Microelectronics Journal 42 (2011) 1176–1185

Contents lists available at ScienceDirect

Microelectronics Journal

journal homepage: www.elsevier.com/locate/mejo

Design of a hard real-time multi-core testbed for energy measurement[☆]

Tongquan Wei [a,*], Xiaodao Chen [b,*], Piyush Mishra [c]

[a] East China Normal University, Shanghai, China
[b] Michigan Technological University, Houghton, MI, United States
[c] GE Global Research, Niskayuna, NY, United States

A R T I C L E I N F O

Article history:
Received 9 May 2010
Received in revised form
28 September 2010
Accepted 7 March 2011
Available online 20 July 2011

Keywords:
Dynamic voltage scaling
Hard real-time systems
Multi-core testbed
Rate monotonic scheduling
Task allocation and scheduling.

A B S T R A C T

This paper presents a systematic methodology for designing a hard real-time multi-core testbed to validate and benchmark various rate monotonic scheduling (RMS)-based task allocation and scheduling schemes in energy consumption. The hard real-time multi-core testbed comprises Intel Core Duo T2500 processor with dynamic voltage scaling (DVS) capability and runs the Linux Fedora 8 operating system supporting soft real-time scheduling. POSIX threads API and Linux FIFO scheduling policy are utilized to facilitate the design and Dhrystone-based tasks are generated to verify the design. A LabView-based DAQ system is designed to measure the energy consumption of CPU and system board of the testbed. A case study of task allocation and scheduling algorithms is also presented that aim to optimize the schedule feasibility and energy consumed by the processor and memory module in the multi-core platform. The experience from the implementation is summarized to serve as potential guidelines for other researchers and practitioners.

1. Introduction

Energy consumption has emerged as an important design constraint in battery-powered portable and embedded hard real-time systems. Since the energy budget of these systems is limited, it is desirable to minimize energy consumption to maximize system lifetime. Moreover, the power density of processors is continuously increasing with the scaling of each technology generation. Energy efficient design at system level is imperative to reduce the average power consumption, and hence the dissipated heat. This is because low heat dissipation is of particular importance to hard real-time applications such as medical implantations, avionic controls, and deep space missions that require systems with small form factors.

Hard real-time systems are typically designed for corner cases where resources are provided for the worst case execution time (WCET). Since, in the average case, tasks do not execute up to their WCET, system level techniques such as dynamic voltage scaling (DVS) could be utilized to achieve energy savings. However, using DVS for energy savings may result in the violation of task timing constraints due to the prolonged task execution time. A tradeoff must be made between energy savings and delay. Energy efficient scheduling schemes for hard real-time systems aim to maximize energy savings by applying DVS while satisfying task timing constraints.

In the past decade, numerous DVS-based energy efficient task allocation and scheduling schemes have been proposed for both uniprocessor and multiprocessor systems [1–4]. Most of these schemes are derived based on idealized operating conditions and energy models. Although theoretically idealized models simplify the analysis and simulations, practical applications must deal with other important constraints added to the models. Hence, designing a real-life test bed could be very valuable to evaluate and validate the modeling assumptions and scheduling solutions under different yet more accurate environmental conditions.

In this paper, a systematic methodology is presented to implement and validate schemes for energy efficient task allocation and scheduling in multi-core hard real-time embedded systems. A real-life test bed comprising the dual-core Intel T2500 processor with DVS capability and running Linux Fedora 8-based hard real-time scheduling has been developed to accurately benchmark energy savings of various task allocation and scheduling schemes. A case study of task allocation and scheduling algorithms is also presented that aim to optimize the schedule feasibility and energy consumed by the processor and memory module in the multi-core platform. Dhrystone benchmark program-based tasks with varying characteristics are generated, allocated, and scheduled on individual cores of the processor using the algorithms implemented. An NI LabView-based data acquisition system is also designed and set up for energy measurement.

[☆] The preliminary version of this manuscript appeared in SAC, 2009. This work was supported in part by the Fundamental Research Funds for the Central Universities of China under the grant No. 78220021.

[*] Corresponding authors.
E-mail addresses: tqwei@cs.ecnu.edu.cn (T. Wei), cxiaodao@mtu.edu (X. Chen).

1.1. Related work

Extensive investigations have been carried out on energy efficient task allocation and scheduling for both uniprocessor and multiprocessor systems [1–4]. With the technology advances towards deep submicro-devices, research on hard real-time scheduling has been extended to jointly optimize schedule feasibility and multidimensional constraints such as energy and fault-tolerance. Melham et al. presented DVS techniques to exploit slacks in task schedules to reduce energy consumption while tolerating faults [5]. They made a simplifying assumption that processor frequency can be scaled in continuous range. Zhang et al. proposed energy efficient techniques based on offline rate monotonic scheduling (RMS) to tolerate faults in hard real-time systems [6]. Wei et al. also proposed low-cost offline RMS-based scheduling algorithms, but their algorithms can be extended to the runtime scenario by combining feasibility analysis and voltage scaling based on the exact characterization of RMS [7]. They also proposed energy efficient task allocation and scheduling schemes with deterministic fault-tolerance capabilities for symmetric multiprocessor systems executing tasks with hard real-time constraints [8]. However, all these energy efficient task allocation and scheduling solutions were validated in simulated environments without considering practical impact of platforms.

Research effort also has been made to explore the methodology of incorporating various scheduling algorithms into operating systems. In [9], Mercer et al. designed a processor capacity reservation mechanism which permits threads to specify their CPU resource requirements and implemented the mechanism in the real-time Mach operating system. The scheme was further improved in [10] to provide dynamic quality of service support. Chu et al. [11] implemented a soft real-time CPU server for continuous media processing in the SUN Solaris 2.5 operating system. In [12], Pedreiras et al. presented a process management library that provides timing services for soft real-time application on top of general operating systems with substantial hardware independence. The library executes completely within user space and provides support for periodic process activations. However, these implementations focus on soft real-time applications where violation of timing constraints does not cause catastrophic results, hence are not well suited for hard real-time systems. Swaminathan et al. proposed an earliest deadline first (EDF)-based energy efficient online scheduling algorithm (LEDF). LEDF was implemented on a laptop equipped with AMD’s Power Now! DVS-capable processor and RTLinux operating system [13]. LEDF is a dynamic priority and non-preemptive scheduling algorithm. Although online EDF algorithm dominates over fixed priority offline scheduling algorithms such as rate monotonic scheduling in feasibility performance, fixed priority and preemptive algorithms are of great practical importance. Most real-time scheduling algorithms, especially in hard real-time systems, use fixed priority assignment due to the relatively lower overhead and higher predictability [14].

This paper describes the practical experience in implementing RMS-based energy efficient algorithms for hard real-time systems on a multi-core embedded platform that support dynamic voltage scaling and Linux operating system of soft real-time scheduling capability. It is assumed that the RMS-based fixed priority scheduling algorithms to be implemented are preemptive and the tasks to be executed are periodic and independent. This experience provides guidelines to implement and validate hard real-time scheduling algorithms on top of an operating system supporting soft real-time scheduling and to benchmark energy consumptions of the algorithms on a multi-core embedded platform. The algorithms presented in [6–8] were implemented and Dhrystone-based real-life tasks were generated to verify the implementation.

1.2. Outline

The rest of the paper is organized as follows. Section 2 describes the construction of the hardware and software platform on which an energy efficient task scheduling algorithm is to be implemented. Section 3 presents the detailed process of implementing the scheduling algorithm on the constructed platform. Section 4 outlines the construction of an energy measurement platform. Section 5 presents the experimental results and Section 6 concludes the paper.

