AFRL-IF-WP-TR-2006-1509

POWERTAP: SYSTEM-WIDE POWER MANAGEMENT THROUGH POWER-AWARE SYSTEM SOFTWARE AND HARDWARE

Babak Falsafi and Raj Rajkumar

Carnegie Mellon University
Computer Architecture Lab (CALCM)
Real-Time Multimedia Systems Lab (RTML)
5000 Forbes Avenue
Pittsburgh, PA 15213

AUGUST 2005

Final Report for 01 June 2002 – 31 December 2004

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

This report was cleared for public release by the Air Force Research Laboratory Wright Site (AFRL/WS) Public Affairs Office (PAO) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

PAO Case Number: AFRL/WS-05-2350, 11 October 2005

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

/s/

RONALD W. BROWER, Ph.D.
Project Engineer
Embedded Information Systems Branch
Advanced Computing Division

/s/

AL SCARPELLI
Team Leader
Embedded Information Systems Branch
Advanced Computer Division

/s/

JAMES S. WILLIAMSON, Chief
Embedded Information Systems Branch
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YY)***
August 2005

2. REPORT TYPE
Final

3. DATES COVERED ***(From - To)***
06/01/2002 – 12/31/2004

4. TITLE AND SUBTITLE
POWERTAP: SYSTEM-WIDE POWER MANAGEMENT THROUGH POWER-AWARE SYSTEM SOFTWARE AND HARDWARE

5a. CONTRACT NUMBER

5b. GRANT NUMBER
F33615-02-1-4004

5c. PROGRAM ELEMENT NUMBER
62301E

6. AUTHOR(S)
Babak Falsafi and Raj Rajkumar

5d. PROJECT NUMBER
M765

5e. TASK NUMBER
40

5f. WORK UNIT NUMBER
04

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie Mellon University
Computer Architecture Lab (CALCM)
Real-Time Multimedia Systems Lab (RTML)
5000 Forbes Avenue
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7334

Defense Advanced Research Projects Agency
DARPA/IPTO
3701 N. Fairfax Drive
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2006-1509

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT

PowerTap is a power-aware system which achieves the best performance within a given power budget, or the best power efficiency under the constraints of each application's deadline in real-time applications. PowerTap dynamically monitors and adjusts power levels to meet application timing and system power/energy demands. PowerTap's Power-Aware Real-Time Operating System (PARTOS), is a morphable real-time operating system that manages power in the system hardware. PARTOS also manages power in the system software services and tunes system software computational requirements. PARTOS morphs to adapt its power management policies depending on the available power source (battery, or AC).

PowerTap uses the Power-Aware Resource Allocation Model (PAQ-RAM) to balance multiple applications' quality of service requirements and to maximize the utility (users' satisfaction) of the overall system. PAQ-RAM takes energy budget into account with other resources and adjusts the quality of service based on criticality and the amount of resources available.

15. SUBJECT TERMS
Power-Aware Computing, Quality of Service, Real-Time Operating System

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Ronald Brower
Unclassified	Unclassified	Unclassified	SAR	52	**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-6548, ext. 3590

Table of Contents

Section Page

List of Figures

Figure **Page**

List of Tables

Table **Page**

1. Introduction

The increasing levels of integration in semiconductor fabrication and the exponential growth in the number of chip transistors has given rise to a myriad of not only handheld and mobile applications but also personal computers and desktop servers. The increase in performance and functionality in those has also been accompanied by unprecedented levels of power dissipation and energy consumption. Unfortunately, power efficiency, battery capacity, size and weight have improved in much slower rates, creating a bottleneck in the utility of portable and mobile digital computer and desktop servers.

There are a large number of technologies for reducing energy/power in digital computers. Current technologies, however, are low-power rather than power-aware and suffer from four key shortcomings:

1) They use ad hoc power management policies that are typically based on stochastic state machines that use device (e.g., memory, disk or network card) activity to decide when to make transitions to low power states. These policies do not take into account application demand for speed and timing (or real-time) constraints, or device usage patterns;
2) They do not exploit demand for resource allocation and scheduling across a variety of applications to find optimal power-/energy-efficiency given a set of application QoS constraints;
3) They offer a system-wide tradeoff between high performance and low energy/power, neglecting an application's/algorithm's variability in demand for performance in individual system resources (e.g., disk bound applications do not benefit from CPU speed);
4) They do not predict system activity thereby incur high overheads (e.g., over two orders of magnitude in latency in Rambus DRAM) when system components make transitions between low-power and active states;

PowerTap achieves the best performance of the system within a given power budget, or the best power efficiency under given constraints of each application's demand such as timing constraints (deadline) in real-time applications, video and audio quality in multimedia applications, high disk bandwidth usage in video servers, application mission-time in the limited power system, etc. The PowerTap is a power-aware system design in which the software (e.g., the operating system and tools) in collaboration with the hardware manages and minimizes power consumption while providing quality of service guarantees to applications and the overall system. In PowerTap system software/hardware dynamically monitor and adjust power levels for all critical system components periodically to meet application timing, and system power/energy demands.

PowerTap's **Power-Aware Real-Time Operating System** (PARTOS), is a morphable real-time operating system that manages power in the system hardware resources such as the disks, the network interface card, and the memory. PARTOS also manages the power levels in the system software services (e.g., TCP/IP stacks), periodic activities (e.g., interrupt handlers), and tunes the system software computational requirements. PARTOS morphability allows for adapting its power management policies depending on the available power source (battery, or AC). For example, when operating under battery power source, it balances the discharging rates and the recharging rates of the batteries. On the other hand, the low average power rate is focused when the power source is AC.

PowerTap exploits the **Power-Aware Resource Allocation Model** (PAQ-RAM), which is an extension of QoS-Resource Allocation Model (Q-RAM) developed at Carnegie Mellon University, to balance multiple applications' quality of service requirements on their usage of multiple finite-resources and maximize the utility (users' satisfaction) of the overall system. Some systems (such as portable computers with battery operated) have a restricted energy usage requirement (the discharge rate of the battery has to be lower than the charge rate to maintain stability). Airplanes and other mobile vehicles are also fallen into this category. Sharing the same goal with Q-RAM, PAQ-RAM, hence, takes this energy budget into account with other resources and adjusts the quality of service of individual application based on its criticality and the amount of resources available.

2. PARTOS: Power-Aware Linux/RK

Energy-Aware Linux/RK stands for Linux/Resource Kernel with Energy-Aware functionality extension, which incorporates real-time and power-aware computing extensions to the Linux kernel to support the abstractions of an energy-efficient resource kernel. An energy-efficient resource kernel is a real-time kernel (operating system) that provides timely, guaranteed and enforced access to system resources for applications whiles minimizing energy consumption of the system. It allows applications to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. This separation of resource specification from resource management allows OS-subsystem-specific customization by extending, optimizing or even replacing resource management schemes. As a result, this resource-centric approach can be implemented with any of several different resource management schemes.

The following represents the key properties of resource kernel.

- Applications can explicitly state their timeliness requirements using the reserve specification *{C, T, D}* where *C* is the amount of maximum required resource, *T* is the period of resource accesses and *D* is the deadline of resource accesses. For example, in case of CPU reservation, *C* is the amount of processor cycles a task requires in each timer interval *T* with deadline *D*.
- The kernel decides whether a reservation can be granted during the admission control when a reservation is created.
- The kernel determines the most energy-efficient operating point both statically and dynamically which can satisfy timing constraints of all guarantees and suit best for a given hardware platform, such as the level of CPU voltage supply, processor speed, energy-aware CPU scheduling policy, memory management scheme, etc.
- Once the reservation is granted, the kernel not only guarantees that timing constraints of resource accesses are satisfied but also minimizes energy.
- The kernel enforces maximum resource usage by applications to maintain temporal isolation among tasks.
- The kernel supports high utilization of system resources which include CPU, memory and energy.
- Applications are allowed to create multiple reservations to timely access different system resources simultaneously.

In this report, we will describe briefly about Resource Kernel (RK) architecture, its energy-aware extension and the description of its API. We will also include the detail of our energy-aware CPU scheduling policies for hard-real-time applications, the integration work of classical real-time scheduling policies with our dynamic voltage scaling schemes.

A new reserve paradigm for multimedia applications, multigranularity reservation (Multi-RSV), is also presented. Multi-RSV is specifically designed for soft-real-time multimedia applications whose timing constraints can be relaxed and their much varying demand consumes much less resource in average compared to their peak requests. For this document, we focus on the implementation of Energy-Aware Linux/RK based upon Linux Kernel 2.4.x across different architecture (2.4.26 for x86, 2.4.19 for iPAQ (arm), 2.4.22 for PowerPC RAD750 (joint project with BAE) and 2.4.19 for BitsyX (joint project with Vitronics).

2.1. Resource Kernel Architecture

Figure 1 depicts the reservation architecture inside Resource Kernel (RK). The system guarantees resource accesses by means of resource reservation. One or more reservations can be grouped together and reside in a resource set which will be bound to one or more tasks. As a result, tasks are allowed to exclusively use their reserved amount of resources inside their resource set. A reserve is created based on the *{C, T, D}* reserve specification. The kernel performs the schedulability analysis of the system whenever a new reserve is created. Appropriate scheduling and enforcement of a reserve by the resource kernel guarantees that the reserved amount is always allocated for all granted reserves and the system achieves high resource utilization.

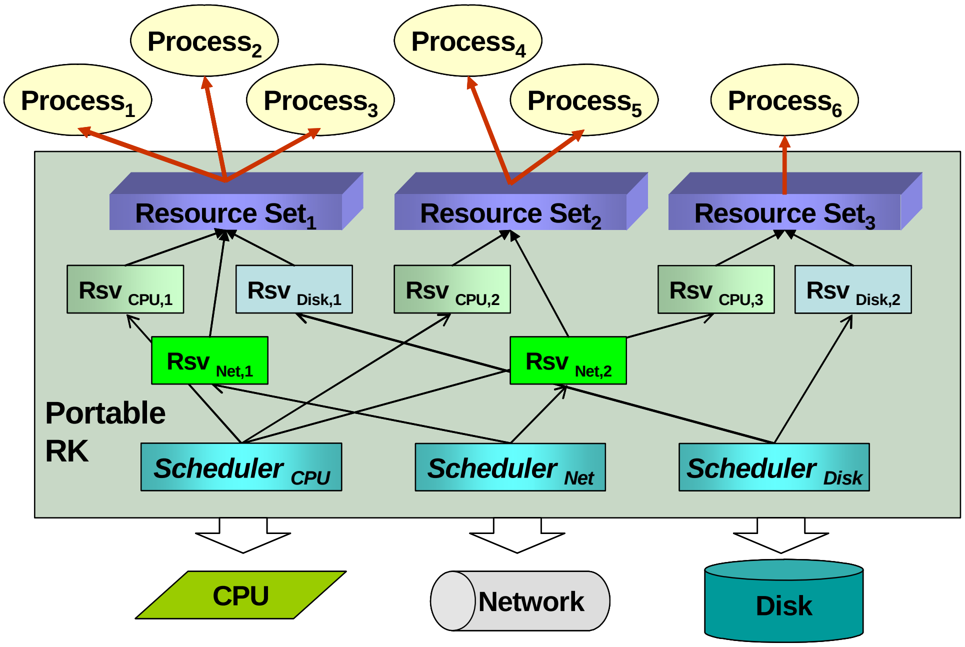


Figure 1: Resource Kernel Reservation Architecture

2.1.1 Reserve Type

The resource enforcement in Resource Kernel ensures that a misbehaving task cannot jeopardize other tasks by overusing its reserve. Whenever a task uses up its reserved resource for each period, the kernel marks the task as a depleted task and the corresponding reserve as a depleted reserve. At the end of each period, a reserve will obtain a new quota and is said to be replenished. Three types of reserves have been defined: hard reserve, soft reserve and firm reserve. Each has different behavior on the resource enforcement and replenishment after the depletion as follows:

- **Hard Reservation:** The reserve will not obtain any more resource until it is replenished. Basically, a task attached to this reserve will always be halted until its next period.
- **Firm Reservation:** The reserve will be allowed to obtain more resources only if no other tasks are ready to access the resource. In other words, a task attached to this reserve will be executed only if no other undepleted and unreserved tasks are ready to run.
- **Soft Reservation:** The reserve will be allowed to share the remaining resources from the undepleted reserves. Therefore, a task attached to this reserve will be executed along with other unreserved tasks and depleted tasks.

2.1.2 Resource Guarantee Mechanisms

Four main mechanisms are contributed to provide fine-grained timing guarantees among tasks.

- **Admission** Control**:** RK uses mathematical analysis to determine whether a new reserve can be granted without jeopardizing timing guarantees of other tasks. The current implementation of Linux/RK supports Rate-Monotonic (RM) and Deadline-Monotonic (DM) scheduling policies. The response time (RT) test (a.k.a. the exact completion time test) is performed during admission control to provide more precise analysis and achieve high system utilization under both schemes. Note that the rest of the documents will focus only on the Deadline-Monotonic scheme.
- **Reserve Accounting and Enforcement:** After a reserve is granted by the admission control, the priorities of all reserves are reassigned according to their deadlines. The reserve with the shorter deadline will be given the higher real-time priority. A task without a reservation is executed with a user-given normal priority which is always lower than any real-time priority assigned to the reservations. In other words, it will be scheduled only if no task with reserves is active. RK keeps track of how much of resources each reserve has been used. One enforcement timer is used to account for the remaining reserved amount of the current task. Once the current

task uses up its reserve (i.e. the enforcement timer expires), it will be removed from the ready queue or assigned a lower priority depending on the type of its corresponding reserve.

- **Reserve Replenishment:** RK **implements** one replenishment timer per reserve to resume a task at the beginning of reserve periods. Its corresponding reserve priority and resource quota (or part of the quota) will be restored.
- **High Resolution Timer:** RK enables the fine-grained resource control by setting a hardware timer to a single-shot timer mode. This setting replaces a periodic 10 ms interval timer mode usually configured in Linux kernel. A software timer queue is used to keep track of all required timer interrupts. Depending on the availability of hardware timers, the software timer queue also maintains the kernel jiffy interrupts (10-ms interval interrupts) if RK and the Linux kernel must share the same interrupt line.

2.1.3. Resource Synchronization

RK currently implements the reserve inheritance protocol to provide bounded priority inversion. Whenever a higher priority reserve is blocked and waiting for a sharing resource from a lower priority reserve, RK transfers the priority and resource quota of the higher priority reserve to the blocking task. The detailed information of the protocol is presented in [1].

2.1.4. Energy-Aware Extension of Linux/RK

Figure 2 depicts the architecture of the energy-aware extension of Linux/RK. Tasks are allowed to create a reservation of CPU cycles with the specification *{C, T, D}* where *C* is the number of processor cycles needed in each interval period *T* with the deadline *D*. During the admission control, the kernel verifies if the new taskset is schedulable. If the reserve is granted, an appropriate voltage-scaling scheme is chosen based on the platform and taskset characteristics to determine the energy-efficient clock frequency (CPU speed) and the minimum voltage supply which can deliver the desired clock frequency in order to minimize energy consumption. The resource accounting, replenishment and enforcement are managed such that not only the temporal isolation but also the energy isolation is maintained.

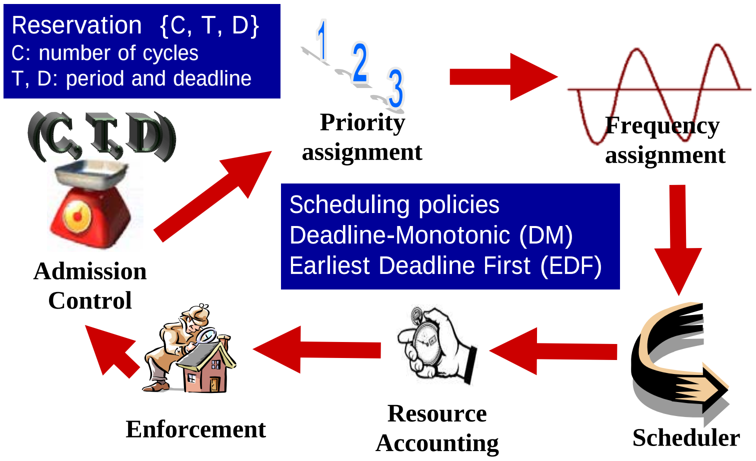


Figure 2: Energy-Aware Linux/RK Architecture

2.2. Voltage Scaling Algorithms in Linux/RK

Energy-Aware Linux/RK incorporates dynamic-voltage-scaling algorithms on CPU to minimize the energy consumption and still maintain temporal isolation for applications. This is based on the fact that power consumption in CMOS circuits is proportional to the product of the frequency and the square of the supply voltage which is given by $P = aC_L V_{DD}^2 f$ where a is the average activity factor, C_L is the average load capacitance, V_{DD} is the supply voltage and f is the operating frequency. Hence, any reductions in the operating frequency and supply voltage of the processor can lead to significant energy savings (and better heat dissipation).

