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

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

**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 multirate 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.

**Index Terms**—Embedded systems, real-time operating systems (RTOS), power and energy modeling, performance modeling, Motorola M-CORE, *μC/OS-II*, Echidna, Chimera.

## 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 [24], 2) the commercial real-time kernel Echidna [12], and 3) a "roll-your-own" style system that has an organization common in today's embedded systems [15], [16].

### 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 become 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 [27], [35]. 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 at which 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 the increasing popularity of hardware/software cosimulation or codesign [21], [1]. 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 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 [9].

Another trend gaining in popularity is the use of real-time operating systems [26], [11], [42]. RTOSs are increasingly used in the development and deployment of real-time

- *K. Baynes is with Verizon, Reston, VA. E-mail: Kathleen.Baynes@imake.com.*
- *C. Collins is with Intel Corp., Mail stop HD2-141, 77 Reed Rd., Hudson, MA 01749-2895. E-mail: christopher.m.collins@intel.com.*
- *E. Fiterman is with Salar, Inc., 5569 Gloucester Ave., Churchton, MD 20733. E-mail: efiterman@yahoo.com.*
- *B. Ganesh, C. Smit, and B. Jacob are with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742. E-mail: {brinda, blj}@eng.umd.edu, christinesmit@hotmail.com.*
- *P. Kohout is with EVI Technology, LLC, 7138 Columbia Gateway Dr., Columbia, MD 21046. E-mail: pkohout@evitechnology.com.*
- *T. Zhang is with 3e Technologies Inc., 19117 Willow Spring Dr., Germantown, MD 20874. E-mail: tiebingzhang@yahoo.com.*

*Manuscript received 2 May 2001; revised 9 May 2002; accepted 17 Sept. 2002. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114089.*

embedded systems. Their benefits are well-known: They provide numerous helpful facilities, including cooperative and preemptive multitasking, multithreading, support for both periodic and aperiodic tasks, fixed-priority/dynamic-priority scheduling, semaphores, interprocess communication, shared memory, and memory management; in doing so, they can dramatically reduce the time to design, develop, and test a product [17], [3], [18].

The third trend is the increasing importance of low-energy systems. There is rapidly growing consumer demand for computing devices that are both compute-intensive and battery operated, including PDAs, cell phones, wearable computers, handhelds, and laptops [6], [33]. 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 [34] and there has been a large number of recent studies modeling the power consumption of microprocessors and applications [22], [19], [20], [14], [7], [43], [38], [39], [31], [32], [13], but ours is one of the few experimental environments that performs both (the only other one of which we are aware is described in [10]).

### 1.2 SimBed

Our group has developed *SimBed*, a high-level 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). In this study, we present a processor model written in C that emulates the Motorola M-CORE microcontroller, a low-power, 32-bit CPU core with 16-bit instructions [40], [41]. All on-chip timers, interrupts, and interrupt handlers used by the operating systems and applications are precisely and accurately simulated. This is essential to the simulation of a system running an operating systems because it will be accessing these components very frequently. 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). The hardware used include 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 [38]. We have verified the simulator's output to measurements of actual hardware and our results are within 10-15 percent of real measurements. This level of accuracy for modeling power at the processor level is about where most current research stands (e.g., [7], [38]).

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 multitasking 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 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 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 Energy Consumption Model

Unlike instruction set simulation, which can be done by using some simple computation statements, there is no obvious way to determine the power and energy consumed by the system. Instead, we need to find out all the relevant factors and construct a mathematical model which will approximately reflect the power and energy consumption of the system.

The way in which the power estimation function was added to the instruction level emulator was first published by a Princeton group [39]. Instead of running simulations, this method is based on experiment data. The power consumption of each instruction is measured by using an infinite loop with only this instruction inside. Because it is an infinite loop, a jump instruction has to be used. In order to minimize the influence of this instruction, several hundred of the tested instructions are included inside the loop. The power number will be the base power consumption number of this specific instruction. This number multiplied by the execution time of the instruction will serve as the basic energy consumption of the instruction.

When a piece of code is running, all of the instructions' energy consumption numbers are simply added together to estimate the total energy consumption. However, according to another paper [7], this number will always be smaller than the real measured number. One explanation is that, during the single instruction test, the state of all the modules inside the processor will not change as much as when the next instruction is different from the previous one. This extra power consumption is called interinstruction overhead, which is different for different pairs of instructions, and this accounts for a big part of the processor's overall power consumption. Measuring all the instruction pairs' overhead power consumption is nearly impossible due to the large number of pairs. It was found that this number stays at a similar level for most instruction pairs. Therefore, a simple alternative method is to add a constant value to all the executed instruction pairs to compensate for this overhead value. With this added, the simulated power consumption number is close to the measured number and the error is within the acceptable region.

Another factor that influences the accuracy of this method is the changing of the operators of each instruction.

Moving an all zero constant and moving an all one constant will result in different circuit state changes and, therefore, different power consumptions. However, because the parameter of an instruction is random, we can use an average number to represent this fluctuation. Test results show that this method is a good approximation.

We verified the accuracy of our power model by running a small program in an infinite loop, then using a digital multimeter to measure the current that the chip consumes. Because the multimeter gives out an average current value and the program only needs several millisecond to run each iteration, the reading number will give out the average current consumption. With a simple multiplication with the power supply voltage, the average power consumption is found. After several such tests and comparing the results with the simulation results, the error is within 15 percent. This is an acceptable error considering that this is an architecture level tool.

### 1.4 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 *online* scheduling (the choices are made at runtime as opposed to compile-time), as they tend to be less amenable to analytical verification than systems with *offline* 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:** A preemptive multitasking RTOS that is in the public domain [24]. 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 (1,700 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 [36] that swaps Chimera's POSIX-like threads in the microkernel for port-based objects [37]; it supports reconfigurable component-based software for microcontrollers and digital signal processors [12]. 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, multirate executive based on descriptions of "roll-your-own" RTOSs given by embedded-systems designers in industry [16]. 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 nonpreemptive 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 multirate 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 multirate executive is much less so [23].

On each of these execution models, we execute several different applications. Following Liu's terminology [28], 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 multitasking to the extent possible in the given OS ($\mu$C/OS provides preemptive multitasking, Echidna provides cooperative multitasking, and NOS schedules work on function boundaries) and use for all data transfer whatever interprocess 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 application kernels 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. Additionally, we also use some applications from the Mediabench suite g721-encode, decode and Adpcm-encode, decode. 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.

### 1.5 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 miss deadlines with predictable regularity and with probability 1, even when the system is under light load. This is due to unexpected interrupts and unaccounted-for task invocations that cause individual job timing to be thrown off, but only occasionally. 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 head-room: 10-20 percent idle CPU cycles is not too much.

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 percent 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 as quickly as the simpler RTOS when the user-level computational load grows. Therefore, the energy consumption and CPU requirements of these more complex RTOSs 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: $\mu$C/OS-II, Echidna, and NOS—the public-domain kernel, the industrial RTOS, and the simple multirate 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 [29], [30], [40], [41]. 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 1$\mu$s. It is a 2-byte counter in I/O space that increments every 1.6$\mu$s. 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 Interprocess Communication.** Periodic interprocess 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 use RTOS-provided IPC (e.g., shared memory). The second job receives that value 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 the last time that second job was 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 a computation intensive benchmark. 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 input by the first job are used to compute an inner product and that value is output to the I/O port.