2. Construction of the implementation platform

The implementation of an RMS-based energy efficient scheduling algorithm entails that a combination of a hardware and software platform on which the algorithm is to be built supports dynamic voltage scaling and multiprocessor scheduling. The integration of the scheduling algorithm into the selected platform should be simple enough so that the implementation and energy measurement are accomplishable. In addition, relevant documents should be available for references.

2.1. Selection of hardware platform

RMS-based fixed priority and preemptive scheduling algorithms for hard real-time systems were to be validated on a real-life test bed comprising a DVS-capable multi-core processor. Even though there are several multi-core development kits available in the market, selecting one that satisfies all the implementation requirements proved quite challenging. For example, most of the existing multi-core development kits do not support hard real-time operating systems. Further, processor cores of the platform must support DVS in order to implement the scheduling algorithm. Since one of the main goals of the project was to investigate and benchmark the energy consumed by an embedded system under various scheduling algorithms, it is desirable that the platform provides an efficient mechanism to measure the energy consumption of different system components.

After much deliberation and extensive surveying, mini-ITX Express motherboard, Endura TP945GM from Radisys Corporation, emerged as the preferred choice. It mainly comprises an Intel core duo T2500 processor, an Intel mobile 945GM Express chipset, an Intel ICH7-M I/O controller hub, and a 512M Micron DDR2 SDRAM-based memory module [15]. Intel T2500 is a DVS-capable processor with 2 MB L2 cache and a VID voltage range from 1.1625 to 1.30 V. It supports 4 CPU frequencies of 2.0, 1.67, 1.33, and 1 GHz under the CPU frequency driver of P4-clockmod. It also supports low power features such as enhanced deeper sleep mode and Intel enhanced speedstep technology. The minimum and maximum power dissipation of T2500 is 8.71 and 44 W, respectively, and the thermal design power of T2500 is 31 W. The Endura TP945GM motherboard also includes an 80 GB hard drive and provides plenty of sockets, ports, and connectors for expansion.

The Endura TP945GM motherboard is powered by a 200 W small form factor flex ATX power supply unit (FSP200-50PLA). Fig. 1 shows the connections between the power supply unit (PSU) and mini-ITX mother board. The PSU is connected to the motherboard and hard drive through ATX connectors. One four-pin ATX connector connects the PSU to the voltage regulator module (VRM) dedicated to CPU, the other four-pin connector connects the PSU to the hard drive, and a 20-pin connector connects the PSU to the system board. The 20-pin ATX connector provides voltage supplies to components on the mini-ITX motherboard, which are listed in Table 1.

Since one goal of the implementation is to benchmark the energy consumption of scheduling algorithms on the Endura TP945GM motherboard, especially the energy consumed by the processor and memory module, modules such as system fans, CPU fan, USB controller, audio device, LAN driver, PCI express ports, and hard drive are disabled to improve the precision of the measurement. Fig. 1 shows the enabled components in green and disabled components in blue.

2.2. Selection of software architecture

State-of-the-art real-time operating systems (RTOSs) have been investigated as options of the operating system for the Endura TP945GM motherboard. These RTOSs can be broadly classified into two categories: commercial and open source. Commercial RTOSs such as Windows CE, Nucleus RTOS, LynxOS, VxWorks, and Radisys OS-9 were not chosen mainly because these RTOSs are not open source and hence their schedulers can not be modified to integrate an energy efficient scheduling algorithm. In addition, most of commercial RTOSs do not meet the design requirements of the scheduling algorithms to be implemented.

There are numerous free open source RTOSs [16]. Examples include eCos, EROS, FreeRTOS, Phoenix-RTOS, RTLinux, SHaRK, and TimeSys Linux. A scheduling algorithm can potentially be integrated into any of these open source RTOSs since they can be accessed at no cost. However, most of these RTOSs do not support multi-core scheduling and DVS, and few well-documented references are available, which makes the implementation difficult and time-consuming.

The Linux Fedora operating system, one of the most widely used operating systems, was chosen as the operating system for Endura TP945GM motherboard. Linux was initially developed by Linus

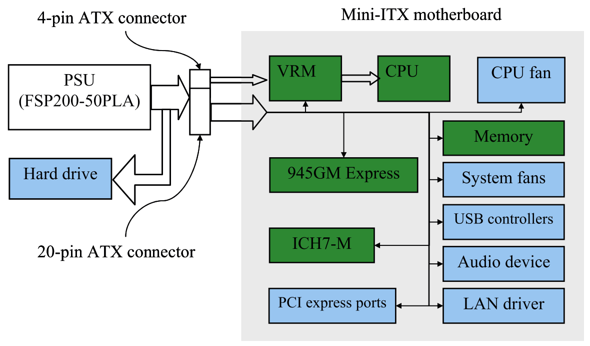

Fig. 1. Connections of power supply unit (PSU) to mini-ITX motherboard. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1
Twenty-pin ATX connector and associated components on mini-ITX motherboard.

Twenty-pin ATX connector	Components on motherboard
+12 V	Fans and expansion slots
+5 V	Voltage regulators for CPU; chipsets and memory; internal logic
+3.3 V	Voltage regulators for chipset interfaces and processor interface; firmware hub (FWH); clock generator; system monitor

Torvalds in 1991 as an operating system for IBM-compatible personal computers based on the Intel 80386 microprocessor [17]. One of the appealing benefits to Linux is that it is not a commercial operating system and is compatible with numerous hardware and software platforms. Its source code under the GNU GPL is open and available to anyone to study, and hence, plentiful materials are available for references [17]. Instead of Linux Fedora Core 6 or 7, Linux Fedora 8 was selected because it supports dynamic voltage scaling.

The scheduling of Linux is based on the time sharing technique where several tasks run in time-multiplexing. Time sharing relies on timer interrupts and is thus transparent to tasks. No additional code needs to be inserted in the programs to ensure CPU time sharing. Linux tasks are priority-based and preemptive. A lower priority task can be preempted by a higher priority task when executing either in kernel or in user mode.

Linux Fedora 8 supports two fixed priority and preemptive real-time scheduling policies: first-in-first-out (FIFO) and round robin (RR) [17]. FIFO assigns CPU to the task with the highest priority. Tasks with the same priorities are arranged to execute in the order of first-in-first-out. The highest priority task continues to use CPU without being preempted until it finishes execution. The round-robin (RR) policy is the application of round robin algorithm in the scheduling domain. Under RR scheduling policy, each real-time task is assigned a fixed time quantum. The execution of a task is either preempted by a higher priority task or suspended when the time quantum is exhausted, whichever occurs first. A suspended task is placed at the end of the same active list of the CPU run queue. The CPU run queue is supposed to have multiple active lists and tasks in an active list is supposed to locate at the same priority level. This policy ensures a fair assignment of CPU time to all RR-based real-time tasks that have the same priority.