A variety of dynamic voltage scaling techniques [3-7] has addressed the tradeoffs between system performance and energy efficiency. These techniques make use of operating system information about the current workload in order to reduce the processor voltage when the full system performance is not necessary. Some of these techniques target real-time systems where the timing constraints of tasks must be satisfied. The above techniques typically assume that (1) the energy consumption is minimized whenever the supply voltage is scaled down and (2) the voltage-scaling overhead is negligible. Unfortunately, these assumptions may not always be correct in practice. In some practical processors, Akihiko et al. [8] showed that there are some energy-inefficient operating frequencies in the sense that operating the same workload with a higher frequency will counter-intuitively consume less energy. Those energy-inefficient operating frequencies therefore must be avoided under all circumstances. In addition, operating frequencies usually cannot be continuously varied, and are only available as a small number of discrete points. To sustain acceptable performance and timeliness guarantees, these processors have to operate at the next higher *energy-efficient* operating frequency if a desired frequency is neither available nor energy-efficient. This inevitably results in more energy consumption. In addition, we must also explicitly consider the effect of the overhead of changing processor frequencies. Whenever frequency scaling is performed, processor internal clocks and DRAM timings need resynchronization. Depending upon the implementation, these frequency-or voltage-scaling delays can be either small enough to be negligible or large enough that they can disrupt taskset timings significantly.

With varying characteristics across hardware platforms, we support four voltage-scaling algorithms, Sys_Clock, PM_Clock, DPM_Clock and Progressive DPM_Clock (or Progressive in short term). Each is suitable for different system characteristics and will be selected automatically by our reservation-based resource kernel. The description of each scheme is as follows:

- **System Clock Frequency Assignment (Sys-Clock):** This scheme is suitable for systems such as iPAQ [9] where the overhead of voltage- or frequency- scaling is too high to perform at every context switch. *One* clock frequency is determined at admission control and kept constant until the taskset changes.
- **Priority-Monotonic Clock Frequency Assignment (PM-Clock):** This scheme is suitable for systems with low voltage- or frequency-scaling overhead such as our modified version of the XScale BRH board [10]. Each task is assigned its own clock frequency. The scheme scales voltage and frequency at every context switch based on the frequency assigned to the next task to run.
- **Dynamic PM-Clock (DPM-Clock):** This scheme is suitable for systems where the average execution time of a task is considerably less than its worst case. Like PM-Clock, this scheme is suitable for systems with low voltage- or frequency-scaling overhead.
- **Progressive DPM-Clock (Progressive):** This objective of this scheme is similar to DPM-Clock scheme but delivers higher performance in energy saving by trading off its algorithm complexity.

We also study the optimal scheme, **Optimal Clock Frequency Assignment (Opt-Clock)**, which uses non-linear optimization techniques to determine the optimal clock frequency for each task in order to minimize the energy consumption. Opt-Clock has high complexity and is not suitable for on-line usage. Several pruning techniques are proposed to dramatically reduce the scheme's complexity for off-line usage. All these algorithms assume the deadline-monotonic scheduling policy. However, they can be easily applied to other fixed-priority preemptive scheduling policies by computing the preemption time based on the assigned task priorities. **Error! Reference source not found.** summarizes the properties of proposed algorithms. We will present more details of each algorithm in the following subsections.

Table 1: Summary of DVS Algorithms

Algorithms	Frequency Assignment	Suitable Platform	Suitable Taskset	Computational Complexity	Platforms
Sys-Clock	*One* freq. for all tasks at admission control	high scaling overhead		low	iPAQ H3760 (15-20 ms overhead)
PM-Clock	One freq. *per task* at admission control (Change freq every context switch)	acceptable scaling overhead	execution time close to WCET	medium	our modified voltage-scaling XScale BRH board
Opt-Clock				high (off-line)	
DPM-Clock/ Progressive	One freq. per task instance at admission control and context switch		varying execution Low AVG/WCET (Multimedia App)	medium	

2.2.1. System Model and Terminology

We use a reservation-based resource kernel [11, 12] as the system model. A task τ_i has its specification as $\{C_i, T_i, D_i\}$, where C_i is its worst-case required processor cycles, T_i is its period and D_i is its relative deadline from arrival time. The clock frequency we refer throughout the paper is a relative frequency normalized to $f_{\max}$. By convention, we assume that $D_1 \le D_2 \le \ldots \le D_n$. In addition, we assume processors have a convex nondecreasing power-frequency relation which is expected to be true in practice. We use the term "frequency-scaling" which actually scales both frequency and voltage simultaneously if a platform is voltage-scaling enabled.

2.2.2. Sys-Clock Algorithm

Sys-Clock applies a well-known property of a processor with a non-decreasing convex power-frequency function: the energy is minimized if the processor executes its workload at the lowest possible *constant* speed. However, in a system using a fixed-priority preemptive scheduling policy, the workload needed to complete a task's request composes of the task's own execution and the preemption by higher priority tasks. When there are multiple task periods, the preemption is not uniformly distributed over the task's critical zone. The critical zone assumes that the requests of all tasks arrive simultaneously. This leads to the worst-case scenario [13]. The workload hence varies in preempting processor cycles and depends on when the task completes. Sys-Clock determines the energy-minimizing clock frequency by finding the minimum clock frequency that allows every task to complete before its deadline. We will illustrate the algorithm by an example. Note that we assume the use of DM scheduling policy.

Consider a taskset that has three tasks $\{\tau_1, \tau_2, \tau_3\}$ with the task specification as shown in Figure 3. The timeline depicts the critical zone of task τ_3 at the maximum clock frequency $f_{\max}$. As can be seen, the completion time of τ_3 is *9*, which we refer to as *the earliest possible completion time*. If the processor operates at a lower clock frequency, not only the execution time needed by all tasks increases but also the number of preempting processor cycles. Since τ_3 can be preempted by τ_1 at time *30* to *32*, the task will definitely miss its deadline if it cannot complete before time *30*. We refer to this last time instant that a task can complete and meet its deadline as *the latest possible completion time*. Since the workload changes at the end of each idle period[1], Sys-Clock determines the clock frequencies which allow a task to complete execution exactly at the end of each idle period between the earliest and latest possible completion time. The energy-minimizing clock frequency of a task is chosen to be the minimum clock frequency among these frequencies.

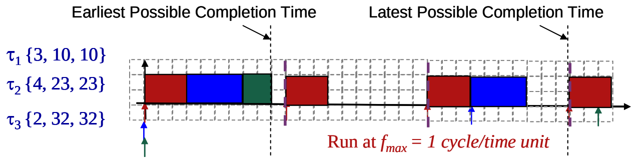

Earliest Possible Completion Time
Latest Possible Completion Time
τ_1 {3, 10, 10}
τ_2 {4, 23, 23}
τ_3 {2, 32, 32}
Run at f_{max} = 1 cycle/time unit

Figure 3: Workload versus Completion Time

[1] An idle period is a time period where the processor is idle.

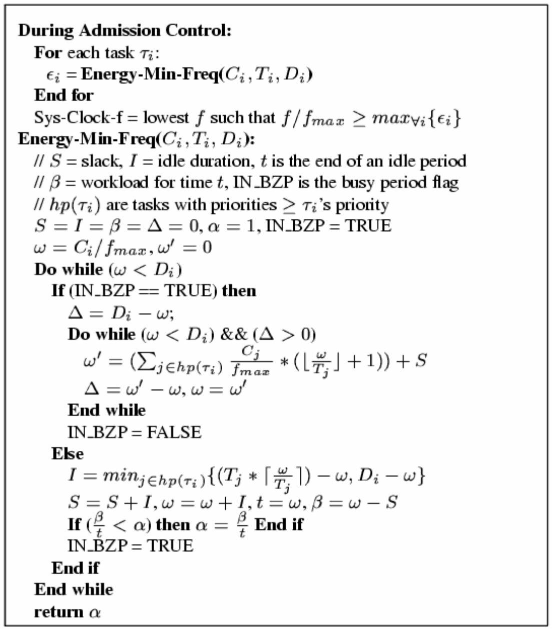
By convention, we denote the workload needed by a task and its total preemption at time *t* as β_i^t , the corresponding constant clock frequency as α_i^t and the energy-minimizing clock frequency as ε_i where i represents an index of task τ_i and *t* represents the end of an idle period. For this example, the ends of idle periods to consider for τ_3 are *10*, *20* and *30*;

$$\beta_3^{10} = \frac{C_1 + C_2 + C_3}{f_{\max}} = 9, \qquad \alpha_3^{10} = \frac{\beta_3^{10}}{10} = \frac{9}{10} = 0.9 \ ,$$
$$\beta_3^{20} = \frac{2C_1 + C_2 + C_3}{f_{\max}} = 12, \quad \alpha_3^{20} = \frac{\beta_3^{20}}{20} = \frac{12}{20} = 0.6 \ ,$$
$$\beta_3^{30} = \frac{3C_1 + 2C_2 + C_3}{f_{\max}} = 19, \quad \alpha_3^{30} = \frac{\beta_3^{30}}{30} = \frac{19}{30} = 0.63 \ . \tag{1}$$

Hence,

$$\varepsilon_3 = \min\{\alpha_3^{10}, \alpha_3^{20}, \alpha_3^{30}\} = \min\{0.9, 0.6, 0.63\} = 0.6 \,. \tag{2}$$

Note that this clock frequency is a relative frequency normalized to $f_{\max}$. The energy-minimizing clock frequency of a task τ_i is the clock frequency needed by the task itself *and* its higher priority tasks to minimize the energy and meet τ_i 's deadline. However, this frequency may not satisfy other tasks' timing constraints. For schedulability, Sys-Clock will set the system clock frequency to the maximum frequency among energy-minimizing clock frequencies of all tasks. For this example, Sys-Clock assigns the system clock frequency to $\max\{\varepsilon_1, \varepsilon_2, \varepsilon_3\} = 0.6$. Figure 4 summarizes the Sys-Clock algorithm.



```
During Admission Control:
  For each task $\tau_i$:
    $\epsilon_i$ = Energy-Min-Freq($C_i, T_i, D_i$)
  End for
  Sys-Clock-f = lowest $f$ such that $f/f_{max} \geq max_{\forall i}\{\epsilon_i\}$
Energy-Min-Freq($C_i, T_i, D_i$):
  // $S$ = slack, $I$ = idle duration, $t$ is the end of an idle period
  // $\beta$ = workload for time $t$, IN_BZP is the busy period flag
  // $hp(\tau_i)$ are tasks with priorities $\geq \tau_i$'s priority
  $S = I = \beta = \Delta = 0, \alpha = 1$, IN_BZP = TRUE
  $\omega = C_i/f_{max}, \omega' = 0$
  Do while ($\omega < D_i$)
    If (IN_BZP == TRUE) then
      $\Delta = D_i - \omega$;
      Do while ($\omega < D_i$) && ($\Delta > 0$)
        $\omega' = (\sum_{j \in hp(\tau_i)} \frac{C_j}{f_{max}} * (\lfloor \frac{\omega}{T_j} \rfloor + 1)) + S$
        $\Delta = \omega' - \omega, \omega = \omega'$
      End while
      IN_BZP = FALSE
    Else
      $I = min_{j \in hp(\tau_i)}\{(T_j * \lceil \frac{\omega}{T_j} \rceil) - \omega, D_i - \omega\}$
      $S = S + I, \omega = \omega + I, t = \omega, \beta = \omega - S$
      If ($\frac{\beta}{t} < \alpha$) then $\alpha = \frac{\beta}{t}$ End if
      IN_BZP = TRUE
    End if
  End while
  return $\alpha$
```

Figure 4: Sys-Clock Algorithm

From an energy perspective, Sys-Clock is optimal for fixed-priority preemptive scheduling policies that use a single clock frequency. The only way for other schemes to consume less energy than Sys-Clock is assigning a lower system clock frequency to a taskset. However, lowering the clock frequency than the Sys-Clock frequency which is an energy-minimizing clock frequency of a task will cause the task to miss its deadline. Sys-Clock thus is optimal in the sense that there is no other scheme that consumes less energy and guarantees timing constraints of a taskset.

2.2.3. PM-Clock Algorithm

PM-Clock is adapted from Sys-Clock. First, it determines energy-minimizing clock frequencies of all tasks like Sys-Clock. With different timing constraints, it is possible that a high priority task τ_H may have a more stringent timing constraint and hence need a higher clock frequency than a low priority task τ_L . To meet τ_H 's deadline, it is necessary to assign a higher clock frequency than that for τ_L . In addition, it is also possible that the desired frequency is neither available nor energy-efficient, and hence a higher frequency needs to be chosen. In either case, PM-Clock will save more energy by recalculating a lower clock frequency for tasks such as τ_L exploiting the slack from these higher frequency assignments for tasks such as τ_H .

Consider a set of two tasks as shown in Figure 5. Following the energy-minimizing clock frequency calculation described in Section 2.2.2, we obtain $\varepsilon_1 = 0.50$ and $\varepsilon_2 = 0.45$. As can be seen, even though τ_2 only requires τ_1 running at *0.45*, τ_1 must run at a higher operating frequency of *0.50* to satisfy its schedulability. We refer to this higher frequency as an *inflated frequency* with respected to τ_2 . The use of an inflated frequency by a higher priority task leads to more available slack for a lower priority task. PM-Clock then recalculates the new energy-minimizing clock frequency of τ_2 by fixing the clock frequency of τ_1 at their inflated frequency. Figure 5 shows the critical zone of τ_2 before and after the effect of inflated frequency. The new slack for τ_2 becomes $20-(4C_1/0.5 * f_{\max}) = 4$. Therefore, the clock frequency for τ_2 can be reduced to $1/4 = 0.25$. Figure 6 summarizes the PM-Clock algorithm.

Note: The term "priority-monotonic" comes from the fact that PM-Clock always assigns task clock frequencies in priority order. In other words, a high priority task will always be assigned a higher or same clock frequency compared to a low priority task.