**G721 Decode.** G721 is an application in the Mediabench suite. It decodes a G.721 voice compressed sample. The first job reads in a previously compressed sample which the second job decodes 4 bits at a time.

**ADPCM Encode.** Adpcm is an application in the Mediabench suite. It performs a conversion from a linear 16 bit PCM sample to a 4 bit ADPCM sample. The first job reads in a single sample which the second job encodes and writes to an I/O port. This is the most computation intensive of the benchmarks used.

**Background Load.** To add some nondeterminism 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 interprocess communication process.

**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 Interprocess Communication.** This task is run when a simulated I/O interrupt is generated by the hardware. It schedules a high-priority user-level job in response that writes to the I/O space. This is the mechanism used to determine system response time under load. The interrupt interarrival times obey a geometric distribution: The emulator generates an interrupt every 100$\mu$s 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.

### 2.3 Characterization of Real-Time Behavior

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

**Jitter.** Jitter is measured by keeping track of interarrival times of periodic output. For example, if a task is scheduled to generate an output value every 10 milliseconds, its average interarrival time should be 10 milliseconds. Any variation in the interarrival time represents output that fails to arrive on time.

Note that this differs slightly from the traditional definition which is based on the RTOS's knowledge of a missed deadline 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. The execution time for NOS and Echidna is divided into user, kernel, handler, semaphore, and idle components. The execution time for $\mu$C/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. We also study the effect of using the doze instruction on the energy consumption of the RTOS.

More detail on the MCORE processor, SimBed's internals, applications, and RTOS models can be found elsewhere [4], [5], [8], [44].

## 2.4 Real-Time Kernels

### 2.4.1 The uC/OS-II Kernel

The $\mu$C/OS-II real-time kernel is a full-featured preemptive multitasking RTOS [24]. It is portable, targeted at both microcontrollers and DSPs, and it currently runs on over 50 different instruction-set architectures. It is designed to have a small footprint: There are roughly 1,700 lines of code in the OS (including comments) and modules are only compiled into the executable if used by the application. Multitasking is preemptive and the kernel can preempt itself. The system can run up to 64 tasks, with eight 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 $\mu$C/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 ($\mu$C/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 wake up messages when the job periods are reached. The interprocess communication method is message-passing.

### 2.4.2 The Echidna RTOS

Echidna is a scaled down version of the Chimera RTOS [36] that replaces Chimera's concept of a process (which is notionally similar to that of POSIX threads) with port-based objects [37]. 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 [12].

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 interprocess 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 100$\mu$s 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 nonpreemptive systems.

### 2.4.3 The NOS Multirate 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 [16]. NOS is a fixed-priority multirate executive for periodic tasks [23] and handles interrupt-driven stimuli via masking interrupts and polling the interrupt status registers when idle. Its main control loop is shown in Fig. 1.

NOS's callout queue is taken from the callout table in Unix [2]; 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.

## 2.5 Simulation Support for Real-Time Operating Systems

For a real-time operating system to succesfully run on a microprocessor, the processor must provide the RTOS with

```
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);
}
```

Fig. 1. NOS main loop—simple multirate executive with fixed priority scheme. Design based on descriptions of RTOSs built by designers in industry [16], 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.

several features. First, the processor must allow the RTOS to accurately determine the current time and how this relates to the external clock time. Second, if interrupts are not polled, the hardware must be able to preempt the current instruction stream upon arrival of an interrupt, save the relevant system state, and start executing a predetermined interrupt service routine. These are the only essential functions that the processor needs to run an RTOS.

The simulator must accurately model all of the processor's functionality that is used by the real-time operating system. In addition to being cycle-accurate, the simulator must correctly support the above-mentioned essential functions of a processor. For the MCORE processor, this includes interrupts, exceptions, and hardware timers, all of which have been implemented and validated.

## 3 Experimental Results

For these studies, we execute the following benchmarks: periodic IPC (*P-IPC*), up-sampling (*UP*), down-sampling (*DOWN*), a 128-tap FIR filter (*FIR*, ADPCM encode and g721 decode. 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 interarrival times obey a geometric distribution (we generate an interrupt every 100$\mu$s with probability 0.01, resulting in an average of 100 AP-IPC interrupts per second). We varied the following parameters:

- RTOSs: {$\mu$C/OS-II, Echidna, NOS}
- Periodic tasks: {P-IPC, UP, DOWN, FIR, ADPCM Encode, G721 Decode}
- 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 500$\mu$s 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 Real-Time 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 nonzero y-values are shown and values have been gathered into 100 $\mu$s intervals.[1]

Fig. 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 interprocess communication 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, Fig. 2d shows that, when Echidna is running eight tasks with 16ms periods (16 jobs), roughly 20 percent of the jobs will execute exactly on-time, 40 percent of the jobs will execute a little early, and 40 percent of the jobs will execute a little late; roughly 1 percent of the time the job executions will be 50$\mu$s off, in either direction. When the system load is four tasks (eight jobs), job executions are on-time roughly 85 percent of the time and missed deadlines are either too early or too late with roughly equal probability and absolute value. When executing one or two 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 Fig. 2c, Fig. 2h, etc.). For Echidna, this point is around 500Hz with eight IPC tasks running; for $\mu$C/OS and NOS, the point is above 1MHz, even for eight tasks running.

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.

Fig. 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 $\mu$C/OS execute a number of tasks too late—and an equal number of tasks too early. Moreover, the number of early/late job invocations does not seem to scale with workload (for example, $\mu$C/OS at task periods of 16ms cannot get more than 50 percent of the tasks to execute on-time when the system is perturbed by occasional interrupts (see Fig. 3l). This behavior is caused in both RTOSs by task self-interference. This is specific to tasks with jobs that run with different 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 $\mu$C/OS is the aperiodic IPC interrupt that happens on average every 10ms. Because $\mu$C/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. The 16ms tasks are upset most by this (see Fig. 3l) because the interrupt displaces a user thread on roughly every other job invocation (thus, only 50 percent 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 Fig. 4. The top row represents Echidna without any background load. The second row of graphs is Echidna with background load. The third row is $\mu$C/OS without background load and the last row is $\mu$C/OS with background load. We see that both RTOSs allow

