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

Kathleen Baynes, Chris Collins, Eric Fiterman, Christine Smit, Tiebing Zhang, and Bruce Jacob

Dept. of Electrical & Computer Engineering
University of Maryland at College Park
{ktbaynes,chriscol,ericf,csmit,zhangtb,blj}@eng.umd.edu

Technical Report UMD-SCA-TR-2000-04
November, 2000

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. Our simulations show what happens when RTOSs are pushed beyond their limits, and they depict situations in which unexpected interrupts or unaccounted-for task invocations disrupt timing, even when the CPU is lightly loaded. In general, there appears no clear winner in timing accuracy between preemptive systems and cooperative systems. The power-consumption measurements show that RTOS overhead is a factor of two to four higher than it needs to be, compared to the energy consumption of the minimal scheduler. In addition, poorly designed idle loops can cause the system to double its energy consumption—energy that could be saved by a simple hardware sleep mechanism.*

1 INTRODUCTION

This paper motivates the use of simulated embedded microcontrollers for system design and presents a simulation-based experimental study comparing the performance and energy characteristics of three real-time operating systems (RTOSs)—(1) the public-domain embedded kernel μC/OS-II [16], (2) the commercial real-time kernel Echidna [5], and (3) a "roll-your-own" style system that has an organization common in today's embedded systems [7, 8].

1.1 Motivation

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. Only a handful of years ago it was easy enough to hook a probe to the memory and I/O buses, but, with the advent of systems on a chip and application-specific integrated circuits, it is no longer possible to obtain those signals, for they never leave the silicon [17, 23]. The only way to debug these systems is to either probe the silicon itself (a bit unrealistic), or to add logic to the chip to bring the desired signal off the chip; the latter option is limited by the number of physical pins that can be put on a chip and spared for simple debug and evaluation purposes. Also, with the speeds that some of today's embedded processors are running, it becomes difficult to find a logic analyzer that can keep up with the processors and not cost something beyond the reach of most academic research groups and small embedded-systems design houses. If there were another method to evaluate these systems early on, both time and money could be saved.

There are three recent trends that are relevant to this observation. First is a design methodology that is gaining wide acceptance in both the embedded and general-purpose worlds—hardware/software cosimulation or codesign [13]. One of the fundamental aspects of this methodology is that, early on in the process, software is developed for and executed on models of the hardware that are implemented in some high-level language. As opposed to the traditional method of developing the hardware and software for a system separately, the hardware/software codesign methodology recognizes the benefits inherent in the designing of the two together, at the same time. 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 [21].

Another trend gaining in popularity is the use of real-time operating systems; RTOSs are increasingly used in the development and deployment of real-time embedded systems. Their benefits are well known: they provide numerous helpful facilities including cooperative and preemptive multitasking, multi-threading, support for both periodic and aperiodic tasks, fixed-priority/dynamic-priority scheduling, semaphores, inter-process communication, shared memory, and memory management; in doing so they can dramatically reduce the programming burden of the system designer [9, 2, 10].

The third trend is the increasing importance of low-energy systems. There is rapidly growing consumer demand for computing devices that are battery operated, including PDAs, cell phones, wearable computers, handhelds, and laptops. Like many things, it is difficult to retro-fit a low-energy philosophy into an existing system architecture; as the StrongARM has shown, energy consumption must be considered from the beginning of the design phase if the system is to be both high-performance and low-power.

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 [22], and there has been a large number of recent studies modeling the power consumption of microprocessors and applications [14, 11, 12, 6, 3, 29, 26], but this is the first study of which we are aware that performs both.

1.2 SimBed

Our group has developed *SimBed*, a C-language model of an embedded hardware system that is accurate enough to run unmodified real-time operating systems (i.e., the binary that runs on the simulator is the same binary that runs on real hardware). The processor model is the Motorola M-CORE microcontroller: a low-power, 32-bit CPU core with 16-bit instructions [27, 28]. 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 (two of Motorola's M-CORE evaluation boards: one for the generic ISA, another for the MMC2001). The numbers presented in this paper correspond to the first evaluation board, which clocks the processor at 20MHz.

We have also instrumented the processor simulator to measure energy consumption, using existing instruction-based techniques [26]. We have verified the simulator's output to measurements of a Motorola M-CORE processor, and our results are within 10–15% of actual numbers. This level of accuracy for modeling power at the processor level is about where most current research stands (e.g. [3, 26]).

This paper presents an experimental study using *SimBed* in which the real-time performance and energy consumption of three different RTOSs are compared: a public-domain preemptive multitasking kernel, an industrial-strength cooperative multi-tasking kernel, and a bare-bones task scheduler (which represents the limiting case of a lightweight cooperatively-scheduled RTOS). We also present the theoretical maximum throughput of the application code *sans* RTOS.

An interesting side note is that some of the measurements represent things that cannot be obtained via traditional means (e.g., 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 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.

1.3 Experiments

This study looks at the behavior of embedded real-time systems, particularly those that use embedded RTOSs. Our initial focus is on systems that use *on-line* scheduling (the choices are made at run-time as opposed to compile-time), as they tend to be less amenable to analytical verification than systems with *off-line* scheduling (those in which the scheduling decisions are made at compile-time). All RTOSs studied handle the simultaneous execution of multiple applications. The RTOSs are also compared to the theoretical maximum throughput values calculated for the benchmark applications. Briefly, these are the execution models studied in this paper:

uC/OS-II: μC/OS-II is a preemptive multitasking RTOS that is in the public domain [16]. 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 multi-tasking RTOSs with footprints small enough for microcontroller systems.

ECHIDNA: A cooperative multitasking RTOS based on Chimera [24] that swaps Chimera's POSIX-like threads in the microkernel for port-based objects [25]; it supports reconfigurable component-based software for microcontrollers and digital signal processors [5]. 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 [8]. 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.

LIMIT: The theoretical performance limit of each application, based solely on the computational requirements of its implementation. This represents the (unattainable) energy and performance limit of a zero-overhead RTOS.

For the realistic performance limit (NOS), we chose a multi-rate executive rather than something simpler, such as a cyclic scheduler, because the behavior of a cyclic scheduler is very sensitive to the execution profile of the application program, while the multi-rate executive is much less so [15].