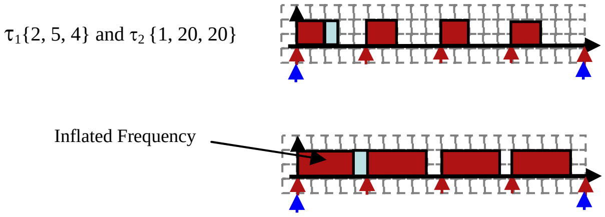
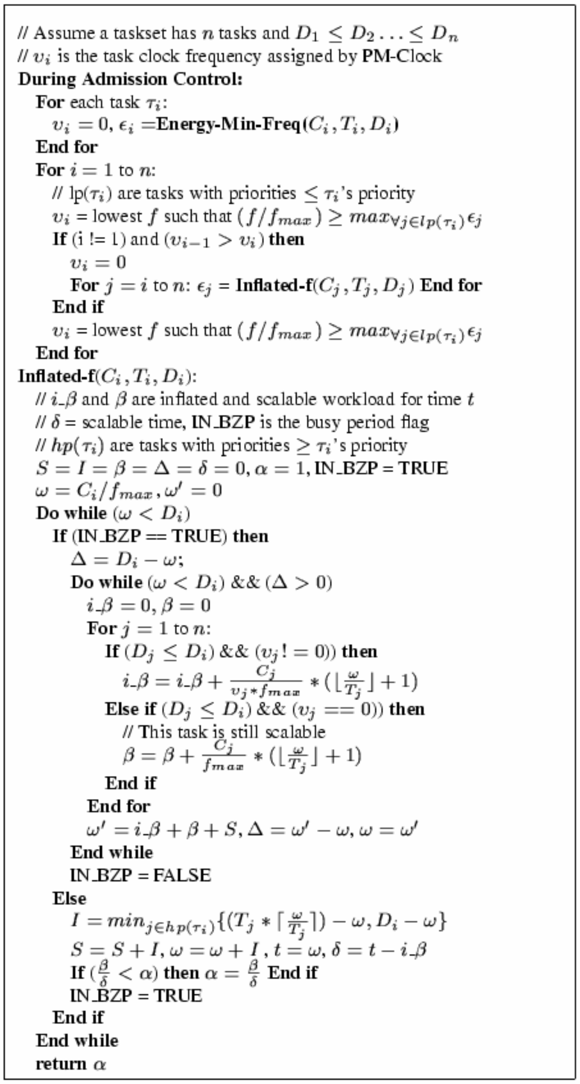


Figure 5: An Example of PM-Clock



```
// Assume a taskset has $n$ tasks and $D_1 \leq D_2 \ldots \leq D_n$
// $v_i$ is the task clock frequency assigned by PM-Clock
During Admission Control:
  For each task $\tau_i$:
    $v_i = 0$, $\epsilon_i =$ Energy-Min-Freq($C_i, T_i, D_i$)
  End for
  For $i = 1$ to $n$:
    // lp($\tau_i$) are tasks with priorities $\leq \tau_i$'s priority
    $v_i$ = lowest $f$ such that $(f/f_{max}) \geq max_{\forall j \in lp(\tau_i)} \epsilon_j$
    If (i != 1) and $(v_{i-1} > v_i)$ then
      $v_i = 0$
      For $j = i$ to $n$: $\epsilon_j$ = Inflated-f($C_j, T_j, D_j$) End for
    End if
    $v_i$ = lowest $f$ such that $(f/f_{max}) \geq max_{\forall j \in lp(\tau_i)} \epsilon_j$
  End for
Inflated-f($C_i, T_i, D_i$):
  // $i\_\beta$ and $\beta$ are inflated and scalable workload for time $t$
  // $\delta$ = scalable time, IN_BZP is the busy period flag
  // $hp(\tau_i)$ are tasks with priorities $\geq \tau_i$'s priority
  $S = I = \beta = \Delta = \delta = 0, \alpha = 1$, IN_BZP = TRUE
  $\omega = C_i/f_{max}, \omega' = 0$
  Do while $(\omega < D_i)$
    If (IN_BZP == TRUE) then
      $\Delta = D_i - \omega;$
      Do while $(\omega < D_i)$ && $(\Delta > 0)$
        $i\_\beta = 0, \beta = 0$
        For $j = 1$ to $n$:
          If $(D_j \leq D_i)$ && $(v_j != 0)$) then
            $i\_\beta = i\_\beta + \frac{C_j}{v_j * f_{max}} * (\lfloor \frac{\omega}{T_j} \rfloor + 1)$
          Else if $(D_j \leq D_i)$ && $(v_j == 0)$) then
            // This task is still scalable
            $\beta = \beta + \frac{C_j}{f_{max}} * (\lfloor \frac{\omega}{T_j} \rfloor + 1)$
          End if
        End for
        $\omega' = i\_\beta + \beta + S, \Delta = \omega' - \omega, \omega = \omega'$
      End while
      IN_BZP = FALSE
    Else
      $I = min_{j \in hp(\tau_i)}\{(T_j * \lceil \frac{\omega}{T_j} \rceil) - \omega, D_i - \omega\}$
      $S = S + I, \omega = \omega + I, t = \omega, \delta = t - i\_\beta$
      If $(\frac{\beta}{\delta} < \alpha)$ then $\alpha = \frac{\beta}{\delta}$ End if
      IN_BZP = TRUE
    End if
  End while
  return $\alpha$
```

Figure 6: PM-Clock Algorithm

2.2.4. DPM-Clock Algorithm

DPM-Clock is an on-line DVS scheme which can monitor actual execution times of tasks, and further minimize energy consumption when those execution times are less than the pre-reserved and guaranteed worst-case execution times. It detects the early completion times and makes use of the additional slack time created at run-time by reducing the processor speed. Even though the available slack from the early completion of a task instance can be shared among multiple tasks, DPM-Clock instead devotes all slack to the next ready task which has lower or same priority.

The new operating frequency of this “lucky” task is computed relative to its current assigned operating frequency. Every new invocation of a task instance always starts with its original operating frequency assigned by PM-Clock and changes its operating frequency dynamically only if it gets the slack. We denote v_j and v'_j as τ_j 's PM-Clock and dynamic clock frequency. We denotes the slack from τ_i 's early completion as S_i . DPM-Clock expands the execution time of left cycles to fill all the slack. Therefore,

$$v'_j = \frac{(LEC_j / v'_j)}{(LEC_j / v'_j) + S_i} * v'_j , \tag{3}$$

$$S_i = (WCEC_i - CEC_i) / v_i \, . \tag{4}$$

Where *LEC*, *WCEC* and *CEC* denote the unexecuted, the worst-case desired and the actual desired processor cycles of the current task instance.

When there is no ready task in the queue after the early completion of an invocation, DPM-Clock decrements the available slack time when the processor remains idle. The next arriving task whose priority is lower than or equal to the early completed task would receive the updated slack time if the slack is still available (i.e., greater than 0).

This dynamic frequency adjustment of DPM-Clock maintains schedulability since there is no priority change. The frequency reduction of τ_j will delay only tasks with same or lower priority. Hence, the preemption time seen by tasks is the same.

2.2.5. Progressive DPM-Clock Algorithm

Like DPM-Clock, Progressive is an on-line DVS scheme which detects run-time slack and further diminishes energy consumption by reducing the processor speed and corresponding voltage supply to expand executions over the slack time. However, instead of transferring slack to the whole demand of the next scheduling task, Progressive aggressively reclaims the available slack for the expected workload towards the next scheduling point. Moreover, by investigating the next two scheduling points, it can claim slack not only from higher priority tasks but also from lower priority tasks, itself and possible idle period.

2.2.5.1. Progressive Slack Detection

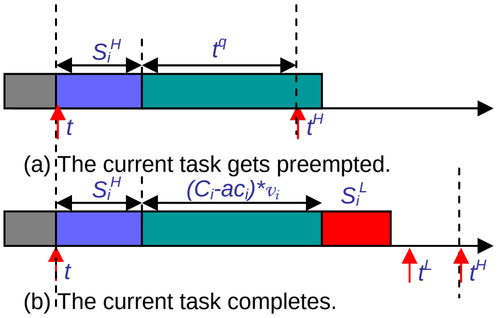
During context switches, Progressive collects runtime slacks from *early-executed cycle slack*, *unused reserved cycle slack* and *unreserved cycle slack* as followings. Progressive prioritizes and always claims slack in the descending order of its priority. This is based on the fact that the preemptive scheduling will always execute high priority task first.

- **Early Executed Cycle Slack:** Slack from active task instances which are executed ahead of the schedule. At context switches, Progressive computes the number of processor cycles the current task is supposed to be execute if all priority slack preempts the task. In other words, it is the amount of processor cycles the current task will be executed if all tasks are scheduled by PM-Clock and always use up their reservations. At the preemption, the scheme then monitors if the task has been executed more than the expectation. These additional cycles are the product of (1) the use of higher operating frequency when a desired frequency is not available due to finite operating frequencies and (2) the overwhelming slack. Let us denote ec_i , c_i and S_i^e as τ_i 's expected processor cycles, actual processor executed and early executed cycle slack at the last context switch, respectively. Therefore the slack with priority of τ_i is given by $S_i^e = (c_i - ec_i) / v_i$ where v_i is the task's clock frequency which is initially assigned by PM-Clock.
- **Unused Reserved Cycle Slack:** Slack from completed task instances which request executions less than their worst-case amounts which are pre-reserved and guaranteed. Let ac_i denote the *accumulated* execution a task τ_i has been executed until its last context switch. Let C_i and S_i^c denote the task's reserved cycles and its slack from early completion. Therefore, when a task is completed, the early completion slack is given by $S_i^c = (C_i - ac_i - ec_i) / v_i$.
- **Unreserved Cycle Slack:** Slack from an idle period after the current task is executed. By investigating the next two scheduling points in advance, Progressive is able to predict whether there will be an idle period after the task's execution and further scale down the task's execution to complete at the end of the idle period.

2.2.5.2. Progressive Slack Distribution

As previously mentioned, Progressive orders slacks by the task priority of their owners, considering that slack from idle periods is the lowest-priority slack. We now describe how Progressive distributes those slacks among tasks. Let τ_i is the current task to be executed.

At context switches, Progressive first determines if the current task τ_i is able to complete its execution before the next context switch under PM-Clock worst-case schedule. If all high priority tasks use up their reservations, the high priority slack S_i^H , which is slack time for higher or same priority will be occupied by those high priority executions. If the task can complete its execution and a lower priority task τ_L is the next task to run, the task will be able to use slack whose priority is higher than τ_L 's priority as well. In addition, the task can claim slack from idle period if available. In summary, the slack distribution of Progressive can be categorized to two main case scenarios as shown in Figure 7.



(a) The current task gets preempted.

(b) The current task completes.

Figure 7: Possible Scenarios in Progressive Slack Distribution

- **When the current task gets preempted:** Let t and t^H be the current time and the next preemption time. If the current task τ_i cannot complete its job within its quota time, all of higher-priority slacks will be distributed to the portion of τ_i 's expected execution. We denote the quota time of a task τ_i as t^q which is given by $t^q = t^H - t - S_i^H$. The current task gets preempted if $t^q < (C_i - ac_i) * v_i$. Therefore,

$$ec_i = (t^H - t - S_i^H) * v_i = (t^H - t - \sum_{j \le H} (S_j^e + S_j^c)) * v_i \,, \tag{5}$$

$$v'_i = \frac{ec_i}{(t^H - t)} = \frac{t^H - t - \sum_{j \le H} (S_j^e + S_j^c)}{t^H - t} * v_i \,. \tag{6}$$

The reasons that we distribute slack to only portion of the current task's execution are as follows:

1. Slack can be exhausted when no use.
2. Adequate slack is needed to step-down the voltage for processors with finite frequencies so aggressive slack is needed for efficient energy saving.
3. The algorithm is much simpler to keep track of all slacks.

Note that it is possible that the current task is way ahead of its schedule, in other words when $S_i^H \ge t^H - t$. For this scenario, Progressive will execute the current task at the lowest speed and the advance execution will be accounted as early executed cycle slack previously described.

- **When the current task completes:** This case happens when the left reserved cycles of the current task executing at the clock frequency v_i will be completed before t^H . Assume that τ_L is the next low priority reserved task to run after the current task. With the same preemption seen by τ_L and other following low priority tasks, the current task can use the slack time not only from middle priority tasks but also from the idle period if available (when $t^L > t + S_i^H + S_i^L + (C_i - ac_i) * v_i$). Let t^S be the elapsed time that the current can occupy and still maintain the schedulability of the task set. Therefore,

$$t^S = \max(t^L - t, S_i^H + (C_i - ac_i) * v_i + S_i^L) \,, \tag{7}$$

$$v'_i = (C_i - ac_i) / t^S \,. \tag{8}$$

After each execution and the end of any idle period, Progressive updates the available slacks by deducting the execution from those in descending order of their priorities.

2.2.6. Opt-Clock Algorithm

Finding the optimal task clock frequency is not a trivial problem but can be a large-scale non-linear optimization problem. Consider a voltage-scaling processor with a convex nondecreasing power-frequency function given by $P(f)=cf^{x}, x\geq 3$ where c is a constant value. Assume that a fixed-priority preemptive scheduling policy is used. We can prove that the optimal task clock frequency assignment algorithm is a convex non-linear minimization problem as follows:

The goal of Opt-Clock scheme is to minimize the energy consumption of a taskset in one hyper-period[2]. Every instance of a task τ_i takes $C_i/(f_i * f_{\max})$ time unit where f_i is its assigned clock frequency. The energy consumed over one hyper-period (*E*) is as follows:

$$\begin{aligned} E &= \sum_{i=1}^{n} c(H/T_i)(C_i/(f_i * f_{\max}))(f_i * f_{\max})^x, x \geq 3 \\ &= cH\sum_{i=1}^{n} U_i f_{\max}^{y} (1/\delta_i)^y, y \geq 2. \end{aligned} \tag{9}$$

Denote (H/T_i) as the number of each task instances in a hyper-period, U_i as its utilization given by C_i/T_i and δ_i as the reciprocal of f_i, respectively. From if f_x has a second derivative in [a, b] then a necessary and sufficient condition for it to be convex on that interval is that the second derivative $f''(x) > 0, \forall x \in [a,b]$. Combine this theorem with a property of convex functions that the combination and the scaling of convex functions are also convex. It can be shown that this problem has a convex objective function.

As described in Section 2.2.2, a task's workload varies with its completion time, i.e. the clock frequency assigned to each task. Consequently, each task has multiple choices of constraints in order to complete the workload at any time before its deadline. For example, consider the taskset shown in Figure 3. The constraint of task τ_2 can be either one of the following conditions:

$$\lceil 10/T_1 \rceil (C_1\delta_1/f_{\max}) + \lceil 10/T_2 \rceil (C_2\delta_2/f_{\max}) \quad \leq 10, \tag{10}$$

$$\lceil 20/T_1 \rceil (C_1\delta_1/f_{\max}) + \lceil 20/T_2 \rceil (C_2\delta_2/f_{\max}) \quad \leq 20. \tag{11}$$

Inequalities (10) and (11) are necessary conditions to complete the task before time *10* and *20*, respectively. These constraints are linear and can be considered as convex. Hence, the optimal task clock frequency assignment is a convex non-linear minimization problem since its non-linear objective function and constraint set are convex.

The problem size of Opt-Clock grows dramatically not only with the number of tasks but also the number of constraint choices of each task. This latter number depends on the number of idle periods in the task’s critical zone and can be very large if the ratio of the task’s and the highest priority task’s period is large. We now present three pruning techniques which can potentially decrease the problem size:

1. **Pruning of High Workload Constraints:** A constraint is said to be a *high-workload constraint* if it requires a shorter completion time β_S and has higher average processor demands from all higher priority tasks than another constraint with a longer completion time β_L. This technique eliminates such redundant high workload constraints. Since the average demands from all tasks decrease if the task completes at β_L, using the optimal frequency set that satisfies the constraint for β_S will definitely generate some slack. Consequently, some task's clock frequencies can always be reduced to save more energy. Therefore, the solution from the constraint set for β_S will impossibly give the optimal solution and can be pruned.

[2] Consider n periodic tasks with $\tau_i = \{C_i, T_i\}, i \in [1,n]$. The hyper-period is given by $LCM\{T_i \mid i \in [1,n]\}$.

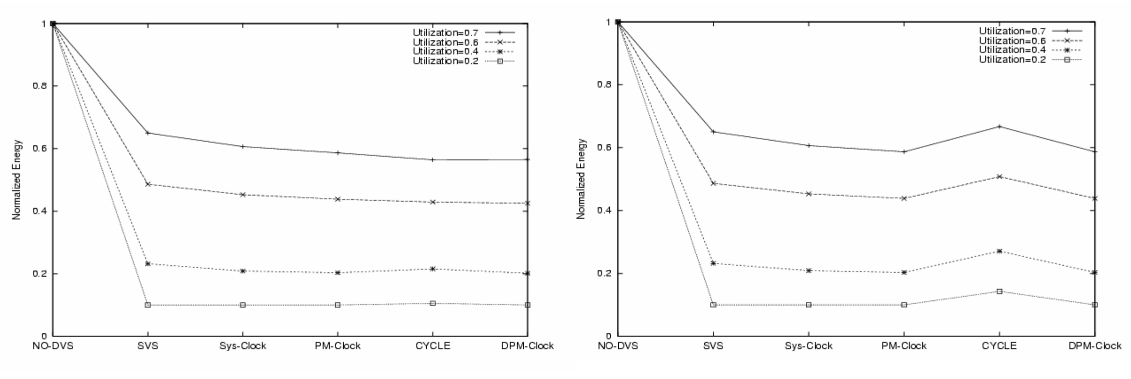
2. **Pruning of Conflict Constraints:** From constraints left over from the previous step, this technique eliminates infeasible combinations of constraints where a lower priority task requires a shorter completion time than a higher priority task, which is not schedulable by fixed-priority preemptive scheduling.