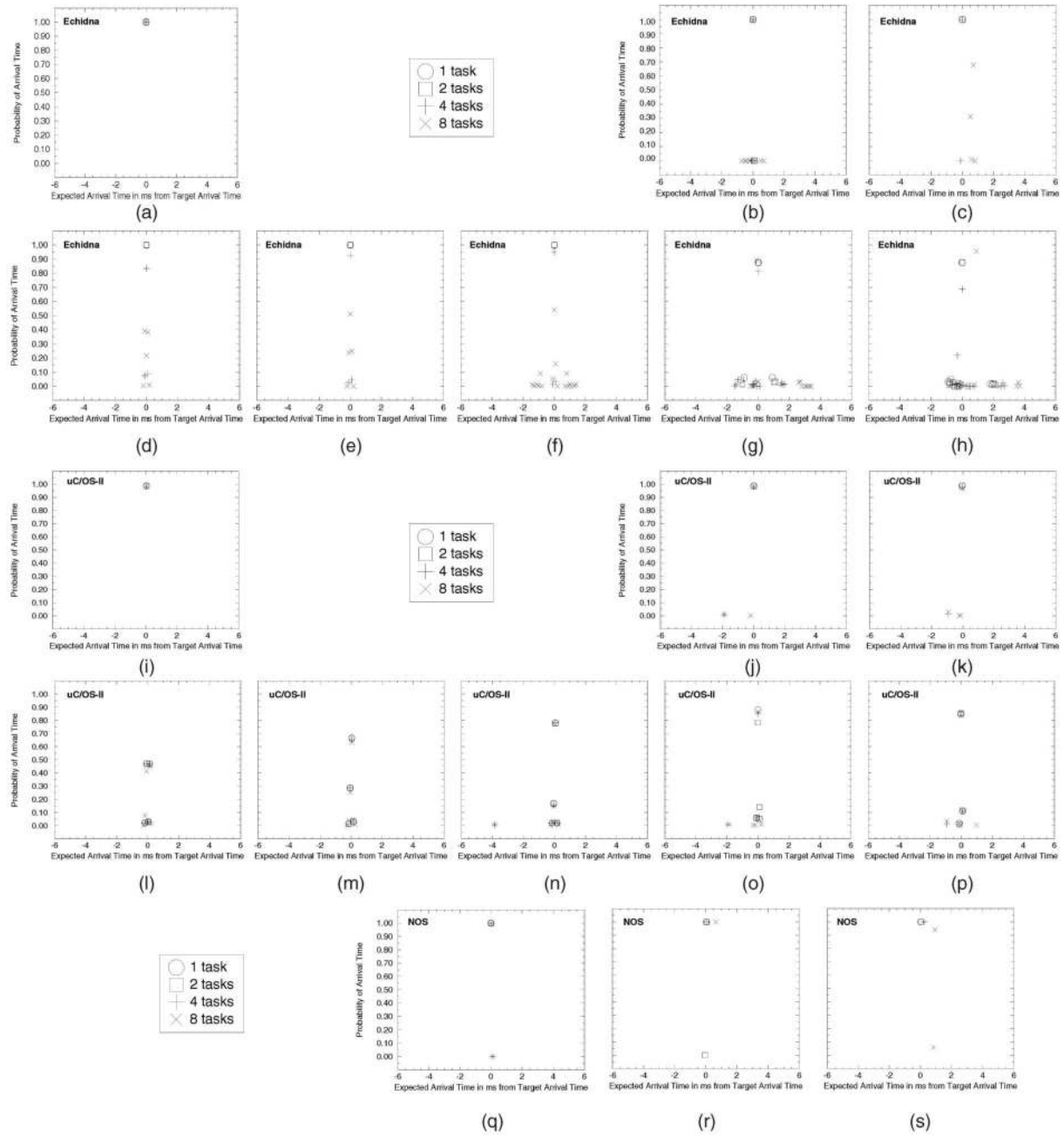


Fig. 2. JITTER probability density graphs for P-IPC. The x-axis represents time deltas btween successive I/O output events as they differ from the expected period. Negative numbers mean a task ran early and positive numbers mean a task has run late in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of one, two, four, and eight simultaneous tasks. (a) Background = None, 16ms period. (b) Background = None, 2ms period. (c) Background = None, 1ms period. (d) Background = AP·IPC+CL, 16ms period. (e) Background = AP·IPC+CL, 8ms period. (f) Background = AP·IPC+CL, 4ms period. (g) Background = AP·IPC+CL, 2ms period. (h) Background = AP·IPC+CL, 1ms period. (i) Background = None, 16ms period. (j) Background = None, 2ms period. (k) Background = None, 1ms period. (l) Background = AP·IPC+CL, 16ms period. (m) Background = AP·IPC+CL, 8 ms period. (n) Background = AP·IPC+CL, 4ms period. (o) Background = AP·IPC+CL, 2ms period. (p) Background = AP·IPC+CL, 1ms period. (q) Background = AP·IPC+CL, 1ms period. (r) Background = AP·IPC+CL, 250$\mu$s period. (s) Background = AP·IPC+CL, 64$\mu$s period.

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 this instance), so task invocations will coincide in time every Nth invocation. Neither operating system manages to spread the tasks out in time.

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 $\mu$C/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 invoations will be on time. However, this is not a panacea: As the workload increases, this