Fig. 2 illustrates the operations of FIFO and RR real-time scheduling policies. Ready tasks in instruction memory are picked and put in the run queue of CPU. Tasks in the run queue are organized according to their priority level and the order in which they are put in the run queue if they have the same priority level. Let $\tau_{p,q}$ denote the qth placed task at the priority level p. Two task lists are arranged in the run queue, as is shown in Fig. 2. Tasks having priority level i are placed in one linked list while tasks having priority level j are put in another linked list. Task $\tau_{i,1}$ and task $\tau_{j,1}$ are at heads of the two linked lists and task $\tau_{i,m}$ and task $\tau_{j,n}$ are at the tails of the two linked lists, respectively. Assume tasks in the first list ($\tau_{i,1}$ through $\tau_{i,m}$) are scheduled using FIFO, tasks in the second list ($\tau_{i,1}$ through $\tau_{j,n}$) are scheduled using RR, and the assigned time quantum of tasks in the RR-list is smaller than task execution times. Tasks $\tau_{i,1}$ through $\tau_{i,m}$ in the FIFO-list

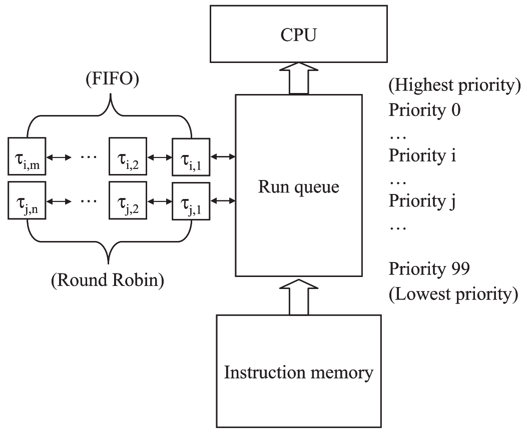

Fig. 2. Operations of FIFO and round robin (RR) scheduling policies.

finish their execution in turn without being preempted by one another. On the contrary, the execution of a task in the RR-list will be terminated and the task will be appended to the tail of the list at the instant when its time quantum expires. For example, task $\tau_{j,1}$ will start its execution after all tasks in the FIFO-list finish their execution since FIFO-list is located at a higher priority level than RR-list. After task $\tau_{j,1}$ in RR-list exhausts its time quantum, it is added to the end of the list. Task $\tau_{j,2}$ becomes the head of the list and starts to execute.

The FIFO policy was selected as the basis for implementing RMS-based scheduling algorithms. RMS, an optimal fixed priority scheduling algorithm in which a task with a shorter period is given a higher priority than a task with a longer period [18,14], is used to assign priorities to tasks with different periods and Linux FIFO policy is used to break ties when tasks have the same periods. The Linux FIFO policy provides a simple yet efficient approach to implementing fixed priority preemptive scheduling algorithms.

3. Implementation of RMS based task allocation and scheduling algorithms

RMS-based task allocation and scheduling algorithms for hard real-time systems were implemented by combining Linux FIFO scheduling policy and POSIX threads API. The implementation consists of three major steps: task generation, task allocation, and task scheduling. In the first step, Dhrystone-based independent and periodic tasks were generated and task characteristics were derived. In the second step, the generated tasks were assigned to processors according to task allocation heuristics such as first fit decreasing (FFD), worst fit decreasing (WFD), and modified worst fit decreasing (MWFD) [8]. In the task scheduling step, the tasks allocated to individual processors are scheduled using uniprocessor scheduling algorithms such as A-DVS [7] and JFTA [6]. Specifically, a priority mapping was performed to synchronize the opposite priority conventions of POSIX threads API and Linux Fedora 8 scheduler, and a voltage schedule for the allocated tasks was derived by calling the uniprocessor scheduling algorithms. The allocated tasks were then synchronized to release their first instances at the critical instant and start to execute on the constructed platform.

3.1. *Task set generation and task characteristics derivation*

A task set with ten tasks was used to facilitate the description of the implementation process. Task utilizations of the tasks in the task set were generated based on the beta-distribution of probability and were limited to less than ln2, the asymptotic bound of RMS. The standard deviation of utilizations was also limited to a maximum value, which is a function of the mean of task utilizations [8]. Ten tasks the execution times of which range from 1 to 1000 ms were generated using the Dhrystone benchmark program, which is described in details in the next paragraph. Task periods were derived using the generated task utilizations and measured task execution times on the Endura TP945GM motherboard.

Dhrystone, a synthetic computing benchmark program developed to be representative of system programming, was selected to generate tasks in the task set. Dhrystone was written in C language code, has small size, and is portable to a large number of platforms and processor architectures. These characteristics make Dhrystone a popular benchmark in embedded applications due to its small memory requirements.

The output of Dhrystone benchmark program is the number of iterations of the Dhrystone main code in unit time, which is derived by dividing a predefined number of iterations of the Dhrystone main code by the corresponding execution time. The value of the predefined number of iterations depends on processor architecture, operating system, and compiler attributes [19,20]. Dhrystone was modified in the implementation to generate tasks in the task set. The modified Dhrystone takes as its input the number of iterations of the Dhrystone main code, which is denoted by *loops*. The corresponding task is therefore represented as *Dhrystone*(*loops*). One *loops* value corresponds to one task in the task set. The execution time of the modified Dhrystone is measured using the GNU C library function *gettimeofday*, hence, measurements have time resolution of 1 ms. This time resolution is fine enough for task execution times in the magnitude of milliseconds. Measurement of the execution time of each task in a task set was performed 10,000 times on the Endura TP945GM motherboard to account for the statistical anomalies and impact of the practical platform on task execution.

Ten *loops* values ranging from 1000 to 950,000 were generated with uniform distribution of probability such that each corresponding task has the same probability of being short (1–10 ms), medium (10–100 ms), and long (100–1000 ms). The *loops* value of 1000 corresponds to the task execution time of 1 ms and the value of 950,000 corresponds to the task execution time of 1000 ms on the motherboard. This approach assumes that each loop of the modified Dhrystone takes fixed execution time.

Fig. 3 shows the function *WCET_period_func* to obtain the worse case execution times and corresponding periods of tasks in the task set. Let *exe_time* denote the execution time of task *taskID*, array component *loops_array*[*taskID*] denote the *loops* counterpart of the task *taskID*, and *TaskNum* denote the number of tasks in the task set. Line 3 of the function derives the execution time of a task by running the task on the practical platform and measuring the time elapsed. Since the execution time of a task varies on the practical hardware and software platform, the execution time obtained in line 3 is adjusted by an experimental factor to derive the worst case execution time (WCET) and corresponding period of the task in a way such that a new task instance will not be released before its predecessor finishes execution. The adjustment factor is a real number greater than 1.0. The 10,000 executions of each generated task on the platform showed that 1.1 is an acceptable adjustment factor since the actual execution times of tasks are close enough to but less than their worst case execution time. This approach does not provides a safe upper bound on the worst case execution time, hence it is only suitable for applications where a safe upper bound is not required. Line 4 adjusts task execution time by the experimental factor 1.1 to obtain the WCET of the task. After the WCET of a task is derived and stored to the global array *WCET* in line 5, the period of the task is derived as the ratio of the corresponding WCET stored in *WCET* to the utilization stored in *util*, as shown in line 6. The hyperperiod of a task set can be obtained by calculating the least common multiple of all task periods.

```
void WCET_period_func( )
1. int $exe\_time, taskID$;
2. for $taskID, 1 \le i \le TaskNum$ do
3.     derive $exe\_time$ of the task $loops\_array[taskID]$;
       {derive WCET of a task}
4.     adjust $exe\_time$ by an experiential factor 1.1;
5.     $WCET[taskID] = exe\_time$;
6.     $period[taskID] = (int)(exe\_time/util[taskID])$;
       {Store WCET and derive period of the task $taskID$}
7. end for
8. return
```

Fig. 3. The function to derive the worst case execution time and period of a task.

Instances of a task are released per constant, periodic interval of time and each instance corresponds to an iteration of the execution of the modified Dhrystone. In the implementation, the release of a task instance was mimicked by calling the modified Dhrystone every interval of the task period. The slack time of a task is the difference between the period and actual execution time of the task. A task is put to sleep during its slack time interval. Waiting tasks can use the processor even if they have lower priorities than the sleeping task. The example code of a task is omitted due to space limitation.