3. **Pruning of Inactive Constraints:** Unlike the first two techniques, this technique uses additional information from a solution of one problem to prune others. It first determines an inactive constraint from the current solution. Note that an inequality constraint $g(\delta) \le D$ is said to be *inactive* at δ_x if $g(\delta_x) < D$. It is *active* at if $g(\delta_x) = D$. Apply the Karush-Kuhn-Tucker (KKT) theorem that the optimal solution subject to a set of active and inactive constraints is the same as that subject to only active constraints. Since a better result is not obtained by adding more constraints to the problem, it is sufficient to eliminate other constraint sets which alter only the inactive constraints.

2.2.7. Algorithm Comparison

We compare the energy consumption of our voltage-scaling algorithms with that of well-known algorithms called the static voltage scaling algorithm (SVS) and the cycle-conserving RT-DVS (CYCLE) proposed in [5]. The processor is modeled based on $P = kf^3$ power-frequency relation, zero idle energy and ten available operating points. A real-time taskset is generated randomly. Each task has a uniform probability of having a short (0.1-1 *ms*), medium (1-10 *ms*) or long (10-100 *ms*) period. Task period is uniformly distributed in each range. Task computation is randomly selected and then adjusted based on the system utilization and the ratio of the best-case (BCET) to worst-case execution time (WCET). For the first experiment, we randomly generate tasks and scaled their utilization equally to achieve the desired total utilization. For the second experiment, we randomly generate tasks such that each task instance requests a random number of processor cycles which is uniformly distributed between BCET and WCET[3].

Figure 8 shows the average energy consumption normalized to energy of no-DVS system from the first experiment. Figure 9 shows the effect of BCET/WCET ratio at system utilization of 0.5 where the energy shown is normalized to energy of SVS algorithm. As can bee seen from both experiments, Sys-Clock and PM-Clock always outperform SVS. This is due to the fact that both algorithms determine the frequency needed to complete the task at the end of the idle period which has minimum workload, not the deadline as SVS does. PM-Clock performs better than Sys-Clock as expected with the tradeoff of additional voltage-scaling overhead during each context switch. With varying BCET/WCET ratio in the second experiment, the cycle-conserving algorithm performs very well when the BCET/WCET is low but somewhat poorly when the BCET/WCET is high. This is because the scheme always executes any task at low speed with the hope of saving energy in the future if the task uses less resource. DPM-Clock performs very close to the cycle-conserving scheme when BCET/WCET is low, and better otherwise even though it has much less complexity. Progressive performs best among all schemes with a little bit more complexity at each context switch compared to DPM-Clock.



(a) BCET/WCET=0.5

(b) BCET/WCET=1.0

Figure 8: Energy vs. System Utilization at BCET/WCET=0.5

[3] These execution times are subjected to $f_{\max}$.

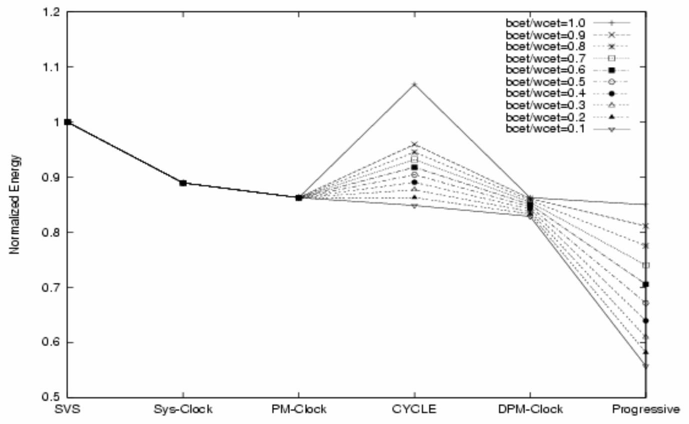


Figure 9: Energy vs. BCET/WCET at U=0.5

2.2.8. The Effect of Finite Frequency Granularity

Practical processors usually provide a finite number of operating frequencies. To sustain timeliness guarantees, these processors have to operate at the next higher operating frequency if a desired frequency is not available. This inevitably results in more energy consumption. We refer to such an energy loss as *the energy quantization error*. We investigate an operating frequency grid which minimizes the worst-case energy quantization error for a processor with *N* operating points. The result is as follows:

$$f_i = \sqrt{\frac{i}{i+1}} f_{i+1}, i \in [1, N-1), \tag{12}$$

$$f_{N-1} = \sqrt{\frac{N-1}{N}} f_{\max}. \tag{13}$$

Our study shows that the minimum energy quantization error varies with the number of operating points as follows:

$$\Delta E = \frac{k\lambda f_{\max}^2}{N} = \frac{E_{noDVS}}{N}. \tag{14}$$

The detail of the proof is presented in [2].

2.2.9. The Implementation

We have implemented these voltage-scaling schemes on CMU's Linux/RK, extensions of which are commercially available from TimeSys Corporation. We refer to the resulting kernel "Power-Aware Linux/RK". We currently have two different hardware targets for Power-Aware Linux/RK. The first of these targets is the 206 MHz Compaq iPAQ H3700 personal digital assistant, a popular and easily accessible target not unlike the ubiquitous PC. On this target, frequency scaling is possible but not voltage-scaling. Also, frequency scaling requires re-synchronization between the CPU and SDRAM timing, which in turn takes a relatively astronomical delay of 20 ms. In other words, once frequency scaling is initiated, the processor becomes unavailable for 20ms! Such a delay is unacceptable for many real-time systems. Linux/RK therefore uses the Sys-Clock algorithm to scale the frequency based on the taskset workload only at admission-control time or on task deletion. The changes to the Linux/RK kernel are therefore confined to the admission control and task exit modules. These changes constitute less than 100 lines of code. Another quirk of the iPAQ hardware is that the entire range of operating frequencies (about 70 MHz - 206 MHz) is not usable: the processor can operate within only one of two non-overlapping ranges (~70 MHz - 140 MHz or ~155 MHz - 206 MHz). The frequency cannot be scaled from a value within one range to a value in another range. Multimedia applications including a music player have been ported to demonstrate the functionality of our kernel. This kernel with its support for frequency-scaling can be downloaded from http://www.cs.cmu.edu/~rtml.

As is already known, more significant energy savings can be obtained by voltage-scaling. We have successfully modified the XScale BRH single-board computer, which boasts a 733 MHz XScale processor a 128 MB memory to support voltage scaling. The Maxim 1855 evaluation kit [14] which is a high-power dynamically adjustable notebook CPU power supply application circuit, was used to provide the programmable supply voltage for the XScale processor card. The kit provides a digitally adjustable voltage from 0.6 to 1.75 V. A total of 11 voltage settings (1.0 to 1.5 V with a step of 0.5 V) is available for the OS. A particular value is chosen by simply writing a 4-bit value to a memory-mapped address. Voltage and resulting frequency changes take a negligible amount of time (of the order of a few microseconds). We have ported Power-Aware Linux/RK with Sys-Clock, PM-Clock and DPM-Clock algorithms to this modified BRH board. The Linux kernel has successfully been loaded and tested. With the low voltage-scaling overhead, the modified XScale BRH board can efficiently save more energy using the PM-Clock and DPM-Clock schemes.

We have also implemented our Energy-Aware Linux/RK on BitsyX platform (a joint project with Vitronics). The BitsyX is a full-featured single board computer using Intel's PXA255 RISC microprocessor with an SA-1111 StrongARM companion chip. Since the BitsyX uses PXA255 which requires time for 20 *ms* to resynchronize an LCD during frequency scaling, we set the Sys-Clock algorithm by default. The software can be downloaded at http://www-2.cs.cmu.edu/~rtml/bitsyx/bitsyx-linux-2.4.19.tgz. The QT voltage and frequency monitoring tool is also available at http://www-2.cs.cmu.edu/~rtml/bitsyx/ bitsyxmon.tgz.

For a joint project with BAE and ISI, we have implemented our Energy-Aware Linux/RK on PowerPC RAD750 platform. This platform has much less timing overhead in voltage and frequency scaling compared to BitsyX and BRH. There are seven operating frequencies available, 4.125, 8.25, 16.5, 33, 99, 115.5 and 132 MHz. The minimum voltage supply needed for each operating frequency varies across hardware from 2.1 *mV* to 2.5 *mV*. The software can be downloaded at http://www-2.cs.cmu.edu/~rtml/bae/bae_linux_2.4.22_rk.tgz.

2.3. Multi-Granularity Reservation for Multimedia

Resource reservation has been recently supported by many real-time operating systems to provide applications with hard-guaranteed and timely access to system resources. Typically, reservations are based on the worst-case requirements, and therefore can inflate resource demands unnecessarily. Many multimedia applications such as MPEG video streams (1) have high worst-case to average-case demand ratio and (2) can tolerate some deadline misses. To support such applications, we propose a "multi-granularity" reservation model. Instead of the classical $\{C, T, D\}$ model of resource reservation, the multi-granular reserve specification is given by $\{\{C, T, D\},\ldots,\{C_x, x{\cdot}T\},\ldots, \{C_y, y{\cdot}T\}\}$ which represents a guarantee of the highest-granularity reserve for C units of resource during every successive periodic interval of T only as long as the resource usage by each of its low-granularity reserves (e.g., C_x units of resource in every recurring time of T) is maintained. This multi-granular reservation approach delivers higher system utilization than the pessimistic strategy of worst-case reservation and better temporal isolation than other stochastic and heuristic guarantees in the literature. We perform a detailed schedulability analysis of this model using deadline-monotonic scheduling and derive an appropriate admission control test. We also present detailed analyses and simulation results comparing our reservation scheme for MPEG-4 streams with average-case resource reservation, constant bandwidth server (CBS), and (m,k)-firm guarantee.

We first now list some important considerations that influence the design of multi-granularity reservation.

- **Flexibility of QoS Specification:** The classical Liu and Layland real-time task model is originally designed for hard real-time systems where any deadline miss is undesirable. This model does not provide good support for QoS demand of multimedia applications whose timing constraints can be relaxed and therefore a more flexible QoS specification model is needed.

- **QoS Isolation and Deterministic Guarantee:** The ability to manage resources for maximum QoS return is a potential objective of real-time and QoS-guaranteed systems. To be able to collaboratively optimize deliverable QoS among applications with a middle-ware QoS manager, OS must be deterministic such that the requested QoS is always delivered regardless of the behavior of other tasks.

- **Efficient Resource Utilization:** The main goal of real-time scheduling is to achieve high utilization and still guarantee timing constraints and QoS requirements for applications. Therefore, the system should satisfy the resource demand just enough to provide guarantees with high system utilization.

- **Varying Demand Tolerance:** Since multimedia applications generally have very high fluctuation on demand, the new reservation scheme should modify resource allocations to suit these characteristics.

2.3.1. The Multi-Granularity Reservation Model

Instead of using one ratio as in the classical resource reservation model, the multi-granular reserve specification is given by $\{\{C_i, T_i, D_i\}, \ldots, \{C_x, xT_i\}, \ldots, \{C_y, yT_i\}\}$. The highest granular reserve $\{C_i, T_i, D_i\}$ denotes an instantaneous resource demand of C_i units every recurring time interval T_i before a relative deadline D_i, while the low-granular reserve $\{C_x, xT_i\}$ denotes the average resource demand of C_x units in the longer time interval of xT_i. With multiple granularities, the scheme gives flexibility to users to specify its average resource requirement in the long run and simultaneously obtain guarantees for bursty requests as long as the average resource consumption over a longer time frame is maintained. Conceptually, the scheme behaves like cascading water tanks. The inflow resource refills the next tank with size, e.g. C_y units, at the rate of a low-granular reserve, e.g. C_y units of resource every yT_i units of time. At the last tank, the outflow rate varies upon the user demand and the availability of the resource inside, and is limited to the rate of the highest-granular reserve $\{C_i, T_i, D_i\}$. Figure 10 depicts the concept.

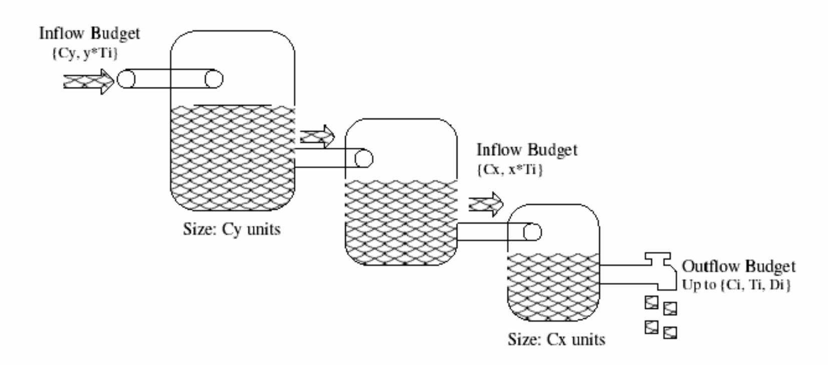


Figure 10: Cascading Water Tanks Concept in Multi-granularity Reserve

With this cascading water tank concept, multi-granularity reservation addresses all the designing goals we previously mentioned. Users can flexibly control the budget in fine-grained fashion, enable the precise resource allocation and therefore achieve efficient system utilization. The tank sizes police resource consumption not to exceed the budget establishing QoS isolation. The resource containment in the last tank supplies the inter-period mutual budget. In other words, it automatically saves underused resources for future use providing varying demand tolerance.

2.3.2. Multi-Granular Resource Replenishment and Enforcement

A *k*-level multi-granularity reserve maintains *k* steps of resource replenishment. The replenishment can be divided into three case scenarios. First, when the available immediate lower-granular budget is larger than C_x, the maximum allowance of C_x will be given. Second, when the available immediate lower-granular budget is smaller than C_x, that total amount will be transferred. Third, when there is no available budget, no refill happens. To ensure QoS isolation, an enforcement mechanism must ensure that no resource consumption in each interval will exceed the budget of any reserve granularity. In each period T_i, an associated task is allowed to access the resource up to its maximum allowance of the highest-granular reserve, which is given by $\max(C_i, c_j)$ where c_j represents the existing budget of the immediate lower granular reserve $\{C_j, T_j\}$.

We now describe the implementation of replenishment and enforcement algorithms. For replenishment, *k* timers will be generated. Each has a period corresponding to its granular reserve period, $T_i, \ldots, kT_i$ respectively. At each granular timer interval (e.g., jT_i), a corresponding reserve budget (e.g., C_j) will be refilled. If its budget is previously depleted, all higher granular reserves will be triggered to be resynchronized with its replenishment timer. For enforcement, when an associated task is executed, the amount of the granted resources is given by $\min(c_i, \ldots, c_k)$. If the task executes more than the granted amount, the task's priority is downgraded to the same level of other non-real-time tasks.

2.3.3. Schedulability Analysis

Our schedulability analysis follows an approach similar to that of Liu and Layland [13] and Mok and Chen [15]. However, the multi-granularity reservation model has some subtle assumptions and constraints. We assume the use of the deadline-monotonic (DM) scheduling policy and the existence of *n* multi-granular reserves, $R_1, R_2, \ldots, R_n$, respectively. Each reserve R_i is assumed to have g_i levels of granularity where $g_i \in I^+$ and its reserve specification is given by

$$\{\{C_i, T_i, D_i\}, \ldots, \{C_x, \varepsilon_x T_i\}, \ldots, \{C_{g_i}, \varepsilon_{g_i} T_i\}\}, \tag{15}$$

where ε_x denotes the ratio of the x^th granular period with the highest granular one. By convention, we assume that $D_1 \le D_2 \le \ldots \le D_n$. One task is associated with each reserve and denoted as $\tau_1, \tau_2, \ldots, \tau_n$ respectively. Note that a traditional reserve can be simply included in the analysis by considering it as a *1*-level multi-granularity reserve. To obtain the analytical results, we assume that tasks are independent, periodic and always ready to run without blocking. We also assume that the reserved amount of low-granular resource is always larger than that of high granularity.

2.3.3.1. Critical Instant of Multi-Granularity Reserve

There are two factors for a task associated with a multi-granularity reserve to miss its deadline: (1) the preemption of higher priority reserves and (2) its own insufficient budget. Assuming a task does not violate its resource quota constraints, one main goal of multi-granularity reserve is to provide the task its promised resource regardless of the behavior of other reserves and non-real-time tasks.