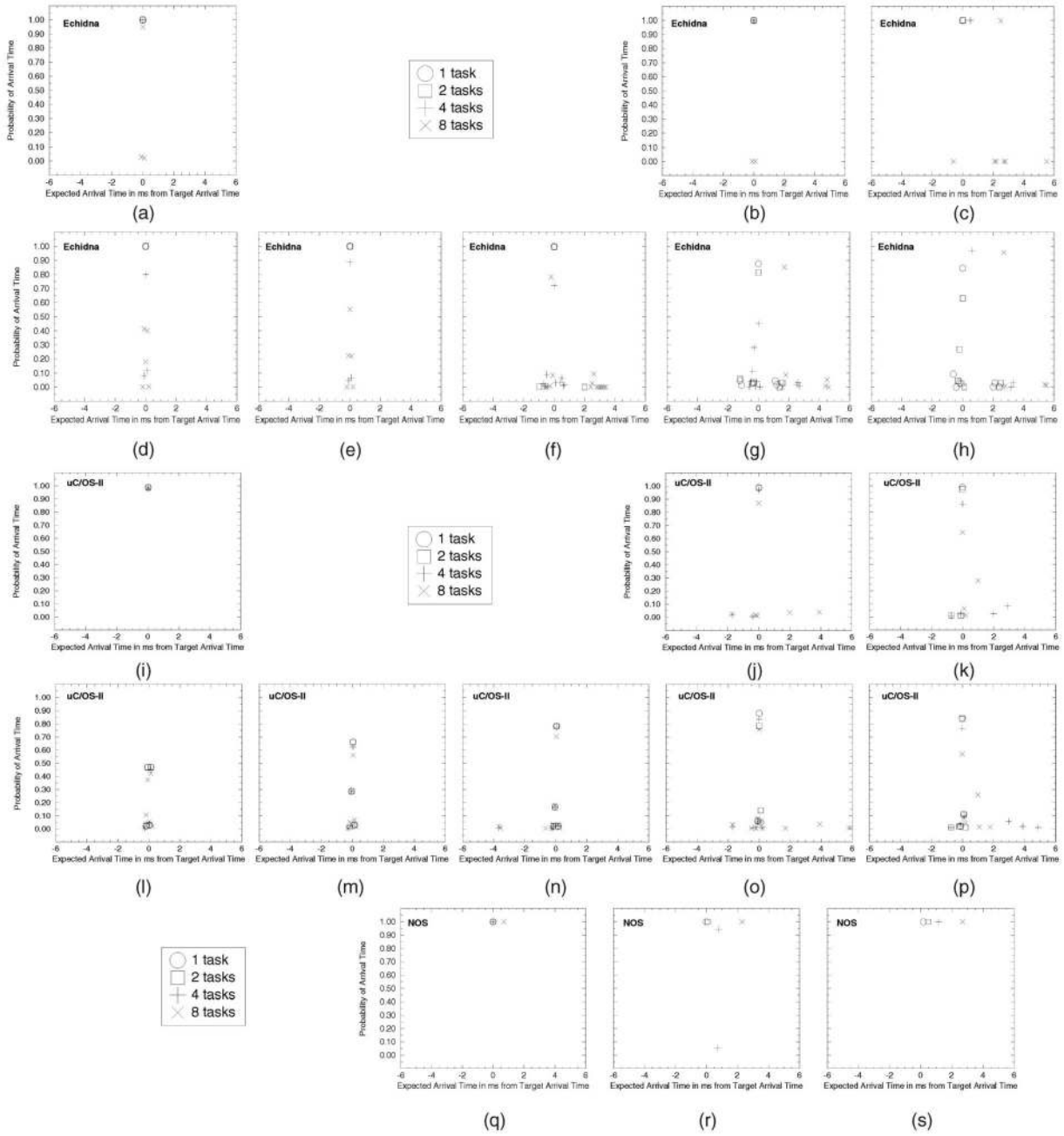


Fig. 3. JITTER probability density graphs for FIR. The x-axis represents time deltas between successive I/O output events as they differ from the expected period. Negative numbers mean a task rean early and positive numbers mean a task has run late in relation to the last task run. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of one, two, four, and eight simultaneous tasks. (a) Background = None, 16ms period. (b) Background = None, 2ms period. (c) Background = None, 1ms period. (d) Background = AP·IPC+CL, 16ms period. (e) Background = AP·IPC+CL, 8 ms period. (f) Background = AP·IPC+CL, 4ms period. (g) Background = AP·IPC+CL, 2ms period. (h) Background = AP·IPC+CL, 1ms period. (i) Background = None, 16ms period. (j) Background = None, 2ms period. (k) Background = None, 1ms period. (l) Background = AP·IPC+CL, 16ms period. (m) Background = AP·IPC+CL, 8ms period. (n) Background = AP·IPC+CL, 4ms period. (o) Background = AP·IPC+CL, 2ms period. (p) Background = AP·IPC+CL, 1ms period. (q) Background = AP·IPC+CL, 1ms period. (r) Background = AP·IPC+CL, 500$\mu$s period. (s) Background = AP·IPC+CL, 250$\mu$s period.

relative scheduling cannot prevent late invocations as NOS is nonpreemptive and, therefore, low-priority tasks can delay high-priority tasks.

### 3.2 Response-Time 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 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 $\mu$C/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 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-

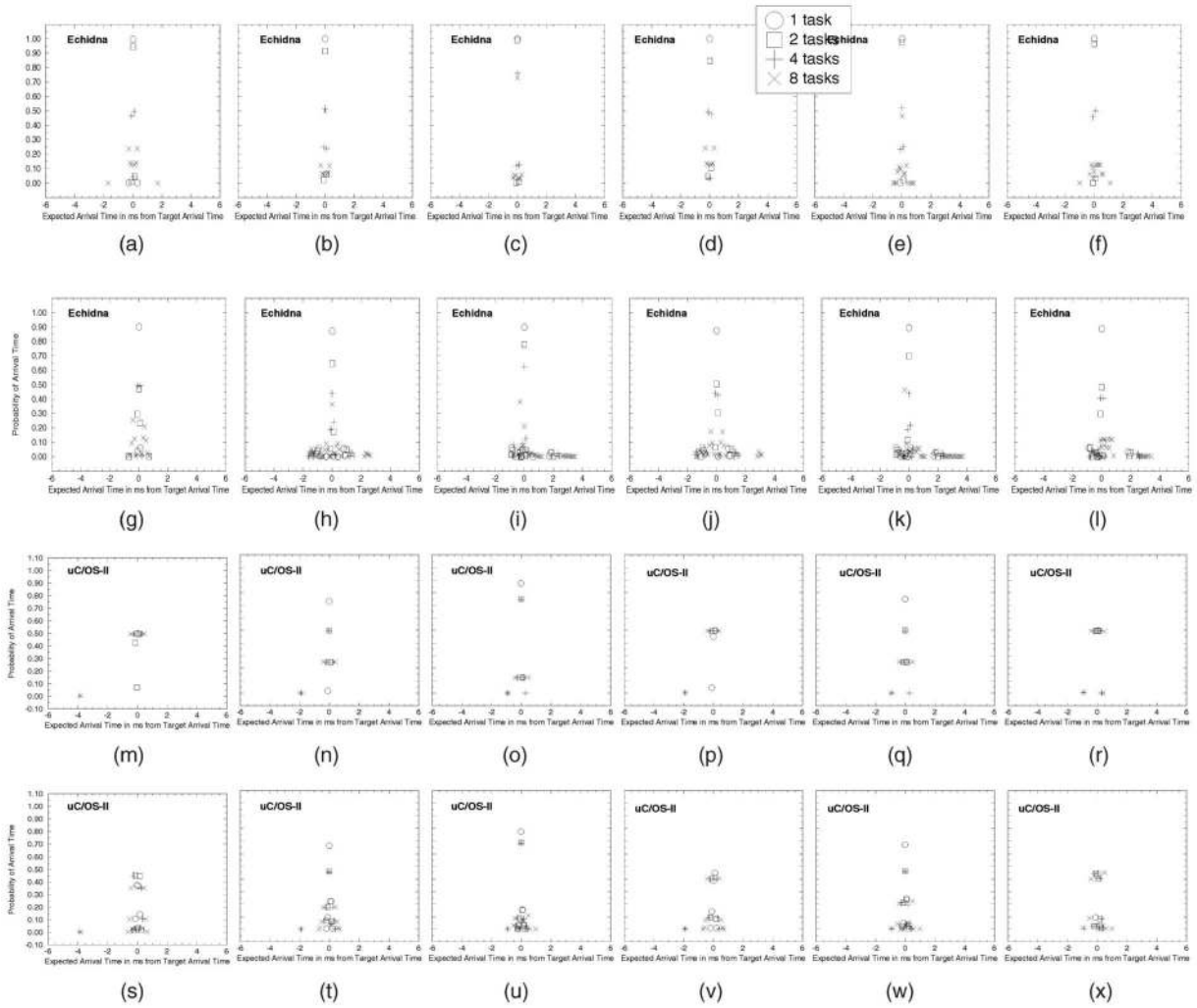


Fig. 4. JITTER probability density graphs for UP. The x-axis represents time deltas between successive I/O output events as they differ from the expected period—negative numbers indicate the output happened early. The y-axis indicates the probability of each delta. The legend shows the symbols used to represent system load of one, two, four, and eight simultaneously executing tasks. (a) Background = None, 8/4ms period. (b) Background = None, 8/2ms period. (c) Background = None, 8/1ms period. (d) Background = 1, 4/2ms period. (e) Background = None, 4/1ms period. (f) Background = None, 2/1ms period. (g) Background = AP·IPC+CL, 8/4ms period. (h) Background = AP·IPC+CL, 8/2ms period. (i) Background = AP·IPC+CL, 8/1ms period. (j) Background = AP·IPC+CL, 4/2ms period. (k) Background = AP·IPC+CL, 4/1ms period. (l) Background = AP·IPC+CL, 2/1ms period. (m) Background = None, 8/4ms period, (n) Background = None, 8/2ms period. (o) Background = None, 8/1ms period. (p) Background = None, 4/2ms period. (q) Background = None, 4/1ms period. (r) Background = None, 2/1ms period. (s) Background = AP·IPC+CL, 8/4ms period. (t) Background = AP·IPC+CL, 8/2ms period. (u) Background = AP·IPC+CL, 8/1ms period. (v) Background = AP·IPC+CL, 4/2ms period. (w) Background = AP·IPC+CL, 4/1ms period. (x) Background = AP·IPC+CL, 2/1 ms period.