3.2. *Allocation of the generated tasks to processors*

The relationship of a process, thread, and task is first clarified in this section. A process is the execution of one or more tasks, and a thread is a light weight process corresponding to the execution of exactly one task. A process can contain multiple threads. In user space of Linux Fedora 8, processes can be mapped to different CPUs or cores while threads of a process can only execute on the CPU or core where the process is mapped.

The task-to-processor allocation strategies can be broadly classified into two categories: global allocation and partitioning allocation. Under global allocation, a task instance can execute on one processor and migrate to other processors if required. On the other hand, partitioning allocation assigns tasks to processors permanently, and migration among processors is prohibited. The global allocation strategy suffers from its large overhead due to task migration among processors and hence has an adverse impact on the feasibility performance of optimal uniprocessor scheduling algorithms such as RMS and EDF. Therefore, RMS-based hard real-time systems usually use partitioning allocation schemes such as FFD, WFD, and MWFD [8] to assign tasks to processors.

The default scheduler of the Linux Fedora 8, however, uses global partitioning. It attempts to keep tasks on the same processor and balances the workload by migrating tasks among processors if the workload of one processor is significantly lower than the workload of another. The *taskset* utility of Linux Fedora 8 was used in the implementation to override the default scheduler of Linux Fedora 8 and set the CPU affinity for processes of tasks. The CPU affinity is a scheduler property that bonds a process to a given processor on the symmetric multiprocessor system. As a result, the Linux scheduler will follow the CPU affinity of the *taskset* utility and the process of a task will not migrate among processors.

The partitioning allocation-based task-to-processor assignment for an n-processor system is achieved by creating n processes in the user space, performing a one-to-one mapping between the n processes and the n CPUs or cores, and creating in each process the threads of tasks that will be assigned to the associated CPU or core. First, processes are created using *fork*, a Linux command that creates a child process differing from the parent process only in process ID. A parent process and its child are inherently synchronous in the sense that the threads of tasks in the parent process start the execution at the same instant as the threads of tasks in the child process. In other words, threads from both the parent and child process start to execute at the instant after the fork command is called. Second, a running process is bound to a specific CPU or core by calling the *taskset*, a utility that utilizes a bitmask to specify the CPU or core where the process is mapped. For example, the bitmask 0×0001 corresponds to processor 0 and 0×0003 represents processor 0 and 1. Finally, the threads of tasks in a process are created using the pthread function *pthreadcreate*, which will be discussed in details in Section 3.3.

```
void Task_Allocation ( )
 1. pid_t pid;
 2. pid = fork ( );
 3. if pid == -1 then
 4.    print" Can't fork, error happens"
 5.    return
 6. end if
 7. if pid == 0 then
 8.    Using Linux taskset utility to set process affnity to core 1;
 9.    Create thread 1 using p thread function p thread_create;
10.    Create thread 2 using p thread function p thread_create;
11.    Synchronize the created threads;
12. end if
13. if pid > 0 then
14.    Using Linux taskset utility to set process affinity to core 2;
15.    Create thread 3 using p thread function p thread_create;
16.    Create thread 4 using p thread function p thread_create;
17.    Synchronize the created threads;
18. end if
19. return
```

Fig. 4. The function to assign tasks to processors.

The *Task Allocation* function shown in Fig. 4 illustrates the allocation of four tasks to a core duo processor. Task 1 and task 2 are supposed to execute on core 1 while task 3 and 4 are specified to run on core 2. The line 1 of the function defines the variable *pid* of the type pid_t. In line 2, the *fork* is called and its return value is assigned to the variable *pid*. If *pid* equals to -1, the call to the fork fails and no child process is created (lines 3–6). If *pid* equals to 0, the call to the fork is successful and the current process is the child (lines 7–12). If the call to the fork returns a positive number, the current process is the parent and the returned positive number is the process ID of the newly created child (lines 13–18).

The child process is bound to core 1 of the core duo processor in line 8. The two threads of task 1 and task 2 are created in lines 9 and 10 using pthread function *pthreadcreat*, and are synchronized in line 11 so that all tasks allocated to core 1 start their executions simultaneously. The similar activity of the parent process is demonstrated in lines 13–17. The synchronization of threads of a process is discussed in Section 3.3.

3.3. *Scheduling of the allocated tasks on processors*

The RMS-based scheduling algorithm performs priority assignment according to periods of tasks in the task set. A task with shorter period has higher priority than a task with longer period. This priority assignment scheme was implemented on the Linux Fedora 8 platform by combining the FIFO policy of Linux and the priority convention of POSIX threads API.

The Linux FIFO real-time scheduler favors a higher priority task over a lower priority one for tasks in a task set with given priorities. Ties are broken according to the first-in-first-out policy. Real-time priorities supported by Linux Fedora 8 scheduler rang from 1 to 99, where 1 represents the highest real-time priority and 99 denotes the lowest real-time priority, as is shown in Fig. 2. Conventional tasks have lower priorities than any real-time tasks. This scheduling property implicitly minimizes interference overheads from the execution of conventional tasks on the platform since no real-time tasks can be preempted by a conventional task. In addition, Linux Fedora 8 utilizes an $O(1)$ scheduler. The time the scheduler takes to find a task to execute does not depend on the varying number of tasks but on the fixed number of priorities supported by the scheduler. In other words, the scheduling overhead of Linux Fedora 8 scheduler is fixed regardless of the number of ready tasks. Therefore, the impact of the scheduler on the RMS-based scheduling algorithm is determined and can be accounted for by the variations in task execution times.

The POSIX threads API is a POSIX standard for threads, which defines an API for creating and manipulating threads [21]. The POSIX threads API supports two real-time scheduling policies (FIFO and RR) and 99 priorities (1–99). Unlike the Linux Fedora 8 scheduler, in the POSIX threads API, a task with a large priority value has higher priority than a task with a small priority value. For example, priority value 99 stands for the highest priority and priority value 1 refers to the lowest priority. Therefore, a priority mapping was performed in the implementation in such a way that a task with shorter period is assigned a higher priority value and a task with longer period is assigned a lower priority value.

POSIX threads API was used in the implementation to manage periodic tasks in the task set. One of its roles is to synchronize periodic tasks in the task set. In the practical implementation, once a real-time task is created and if there are no higher priority tasks in the run queue of CPU, the task will be executed immediately. This may lead to a situation where a task created first is executed first regardless of its relatively lower priority than a task that is created later. Since the CPU runs fast and there exists a time delay between creation of two tasks, the first task may finish its execution before the second task is created. Therefore, periodic tasks in the task set were synchronized to release their first instances simultaneously in the implementation. The proposed synchronization strategy consists of three steps: (1) POSIX threads API routines are called to create a condition initialized to false, on which all tasks in a task set have to wait, (2) POSIX threads API routines are called to instantiate tasks and assign priorities to tasks, and (3) the condition is set to true and all tasks in the task set become ready for execution. Synchronized tasks in a task set compete for CPU and the task having the highest priority executes first.

The RMS-based scheduling algorithm is intended for hard real-time systems while the Linux Fedora 8 only supports a soft real-time scheduler, in which tasks are allowed to miss their deadlines without necessarily causing system failure. In this implementation the hard real-time scheduling property was achieved on top of the Linux Fedora 8 soft real-time scheduler by comparing task deadlines with current time and exiting task executions if task deadlines are missed. The time instant denoting the critical instant of tasks in a task set is stamped before the synchronization condition is set true. The time instant is utilized as base time to derive absolute deadlines of tasks. A comparison is performed at the end of the execution of a task instance to determine if current time exceeds the absolute deadline of the task instance. A task is considered unschedulable if one of its instances can not meet its absolute deadline, and then the task set containing the task is deemed infeasible.