Within each of these execution models, we execute several different application kernels. These are periodic applications, for which an absolute deadline is less important than a relative deadline—i.e. these are applications for which a 500Hz task requires its i+1th invocation to run exactly 2ms after its ith invocation and could care less whether the very first invocation started at time t_0 or t_0 plus some small delta. These 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 for 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.

Following Liu's terminology [18], we use the term "job" to mean *an independently scheduled block of code* and the term "task" to mean *a collection of logically related jobs* that together perform some function. The embedded applications studied exploit multi-tasking 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. The applications differ primarily in the amount of computation and include *raw IPC* (both periodic and aperiodic), *up-sampling*, *down-sampling*, and a 128-tap *FIR filter.* The applications are chosen to be simple so that they can be sped up and/or layered atop each other to gradually increase the total system workload. 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. The same application code is executed on all three operating systems (with minor RTOS-specific modifications) and is used to determine the theoretical computational limit as well. 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.

1.4 Results

The performance measurements yield both predictable and surprising results. Predictably, as system load is increased, the RTOSs studied 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. Also predictably, the fixed priority scheduler in NOS leads to complete denial of service for lower-priority jobs when the critical system load is reached. The surprising results include situations where the industrial RTOSs schedule a substantial number of application tasks too early, even under light system load. This is due to unexpected interrupts and unaccounted-for task invocations that cause individual job timing to be thrown off, but only occasionally. The problem is that the RTOSs studied attempt to schedule jobs against a universal clock instead of a relative clock.

The energy-consumption measurements show some interesting results. RTOS energy overheads can be extremely high when running low-overhead tasks; if the task requires very little computation time for each job invocation, the RTOS can easily account for 90% of the processor's energy consumption, and poorly considered idle loops can double the system's energy requirements. As a periodic task's complexity and CPU requirements grow, the proportion of the energy spent in the RTOS diminishes significantly, and the effect of the idle loop is also diminished. There is also 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.

2 EXPERIMENTAL SET-UP

We use an execution-driven simulation of the Motorola M-CORE processor that can run unmodified RTOSs. On this simulator we run three different software configurations: µC/OS-II, Echidna, and NOS—the public-domain kernel, the industrial RTOS, and the simple multi-rate executive, respectively. We run several benchmarks atop each of these, increasing the workload to the point where the system fails to meet deadlines. We also ran the benchmarks without any RTOS support, to obtain performance and energy-consumption limits.

2.1 Motorola M-CORE Processor

The M-CORE is a low-power, compiler-friendly core designed specifically for the embedded market [19, 20, 27, 28]. It is a RISC-based design that uses 16-bit instructions and operates on 32-bit data. It has a simple four-stage single-issue pipeline, memory-mapped I/O, an orthogonal general-purpose register file with 16 registers, and a duplicate "shadow" register file that privileged software can enable instead of the regular register file. For this study, we simulate the processor at 20MHz, the same clock frequency as the evaluation hardware. The timing mechanism on the M-CORE evaluation board is simple and offers precision on the order of 1µs. It is a 2-byte counter in I/O space that increments every 1.6µs. Every 100ms (every 62,500 ticks of the counter), the counter wraps around and raises a timer interrupt to the CPU.

2.2 Application Code

The following describe the range of user-level code run in the experiments.

Periodic Inter-Process Communication. Periodic inter-process communication (IPC) is the simplest of the benchmarks that was used to evaluate performance. As mentioned above, the first job grabs data off of the input I/O port and stores it into shared memory. The second job takes that value from shared memory and writes it to the output I/O port. There is no computation, only the movement of data. This task represents the simplest possible two-job task possible.

Up/Down Sampling. With up sampling (UP), the second job runs at a higher frequency than that of the first job. Only a fraction of times that the second job has run will there be any new information. Therefore the second job carries out a basic form of interpolation. In down sampling (DOWN), the first job runs at a higher frequency than the second job. The second job takes all of the values that have been brought in by the read job since last time that second job has run, averages them, and then outputs that average to the output I/O port.

Finite Impulse Response Filter. The finite impulse response (FIR) filter is the most computation intensive of the four benchmarks. The second job runs a 128-tap filter on the data that has been collected by the first task. For each run of the second job, the last 128 values to be inputted by the first job are used to compute an inner product, and that value is outputted to the I/O port.

Background Load. To add some non-determinism to the evaluation of these two operating systems, and to offer more realistic simulations indicative of real-world systems, two different additional tasks were created. These tasks can be run concurrently with the above listed benchmarks to provide a background load. These two tasks are a periodic control loop and an aperiodic inter-process communication process.

Control Loop: The control loop (CL) was created to run 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 every so often to refresh its LCD display. This control loop performs several memory lookups with an index that is randomly generated.

Aperiodic Inter-Process Communication: The aperiodic inter-process communication (AP-IPC) task is run when a simulated I/O interrupt is generated by the hardware. It calls a user-level function 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.

2.3 Characterization of Real-Time Behavior

As mentioned earlier, we measure three things: jitter, delay, and cycle-by-cycle energy consumption.

Jitter: Jitter is measured by keeping track of inter-arrival times of periodic output. For example, if a task is scheduled to write an output value 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 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. We categorize all behavior into *user, kernel, handler, semaphore,* and *idle* components. Note that in the theoretical limits there are no kernel, semaphore, or handler components.

2.4 Real-TIme Kernels

The uC/OS-II Kernel. The µC/OS-II real-time kernel is a full-featured preemptive multitasking RTOS [16]. It is portable, targeted at both microcontrollers and DSPs, and it currently runs on over fifty different instruction-set architectures. It is designed to have a small footprint: there are roughly 1700 lines of code in the OS (including comments), and modules are only compiled into the executable if used by the application. Multi-tasking is preemptive, and the kernel can preempt itself. The system can run up to 64 tasks, with 8 of those tasks reserved for the kernel's use. It provides traditional OS services such as IPC, semaphores, and memory management, and it also provides time-related features such as the ability to sleep until a specified time and callout functions in which an application can specify code to execute on task creation, task deletion, context switch, and system timer tick.

Because µC/OS-II has no concept of a periodic task, we used two facilities within the kernel to implement periodic job invocations. Each job sleeps on a unique semaphore, and a user-level task is attached to the clock interrupt (µC/OS-II allows user-level code to be attached to arbitrary events). This user-level task keeps track of the job invocation times and generates wakeup messages when the job periods are reached. The inter-process communication method is message-passing.