handler 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 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 Fig. 5. These represent the range of CPU load from very light (1 G721 Decode task, 16ms period) to very heavy (8 Adpcm Encode tasks, 80 ms 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 $\mu$C/OS-II kernel handles interrupts with absolute precision that is independent of application load. The NOS system has the simplest mechanism of all because its cooperatively scheduled nature means no state needs to be saved on task switch, so we should expect to see zero response time at light workloads. But, using the "idle" mode affects this response time adversely, as we see in Fig. 5g. When the system idles in NOS, the kernel executes the "doze" instruction. The system will awake from doze mode at every timer interrupt. As a result, the response time at light workloads is distributed uniformly in a time interval of one

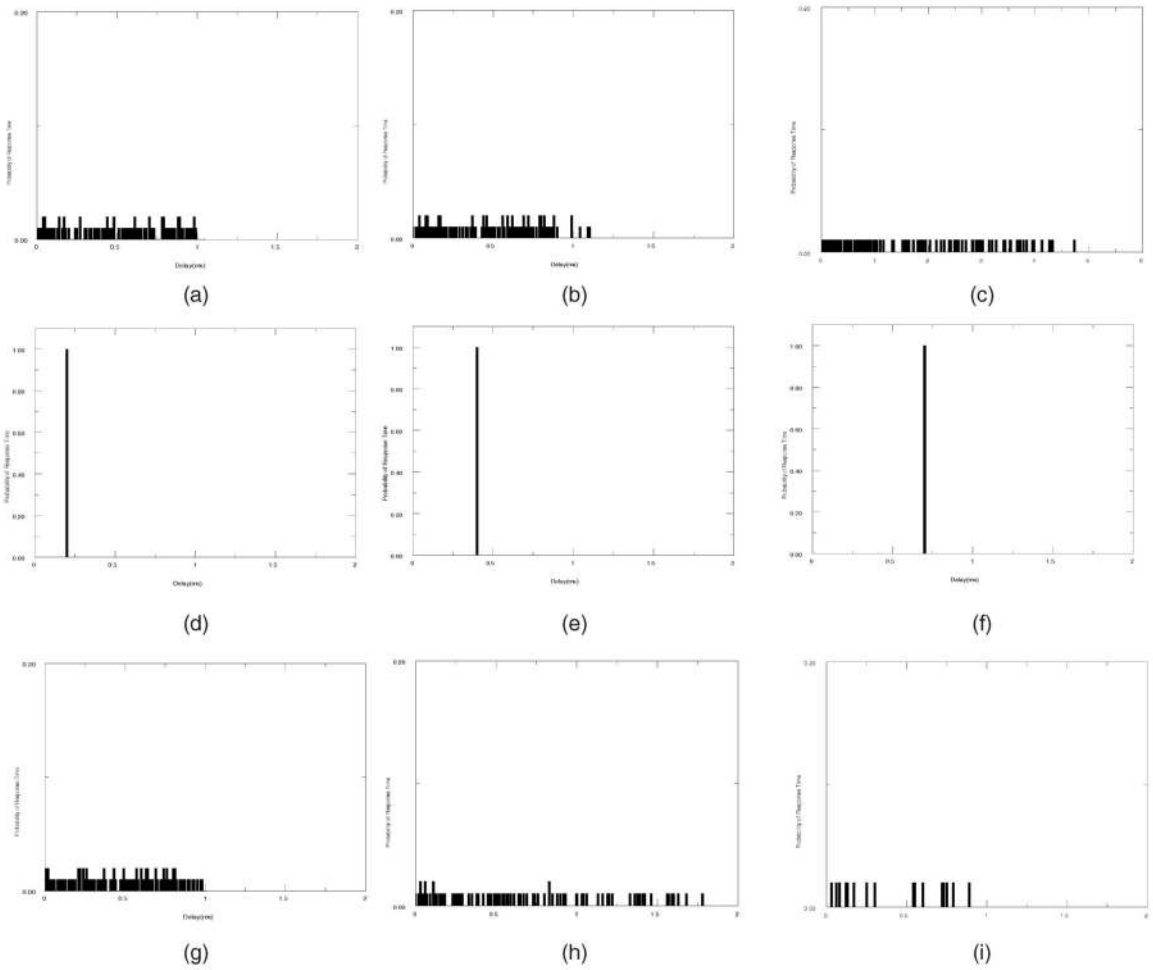


Fig. 5. DELAY probability density graphs for ECHIDNA and NOS. The x-axis represents time between an interrupt being generated by an I/O device and the corresponding output to an I/O port of the responding thread. The y-axis indicates the probability of each delta. All measurements are for configurations with both types of background load (3Hz 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 (one G721 Decode task) to heavy foreground load (eight Adpcm Encode tasks). Note that the y-axis scale is different for the uC/OS graphs and that the x-axis scales are different in (c) and (i). (a) ECHINDA: 1 G721 Decode task, atms period. (b) ECHIDNA: 4 G721 Decode tasks, 2ms period. (c) ECHIDNA: 8 ADPCM Encode tasks, 80ms period. (d) uC.OS-II: 1 G721 Decode task, 16ms period. (e) uC/OS-II: 4 G721 Decode tasks, 1ms period. (f) uC/OS-II: 8 ADPCM Encode tasks, 80ms period. (g) NOS: 1 G721 Decode task, 16ms period. (h) NOS: 4 G721 Decode tasks, 1ms period. (i) NOS: 8 ADPCM Encode tasks, 80ms period.

operating system time tick - 1ms. 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.

The preemptive $\mu$C/OS-II kernel handles interrupts with relative precision. Yet the figures show that its overhead varies slightly from benchmark to benchmark. The variation is due to the RTOS's implementation of preemptive scheduling: A task can made ready when the hardware timer tick occurs. This event causes 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 the scheduler traverses the entire TCB list on a timer tick, the time to complete the tick 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.

### 3.3 Energy Consumption

To measure energy consumption, we ran each configuration for the same number of application iterations. The results are shown in Figs. 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 [10]. Results are only shown for the applications with the least (IPC) and greatest (FIR) overhead per job invocation.

The IPC results in Fig. 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 20; Echidna and $\mu$C/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.

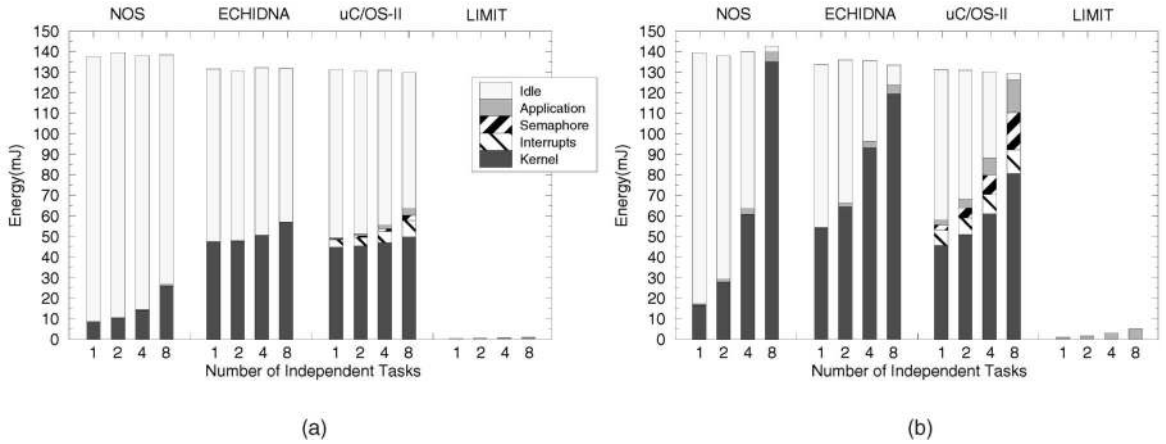


Fig. 6. Energy consumption graphs for IPC. The x-axis represents increasing workloads as a result of increasing the number of executing tasks or. The y-axis represents the total CPU energy consumption and breakdowns for how much energy is consumed by executing kernel code, executing user application 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 Echnidna. (a) Task period: 16ms. (b) Task period: 1ms.