4. Construction of energy measurement platform

Since the ATX 4-pin connector exclusively provides 12 V voltage to the VRM of CPU and the ATX 20-pin connector provides 12, 5, and 3.3 V voltages to components on system board, the energy consumption of CPU and system board can be approximated by the energy delivered from ATX power supply connectors, which can be derived by measuring the current flowing through ATX connectors. Considering the fact that a VRM can achieve energy efficiency of up to 95% [22] and difficulties to directly measure energy consumption of an onboard device, this strategy for energy estimation can be used to sketch energy consumption of CPU and system board.

Measurement of energy consumption is accomplished by using a DAQ system and Tektronix A622 AC/DC current probe. The DAQ system is composed of an NI PCI-6040E DAQ, NI BNC-2110 connector block, and a host computer with LabView, as is shown in Fig. 5.

The Tektronix A622 AC/DC current probe uses a Hall-effect current sensor to measure the strength of the magnetic field of current flow and then transform it to a voltage output. It can measure AC/DC currents from 50 mA to 70 A over a frequency range of DC to 100 kHz. It provides 10 or 100 mV output for each ampere measured. The 100 mV scale is used in the experiment to improve measurement accuracy. As is shown in Fig. 5, the probe acquires the currents flowing through ATX power supply wires and feeds its voltage outputs to the NI PCI-6040E DAQ device via a NI BNC 2110 connector block.

The NI PCI-6040E multifunction DAQ is a fast and accurate multiplexing data acquisition device ideal for continuous high-speed data logging. It includes connectivity for 16 analog inputs, two analog outputs, eight digital input/output lines, and two counter/timer signals. Four of the analog inputs were used in the experiments to accept voltage outputs from current probes attached to PSU power wires. The combination of the NI PCI-6040E DAQ and the LabView real-time module constitutes the basic framework of the measurement platform. LabView program is designed in this experiment to access the NI PCI-6040E DAQ device to acquire voltage data, which are then transformed to the corresponding currents flowing through ATX connector wires. The input power to CPU and the system board are derived using the ATX connector voltage values and the measured current values.

Fig. 6 shows the graphical LabView program to derive the energy consumption of CPU. The analog voltage acquired by the DAQ assistant from the current probe, which is attached to the four-pin ATX connector power wires dedicated to CPU, is fed to the A/D converter and converted to a digital voltage value. Since current probes are set at 100 mV output scale for each ampere measured in the experiment and the supply voltage of CPU from the four-pin ATX connector is 12 V, the power delivered to CPU is expressed as $12\,\text{V} \times 10 \times \mathit{OutputA/D}$, where $\mathit{OutputA/D}$ denotes the output of A/D converter. The energy delivered to CPU is the accumulated power in the duration of task executions.

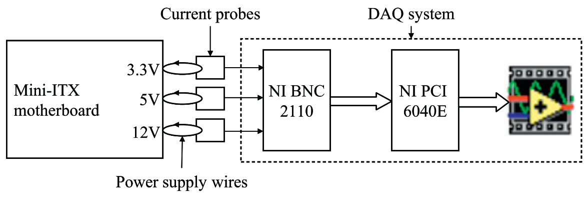

Fig. 5. DAQ system and current probes for energy measurement.

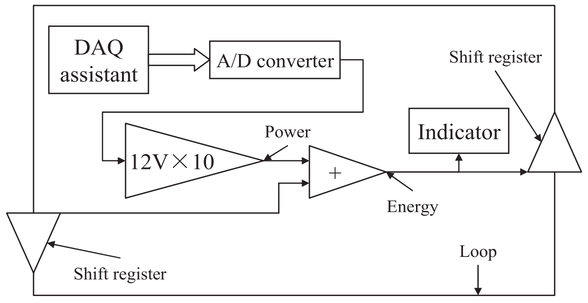

Fig. 6. The LabView graphical program to derive energy consumption of CPU.

The real-time energy value is displayed by an indicator, passed to the next iteration through a local variable called shift register, and retrieved at another shift register on the main loop for accumulation. The iteration of the loop is controlled by task execution time. The energy consumption of the system board can be measured using the similar approach. Components on the system board such as the system fans, CPU fan, USB controllers, audio device, LAN driver, PCI express ports, and hard drive are disabled during system board energy measurement since they are not related to system scheduler.

5. Experimental results

Extensive experiments were performed on the multi-core embedded board to validate the feasibility and efficiency of energy aware task allocation and scheduling schemes for hard real-time systems. The multi-core embedded board is configurable such that one of the two cores of the Intel T2500 can be disabled and the board could be used for both uniprocessor systems and multi-core systems. An energy measurement platform was set up in the experiment to measure energy consumptions of the generated task set on the cores and system board, respectively. The energy measurement was performed under different allocation and scheduling schemes.

5.1. Results for uniprocessor systems

One core of the Core Duo processor Intel T2500 was disabled for uniprocessor implementation. Two offline energy efficient task scheduling algorithms for uniprocessor systems, JFTA [6] and A-DVS [7], were implemented on the test bed. Both JFTA and A-DVS are DVS-based application level task scheduling algorithms where all tasks run at the same processor speed. JFTA exhaustively examines all processor frequency levels to determine the lowest frequency that satisfies timing constraints. Time complexity of the algorithm is $O(n^2RL)$, where n is the number of periodic tasks in a task set, R is the ratio of the largest task period to the smallest task period, and L is the number of frequency levels supported by the processor. Unlike the JFTA, the A-DVS algorithm starts feasibility analysis from the lowest processor speed and the first schedule found feasible is proved to be energy optimum. It exploits a binary search-based approach to find the lowest possible processor speed at which the task schedule is feasible. The time complexity of the A-DVS algorithm is $O(n^2R\log_2^L)$.

The A-DVS algorithm utilises the time-demand-based approach in feasibility analysis, which enables the adaptation of the A-DVS algorithm to the runtime behavior of task execution. The dynamic component of the A-DVS algorithm, referred to as D-ADVS, is also implemented in this section. Both the JFTA and the A-DVS are fault-tolerance scheduling algorithms. The effect of fault is emulated in the implementation by incorporating fault recovery overheads into task execution times to facilitate energy measurement. A more detailed description of the JFTA and the A-DVS algorithms can be found in [6,7], respectively.

The 10 generated Dhrystone-based tasks were scheduled to execute on the test bed using the JFTA and A-DVS, respectively. Table 2 shows the energy consumptions of the core and system board, respectively. K denotes the maximum number of faults the system is designed to tolerate and $E_{JA} = (E_J - E_A)/E_J \times 100\%$, where E_J and E_A represent energy consumptions of the task set under JFTA and A-DVS, respectively. *NF* denotes that the tasks in the task set can not be feasibly scheduled.

Table 2 shows that as compared to A-DVS, JFTA consumes about 20% more core energy in the presence of fault occurrences and consumes the same core energy in the absence of fault occurrences. This is because JFTA is an offline scheduling algorithm that considers the worst case fault occurrences in the design phase while A-DVS is an efficient offline scheme that can utilize uncertainties in fault occurrences at runtime to enhance energy savings.

Table 2
Energy consumptions of CPU and system board under JFTA and A-DVS scheduling schemes.