The Echidna RTOS. Echidna is a scaled down version of the Chimera RTOS [24] that replaces Chimera's concept of a process (which is notionally similar to that of POSIX threads) with port-based objects [25]. It is designed to support dynamically reconfigurable real-time software and is targeted for 8-bit to 32-bit microcontrollers as well as DSPs, whereas Chimera was intended for 32-bit multiprocessor systems due to its relatively high overhead. Echidna, like Chimera, provides cooperative multitasking. It offers a good deal of functionality in a small footprint—as little as 6KB, depending on the configuration. The design concepts embodied in the RTOS are described in more detail in [5].

Echidna is designed to support only periodically scheduled tasks, and its periods are defined in terms of milliseconds (no finer granularity is supported by the OS). The inter-process communication method used is shared memory. To calculate delay times, we create a process with the smallest period possible (1ms) that checks to see if an AP-IPC interrupt has occurred. If such is the case, then the AP-IPC code will run. It is important to note that since an interrupt is possible (though not likely) every 100µs, and the interrupt is checked only every 1ms, it is possible for two or more interrupts to happen before any of them are serviced. This is an expected behavior of non-preemptive systems.

The NOS Multi-Rate Executive. NOS represents the type of "roll-your-own" RTOS often produced in the embedded-systems industry—it was designed in-house and is based entirely on descriptions of home-grown embedded system software given by practicing engineers in the embedded-systems industry [8]. NOS is a fixed-priority multi-rate executive for periodic tasks [15] and handles interrupt-driven stimuli via masking interrupts and polling the interrupt status registers when idle. Its main control loop is shown in Figure 1.

NOS's callout queue is taken from the callout table in UNIX [3]; events to happen in the future are placed in the queue keyed by the time at which they are expected to execute, and the *delta* field in the *event* structure represents the time difference between the event in question and the one before it in the queue. The delta field of the first event represents the invocation time relative to *now*. If the value is negative, the deadline for the first task (and perhaps following tasks as well) has been missed; if the value is zero, it is time to execute the first task; if the value is positive, the first event is to happen at some point in the future. One nice feature of this organization is that a periodic task can easily be created by having a function place itself back on the queue at the end of its execution.

NOS only handles a job or interrupt if there are no jobs or interrupts waiting at higher priority levels. Therefore, at levels beneath priority 1 (HARD jobs that have reached their time to execute), only one job is executed before jumping back to the top of the control loop—e.g., only one interrupt is handled before checking the callout queue to see if any more HARD jobs are ready to run. It is a simple fixed-priority scheduler with the expected weakness that low priority jobs will be ignored indefinitely if there is enough work to do at a higher priority.

3 EXPERIMENTS

For these studies, we execute the following benchmarks: periodic IPC (*P-IPC*), up-sampling (*UP*), down-sampling (*DOWN*), and a 128-tap FIR filter (*FIR*). We also have a periodic control-type administrative loop (*CL*) and interrupt-driven aperiodic IPC (*AP-IPC*) that can be run concurrently with the benchmarks to provide background load. The CL background task runs at 32Hz, and the AP-IPC inter-arrival times obey a geometric distribution (we generate an interrupt every 100µs with probability 0.01, resulting in an average of 100 AP-IPC interrupts per second). We varied the following parameters:

- RTOSs: {µC/OS-II, Echidna, NOS}
- Periodic tasks: {P-IPC, UP, DOWN, FIR}

```
struct event {
     struct event *prev, *next;
     time_t delta;       // invocation time delta from previous task; value for first task is relative to “now”
     void (*execute)();  // function to execute at invocation time
     char *data;         // data to pass to function at invocation time
     int priority;       // HARD_DEADLINE or SOFT_DEADLINE
};

struct event *calloutq;  // global linked list of tasks to perform

time_t update_calloutq(time_t t_now, time_t t_then)
{
     if (calloutq) {
          calloutq->delta -= (t_now - t_then);
     }
     return t_now;
}

// ... buried down in main() somewhere:
struct event *eventp;
time_t t, time = now();
while (1) {
     for (entryp=calloutq, t=(calloutq ? calloutq->delta : 1); entryp && t<=0; t=(entryp ? t + entryp->delta : 1)) {
          if (entryp->priority == HARD_DEADLINE) {
               entryp->execute(entryp->data);
               entryp = free_entry(entryp);  // returns entryp->next or NULL if last in list
               time = update_calloutq(now(), time);
          } else {
               entryp = entryp->next;
          }
     }

     if (HIGH_PRIORITY(interrupt_status())) {
          handle_interrupt(HIGH_PRIORITY(interrupt_status()));
          time = update_calloutq(now(), time);
          continue;
     }

     if (calloutq && calloutq->delta <= 0) {
          calloutq->execute(calloutq->data);
          free_entry(calloutq);
          time = update_calloutq(now(), time);
          continue;
     }

     if (LOW_PRIORITY(interrupt_status())) {
          handle_interrupt(LOW_PRIORITY(interrupt_status()));
          time = update_calloutq(now(), time);
          continue;
     }

     if (calloutq) {
          delta = calloutq->delta; // has to be positive if we have gotten this far
     } else {
          delta = INDEFINITE;
     }

     sleep(delta);      // wakes up only for interrupt or timeout

     time = update_calloutq(now(), time);
}
```

Figure 1: NOS main loop—simple multi-rate executive with fixed priority scheme. Design based on descriptions of RTOSs built by designers in industry [8], e.g. “The dispatch mechanism is a while(1) loop that does the highest priority thing, then the next highest, then the next highest, etc., in each case repeating the loop without touching lower priority tasks if there is more to do on that priority ... This can be interrupt-based or completely polled depending upon hardware.” In this case, all I/O is polled.

- Workload: {1, 2, 4, 8 tasks}
- Periods: {16, 8, 4, 2, 1, 0.5, 0.25, 0.125, 0.064 msec}
- UP/DOWN Sampling ratios:{2:1, 4:1, 8:1}
- Background load: {AP-IPC, AP-IPC+CL, CL}

The studies represent the effective cross-product of these variations, minus those configurations that lie beyond the point where the system in question failed to meet deadlines. Also, Echidna will not schedule periodic tasks with periods less than 1ms; therefore, we do not have results for periods at 500µs or below for Echidna. Remember that, by design, no job performs both reads and writes to I/O; therefore, each task is actually two separately scheduled jobs.