The FIR results in Fig. 7 show that, for more realistic applications (bear in mind that FIR is still relatively light in computation time at ~233$\mu$s per invocation), RTOS kernel overhead is more reasonable. The use of the NOS scheduler increases energy consumption by less than a factor of two and the Echidna and $\mu$C/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 $\mu$C/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 $\mu$C/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 component of the clock-tick interrupt handler in $\mu$C/OS-II that runs every clock tick and wakes up sleeping threads when it determines that their periods have expired. 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 McORE 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 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, in the FIR graphs, compare NOS:8 in Fig. 7a, NOS:4 in Fig. 7b, NOS:2 in Fig. 7c, and NOS:1 in Fig. 7d, which represent different trade offs of speed and number of tasks. The user components are the same for these configurations as the configurations all represent the same amount of work: 2,000 job invocations per second, broken down as (respectively) 16 jobs, each scheduled every 8ms, eight jobs, each scheduled every 4ms, four jobs, each scheduled every 2ms, and two 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.

We study the effect of using the doze mode provided by the processor on the energy consumption of two operating systems—NOS and UCOS. The graphs in Fig. 8 are for the two operating systems using the idle instruction.

The idle mode on the Mcore can be reached by executing a simple doze instruction. This reduces the current drawn by the processor by nearly 50 percent. The processor leaves this mode only when an interrupt arrives.The two operating systems—NOS and $\mu$COS can use this mode very easily. These operating systems maintain a system clock whose granularity is determined by the rate of the timer interrupt.

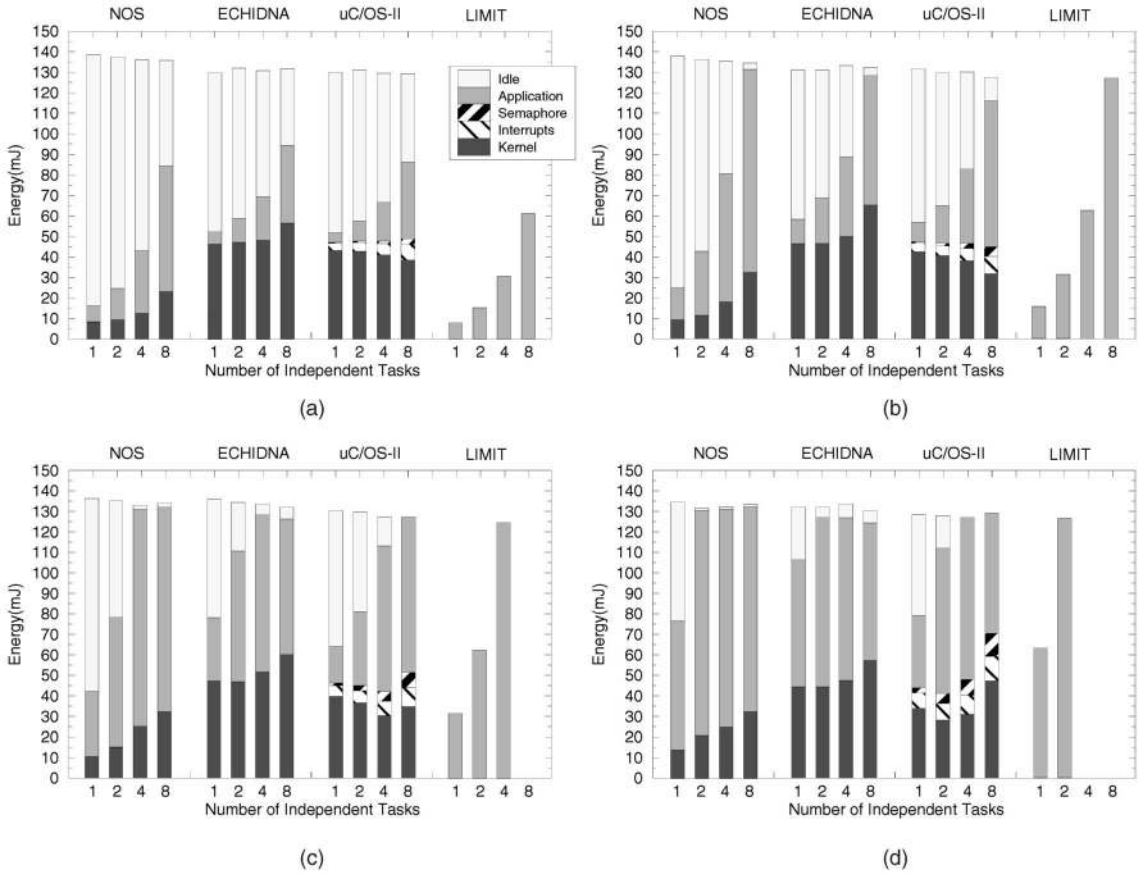


Fig. 7. Energy consumption graphs for FIR. The x-axis represents increasing workloads as a result of increasing the number of executing tasks or. The y-axis represents the total CPU energy consumption and breakdowns for how much energy is consumed by executing kernel code, executing user application 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. (a) Task period: 8ms. (b) Task period: 4ms. (c) Task period: 2ms. (d) Task period: 1ms.

We peg this to 1 kHz or a system clock granularity in the order of milliseconds. When the system executes the idle task, it implies that there are no tasks on the task queue which need to be executed in that particular timer tick. The operating system can thus safely execute the doze instruction at this juncture. The system will stay in low power mode till the next interrupt—when it will update its clock and timeout queue. As the systems sense of time is tied directly with the timer interrupt, one can safely use the doze instruction without losing time.

By contrast, Echidna's system clock is updated by a regular timer interrupt every 1/5th of a second. The time in between is computed by interpolation using the value of the timer interrupt counter. When the system is idle, the scheduler polls the system clock continuosly in order to determine when the first waiting task can be released. Since the order of the delays is finer than that of the timer interrupt, invoking the doze mode may cause tasks in the queue to miss their deadlines. This prevents us from using the doze instruction for Echidna.

- The graphs represent data for the adpcm encode benchmark from the media bench suite. Utilization of the doze mode yields great energy savings. When the system is lightly loaded, as in the case of one instance of the application running at the different rates, the energy consumed by the system is in the order of 1200mJ. This can be contrasted with a fully loaded system which has eight adpcm encode tasks running with periods of 80 ms each when the system consumes roughly three times the energy.
- We also have a break up of the energy consumed by the kernel in the case of $\mu$COS. The overhead of the kernel scales with workload. The bulk of the RTOS overhead is seen in the timer interrupt. This is because every timer tick the RTOS walks through the task queue and this operation scales linearly with the number of tasks in the system. Thus, the timer interrupt energy is the same across all configurations for a given number of tasks, as can be seen for the various instances in the graphs a, b, c, d.

## 4 Future Work

This paper characterizes the performance 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 digital signal processor.