	K	JFTA E_J (J)	A-DVS E_A (J)	$(E_J-E_A)/E_J$ E_{JA} (%)
CPU	0	312.1	312.1	0
	1	396.7	318.5	19.7
	2	408.4	318.5	22.0
	3	418.1	325.6	22.1
	4	436.2	330.7	24.2
	5	*NF*	341.2	–
System board	0	489.1	489.1	0
	1	488.5	490.4	−0.38
	2	488.7	490.8	−0.45
	3	489.1	491.4	−0.46
	4	491.6	490.4	0.25
	5	NF	491.0	–

Table 3
Overhead in energy and execution time of JFTA and A-DVS when scheduling Dhrystone-based task set at different CPU speed.

CPU speed (GHz)	JFTA		A-DVS	
	Energy (μJ)	Execution time (μs)	Energy (μJ)	Execution time (μs)
1.00	20.7	3497	6.31	1663
1.33	23.0	2421	7.02	1132
1.67	25.4	1366	7.77	864
2.00	29.7	1013	9.02	699

The energy consumptions of the system board excluding the processor are close for the two scheduling algorithms under different numbers of fault occurrences. For example, the difference in the energy consumption of the system board between the JFTA and A-DVS is less than 0.5% with the number of faults ranging from 0 to 5, as is shown in Table 2. Furthermore, the energy consumption of the system board when it is idle is 479.8 J, which is about 10 J less as compared to the energy consumption of the system board under the load of the Dhrystone-based task set.

There are three possible reasons for the relative stableness in the energy consumption of the system board. First, the Dhrystone is a CPU-intensive benchmark program and it does not intensively exercise the system board, especially the memory system to store and load data. Second, the JFTA and A-DVS scheduling algorithms are also CPU-intensive and their impact on the system board energy consumption is small. Finally, the total size of the instructions of the schedulers and the Dhrystone-based tasks in the form of an executable file is about 20 K. This file could be readily fit in the 2 MB L2 cache of the Intel T2500 processor, which reduces the memory access overheads to fetch instructions.

Table 3 shows the overhead in energy consumption and execution time of JFTA and A-DVS to schedule at different CPU speed the 10 Dhrystone-based tasks. The energy consumption of JFTA and A-DVS is in the range of 20–30 mJ and 6–10 mJ, respectively, and the execution time of JFTA and A-DVS is in the range of 1000−3500 μs and 600−1700 μs, respectively. The execution time of A-DVS is about 1/2 of that of JFTA, which is consistent with the time complexity of the two algorithms.

The energy consumption of A-DVS is about 1/3 of that of JFTA. This is primarily due to the lower complexity of A-DVS when compared to JFTA.

To further compare the scheduling overhead of the two algorithms, several task sets with different numbers of tasks are generated and scheduled using JFTA and A-DVS. Table 4 shows the scheduling overhead of the two algorithms at the processor speed of 1 GHz. Overall, JFTA incurs larger overhead when compared to A-DVS. The scheduling overheads of the two algorithms grow with the increase in the number of tasks to be scheduled and come close to each other with the total task utilizations approaching the processor utilization bound. For example, the overhead of JFTA to schedule 10 tasks is about two times the overhead of A-DVS to schedule the 10 tasks while both the algorithms take about 8000 μs to schedule 70 tasks. This is because the time complexity of the two algorithms is proportional to n^2, where n is the number of tasks to be scheduled, and JFTA incurs the worst case scheduling overhead due to its exhaustive search property.

D-ADVS and the D-TDVS [7] are the dynamic components of the A-DVS and T-DVS algorithm, respectively. The scheduling overheads of the D-ADVS and the D-TDVS are negligible when compared to task execution times. For example, it is shown in Table 4 that the runtime overhead of the D-ADVS is 2 μs for 10 tasks and 26 μs for 70 tasks. The runtime overhead of the D-TDVS is about 7 μs, which is independent of the number of tasks and remains the same for task set with varying sizes.

Since Intel does not provide power characteristics of Core Duo processors such as T2500 at discrete frequency or voltage levels, the simulation for energy consumption can not be performed for Dhrystone-based task set; thus, the comparison between actual energy measurements and simulation results for Dhrystone-based task set was not performed. However, the trend of the experimental results for Dhrystone-based task set conforms to the trend of the simulation results for computer numerical control (CNC) and inertial navigation system (INS) task sets [7] in feasibility performance and energy consumption. In other words, the testbed can be utilized to accurately benchmark various task scheduling algorithms on weather tasks in a given task set can finish their execution on time and energy consumed by schedulable tasks.

5.2. Results for multi-core systems

Task allocation heuristics FFD, WFD, and MWFD [8], and the optimistic fault-tolerance task allocation and scheduling scheme OFT-MWFD [8], were implemented on the multi-core embedded board and validated in terms of the feasibility performance and energy consumption. Both FFD and WFD assume that only one processor is open for task assignment at the beginning. For a given task, the FFD heuristic selects the first processor that can feasibly schedule the task while the WFD heuristic selects the processor with the maximum remaining capacity. MWFD assumes all processors are open at the beginning. Instead of the processor with the maximum remaining capacity, MWFD selects the processor with the minimum workload for task assignment. Allocated tasks are scheduled on individual processors using the exact characterization of the rate monotonic scheduling algorithm, referred to as ECRMA [14].

Ten task sets were generated to validate the feasibility performance and energy consumption of the task allocation and scheduling schemes under different processor workloads. Each of the task sets contains 20 Dhrystone-based independent and periodic tasks with varying characteristics. These task sets were run on the presented real-life testbed, and Fig. 7 shows the percentage of the feasible task sets when the FFD and MWFD allocation schemes are used in conjunction with the ECRMA-based scheduling scheme, where σ/σ_{max} denotes the ratio of the standard deviation of the utilizations to a predefined maximum value and U_{tot}/n denotes the average utilizations on each core. Similarly, Fig. 8 shows the percentage of the feasible task sets under a simulated environment, which was described in [8]. With regard to the feasibility performance of the FFD and MWFD allocation schemes, the trend on the real-life testbed given in Fig. 7 was observed to be consistent with the trend under the simulated environment given in Fig. 8. That is, the feasibility performance of FFD and MWFD is comparable under the varying U_{tot}/n, FFD consistently outperforms MWFD when σ/σ_{max} is large (e.g., $\sigma/\sigma_{max}=0.8$), MWFD consistently outperforms FFD when σ/σ_{max} is small (e.g., $\sigma/\sigma_{max}=0.2$), and MWFD is more resilient to the variations in σ/σ_{max} (i.e., its performance does not significantly change with the variations in σ/σ_{max}). However, the scheduling capacity of the cores decreases by about 0.1 when compared to the simulation results given in Fig. 8. For example, both FFD and MWFD can feasibly schedule all the task sets with $\sigma/\sigma_{max}=0.2$ when $U_{tot}/n<0.75$ and all the task sets with $\sigma/\sigma_{max}=0.8$ when $U_{tot}/n<0.79$, while the corresponding U_{tot}/n reported in the simulation results is about 0.85 and 0.89, respectively. This is because the overheads incurred in the implementation degrade the feasibility performance of the FFD and MWFD.

The implementation overheads include the cost incurred when task instances are created and the context switching overhead incurred when the execution of a task instance is preempted. The

Table 4
Scheduling overhead for varying task set sizes.