3.1 Experimental Results: JITTER

As described above, jitter measurements represent the time deltas between successive output seen at the I/O device for a given executing task. 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.

The graphs shown are 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, and values have been gathered into 100 µs intervals.[1]

Figure 2 presents the jitter measurements for the periodic IPC, with background load and without. The periodic IPC task represents the simplest possible case of two interacting jobs: the input job reads input from I/O space and uses RTOS-supplied inter-process commu-

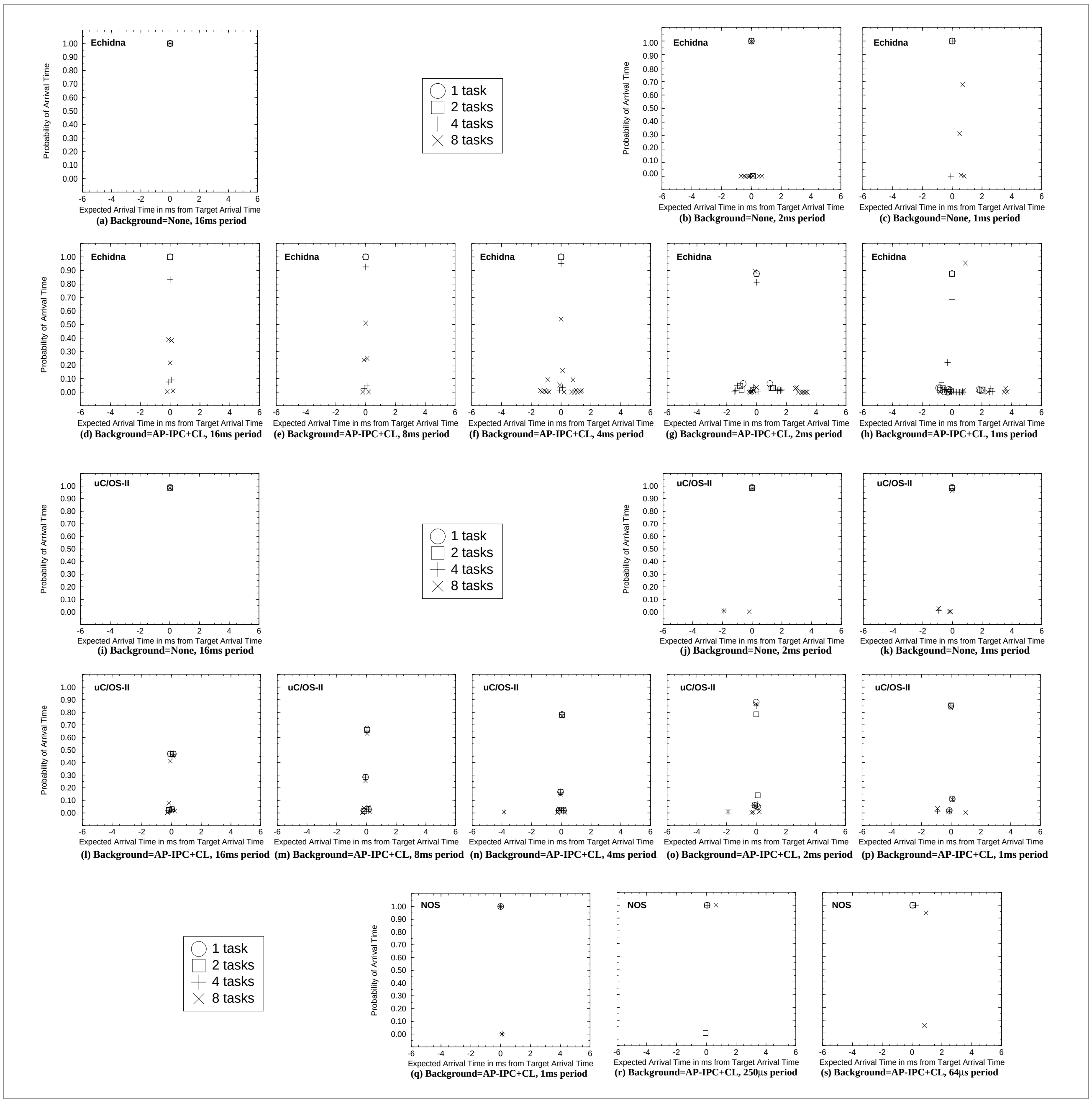

Figure 2: JITTER probability density graphs for P-IPC. The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates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

nication to send the data to the output job, and the output job sends the received datum to another I/O port. There is no computation performed other than moving data; this therefore represents the smallest workload that a realistic application would schedule on an RTOS. It is thus likely to exhibit the highest possible RTOS overhead.

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 2(d) shows that when Echidna is running 8 tasks with 16ms periods (16 jobs), roughly 20% of the jobs will execute exactly on-time, 40% of the jobs will execute a little early, and 40% of the jobs will execute a little late; roughly 1% of the time the job executions will be 50μs off, in either direction. When the system load is 4 tasks (8 jobs), job executions are on-time roughly 85% of the time, and missed deadlines are either too early or too late with roughly equal

1. 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.