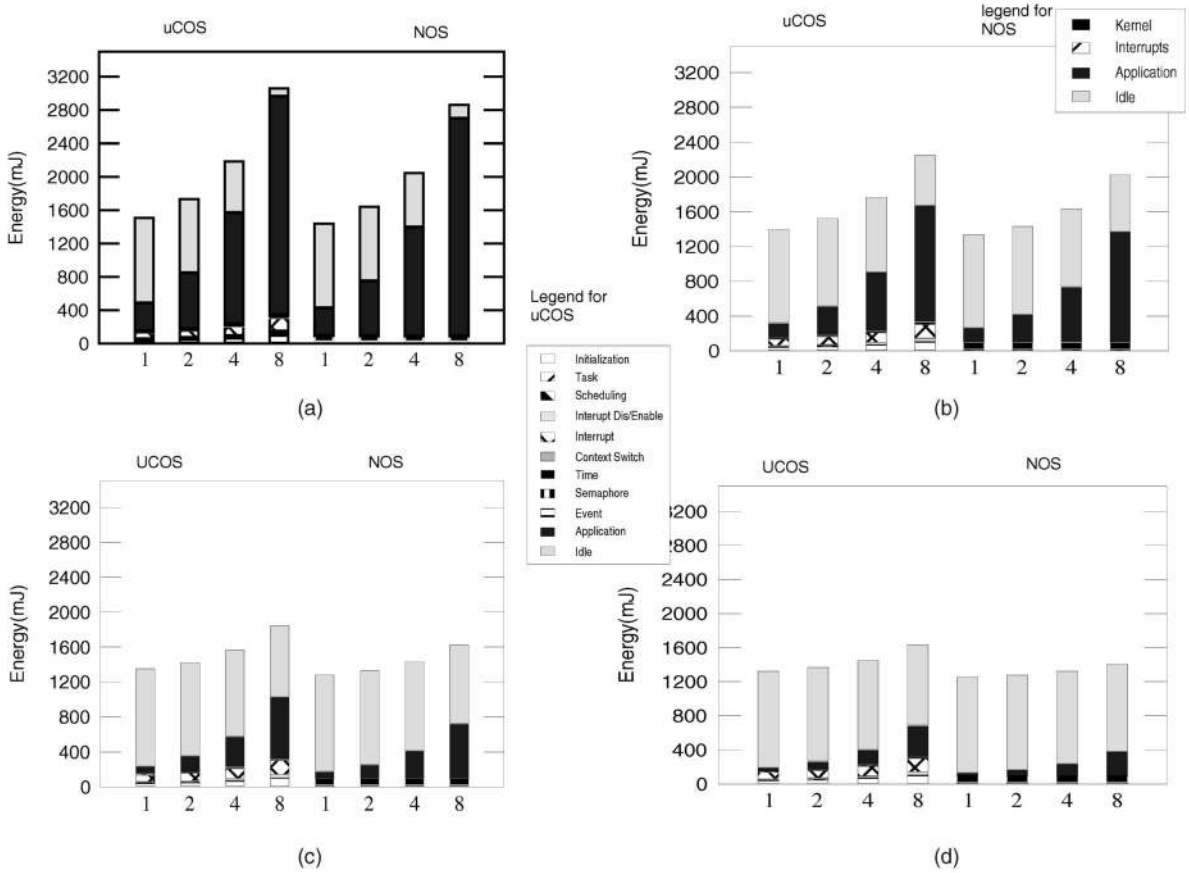


Fig. 8. Energy consumption graphs for ADPCM ENCODE. The x-axis represents increasing workloads as a result of increasing the number of execting tasks or. The y-axis represents the total CPU energy consumption and breakdowns for how much energy is consumed by executing kernel code, executing user application code, handling interrupts, performing semaphore handling, and sitting idle for NOS. For uCOS, the kernel breakdown is further divided. (a) Task period: 80ms. (b) Task period: 160ms. (c) Task period: 320ms. (d) Task period: 640ms.

This common DSP appears in several modern products, including devices for wireless communication, broadband communication, audio/video processing, and encryption. 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 1,600 MIPS.

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 [25], are being utilized.

Another goal of future research is to propose new hardware components intended to improve the performance of RTOSs running on embedded microprocessors and digital signal processors. Such hardware components are often found on desktop computer processors, but rarely in the embedded market. Detailed analyses of the effect of these components will be performed on various RTOSs that fairly represent those used in industry.

This research will significantly contribute to the development of embedded microprocessors, as well as real-time operating systems.

## 5 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 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 percent.

We presented a study of preemptive and nonpreemptive 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 nonpreemptive 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 percent or more—but that the overhead diminishes significantly for more compute-intensive applications (down to 50 percent for Echidna and $\mu$C/OS-II, 30 percent 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.

We also saw that utilization of the idle mode can effectively lower the energy consumption in the system, but may come at the cost of worse response to polled interrupts as seen in the case of NOS.

## ACKNOWLEDGMENTS

The work of Kathleen Baynes and Christine Smit was supported in part by the US National Science Foundation's (NSF) 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. Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, and Tiebing Zhang were students at the University of Maryland while working on the research presented in this paper.

## REFERENCES

[1] A. Allara, C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "System-Level Performance Estimation Strategy for SW and HW," *Proc. Int'l Conf. Computer Design,* Oct. 1998.

[2] M.J. Bach, *The Design of the UNIX Operating System.* Englewood Cliffs, N.J.: Prentice Hall, 1986.

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

[4] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, "The Performance and Energy Consumption of Three Embedded Real-Time Operating Systems," *Proc. Fourth Workshop Compiler and Architecture Support for Embedded Systems (CASES '01),* pp. 203-210, Nov. 2001.

[5] K. Baynes, C. Collins, E. Fiterman, C. Smit, T. Zhang, and B. Jacob, "The Performance and Energy Consumption of Embedded Real-Time Operating Systems," Technical Report UMD-SCA-TR-2000-04, Univ. of Maryland Systems & Computer Architecture Group, Nov. 2000.

[6] L. Benini and G.D. Micheli, "System-Level Power Optimization: Techniques and Tools," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 288-293, Aug. 1998.

[7] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 83-94, June 2000.

[8] C.M. Collins, "An Evaluation of Embedded System Behavior Using Full-System Software Emulation," Master's thesis, Univ. of Maryland at College Park, May 2000.

[9] Design & Test Roundtable, "Hardware-Software Codesign," *IEEE Design and Test of Computers,* vol. 14, no. 1, pp. 75-83, Jan.-Mar. 1997.

[10] R.P. Dick, G. Lakshminarayana, A. Raghunathan, and N.K. Jha, "Power Analysis of Embedded Operating Systems," *Proc. 37th Design Automation Conf.,* pp. 312- 315, June 2000.

[11] C. Ellis, "The Case for Higher-Level Power Management," *Proc. Workshop Hot Topics in Operating Systems,* 1999.

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

[13] J. Flinn and M. Satyanarayanan, "Powerscope: A Tool for Profiling the Energy Usage of Mobile Applications," *Proc. Workshop Mobile Computing Systems and Applications,* pp. 2-10, Feb. 1999.

[14] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughy, D. Patterson, T. Anderson, and K. Yelick, "The Energy Efficiency of IRAM Architectures," *Proc. 24th Ann. Int'l Symp. Computer Architecture (ISCA '97),* pp. 327-337, June 1997.

[15] J. Ganssle, "Conspiracy Theory," *The Embedded Muse,* no. 46, 3 Mar. 2000.

[16] J. Ganssle, "Conspiracy Theory, Take 2," *The Embedded Muse,* no. 47, 22 Mar. 2000.

[17] J.G. Ganssle, "An OS in a Can," *Embedded Systems Programming,* Jan. 1994.