Number of tasks	Time overhead (μs)			
	JFTA	A-DVS	D-ADVS	D-TDVS
10	607	319	4	7
20	1141	634	6	7
30	2080	1300	8	7
40	3473	2481	11	7
50	4937	4114	15	7
60	6031	5483	22	7
70	8061	7971	26	7
80	NF	NF	NF	NF

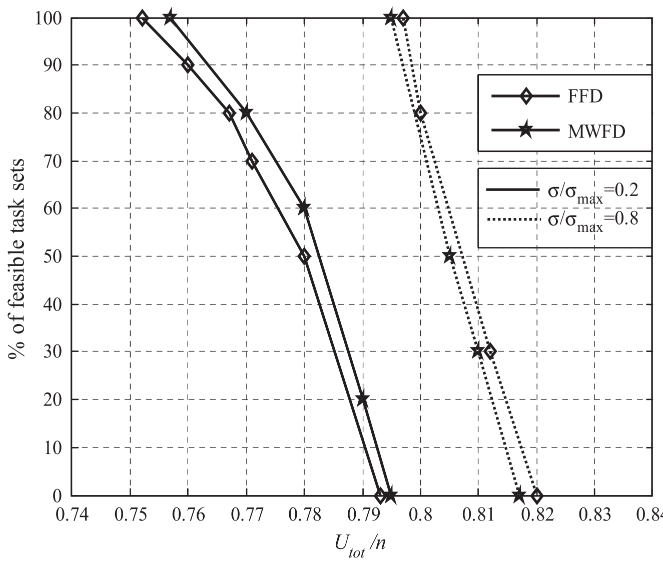

Fig. 7. On the presented real-life testbed with a Core Duo processor, the feasibility performance of FFD and MWFD allocation schemes when used in conjunction with the ECRMA-based scheduling scheme.

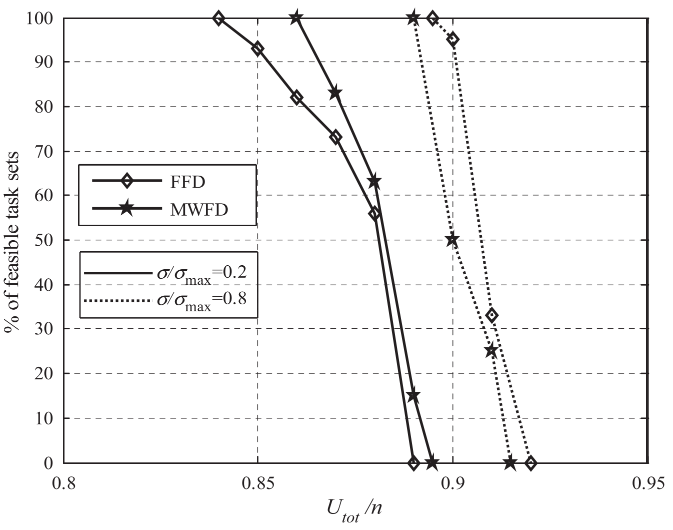

Fig. 8. Under the simulated environment of a two-core processor and 20 tasks [8], the feasibility performance of FFD and MWFD allocation schemes when used in conjunction with the ECRMA-based scheduling scheme.

Table 5
On the real-life testbed of a Core Duo processor, the energy consumption of FFD, WFD, and MWFD allocation schemes when used in conjunction with ECRMA-based A-DVS scheduling algorithm (energy unit in J).

	$\underline{U_{tot}}$ (n)	FFD (E_F)	WFD (E_W)	MWFD (E_M)	E_{FW} (%)	E_{FM} (%)	E_{WM} (%)
Cores	0.31	173.8	173.8	60.1	0	65.4	65.4
	0.50	276.3	227.9	170.3	17.5	38.4	25.3
	0.63	326.3	280.6	248.5	13.8	23.8	11.4
Board	0.31	452.9	452.9	215.0	0	52.5	52.5
	0.50	488.7	447.3	317.5	8.5	35.0	29.0
	0.63	492.5	453.9	407.8	7.8	17.2	10.2

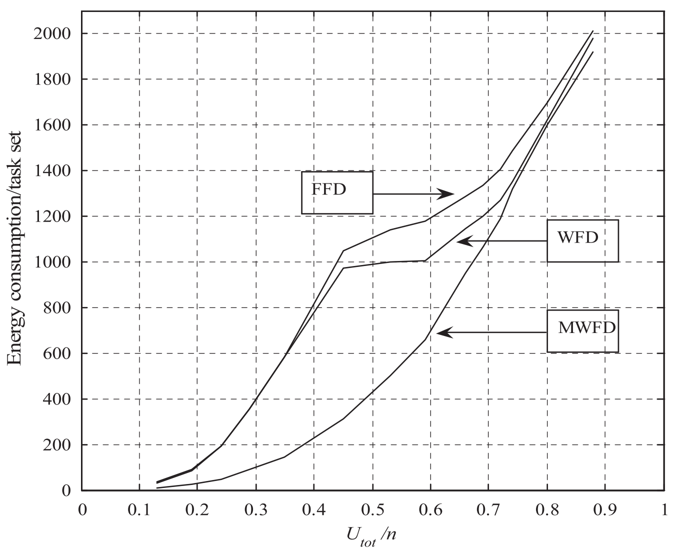

Fig. 9. Under the simulated environment of a two-core processor and 20 tasks [8], the energy consumption of the FFD, WFD, and MWFD allocation schemes when used in conjunction with the ECRMA-based scheduling scheme.

instances of a task are released periodically. Assuming a released instance of the task finishes its execution without being preempted, the processor is put to sleep when the task instance finishes its execution so that other tasks can utilize the processor during the slack of the task instance. This is achieved by calling the *usleep* function. The invocation overhead of the *usleep* function is in the order of microseconds. In addition, task preemption may occur during the execution of the current task. Preemption may also occur during the slack of the current task where conventional background tasks or other real-time tasks execute. Consequently, context switching overhead is incurred as a result of the task preemption and increases with the growth in the number of the preemption.

Although the generated real-time tasks are assigned to cores using the task allocation heuristic presented in [8], the conventional non-real-time background processes are allocated to cores by the default Linux scheduler. The default Linux scheduler performs load-balancing every clock tick and ensures that the run queues of different cores contain approximately the same number of conventional processes. This allocation strategy may lead to a scenario where some cores contain significantly longer conventional processes than other cores even though the number of conventional processes on each core is the same. As a result, the cores containing the longer conventional processes are relatively busier than the cores containing the shorter conventional processes. Since a long process is more likely to be preempted by a real-time task, the cores containing the longer conventional processes possibly incur more preemption overheads. This is consistent with the observation in the experiment that the preemption overheads are different on the two cores although real-time tasks are balanced between the two cores.

Both the context switching overhead and the *usleep* invocation overhead are deterministic for a real-time task. Since a task preempts at most one task [23], the incurred preemption cost of the task is upper-bounded by the overhead from one context switching. In addition, a task instance incurs overhead from exactly one *usleep* invocation. These overheads are not taken into account in the feasibility analysis of both simulation and implementation while are essentially incorporated into task execution time in actual implementation. This discrepancy leads to a plausible degradation in the core scheduling capacity.

Table 5 shows the energy consumption of a task set under FFD, WFD, and MWFD allocation schemes when used in conjunction with ECRMA-based A-DVS scheduling algorithm. The energy consumption was measured under the workload $U_{tot}/n=0.31$, 0.5, and 0.63, respectively, and averaged over 10 feasible task sets with $\sigma/\sigma_{max}=0.8$. E_F, E_W, and W_M denotes the energy consumption of FFD, WFD, and MWFD, respectively. Similarly, E_{FW} denotes $(E_F-E_W)/E_F\times 100\%$, E_{FM} denotes $(E_F-E_M)/E_F\times 100\%$, and E_{WM} denotes $(E_W-E_M)/E_W\times 100\%$. Table 5 shows the energy consumption of the cores and the system board. When the average utilization on cores is low, for example, $U_{tot}/n=0.31$, FFD and WFD consume approximately the same core energy while the MWFD consumes about 65% less core energy. When the average workload is medium, the energy consumptions of FFD, WFD, and MWFD increase and the difference in their energy consumption becomes large. For example, for $U_{tot}/n=0.5$, FFD consumes about 17% more core energy than the WFD, and both FFD and WFD consume 25% more core energy than the MWFD. As the average workload increases, the energy consumptions of FFD, WFD, and MWFD continue increasing but the differences in energy consumptions reduce. For instance, for $U_{tot}/n=0.63$, the core energy consumption of FFD is about 13% more than that of the WFD and 23% more than that of the MWFD, and the core energy consumption of WFD is about 11% more than that of the MWFD. The energy consumption of a task set under FFD, WFD, and MWFD schemes in the simulated environment is given in Fig. 9. It can be seen in the figure that the trend shown in the simulation results conforms well to the trend given in the actual measurements.