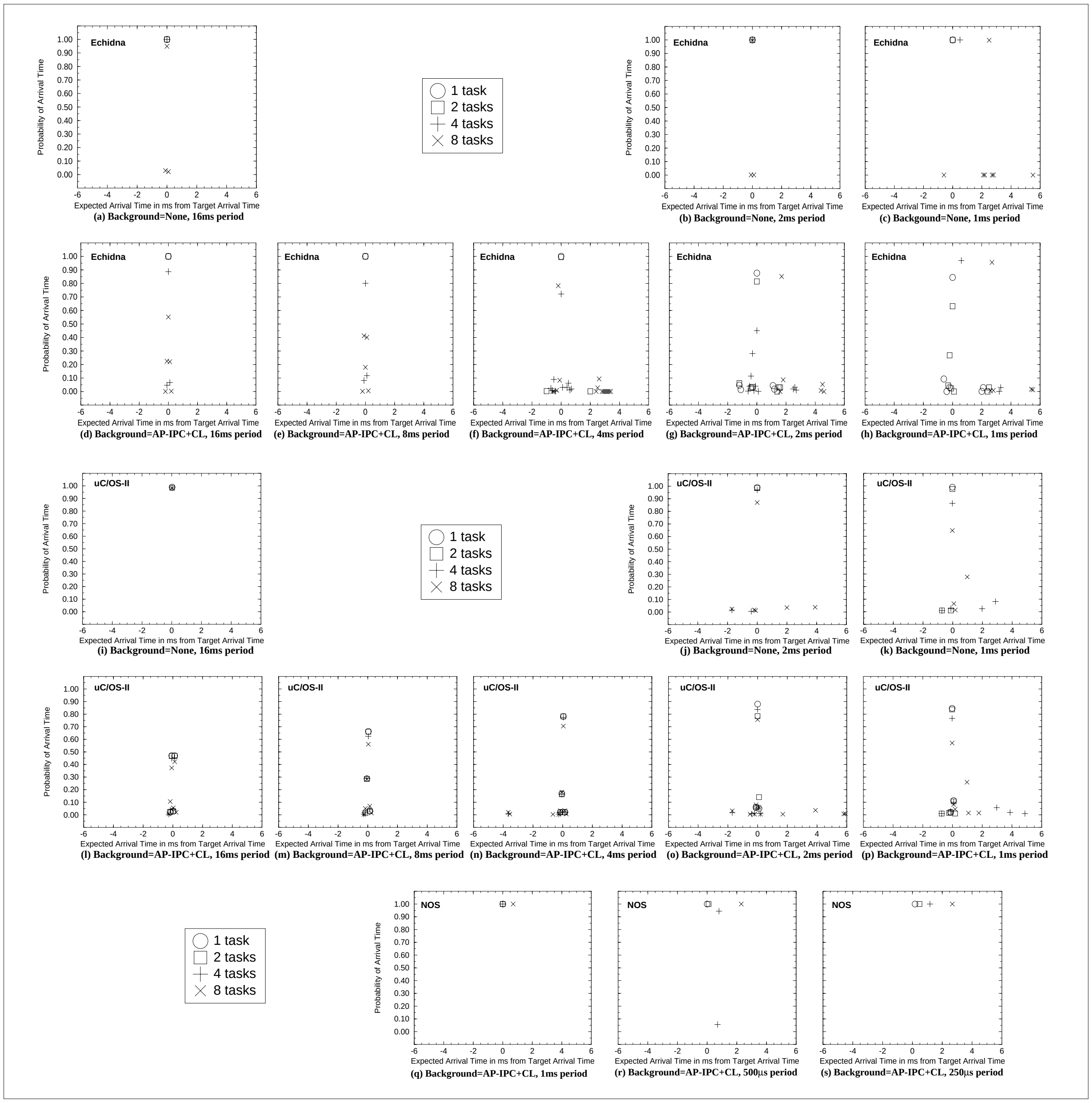

Figure 3: JITTER probability density graphs for FIR. The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early, and positive numbers mean a task has run late, in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneous tasks.

probability and absolute value. When executing 1 or 2 tasks, job execution is always on time.

There are some obvious RTOS behaviors shown in the figure: there is a workload level at which point the RTOS fails to meet deadlines. Once this line is crossed, most if not all of the output arrives late every time (e.g. 8-task output in Figures 2(c), 2(h), etc). For Echidna, this point is around 500Hz with 8 IPC tasks running; for µC/OS and NOS, the point is above 1MHz, even for 8 tasks running.

Figure 3 presents the jitter measurements for the FIR filter. This benchmark represents the largest computational overhead per job invocation; as expected, it shows the same behavior as the IPC benchmark, only at different periods—the system is overloaded sooner, compared to IPC. The results are very similar to the IPC results, except that they display slightly more variation in the timing.

An interesting result seen in the graphs is that, even at light workloads (e.g. tasks running with 16ms periods), Echidna and µC/OS execute a number of jobs too late—and an equal number of tasks too early. We see that µC/OS at task periods of 16ms cannot get more than 50% of the tasks to execute on-time when the system is perturbed by occasional interrupts (see Figure 3(l)). What is happening is that both RTOSs attempt to schedule tasks against a universal clock. Future job invocations are not scheduled relative to the job