Under the multi-granularity reservation model, a critical instant for a task occurs whenever the task's request arrives simultaneously with requests from all higher-priority tasks, they request resources at their maximum allowances, and their first preempting requests in the critical zone are synchronous with their lowest-granular periods.

2.3.3.2. The Worst-case Response Time Test

We now demonstrate the worst-case response time test by an example and generalize the worst-case response time test for *n* tasks at the end of this section. Consider a set of two tasks, τ_1 and τ_2, with reserves *{{3, 5, 5}, {7, 20}, {13, 50}}* and *{{40, 80, 80}, {60, 160}}* respectively.

Figure 11 illustrates τ_2's critical zone as defined in Section 2.3.3.1. The preemption pattern of τ_1 during time interval [0, t] is shaped in the ascending order of its multi-granular level of enforcement. Consider the case when t = 56. Due to its lowest granular enforcement, τ_1's execution during time interval [0, 56] cannot be larger than twice of its lowest granular budget, which is given by

$$(\left\lfloor \frac{56}{50} \right\rfloor + 1) \cdot 13 = 26. \tag{16}$$

While τ_1 will obtain the resource for 13 time units for the first interval of 50 time units, in the second interval its execution may be enforced by other higher granular enforcements. The next higher granular enforcement limits the execution in its second interval to

$$(\left\lfloor \frac{56-50}{20} \right\rfloor + 1) \cdot 7 = 7. \tag{17}$$

However, the highest granular enforcement ensures that the task must not obtain resources more than 3 units each during time [50, 55] and [55, 60]. In addition, during time interval [55, 56], the maximum demand τ_1 can impose is 1 time unit. Consequently, the maximum time τ_1 can preempt τ_2 during time interval [0, 56] is 17, which is given by

$$\left\lfloor \frac{56}{50} \right\rfloor \cdot 13 + \left\lfloor \frac{56-50}{20} \right\rfloor \cdot 7 + \left\lfloor \frac{56-50-0}{5} \right\rfloor \cdot 3 + \min(5, 56-50-0-5). \tag{18}$$

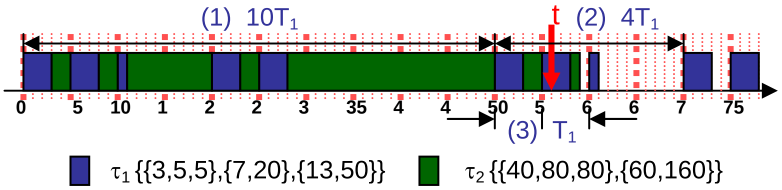


Figure 11: Multi-RSV Critical Zone

The first three terms of Expression (18) basically represent the resource amount that τ_1 has been fully guaranteed by multi-granular periods, labeled as (1), (2) and (3) in Figure 11 respectively. The last term represents the maximum imposed demand for the partially guaranteed resources.

We can generalize the computation of a high priority task τ_i's preemption time to a low priority task during timer interval [t_0, t_0+t] assuming that both tasks' requests arrive at time t_0. Let

$$m_i = \left\lfloor \frac{t - m_{i+1}}{\varepsilon_i T_i} \right\rfloor \cdot \varepsilon_i T_i \qquad (19)$$

and

$$m_{g_i+1} = 0\,. \qquad (20)$$

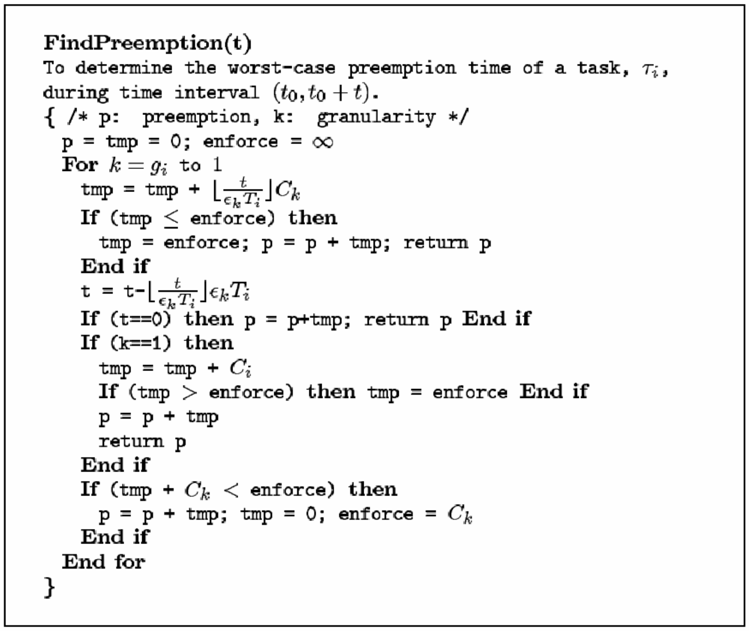
The preemption, denoted as $P^{(t_0,t_0+t)}$, will be shaped by multi-granular enforcements, L_1 to L_{g_i}, as follows:

$$\begin{aligned}
L_{g_i} : \P^{(t_0,t_0+t)} &\geq \lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i}\\
\P^{(t_0,t_0+t)} &\leq MIN(\lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + (t - m_{g_i}), \lceil\frac{t}{\epsilon_{g_i}T_i}\rceil C_{g_i})\\
L_{g_i-1} : \P^{(t_0,t_0+t)} &\geq \lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1}\\
\P^{(t_0,t_0+t)} &\leq MIN(\lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1} + (t - m_{g_i} - m_{g_i-1}),\\
&\quad \lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lceil\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rceil C_{g_i-1}, \lceil\frac{t}{\epsilon_{g_i}T_i}\rceil C_{g_i})\\
&\quad \vdots\\
L_1 : \P^{(t_0,t_0+t)} &\geq \lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1} + \ldots + \lfloor\frac{t - m_{g_i} - \ldots - m_2}{\epsilon_1 T_i}\rfloor C_1 +\\
&\quad (t - m_{g_i} \ldots - m_1)\\
\P^{(t_0,t_0+t)} &\leq MIN(\lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1} + \ldots + \lfloor\frac{t - m_{g_i} - \ldots - m_2}{\epsilon_1 T_i}\rfloor C_1 +\\
&\quad (t - m_{g_i} \ldots - m_1),\\
&\quad \lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1} + \ldots + \lceil\frac{t - m_{g_i} - \ldots - m_2}{\epsilon_1 T_i}\rceil C_1,\\
&\quad \ldots, \lceil\frac{t}{\epsilon_{g_i}T_i}\rceil C_{g_i})
\end{aligned} \qquad (21)$$

Note that at the highest granular enforcement, the right hand term of the first inequality for L_1 granularity is the same as the first term in MIN () operation of second inequality. Therefore,

$$\begin{aligned}
\P^{(t_0,t_0+t)} = {} & MIN(\lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1} + \ldots + \lfloor\frac{t - (\sum_{j=2}^{g_i} m_j)}{\epsilon_1 T_i}\rfloor C_1 + (t - (\sum_{j=1}^{g_i} m_j)),\\
& \lfloor\frac{t}{\epsilon_{g_i}T_i}\rfloor C_{g_i} + \lfloor\frac{t - m_{g_i}}{\epsilon_{g_i-1}T_i}\rfloor C_{g_i-1} + \ldots + \lceil\frac{t - (\sum_{j=2}^{g_i} m_j)}{\epsilon_1 T_i}\rceil C_1, \ldots, \lceil\frac{t}{\epsilon_{g_i}T_i}\rceil C_{g_i})\,.
\end{aligned} \qquad (22)$$

Figure 12 shows the polynomial algorithm to determine the preemption time by a multi-granularity reserve. The worst-case response time of a task thus is the summation of its required computation time and possible preemption from all higher priority tasks. This leads to Theorem 1.



```
FindPreemption(t)
To determine the worst-case preemption time of a task, $\tau_i$,
during time interval $(t_0, t_0 + t)$.
{ /* p:  preemption, k:  granularity */
  p = tmp = 0; enforce = $\infty$
  For $k = g_i$ to 1
    tmp = tmp + $\lfloor \frac{t}{\epsilon_k T_i} \rfloor C_k$
    If (tmp $\leq$ enforce) then
      tmp = enforce; p = p + tmp; return p
    End if
    t = t-$\lfloor \frac{t}{\epsilon_k T_i} \rfloor \epsilon_k T_i$
    If (t==0) then p = p+tmp; return p End if
    If (k==1) then
      tmp = tmp + $C_i$
      If (tmp $>$ enforce) then tmp = enforce End if
      p = p + tmp
      return p
    End if
    If (tmp + $C_k$ $<$ enforce) then
      p = p + tmp; tmp = 0; enforce = $C_k$
    End if
  End for
}
```

Figure 12: Multi-Granularity Preemption Computation Algorithm

Theorem 1

For a multi-granularity resource reservation system, the worst-case response time for a task τ_i is the smallest solution to the following equation:

$$\omega^{k+1} = C_i + \sum_{j<i} P_j^{(0,\omega^k)}, \tag{23}$$

where $P_j^{(0,\omega^k)}$ can be determined by the equation (1) or the algorithm listed in Figure 12. This formula must be solved recursively starting with $\omega^0 = C_i$ and finishing when $\omega^{k+1} = \omega^k$ on success or when $\omega^{k+1} > D_i$ on failure.

2.3.3.3. The Utilization Bound Test

We now derive a simple utilization bound test for a multi-granularity reservation. Consider two tasks in the system, τ_i and τ_j using the same notation previously described. The preemption rate seen by τ_j varies with its period T_j and is given by $\frac{C_x}{\varepsilon_x T_i}$ where $x = MAX(k \in I^+ \mid (1 \leq k \leq g_i) \cap (\varepsilon_k T_i \leq T_j))$. We denote this effective granularity as y_{ij}. Therefore, considering its possible preemptions from all higher-priority tasks, the schedulability of a task, τ_j, can be determined as follows:

$$\sum_{i|D_i \leq D_j} \frac{C_{y_{ij}}}{\varepsilon_{y_{ij}} T_i} \leq n(2^{1/n} - 1). \tag{24}$$

where *n* is the number of tasks in consideration. Note that to determine the schedulability of the task set, the utilization bound test of all tasks must be performed.

2.3.4. Multi-granularity Reservation Deployment

Multi-granularity reservations are specifically designed for soft real-time guarantees where requests can miss deadline occasionally. Specifically, MPEG-4 streams [16] use frames are encoded into three types: *I*, *P* and *B*-frames. An *I*-frame is encoded as a single image with no reference to any frames and typically has larger size than other frames. A *P*-frame is encoded relative to the past reference *P* or *I*-frame. A *B*-frame is encoded relative to the past reference frame, the next future frame, or both frames which again are the closest *I* or *P* frames. Each video sequence is composed of a series of Groups of Pictures (GOP). A GOP is an independently decode-able unit that can be of any size as long as it begins with an *I*-frame.

One effective and simple example of a multi-granularity reserve for an MPEG-4 video decoder is reserving its maximum decoding time in frame period granularity and its average decoding time per frame for each GOP period as given by

$$R_{mpeg4} = \{\{max\_decoding, T\}, \{avg\_decoding * GOP\_size, GOP\_size * T\}\} \quad (25)$$

Since a multi-granular reserve always delivers resources as long as the budget is available, the above reserve always guarantees the successful decoding of *I* frame for all GOPs and any GOP that requires smaller decoding time than *(avg_decoding * GOP_size)* is also guaranteed.

Note that a more stringent and relaxed QoS specification and even probabilistic guarantees can be easily obtained by detailed profiling of the video stream.

2.3.4.1. Multi-granularity Reserve vs. *(m, k)*-Firm Guarantees

All *(m, k)*-firm guarantee schemes mark *m* out of *k* consecutive requests as mandatory requests. Only these requests are considered as preemption to low priority tasks assuming the worst-case demand. Some *(m, k)* schemes (e.g.[17]) subtly assign mandatory requests to reduce interference among tasks. However, these schemes need the knowledge of phasing among applications which is somewhat difficult in practice.

Our scheme does not assume such knowledge and enables budget to be shared across periods. Consider an MPEG-4 multi-granular reserve as given in Equation (25). If we assume that all requests require the maximum demand, at least *mrsv_guarantee = (avg_decoding*GOP_size)/max_decoding* frames are guaranteed. In the performance evaluation, we will compare this reserve with *(mrsv_guarantee, GOP_size)*-firm guarantee. We will show that our scheme always performs better than *(m, k)*-firm guarantee with the same amount of allocated resource. Note that a Pfair schedule [18] of multi-granularity reserve can also be modeled by adding one more middle-granular reserve to equally distribute budget its lower-granular period.

2.3.4.2. Multi-granularity Reserve vs. Generalized Multiframe Model

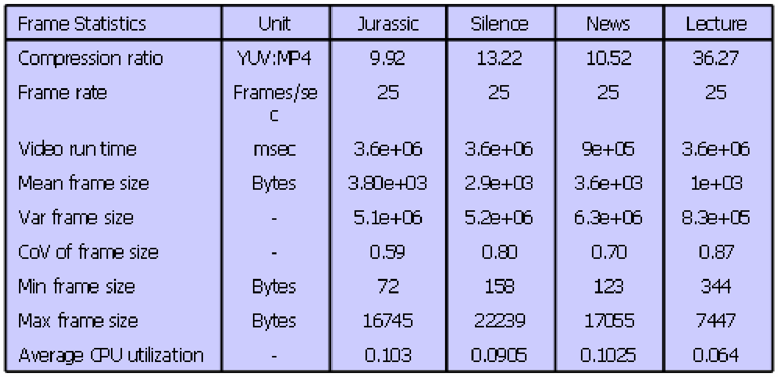
A multi-granularity reserve can be considered as a special case of generalized multiframe model. Any multi-granularity reserve can be conservatively converted into a generalized multiframe task model. Fundamentally, the worst-case execution time patterns of consecutive frames will be conservatively assumed by our worst-case response-time test. For instance, a reserve given by *{{3, 5, 5}, {7, 25}}* can be converted to *< (3, 6, 7, 7, 7), 5>*. Note that our model also allows tasks to have deadlines different from their periods.

2.3.4.3. Multi-granularity Reserve vs. Constant Bandwidth Server (CBS)

Instead of assuming fixed-priority scheduling policies, CBS uses earliest deadline first (EDF) scheduling. Our scheme shares the same concept with CBS in the sense that it is tolerant to varying instantaneous demand yet in the long run the multimedia workload will be shaped into its average-case specification. Through the dynamic adjustment of requests' deadlines, CBS handles high-demand requests as the second-priority class in EDF fashion starving all other non-real-time tasks. Moreover, its guarantee for the multimedia stream is not deterministic. In other words, a highly dynamic real-time workload can cause an unacceptably high failure rate. On the contrary, in the same scenario, multi-granularity reservations will be able to detect this failure rate and reject the guarantee or negotiate the QoS. Such predictability property is crucial for real-time and QoS-guaranteed systems.

2.3.5. Performance Evaluation

We evaluate the performance of multi-granularity reservation scheme with average-case reservation, *(m, k)*-firm guarantee and CBS, which we will call in short MULTI-RSV, AVG-RSV, MK-RSV and CBS respectively. A series of simulations is performed to study QoS and temporal isolation, the response time of non-real-time tasks and the system utilization among those schemes. Four MPEG-4 video traces from [19]: *Jurassic Park*, *Silence of the Lamb*, *News* and *Lectures Room Cam* are chosen to represent four different kind of movies. The trace lengths are *60, 60, 15* and *60* minutes respectively. All video streams are encoded using the GOP pattern given by *IBBPBBPBBPBB* while data in the video streams typically occur as *IPBBPBPBBPBB*. Table 2 summarizes the frame statistics.