[18] J.G. Ganssle, "The Challenges of Real-Time Programming," *Embedded Systems Programming,* vol. 11, no. 7, pp. 20-26, July 1997.

[19] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *IEEE J. Solid-State Circuits,* vol. 31, no. 9, pp. 1277-1284, Sept. 1996.

[20] J.K.M. Gupta and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," *Proc. 30th Ann. Int'l Symp. Microarchitecture (MICRO '97),* pp. 184-193, Dec. 1997.

[21] J. Hennessy and M. Heinrich, "Hardware/Software Codesign of Processors: Concepts and Examples," *Hardware/Software Co-Design,* G. De Micheli and M. Sami, eds., pp. 29-44, Kluwer Academic, 1996.

[22] M. Horowitz, T. Indermaur, and R. Gonzalez, "Low-Power Digital Design," *Proc. IEEE Symp. Low Power Electronics,* pp. 8-11, Oct. 1994.

[23] D. Kalinsky, "A Survey of Task Schedulers," *Proc. Embedded Systems Conf.,* Sept. 1999.

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

[25] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proc. 30th Ann. Int'l Symp. Microarchitecture (MICRO '97),* pp. 330-335, Dec. 1997.

[26] Y. Li, M. Potkonjak, and W. Wolf, "Real-Time Operating Systems for Embedded Computing," *Proc. Int'l Conf. Computer Design,* Oct. 1997.

[27] 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, Apr.-June 1997.

[28] J.W.S. Liu, *Real-Time Systems.* Upper Saddle River N.J.: Prentice Hall, 2000.

[29] Mcore, *M-CORE Reference Manual.* Denver, Colo.: Motorola Literature Distribution, 1997.

[30] Mcore, *M-CORE MMC2001 Reference Manual.* Denver, Colo.: Motorola Literature Distribution, 1998.

[31] K. Roy and M.C. Johnson, "Software Design for Low Power," *NATO Advanced Study Inst. on Low Power Design in Deep Submicron Electronics,* Aug. 1996.

[32] J. Russell and M. Jacome, "Software Power Estimation and Optimization for High Performance, 32-Bit Embedded Processors," *Proc. Int'l Conf. Computer Design,* Oct. 1998.

[33] J. Scott, L. Lee, A. Chin, J. Arends, and B. Moyer, "Designing the M.CORE M3 CPU Architecture," *Proc. Int'l Conf. Computer Design,* Oct. 1999.

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

[35] M.J. Smith, *Application-Specific Integrated Circuits.* Reading Mass.: Addison-Wesley, 1997.

[36] D.B. Stewart, D.E. Schmitz, and P.K. Khosla, "The Chimera II Real-Time Operating System for Advanced Sensor-Based Applications," *IEEE Trans. Systems, Man, and Cybernetics,* vol. 22, no. 6, pp. 1282-1295, Nov./Dec. 1992.

[37] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Trans. Software Eng.,* vol. 23, no. 12, pp. 759-776, Dec. 1997.

[38] V. Tiwari and M.T.-C. Lee, "Power Analysis of a 32-bit Embedded Microcontroller," *VLSI Design J.,* vol. 7, no. 3, 1998.

[39] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step towards Software Power Minimization," *IEEE Trans. VLSI Systems,* vol. 2, no. 4, pp. 1277-1284, Dec. 1994.

[40] J. Turley, "M.Core Shrinks Code, Power Budgets," *Microprocessor Report,* vol. 11, no. 14, pp. 12-15, Oct. 1997.

[41] J. Turley, "M.Core for the Portable Millenium," *Microprocessor Report,* vol. 12, no. 2, pp. 15-18, Feb. 1998.

[42] A. Vahdat, A. Lebeck, and C. Ellis, "Every Joule Is Precious: The Case for Revisiting Operating System Design for Energy Efficiency," *Proc. SIGOPS European Workshop,* Sept. 2000.

[43] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye, "Energy-Driven Integrated Hardware-Software Optimizations Using Simplepower," *Proc. 27th Ann. Int'l Symp. Computer Architecture (ISCA '00),* pp. 95-106, June 2000.

[44] T. Zhang, "RTOS Performance and Energy Consumption Analysis Based on an Embedded System Testbed," Master's thesis, Univ. of Maryland at College Park, May 2001.

**Kathleen Baynes** received the Bachelor of Science degree in computer engineering from the University of Maryland in May of 2001. She is currently doing consulting work involving systems integration at Verizon in Reston, Virginia.

**Chris Collins** received the BS degree with honors in electrical engineering from the University of Virginia in 1998 and the MS degree in electrical engineering from the University of Maryland, College Park in 2000. He is currently a senior architecture modeling engineer at Intel Corporation in Hudson, Massachusetts, and is working on the Intel IXP2800 Network Processor.

**Eric Fiterman** received the BS degree in computer science from the University of Maryland, College Park in 2000, and the BS degree in electrical engineering from the University of Maryland, College Park in 2000. He has worked as an embedded software engineer and protocols engineer for Ericsson IP Infrastructure, developing unicast routing and tunnel protocols. He presently works for Salar, Inc., a successful startup company, developing enterprise database applications and wireless/mobile software for the healthcare industry. He is a member of the IEEE.

**Brinda Ganesh** received the BE degree in electronics and communication from the Karnataka Regional Engineering College, Suratkal, India, in 1999 and the MS degree in computer engineering from the University of Maryland, College Park in 2002. She is currently pursuing the PhD degree in the Department of Electrical and Computer Engineering at the University of Maryland, College Park. Her research interests include hardware and software for embedded and real-time systems.

**Paul Kohout** received the BS and MS degree in electrical engineering from the University of Maryland in 2000 and 2002, respectively. While at Maryland, he studied computer architecture and embedded systems and he wrote his thesis on hardware support for real-time operating systems. He recently started working as an electrical engineer for EVI Technology in Columbia, Maryland. EVI specializes in designing and manufacturing digital and RF devices for the government. At EVI, Paul is responsible for designing embedded hardware and software. He is a member of the IEEE and the IEEE Computer Society.

**Christine Smit** is a senior computer engineering and vocal performance double major at the University of Maryland, College Park. She has done research internships at the University of Cincinnati and NASA Goddard in addition to the University of Maryland. She plans to attend graduate school in electrical engineering once she has finished her undergraduate degrees. She is a student member of the IEEE.

**Tiebing Zhang** received the MS degree in electrical engineering from the University of Maryland, College Park in 2001, the MS degree in automation from Tsinghua University, People's Republic of China in 1999, and the BS degree in automation from China Textile University (now East China University), People's Republic of China, in 1996. He is currently working as a senior software engineer at 3e Technologies Inc. His interests include embedded Linux, RTOSs, and security of wireless communications.

**Bruce Jacob** received the AB degree cum laude in mathematics from Harvard University in 1988 and the MS and PhD degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1995 and 1997, respectively. At the University of Michigan, he was part of a design team building high-performance, high-clock-rate microprocessors. He has also worked as a software engineer for two successful startup companies: Boston Technology and Priority Call Management. At Boston Technology, he worked as a distributed systems developer and, at Priority Call Management, he was the initial system architect and chief engineer. He is currently on the faculty of the University of Maryland, College Park, where he is an associate professor of electrical and computer engineering. His present research covers memory-system design, DRAM architectures, virtual memory systems, and microarchitectural support for real-time embedded systems. He is a recipient of a US National Science Foundation CAREER award for his work on DRAM architectures and systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.