7, no. 3, 1998.

[32] V. Tiwari, S. Malik, and A. Wolfe. “Power analysis of embedded software: A first step towards software power minimization.” In *International Conference on Computer-Aided Design*, San Jose CA, November 1994.

[33] J. Turley. “M.Core shrinks code, power budgets.” *Microprocessor Report*, vol. 11, no. 14, pp. 12–15, October 1997.

[34] J. Turley. “M.Core for the portable millenium.” *Microprocessor Report*, vol. 12, no. 2, pp. 15–18, February 1998.

[35] A. Vahdat, A. Lebeck, and C. Ellis. “Every joule is precious: The case for revisiting operating system design for energy efficency.” In *SIGOPS European Workshop*, September 2000.

[36] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. “Energy-driven integrated hardware-software optimizations using simplepower.” In *Proc. 27th Annual International Symposium on Computer Architecture (ISCA’00)*, Vancouver BC, June 2000, pp. 95–106.

[37] T. Zhang. “RTOS Performance and Energy Consumption Analysis Based on an Embedded System Testbed.” Master’s Thesis, University of Maryland at College Park, May 2001.

[38] C. Lee, M. Potkonjak and W. H. Mangione-Smith. In *Proc. 30th Annual International Symposium on Microarchitecture (MICRO’97)*, Research Triangle Park NC, December 1997, pp. 330-335.