Frame Statistics	Unit	Jurassic	Silence	News	Lecture
Compression ratio	YUV:MP4	9.92	13.22	10.52	36.27
Frame rate	Frames/sec	25	25	25	25
Video run time	msec	3.6e+06	3.6e+06	9e+05	3.6e+06
Mean frame size	Bytes	3.80e+03	2.9e+03	3.6e+03	1e+03
Var frame size	-	5.1e+06	5.2e+06	6.3e+06	8.3e+05
CoV of frame size	-	0.59	0.80	0.70	0.87
Min frame size	Bytes	72	158	123	344
Max frame size	Bytes	16745	22239	17055	7447
Average CPU utilization	-	0.103	0.0905	0.1025	0.064

Table 2: Video Frame Statistics

Unless explicitly stated, five real-time tasks are randomly generated to achieve the given total real-time utilization. Each task has a uniform probability of having a short (*1-10 ms*), medium *(10-100 ms)* or long *(100-1000 ms)* period. Task period is uniformly distributed in each range. Five non-real-time tasks are randomly generated in the same fashion but without reservations. We estimate the video decoding time of each frame using the linear relationship with its corresponding frame size as suggested in [20]. Four parameters will be measured as follows:

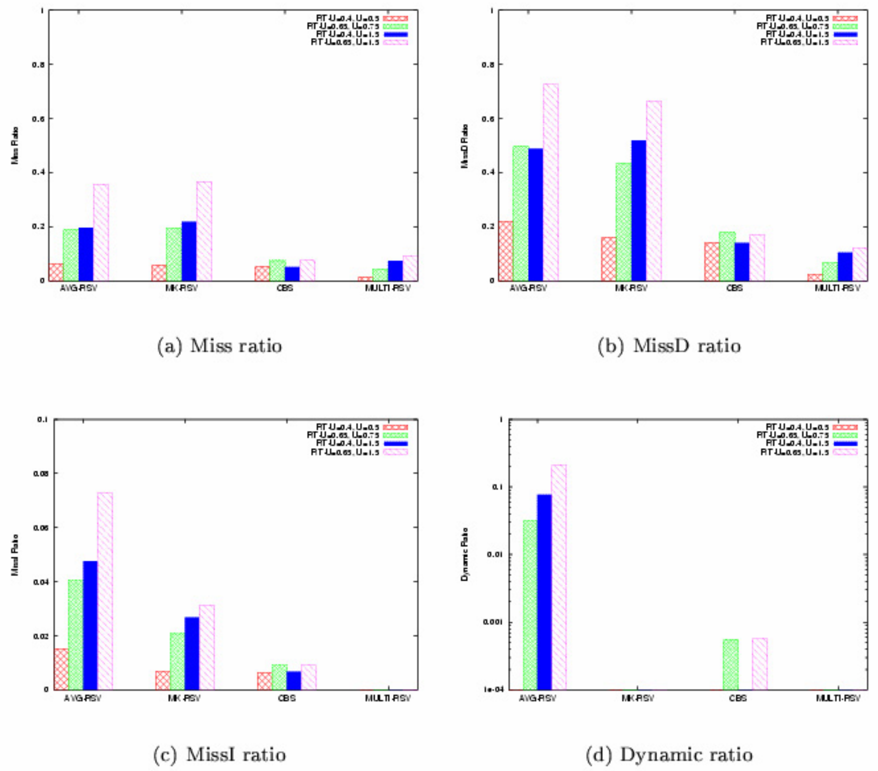
- **Miss:** the ratio of the number of deadline-missing frames to the total number of frames.
- **MissI**: the ratio of the number of deadline-missing *I*-frames to the total number of frames (of all types).
- **MissD:** the ratio of the number of undecodeable frames (due to a deadline miss or the failure of its reference frames) to the total number of frames.
- **Dynamic:** the ratio of dynamic errors [21][4] to the total number of possible *k*-consecutive frame sets.

2.3.5.1. Single Stream Evaluation

We first evaluate the performance of one single video stream competing with real-time and non-real-time workload using four different schemes. We believe that this scenario is most likely to happen in most handheld devices and personal computers. We create a multi-granularity reserve and a (m, k)-firm guarantee as described in Subsection 2.3.4. For CBS and average-case reserve, we allocate resources based on its average demand. Note that the missing video requests are immediately dropped at their deadlines for all schemes in order to avoid a queueing delay effect.

Figure 13 shows the performance of one Jurassic movie at four different real-time and total system utilization scenarios: low workload (RT-U=0.4, U=0.5), low workload high background (RT-U = 0.4, U = 1.5), high workload (RT-U = 0.65, U = 0.75) and full system load (RT-U = 0.65, U = 1.5). RT-U and U denotes the real-time task utilization and the total system utilization, both numbers exclude the video stream utilization. For clearer view, we zoom in the missI ratio axes in Figure 13(c) and plot the dynamic ratio axes shown in Figure 13(d) using logarithmic scale. As expected, AVG-RSV and MK-RSV perform poorly at high system utilization. At high workload and full system load, more than 35 persent of frames miss the deadline and more than 65 percent of frames are undecodeable under both schemes. MK-RSV delivers more *I*-frames for 4 percent due to the worst-case demand assumption on resource allocation and therefore achieves a little bit better *MissD* ratio. MULTI-RSV and MK-RSV satisfy the (m, k) constraint as expected. Even though our MULTI-RSV allocates resources for approximately the same amount as MK-RSV, it always achieve lower failure rates and yields better system utilization, ~4-30 percent and ~10-57 percent less *Miss* and *MissD* ratios respectively. Unlike other schemes, the performance of CBS does not suffer from the presence of non-real-time tasks. This is due to its preference of high-demand workload over non-real-time tasks. Note that MULTI-RSV has no missing *I*-frames at all, which results in a better *MissD* ratio than CBS.

[4] Dynamic error is defined by Hamdaoui et al. as the failure of a system to satisfy timing constraints of at least *m* frames out of any *k* consecutive frames.



(a) Miss ratio

(b) MissD ratio

(c) MissI ratio

(d) Dynamic ratio

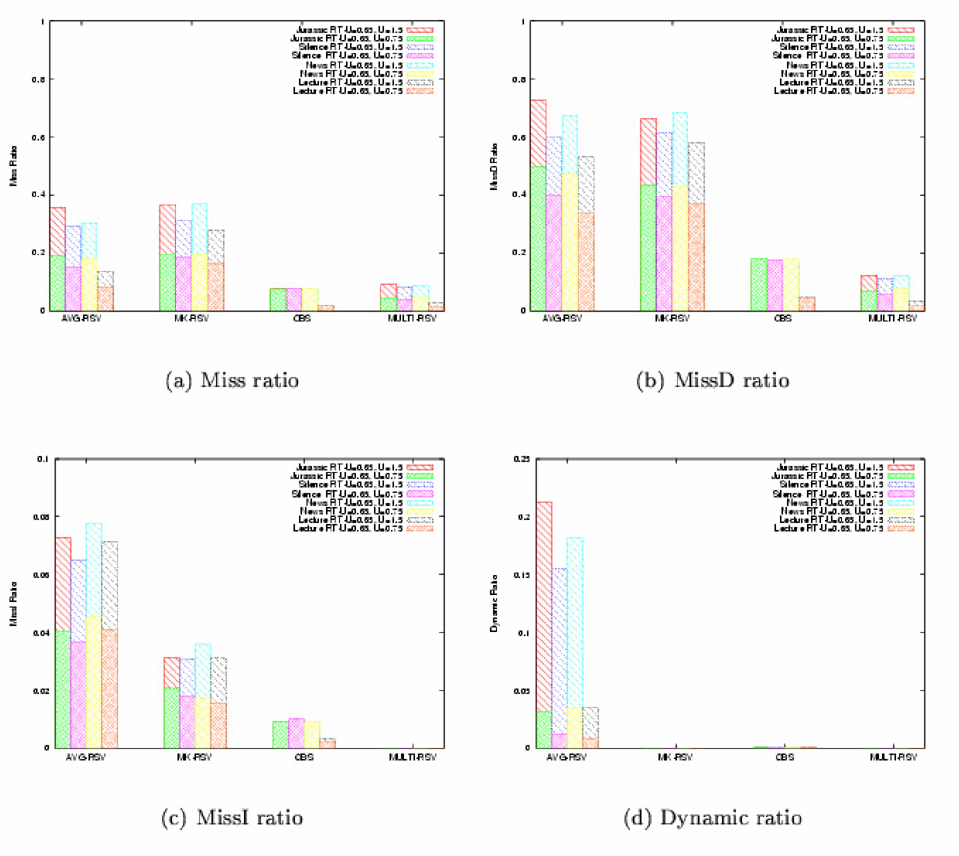
Figure 13: Miss Ratio versus Utilization for Jurassic Stream

We repeat the same experiment with the other three video streams. Figure 14 shows the results. For clearer view, we zoom in the missI and dynamic ratio axes to 0.1 and 0.25 respectively. Overall, *Lecture* obtains the better performance than others due to its much smaller variance in frame size. However, since most of its *I*-frames represent the group of the largest frame size and are likely to miss deadlines, any loss in the major reference frames results in bad *MissD* ratios of the *Lecture* stream. Again, note that the performance under CBS is independent from the existence of the non-real-time workload and therefore the graphs respected to low and high background workload are completely the same.

Note that even though the buffering technique can relax the timing constraints and reduce the variation of resource demands, due to the long-range-dependence property of MPEG4 streams a very large buffer resulting in long delay is required to effectively smooth the burst. Therefore, in practice the combination of buffering with multi-granularity reserve is necessary to compromise delay with high system utilization.

2.3.5.2. Multiplexing Streams Evaluation

We now simultaneously run three streams, *Jurassic, News* and *Lecture* with RT-U = 0.35 and U = 1.5. This utilization is chosen to fully utilize the system reserve capacity. Figure 15 shows the performance of algorithms on multiplexing video streams. Comparing both experiments, multiplexing video streams yields better performance overall, especially the reduction of the *MissI* ratio in MK-RSV. This is because of two main reasons. First, there is typically a low-probability that video streams that are statistically independent of each other will request resources at the peak rate simultaneously. Second, the worst-case resource allocation per frame used by MK-RSV more properly manages priorities for a complete *I*-frame execution. Again, *Lecture* obtains better performance than other streams due to its smaller fluctuation in demand. MULTI-RSV successfully delivers all *I*-frames and achieves a low *missD* ratio. In addition, it maintains (m, k) constraints. In summary, this experiment shows that despite the benefit of the video multiplexing in a server where multiple streams can share resources, the efficient deterministic guarantees is nevertheless crucial for protecting unpredictable high failure rate during overload. Note that we plot the *missI* and *dynamic* ratio axes using logarithmic scale.



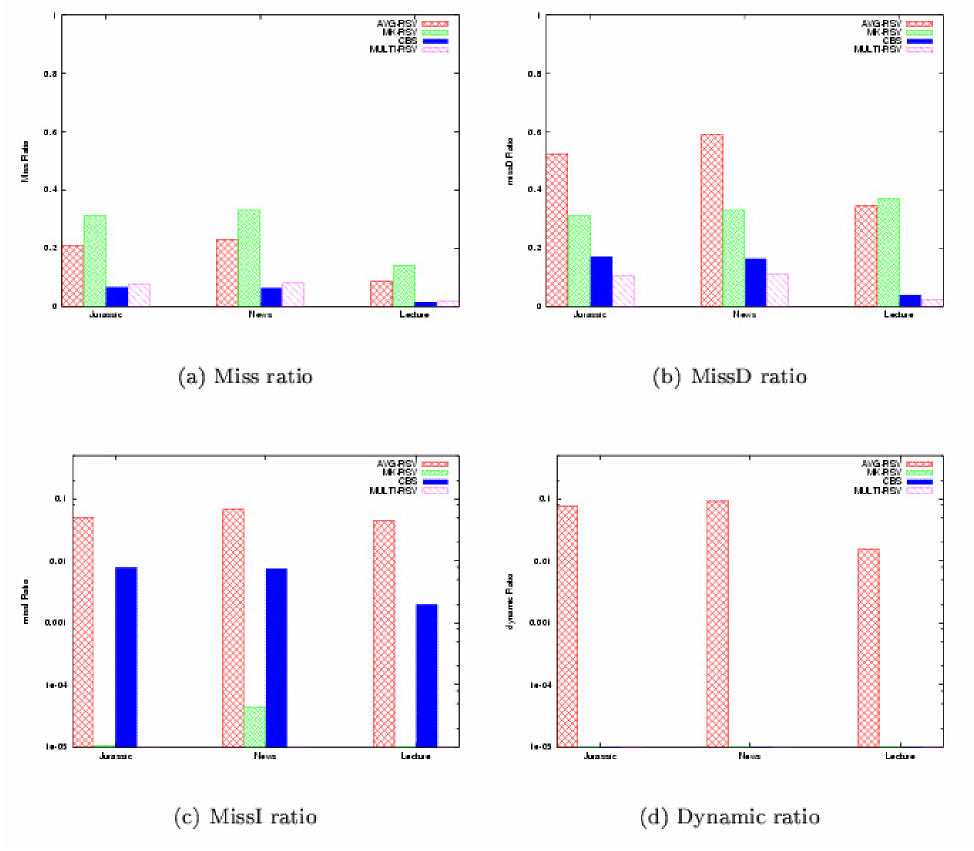
(a) Miss ratio

(b) MissD ratio

(c) MissI ratio

(d) Dynamic ratio

Figure 14: Performance Comparison on Different Video Streams



(a) Miss ratio

(b) MissD ratio

(c) MissI ratio

(d) Dynamic ratio

Figure 15: Performance Comparison on Multiplexing Video Streams

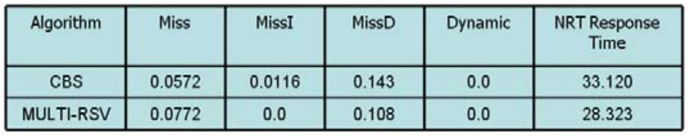
2.3.6. System Response Time Comparison

In this section, we study the effect of CBS and MULTI-RSV algorithms on the system response time. A random number of short non-real-time requests, 0 to 3 requests, are generated every *40 ms* to compete for CPU resources with one video stream *Jurassic*. The sizes of the requests are uniformly distributed between *10* and *20 ms.* The average CPU utilization of the video stream is about 0.11. A set of real-time tasks with total utilization of 0.5 is generated in the same fashion as in the first experiment. We measure the normalized response time defined as

normalized_response_time = completion_time/required_decoding_time

As can be seen in Table 3, even though both schemes deliver comparable performance (5 percent difference) on multimedia streams, CBS delays the non-real-time responses by approximately 17 percent.

Table 3: CBS and MULTI-RSV Comparison



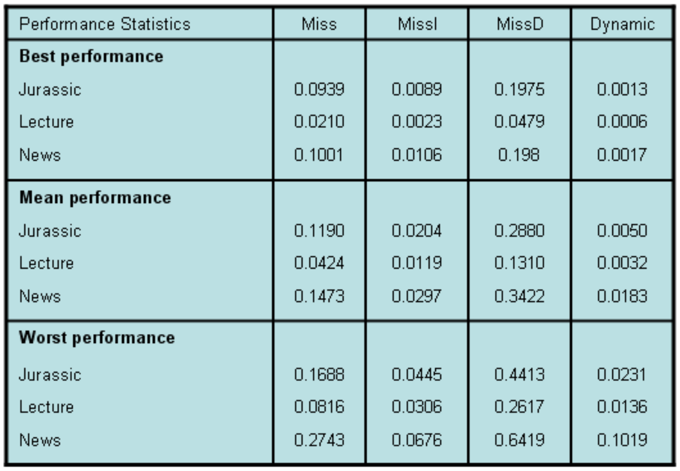
Algorithm	Miss	MissI	MissD	Dynamic	NRT Response Time
CBS	0.0572	0.0116	0.143	0.0	33.120
MULTI-RSV	0.0772	0.0	0.108	0.0	28.323

2.3.5.3. Performance Predictability

Unlike CBS, multi-granularity reservation trades some system capacity against the provision of a (m, k)-firm QoS guarantee. In this section, we confirm the necessity of predictability in scheduling policies. We experiment with multiple video streams using CBS. A real-time task set again is randomly generated as in previous experiments with the highest utilization allowed by CBS. None of non-real-time task is created since it does not affect CBS performance as previously discussed.

As can be seen in Table 4, the performance of video streams uncontrollably depends on competing real-time tasksets despite of the same utilization. The worst-case performance[5] may even have 27% of frames missing with 10% of dynamic errors. Such unpredictability is generally unacceptable in many QoS-guaranteed systems.