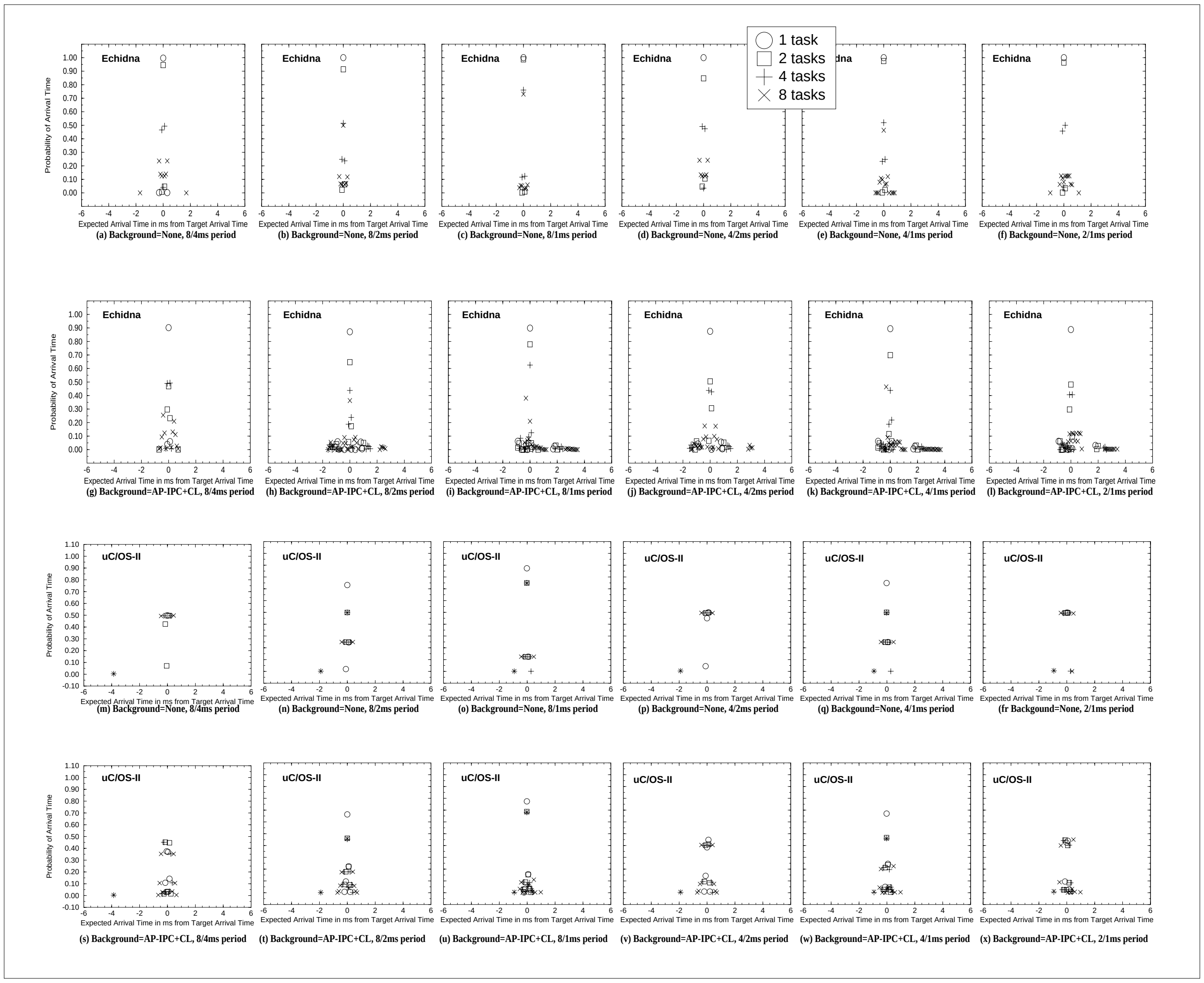

Figure 4: JITTER probability density graphs for UP. The x-axis represents time deltas between successive I/O output events as they differ from the expected period—negative numbers indicate the output happened early. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of 1, 2, 4, and 8 simultaneously executing tasks.

invocation time, but relative to the *intended* invocation time. Therefore, if a job is occasionally perturbed, it will be scheduled late on one instance and "on-time" the next instance. Though this results in the system missing a beat occasionally, it also means that if the disturbance is periodic, as is the case if multiple periodic tasks want the same invocation time, the missed beats will happen with probability 1, even if the workload is light. The systems would benefit from better load-distribution algorithms. 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. We see exactly this behavior with the NOS scheduler.

So what are the periodic disturbances? The most obvious disturbances are the tasks executing as background load. In Echidna, the background control loop is a periodic task with period 32ms. Therefore, it is executed every other job invocation for 16ms tasks, every fourth job invocation for 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.

The disturbance in μC/OS is the aperiodic IPC interrupt that happens on average every 10ms. It has a higher priority than any of the periodic application tasks, so it preempts application threads whenever it runs. The 16ms tasks are upset most by this (see Figure 3(l)), 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.

Timing disturbances in real-time schedulers do not require unpredictable background load, however. The UP and DOWN benchmarks exhibit this interference even without any background load. The simulation results for up-sampling are shown in Figure 4. The top row represents Echidna without any background load. The second row of graphs is Echidna with background load. The third row is μC/OS without background load, and the last row is μC/OS with background load. We see that both RTOSs allow applications to interfere with themselves, even when tasks are scheduled with relatively low frequencies. This is because the periods are not the same, but they are not relatively prime (they are multiples of each other in