Table 5 also shows that the energy consumption of the system board is roughly proportional to the length of the schedule. For example, when $U_{tot}/n=0.31$, both FFD and WFD could feasibly allocate and schedule the task set on one core, hence generate a schedule with the same length. On the contrary, for $U_{tot}/n= 0.31$, MWFD balances the workload between two cores and the resultant schedule is about 50% long when compared to that of FFD and WFD. Consequently, the system board energy consumption of FFD and WFD is about the same (452.9 J) and is about two times the system board energy consumption of MWFD (215.0 J). Overall, the system board energy consumption of FFD is larger than that of WFD, which is in turn larger than that of MWFD. This corresponds to the load-balancing property of the three schemes that dictates the length of the resultant schedules.

In Table 5, the average utilizations on cores, U_{tot}/n, do not incorporate the context switching overhead and the *usleep* invocation overhead. The relationship of FFD, WFD, and MWFD in the core energy consumption conforms better to the simulation results given in Fig. 9 when these overheads are considered. In addition, the corresponding energy costs from task preemption and *usleep* invocation are in fact included in the energy measurements. The time overhead of *usleep* invocation can be estimated by the overhead of *usleep*(0), which is in the order of microseconds on the test bed. Hence, the corresponding energy cost of the *usleep* invocation does not pose significant impact to the core or system board energy that is in the order of Joules. Similarly, the energy cost from task preemption was shown to be in the order of micro-Joules [24–26] and hence could be negligible when the core and the system board energy are investigated. In other words, the energy costs from task preemption and *usleep* invocation do not affect the relationship of FFD, WFD, and MWFD in the core energy consumption, which was confirmed in the implementation results shown in Table 5.

6. Conclusions

This paper presents practical experience in designing a hard real-time multi-core testbed to benchmark various energy efficient task allocation and scheduling schemes. The testbed was built on the Endura TP945GM Mini-ITX motherboard that comprises Intel T2500 processor with DVS capability and on top of Linux Fedora 8 that only supports soft real-time scheduling. The independent and periodic tasks used to verify the design were generated based on Dhrystone benchmark program. POSIX threads API and Linux FIFO scheduling policy were utilized to facilitate the design and a LabView-based DAQ system was set up to measure the energy consumption of the CPU and system board. A case study of task allocation and scheduling schemes was also presented.

The key to the success of this implementation is to properly select hardware and software platforms. Common platforms, such as $\times$86 hardware platform and open source Linux Fedora 8 software platform, facilitate the implementation since free support and well-documented information are available. This implementation experience provides guidelines to validate RMS-based energy efficient scheduling algorithms and to benchmark energy consumptions of the algorithms on a real-life multi-core embedded systems.

References

[1] Y. Shin, K. Choi, Power conscious fixed priority scheduling for hard real-time systems, in: Proceedings of the DAC, 1999.

[2] C. Krishna, Y. Lee, Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems, IEEE Transactions on Computers 52 (12) (2003) 1586–1593.

[3] J. Chen, H. Hsu, K. Chuang, C. Yang, A. Pang, T. Kuo, Multiprocessor energy-efficient scheduling with task migration consideration, in: IEEE Euromicro Conference on Real-Time Systems, 2004, pp. 101–108.

[4] T. AlEnawy, K. Aydin, Energy-aware task allocation for rate monotonic scheduling, in: IEEE Real-time and Embedded Technology and Applications Symposium, 2005, pp. 213–223.

[5] R. Melhem, D. Mosse, E. Elnozahy, The interplay of power management and fault recovery in real-time systems, IEEE Transactions on Computers (2004).

[6] Y. Zhang, K. Chakrabarty, A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25 (1) (2006) 111–125.

[7] T. Wei, P. Mishra, K. Wu, H. Liang, Online task-scheduling for fault-tolerant low-energy real-time systems, in: Proceedings of the International Conference on Computer-Aided Design (ICCAD), 2006.

[8] T. Wei, P. Mishra, K. Wu, H. Liang, Fixed-priority allocation and scheduling for energy-efficient fault-tolerance in hard real-time multiprocessor systems, IEEE Transactions on Parallel and Distributed Systems 19 (11) (2008) 1511–1526.

[9] C. Mercer, S. Savage, H. Tokuda, Processor capacity reserves: operating system support for multimedia applications, in: IEEE International Conference on Multimedia Computing and Systems, 1994.

[10] C. Lee, R. Rajkumar, C. Mercer, Experiences with processor reservation and dynamic QoS in real-time Mach, in: Proceedings of Multimedia, 1996.

[11] H. Chu, K. Nahrstedt, A soft real-time scheduling server in unix operating system, in: Proceedings of European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, 1997.

[12] P. Pedreiras, L. Almeida, Task management for soft real-time applications based on general purpose operating systems, in: The 9th Workshop on Real-Time Systems, 2007.

[13] V. Swaminathan, C. Schweizer, K. Chakrabarty, A. Patel, Experiences in implementing an energy-driven task scheduler in RT-Linux, in: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2002.

[14] J. Lehoczky, L. Sha, Y. Ding, The rate monotonic scheduling algorithm: exact characterization and average case behavior, in: IEEE Real-Time Systems Symposium, 1989.

[15] Radisys Corporation, Endura TP945GM motherboard, 2002, available at ⟨http://www.radisys.com⟩.

[16] RTOS List, List of real-time operating systems, available at ⟨http://www.wikipedia.org⟩.

[17] D. Bovet, M. Cesati, Understanding the Linux Kernel, third ed., O'Reilly, 2005.

[18] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in a hard real time environment, Journal of the ACM (1973).

[19] Dhrystone source code, available at ⟨http://www.cs.helsinki.fi⟩.

[20] R. Weicker, Dhrystone: a synthetic systems programming benchmark, Communications of the ACM 27 (10) (1984) 1013–1030.

[21] B. Lewis, D. Berg, Threads Primer—A Guide to Multithreaded Programming, Prentice-Hall, 1996.

[22] Wikipedia, available at ⟨http://www.wikipedia.org⟩.

[23] J. Liu, Real Time Systems, Prentice-Hall, 2000.

[24] K. Nowka, G. Carpenter, E. Donald, H. Ngo, B. Brock, K. Ishii, K. Nguyen, J. Burns, A 0.9 V to 1.95 V dynamic voltage-scalable and frequency-scalable 32b PowerPC processor, IEEE International Conference on Solid State Circuits (2002) 340–341.

[25] D. Grunwald, P. Levis, C. Morrey, M. Neufeld, K. Farkas, Policies for dynamic clock scheduling, in: The Symposium on Operating Systems Design and Implementation, 2000, pp. 73–86.

[26] T. Burd, R. Broderson, Design issues for dynamic voltage scaling, International Symposium on Low Power Electronics and Design (2000) 9–14.