Table 4: CBS Performance at highest allowance load



Performance Statistics	Miss	MissI	MissD	Dynamic
Best performance				
Jurassic	0.0939	0.0089	0.1975	0.0013
Lecture	0.0210	0.0023	0.0479	0.0006
News	0.1001	0.0106	0.198	0.0017
Mean performance				
Jurassic	0.1190	0.0204	0.2880	0.0050
Lecture	0.0424	0.0119	0.1310	0.0032
News	0.1473	0.0297	0.3422	0.0183
Worst performance				
Jurassic	0.1688	0.0445	0.4413	0.0231
Lecture	0.0816	0.0306	0.2617	0.0136
News	0.2743	0.0676	0.6419	0.1019

[5] We define this as the highest average ratio among all performance matrices.

3. Memory Reservation

Multitasking operating systems manage the sharing of multiple system resources including CPU cycles, physical memory, disk space, disk bandwidth and network bandwidth. Applications such as multimedia tasks that require timely response must be given predictable access to these system resources. In recent years, significant research has been conducted on the use of time-multiplexed resources including processor cycles, disk bandwidth and network bandwidth. However, to our knowledge, work on spatial resources (and memory usage) particularly in the context of demand paging has not been studied at length. In a multitasking environment, background and other non-real-time applications can steal memory pages forcing the working set of a high importance multimedia application to be swapped out. Consequently, when the multimedia application resumes, it might be forced to pay a timing penalty for swapping pages back in, and encounter jitter and even unacceptable delays. Operating system support is required for ensuring that tasks with QoS and predictability requirements get guaranteed access to time-shared and spatial resources. We also expect that such schemes will also gain more importance as consumer devices become smaller and more cost-sensitive.

In this project, we propose a *memory reservation* abstraction for application tasks. Specifically, memory reservations can significantly enhance predictability and hence satisfy QoS needs. Variants of the same mechanism can be utilized as different types of sand-boxes to enforce limits on memory usage.

Limiting the energy consumption in mobile/embedded systems such as laptops, PDA's and cell phones is becoming increasingly important. Similarly, power-aware mechanisms are increasingly critical in the context of high-end server systems where heat levels and cooling requirements are becoming important considerations. Current hardware technology allows various system components to operate at different power levels and corresponding performance levels. We have implemented energy-aware extensions to our memory reservation implementation in the Linux operating system, wherein memory reserves can be bank-aligned to RAMBUS banks and can be associated with particular power-states at particular times, thus maximizing power savings while minimizing performance degradation. The banks used by the reservation can then be put into a lower-energy state when the application is not in use. In order to reduce page-scattering effects, we pack reservations into the minimum number of banks possible without affecting performance.

3.1. Shortcomings with Current Memory Systems

We illustrate the problems with current memory architectures with a simple example. Consider sorter (a sorting routine that is representative of many database applications) that is running in a 16 MB memory system in parallel with a malicious adversary swapper, which declares a large array and then accesses array elements interleaved by the page frame size. When run in isolation, sorter incurs zero capacity misses. A capacity miss is a request for a page that has already been accessed at some point in the past and has been evicted from memory by the paging policy and hence has to be fetched from disk. As shown in Figure 16, swapper commences at t = 2 s and stops at t = 12 s, stealing pages of the memory. This results in capacity misses and unpredictable page flushout delays for sorter which causes a 200 percent increase in run-time for sorter. Similarly, for a convex optimization algorithm amrmd [30] running in parallel with swapper in a 32 MB memory system, there is an 80 percetn increase in run-time. These problems are discussed in detail in [28]. The problem here is that of the lack of memory isolation in the operating system due to the spatial multiplexing of the shared memory resource. Similar problems also plague the swap space. Our proposed scheme of memory and swap reservations addresses the above problems by isolating the effects of memory management of different applications from each other. Further more, it gives each application control over its own memory availability and usage, thus enhancing timing predictability and performance.

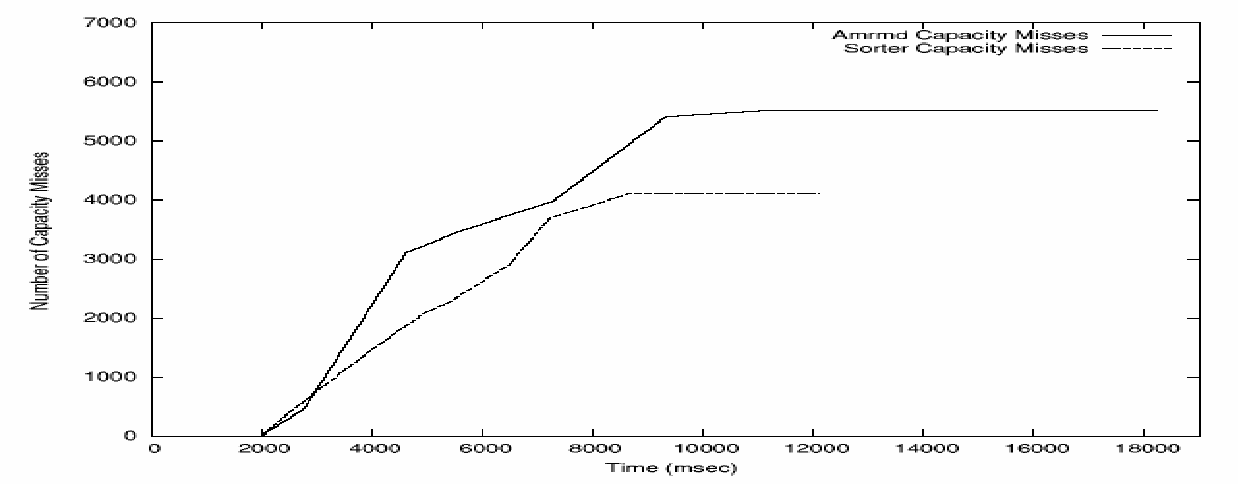


Figure 16: The problems in current memory subsystems

3.1.1. Related Work

Extensive work has been done in the domain of memory management for general-purpose operating systems. While application controlled page-replacement [22,23,24,25] and adaptive page-replacement [26,27] are very well-researched topics, our work focuses on the orthogonal topic of firewalling memory behavior of each task to a separate physical domain. The closest work related to ours in the domain of general-purpose operating systems is [28] by Hand. While they study the impact of firewalling on the performance of normal applications, our focus is on soft real-time tasks. In particular, we allow applications with QoS need to *specify* their memory requirement which is then guaranteed by the OS. Besides, in our case, the paging is implicitly triggered by the operating system rather than by an application handler. Our "self-clocking" paging mechanism focuses on complete predictability with regard to the paging operations.

In the domain of real-time systems, memory pages of (real-time) applications could be wired down in the Mach [32] and RT-Mach operating systems, and therefore could not be paged out causing unpredictable future delays. However, for large applications and resource-constrained systems, such wiring may be very expensive (when possible) and may prevent other applications from running at all. In [29], Nakajima et al. added the ability to mark memory pages as belonging to a particular real-time application, which when used up, generated a trigger to the system. In contrast, each memory reservation in our case is a complete "virtual" memory subsystem which automatically self-pages only within that reserve, thus not affecting other tasks.

While extensive research has been carried out in power-aware management for computing systems, energy-aware memory management is a relatively new area of research. In [34] and [35], combined hardware-software hybrid schemes are proposed to determine when particular banks are turned off using trace-driven experimental set-up. Our scheme leverages information that is exclusive to the OS and we have implemented our changes in the Linux kernel. Energy-aware memory optimizations using compiler techniques has been covered for generic embedded systems in [39,38] and for real-time systems in [37]. These optimizations are complementary to our work, which targets multitasking systems and relies on OS only strategies. In [36], Pillai et al. have implemented power-aware memory management in general-purpose OS scenarios without reservation support and have validated the phenomenon of "page-scattering". While their work does not characterize performance effects due to the energy optimizations, our scheme saves energy with *zero* performance hit. Besides, since we leverage the concept of reservations to ensure that applications are packed into a minimum number of RDRAM banks, we minimize "page-scattering" for long running applications, thus saving significantly more energy over long periods of time. While our work assumes all active banks to be turned on with *zero performance degradation*, we could use a control framework and algorithms such as that proposed in [40] for more aggressive power-optimizations, albeit at the cost of a bounded performance hit.

Finally, our work augments the *Resource Kernel* (RK) paradigm [33], which provides a unified abstraction for resource reservation, to accommodate memory which is a spatially multiplexed (as compared to time-multiplexed) resource.

3.2. Memory Reservation Architecture

3.2.1. Design Objectives

Our design goals for the new memory subsystem architecture are as follows:

- Timeliness and Predictability: An application must be able to specify its memory (resource) demands from a resource kernel. Once granted, the requested number of memory page frames must be exclusively available to it. Failure to honor a request will be used by user-level managers for appropriate recovery action.
- Enforcement and Protection: The operating system must ensure that potential abuse/misuse of system resources by any application does not affect the performance of other well-behaved applications.
- Optimal memory utilization: The admission control mechanism should ensure that the available system memory is well-used. The system should also ensure that the apportioning of memory among applications is such that precious memory pages are allocated where their need is most.
- Customizable Page Update Policy for application: Applications may exhibit special access patterns that the page-replacement policy would like to exploit. For example, the Least Recently Used (LRU) policy is known to perform extremely poorly for applications with predictable access patterns. Besides, for a given memory policy, the various parameters that affect the policy can be tuned to suit the application profile best w.r.t predictability and performance.
- Compatibility with RK Abstractions: The *Resource Kernel* (RK) paradigm [33] provides a unified abstraction for managing multiple resources such as CPU, bandwidth and disk subsystem. We would like the abstractions for memory management to be compatible with that of other (time-multiplexed) resources in RK.

The resource parameter that the application specifies in the memory reservation model is the size of the memory required by the application in bytes. This, in turn, is translated to the number of pages depending on the page size supported by the memory architecture. The memory subsystem may internally use efficient management structures such as the buddy system, and therefore the number of pages reserved may be higher (but never smaller) than the requested size.

When the set of free pages of a reservation is used up completely by the application, the memory reservation is said to have “filled up”. We associate two types of policies with a memory reservation that is currently filled up.

Hard Reservation: In a hard reservation, a process bound to it is *not* allowed to use pages beyond what is available in the memory reservation. This is true even if unused/unreserved physical memory is abundantly available. In case the reserve is filled up, the process starts swapping pages within its own reserve to satisfy subsequent requests. This provides a powerful sand-boxing mechanism for enforcing the memory usage of untrusted and potentially malicious applications, or even for the use of applications which must exhibit deterministic and predictable behavior every time they are run, while not compromising on performance optimizations such as page sharing of dynamic linked libraries. Strictly speaking, even an application that uses a hard memory reserve will not have the exact memory behavior when it is re-run because the disk buffering system is shared with other tasks. However, these variations will still be significantly lower than without the use of a hard memory reserve.

Firm Reservation: In a firm reservation, when a reservation is filled up, a process bound to the reservation is allowed to use pages from the unreserved freelist. However, when the unreserved pages are needed by others (unreserved or soft reserved applications), these “borrowed” pages have to be returned. Applications such as multimedia tasks that require a lower bound on performance but could use improved performance should use firm reservations.

An application can reserve the amount of swap space that it requires. It can therefore not be affected by the abuse of swap resources by contending threads. We treat swap reservations only as hard reservations, and do not support firm reservations.

Multiple application threads/processes can be bound to a single reservation. This is consistent with other resource kernel [33] abstractions for time-multiplexed resources, and allows one or more applications to “reside” within a virtual machine that comprises a fraction of the resources of the underlying physical machine.

Memory reservation provides the mechanisms for memory partitioning. This abstraction of resource partitioning is also extensible in a hierarchical fashion. That is, an application can create a "child" reservation (recursively) carved from its parent memory reserve and can specify appropriate policies for its use. For example, this type of functionality could be targeted at large multimedia applications which are multi-threaded, where we would like to apportion the total memory reservation in a controlled predictable manner among the various children threads.

3.3. Customizing Memory Management

Applications such as multimedia streaming players or data compression tools always tend to access new data blocks. We call such applications *capacity-insensitive* applications or *look-forward* applications because typically they never look back very far into the past to access old data or code. In contrast, applications like databases that implement mergesort algorithms regularly process previously used data. They may also transition to different phases of execution, where new code pages will be brought in, but earlier code/data may be revisited for aggregation purposes. We call such applications capacity-miss-sensitive applications or look-back applications.

It is natural to assume that given the QoS support added to real-time operating systems in the recent past, memory reservations are useful for real-time and multimedia applications. In order to further enhance the predictability and performance of real-time tasks, we next introduce a novel *self-clocking* scheme for triggering memory management. This harnesses the isolation of memory usage of each application in order to provide deterministic paging characteristics for each application.

3.3.1. Application Behavior, Self-Clocking and Memory Policies

The knobs that can be used to characterize the memory behavior of applications in a reservation-based scenario are the size of the reservation and the size of the look-back window, which corresponds to the memory pages that were accessed by an application in the past but still retained in the memory subsystem as *active pages*. These two parameters affect the extent to which capacity misses can occur for an application. We try to match the parameters of the page-replacement algorithm with the application profile in order to minimize the number of capacity misses.

Like standard implementations of page-eviction, we associate a parameter called *Age* which is decremented periodically by the operating system unless it was touched during the last “aging interval” in order to maintain a look-back window which includes the working set of a process. In our approach, however, rather than using *wall-clock time* for triggering aging, we use the number of remaining free pages in the reservation as an implicit trigger to conduct the aging operation. This results in significantly reduced number of capacity misses since in-use pages are evicted only upon need. We call this implicit predictable triggering of page-eviction in reservation-based systems *self-clocking,* since page evictions are automatically adjusted based on the application memory needs.

As mentioned, for capacity-miss-sensitive applications, beyond large reservation sizes, the look-back window should be as large as possible, meaning the size of its memory reservation. However, this implies that the number of pages that need to be evicted when there is need for a free page is rather small. When the reservation has depleted its pages, every time a page fault occurs, a small number of pages must have their buffers flushed out to disk before being returned to the freelist. This flushout operation is expensive due to the slow relative speed of the disk. Thus, we need to balance the size of the look-back window against the size of the pages that are evicted when the freelist becomes (nearly) empty.

The above discussion suggests two different degrees of freedom in self-clocking policies:

- **Forced Eviction Policy:** Retain all accessed pages until a new page is needed. At this point, forcibly evict exactly N pages where N is an application-specific parameter. The aging of pages is used to decide which pages are picked for eviction.

- **Aging Only Policy**: Use a look-back window. Pages that are beyond this window of look-back are eligible to be evicted at any time. Window sizes should be chosen such that there is no possibility of a scenario where all pages in the reservation are in the look-back window and there is the need for a new page.

The above two options can also be combined to form a hybrid scheme:

- **Automatic Decay Policy:** Use a look-back window and allow forced evictions under scenarios where there are no free pages. This offers the leeway of choosing larger window sizes since the forced evictions can handle bursts of requests before the next aging interval.

We evaluate these schemes in Section 3.8.

3.4. Energy-aware Reservations

Modern memory architectures such as RAMBUS provide multiple states for power control. There are four power levels defined in the RDRAM specification - Attention, *Standby*, *Nap* and *Powerdown*. Banks can be put into lower power levels by disabling auxiliary sub-components of the memory circuitry. Our techniques incorporate information that is available exclusively to the operating system to significantly augment power savings.

The memory reservation paradigm can naturally be extended to achieve energy-awareness. The basic observation used is simple - all unused banks can be turned to *PowerDown* mode (maximum power savings mode) and hence the energy incurred by these banks being on is saved. The reason that we consider only the *PowerDown* mode is that the bank turn-off decision is coupled with the scheduling decision. Thus when the operating system schedules a particular task, it knows for sure that the banks of all other processes will not be used and hence can safely turn them off to the lowest power state. At any instant of time, only the banks associated with the currently active process are kept on.

Typical programs may, however, use memory-mapped files belonging to other processes, dynamically linked library files etc. This results in a process touching pages outside its reservation banks. This page-scattering effect has been confirmed by [36] in non-reservation-based scenarios and results in sub-optimal energy savings. Our solution to page-scattering is collocation of tasks i.e. we try to minimize the total number of banks used, thereby containing page-scattering effects.

3.5. Memory Reserves Implementation

In this section, we describe our implementation of Linux memory reservation in the Linux 2.4.18 kernel.