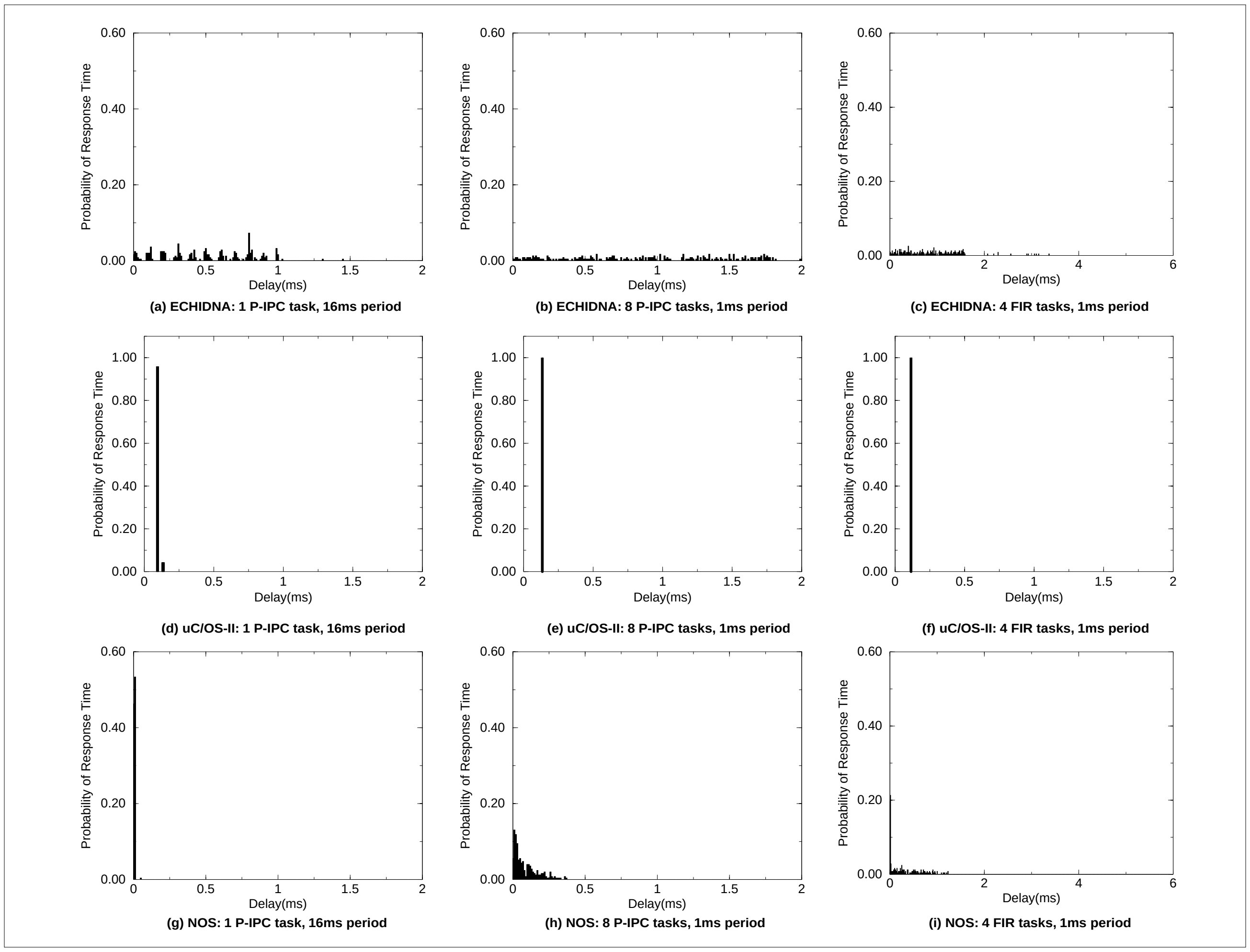

Figure 5: DELAY probability density graphs for ECHIDNA and NOS. The x-axis represents time between an interrupt being generated by an I/O device and the corresponding output to an I/O port of the responding thread. The y-axis indicates the probability of each delta. All measurements are for configurations 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).

this instance), so task invocations will coincide in time every Nth invocation. Neither operating system manages to spread the tasks out in time. Again, this is because both RTOSs schedule jobs against a universal clock and not relative to previous invocation.

In summary, the dynamic-priority schedulers in Echidna and µC/OS add a bit of intelligence to their decision-making that is not found in NOS. However, NOS schedules jobs on-period when lightly loaded and fails in highly predictable ways when overloaded (i.e. missing deadlines), whereas Echidna and µC/OS both fail in surprising ways when overloaded or even lightly loaded by executing tasks earlier than expected, sometimes by several milliseconds.

3.2 Experimental Results: DELAY

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

The µC/OS-II kernel handles interrupts preemptively; both Echidna and NOS use a polling technique. The difference between Echidna and NOS is that the Echidna RTOS will not spawn a new task as a result of an interrupt; this must be done by an application task. Therefore, our Echidna interrupt-handler task is periodic with the shortest possible period (1ms) and 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.

The delay times are shown in Figure 5. These represent the range of CPU 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). Also as expected, the preemptive µC/OS-II kernel handles interrupts with absolute precision that is independent of application load. 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, it is even faster than the preemptive µC/OS-II kernel, because its cooperatively scheduled nature means that on task switch, no state needs to be saved. As the system load increases, the average response time of the NOS system increases, and it obeys a geometric distribution corresponding to the average execution time of the application's jobs.

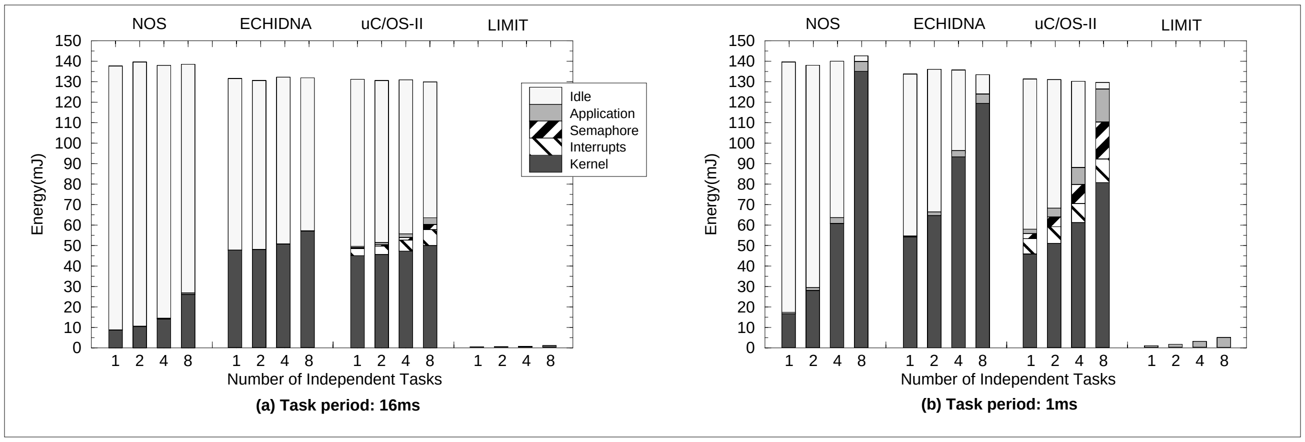

Figure 6: ENERGY COMSUMPTION graphs for IPC. The x-axis represents increasing workloads, as a result of increasing the number of executing tasks or. The y-axis represents the total CPU energy consumption and breakdowns for how much energy is consumed by executing kernel code, executing user application code, handling interrupts, performing semaphore handling, and sitting idle. 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.

3.3 Experimental Results: ENERGY CONSUMPTION

To get energy consumption, we ran each configuration for the same amount of application iterations. The results are shown in Figures 6 and 7, which show the energy overhead one pays for an RTOS. This closely mirrors the overhead one pays in terms of execution time as well [4]. Results are only shown for the applications with the least (IPC) and greatest (FIR) overhead per job invocation.

The IPC results in Figure 6 indicate several things very clearly. First, at the extreme of performing essentially no computation at all per job invocation, using an RTOS is overkill, even for a simple task scheduler. For NOS, the kernel overhead increases energy consumption by roughly a factor of twenty; Echidna and μC/OS-II eat up even more. The implications are obvious: simply keeping track of time and what task to execute at what time consumes considerable energy and CPU resources, compared to simple I/O operations. Note that the measurements are for a 20MHz microcontroller.

The FIR results in Figure 7 show that, for more realistic applications (bear in mind that FIR is still relatively light in computation time at ~233μs per invocation), RTOS kernel overhead is slightly reasonable. 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 results 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—and less than the limit, as well! This simply represents the trade-off of being able to move some of the functionality from the application into the kernel. However, in the IPC graphs, the user components are higher—the low computation requirements of IPC expose the user-level clock-tick handler in μC/OS-II that runs every clock tick and wakes up sleeping threads when their periods expire. This is present in all applications.
- The systems all consume an enormous amount of energy doing nothing, as represented by the idle components. This is because none of the systems have an intelligent sleep mechanism that can use less power when there is nothing to do; though the M-CORE has such a facility (a doze mode that can be awakened by a watch-dog timer interrupt), no system uses it. If implemented, this would save considerable energy resources. Note, however, that there is very little idle time as the system is pushed up to but not beyond its limits, which is where embedded-system engineers would like their systems to be, as this makes most effective use of the CPU resources.
- The kernel overhead in NOS scales with the application workload, while the kernel components in the other RTOSs is 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, in the FIR graphs, compare NOS:8 in Figure 7(a), NOS:4 in Figure 7(b), NOS:2 in Figure 7(c), and NOS:1 in Figure 7(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 CONCLUSION

We have described *SimBed*, a simulation-based environment for evaluating the performance and energy consumption of embedded real-time operating systems. The simulation environment was built

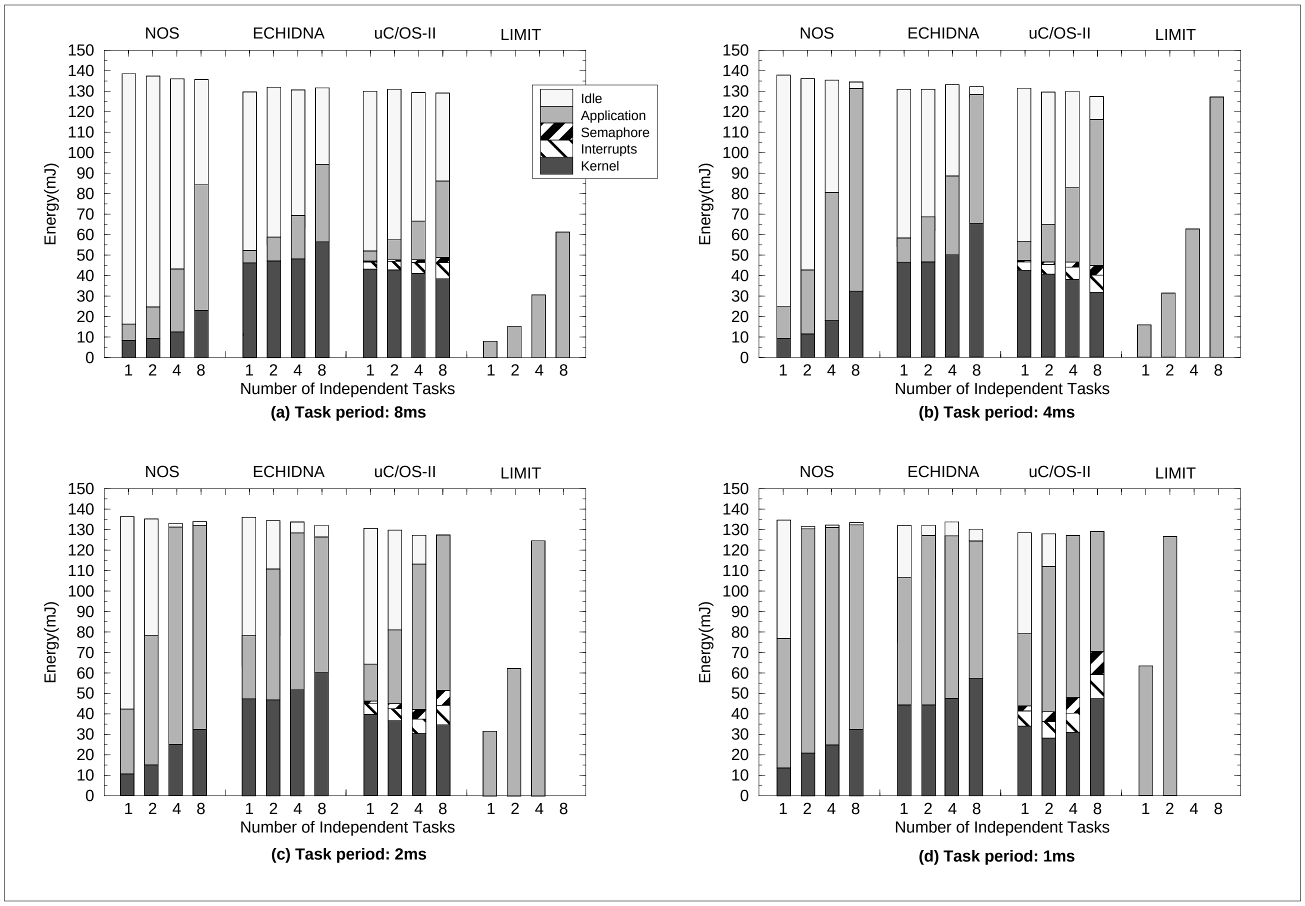

Figure 7: ENERGY CONSUMPTION graphs for FIR. The x-axis represents increasing workloads, as a result of increasing the number of executing tasks or. The y-axis represents the total CPU energy consumption and breakdowns for how much energy is consumed by executing kernel code, executing user application code, handling interrupts, performing semaphore handling, and sitting idle. 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.

to study hardware mechanisms that help facilitate low-power real-time processing, as well as to quantify differences between design and implementation in existing RTOSs. 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 aimed at microcontrollers as well as DSPs. 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. We find that RTOS overheads for lightweight applications are very high—95% or more—but that the overhead diminishes significantly for more compute-intensive applications (down to 50% for Echidna and µC/OS-II, 30% for the limit). There is also 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, Chris Collins, Eric Fiterman, Christine Smit, and Tiebing Zhang was supported in part by NSF grants EIA-9806645 and EIA-0000439. Bruce Jacob was supported in part by these grants, NSF CAREER Award CCR-9983618, and by Compaq and IBM.

REFERENCES

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

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

[3] 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.

[4] 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.

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

[6] 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.

[7] J. Ganssle. "Conspiracy theory." *The Embedded Muse newsletter, no. 46*, March 3, 2000.

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

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

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

[11] 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.

[12] 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.

[13] 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.

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

[15] D. Kalinsky. "A survey of task schedulers." In *Embedded Systems Conference 1999*, San Jose CA, September 1999.

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

[17] 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.

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

[19] Mcore. *M-CORE Reference Manual*. Motorola Literature Distribution, Denver CO, 1997.

[20] Mcore. *M-CORE MMC2001 Reference Manual*. Motorola Literature Distribution, Denver CO, 1998.

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

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

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

[24] 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.

[25] 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.

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

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

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

[29] 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.