3.5.1. Memory Reserves

A physical memory reservation is created upon the invocation of a create() system call as follows. First, an unreserved portion of system memory sufficient to hold the requested size is chosen. The data structures associated with each reservation consists, among other fields, of a set of freelists (one associated with each size of memory chunk as explained in Section 3.1 and a buddy bitmap for handling the merging and splitting of memory chunks. Each page is, in turn, populated onto the reservation freelists at the appropriate level, and is marked as belonging to the reservation. If a page is not free (but dirty), it is flushed to the disk before being added to the freelist. At this stage, the memory reservation has been created. When a process binds to a reservation with another system call, the task control block is attached to the particular reservation, with another pointer from the task to its memory reservation. Multiple processes can attach to the same reservation.

3.5.1.1. Paging Policy: Lessons Learned

There are several subtle but important design issues involved in page eviction. Our initial design was as follows. When the reserve is nearly filled up, the pages in the inactive list need to be freed and reused. As a part of the API set, we supported two thresholds: a low watermark and a high watermark. If the number of free pages in the reservation fell below the low watermark, the reservation started cleaning itself by evicting some aged pages. The API also allowed the specification of a parameter NUMPAGES-EVICTED that denoted how many pages will be evicted, after eviction is triggered. The intent was that this parameter can be chosen depending on the waiting time that the application is willing to tolerate. Once the low watermark is encountered, for every page fault, NUMPAGES-EVICTED pages are cleaned until the high watermark is reached. At that point, eviction is disabled for the reservation.

Later, we found that this scheme is not desirable for two reasons. One, it does not clearly capture the look-back window control that we seek. In this scheme, aging takes place only when memory pressure is detected in the reservation (the low watermark is reached), after which all pages are treated equally. There may be many pages that could be beyond the lookback window of this application that could be reclaimed that are not reclaimed. Two, once memory pressure is detected and eviction is triggered, *every* page fault involves cleaning out pages which requires writing buffers to disk, an expensive operation.

3.5.1.2. Paging Policy: Implementation

Our first implementation made it evident that reservation pages must be aged and evicted not on every page fault, but periodically every few page faults. We therefore designed and implemented the self-clocking scheme described earlier.

For every page fault that the process encounters, pages from the reservation are used to satisfy the page requests and are added to the *Active page* list of the reservation. Aging of the pages in the reservation is carried out after a constant number of page faults occur in *this* reservation. Pages are moved to the inactive list of the reservation according to the paging policy in use as described earlier. As a performance optimization, we ensure that the *buffer cache is shared* so that pages swapped in by some application can be used by other applications without having to swap in a local copy of the data.

For firm reserves, once the reserve is filled up, they can potentially steal unused pages from the unreserved portion, marking them as borrowed. When the number of pages in the unreserved portion falls below a particular threshold, we return the most aged inactive borrowed pages to the LRU lists of the unreserved portion (the default system freelist). This ensures that an application using a firm memory reservation is always guaranteed not to experience higher response times than when using a hard reservation of the same size.

3.5.2. Swap Reserves

Linux implements swap space as a simple linear array, which makes implementing a swap reservation relatively straightforward. We recommend that the swap reservation value specified by an application be at least the stable steady-state swap usage obtained over long periods of runtime. To be safe, an extra safety margin will normally need to be applied in practice. Our swap reservation implementation allows an application to reserve a sub-section of the array for exclusive use to the particular swap reservation. This array is collocated as much as possible for batching efficiency of disk writes/reads. If contiguous blocks are not found, then the swap reservation creation API returns a failure message.

3.5.3. Hierarchical Memory Reserves

The implementation of hierarchical memory reservation in the Linux kernel is also fairly straightforward. In this case, when a forked child makes a hierarchical reservation using our hierarchical memory reserve interface, a child reservation is created from the memory reserved for the parent reservation. Subsequently, the forked child can get exclusive access to the child reservation which behaves exactly like a level-1 reservation. This concept can be recursively applied for hierarchical (level-i) reservations. If the requested number of pages exceeds the available number of pages in the parent reservation, the reservation request is rejected.

3.5.4. Implementation of Energy-Aware Reservations

When the first task's reservation is created, it is bank-aligned. The system has information of the number of pages in each of the bank that has been filled up (and hence the residual space in each bank). Each of the partially filled banks is equally eligible for packing new reservation requests, thereby reducing the page-scattering effect. All banks belonging to all tasks other than the currently scheduled task are put into power-down mode.

When the reservation spans multiple banks, each memory bank has its own data structures that manage the pages of that bank. This ensures that within a reservation, memory management mechanisms such as buddy management and freelist management are contained within the particular banks and do not propagate to neighbor banks.

In addition, we use profiling information to further augment the energy savings when reservations span multiple memory banks. Using the application "look-back" profile, when we do a context swap, we keep track of the "working banks" of the just-swapped-out task which are the few banks in the immediate past that service most recent page-faults. In cases where the programs exhibit locality of reference, we can ensure that only the "working banks" are turned on while all other banks of the reservation can also be turned off, thus augmenting savings. Hierarchical reservations can be used to collocate all threads of a multi-threaded application to the same bank(s).

3.6. Evaluation of Memory Reserves

In this section, we evaluate our implementation of the memory reservation paradigm in the Linux 2.4.18 kernel. We have implemented memory reservation, hierarchical memory reservation and swap reservation by incorporating changes to the Linux memory subsystem. Our measurements were conducted on a Dell Dimension 8200 2.0 GHz Pentium-4 machine with 512 MB RAM. Our implementation supports reservation sizes from 2 MB to 128 MB.

3.6.1. Hard Reservations

We first demonstrate the ability of memory reservations to enforce limits on the usage of memory pages of applications. In this experiment, we create a hard reservation of 32 MB for the popular cycle-accurate simulator tool SimpleScalar running a simulated execution of the bzip utility. The look-back window was set to 6000 accesses and when the application encountered memory pressure in the form of lack of free pages in the reservation list, exactly 128 pages were evicted. Figure 17 tracks the number of page frames used by this application over time. As can be seen, the total number of pages used by the application never exceeds 32 MB (8192 pages of 4 KB each). This experiment also depicts the usage of reservations as an instrumentation framework that can be used to profile application memory characteristics. There are marked phases in the program which denote the vast drops in page usage (many pages are not used and hence age to zero). As can be seen from the intervals 5s..7s and 17s..20s, when there were no pages in the freelists, only 128 pages are returned (the height of the sawtooth-like waveforms at those points). The slope of the graph for any time *t* on the graph denotes the rate at which the application is generating page faults at that point in time.

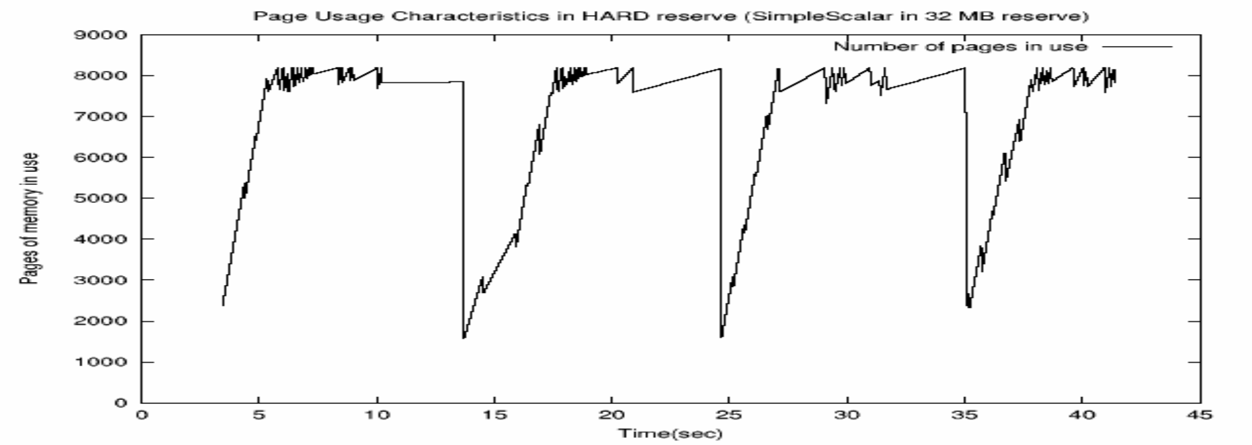


Figure 17: Hard Reservations

3.6.2. Firm Reservations

We next demonstrate the effect of firm reservations which provide guaranteed access to a specified amount of physical memory allocated to a reservation but which allow access to more memory (if available). All parameters are the same as in the previous experiment. In Figure 18, we simultaneously plot the number pages used by the application and the number of pages that belong to a reservation that the application attaches to. The difference in the heights of the two graphs at any point corresponds to the number of pages beyond the reservation size that is borrowed from the system freelist. For firm reservations, whenever the reservation runs out of free pages, it steals pages from Linux's page allocator. Hence, there is no need to forcibly evict any reservation pages that have not aged. This is reflected in the lack of any 128-page high saw tooth waveforms at the reservation's size limit. Borrowed pages are lazily returned to the OS (rather than the application) freelists once they have aged.

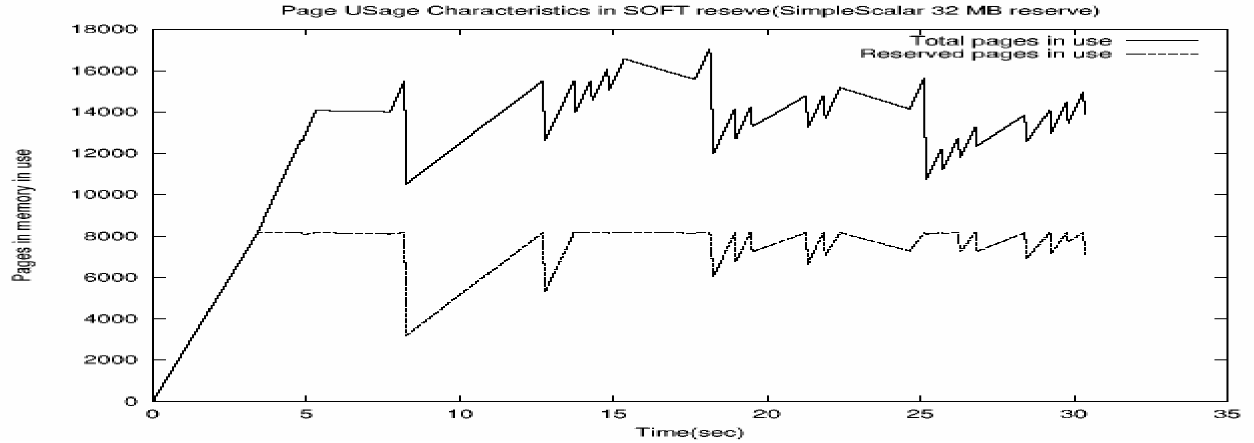


Figure 18: Firm Reservations

3.6.3. Hierarchical Memory Reservations

We next demonstrate the effects of hierarchical reservations where a reservation can, in turn, be split into 2 (or more) child reservations. In this example, a reservation of size 64 MB was split into 2 equal 32 MB reservations that run in parallel.

Amrmd [30] is a convex optimization algorithm with polynomial complexity that has distinct execution phases. During the first phase, multiple files are opened and their contents read into RAM. During the next phase, data elements in the files are merged together and processed, after which the program moves on to a third phase where values are output.

We utilize this opportunity to depict the self-clocking property (where the number of residual pages rather than absolute time is used to trigger paging operations) by plotting the graph in Figure 19 for page fault number along the x-axis rather than wall-clock time. Note that the aging operation is deterministic and occurs once every 1500 accesses, as denoted by the breaks in the graph other than at instants when the reservation is depleted and pages are forcibly evicted. This value was chosen for the following reason: given that we want to minimize capacity misses in the merging phase, we want a large window (we chose 6000 accesses) before a page is aged and is deemed to be inactive. This, in turn, is factored into a page age limit of 4 and an update frequency of once every 1500 accesses. We also want to ensure that the number of pages evicted under memory pressure is low. Note that there are very few points for amrmd where the memory pages have depleted (at around page fault index 10000), and 16 pages are forcibly released every time that there are no pages in the reservation's freelists. The various execution phases of amrmd when files are read in (interval 0..10000), merging algorithms are executed (interval 10000..19000) and results are aggregated, are clearly visible in the graph.

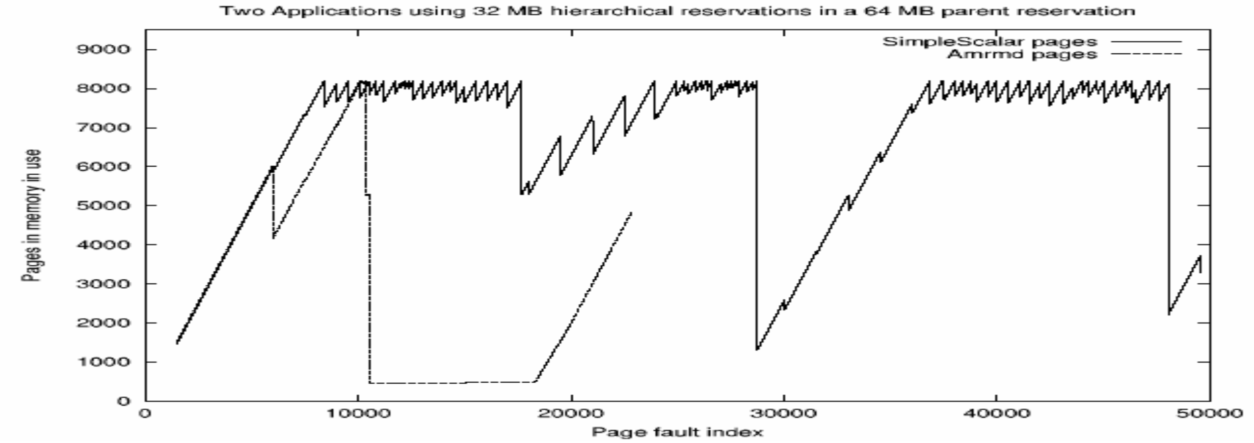


Figure 19 : Hierarchical Reservations

Next we run the amrmd (using a 32 MB reserve) in parallel with the SimpleScalar simulator application (again using 32 MB) in a parent reservation of 64 MB. In particular, we choose the same parameters as before. The graphs are plotted in Figure 19. Note that each reservation does not use the complete 64 MB parent reservation but is constrained only to the child reservation portion of 32 MB each.

3.6.4. Swap Space Reservations

Long-running applications such as server tasks or cron-jobs can benefit from our support for swap space reservation. Even if other malicious or buggy applications start thrashing and use up swap space, we want to be able to ensure that other applications get access to necessary swap space. Since the swap requirements are very application-specific, we suggest that the user (or preferably the application developer) profile the application to determine its swap usage. For example, the swap usage of the X Server was less than 50 pages, that of using the make utility to build the Linux 2.4.18 kernel was 0 pages, that of SimpleScalar irrespective of the benchmark was found to be around 500 pages, while that of amrmd was up to 35,000 pages for bad choices of window and forced eviction threshold values. Typically, good choices of look-back window sizes and eviction counts minimize capacity misses and lead to reduced swap usage.

3.7. Determining Application Reservation Sizes

In this section, we propose a novel scheme to determine reservation sizes for an embedded task-set such that the sum of run-times of all tasks in the system is minimized. The intuition behind our scheme is that the run-time is related to its memory requirement by a concave function. In particular, while increased memory allocation to a process results in lower run-time, the returns in reduction of run-time diminishes with increased memory allocation.

We define the *runtime insensitivity* of a task, given an allocation of *M* MB of memory as (1 - C(M)/ C(M_{min})) where C(M) is the application run-time, and M_{min} denotes the minimum memory requirements provided to any task in the system. M_{min} equals the smallest possible reservation size of the system which equals 2 MB. In Figure 20, we show the run-time insensitivity profile of two applications amrmd and SimpleScalar with variation of reservation size. Applications such as amrmd which are more sensitive to memory needs have a steady slope as compared to an abrupt steep slope that is characterized by tasks such as Simplescalar. For a system with unconstrained memory resources, the point at which the concave curve flattens corresponds to its memory reservation requirement because at this stage, the number of capacity misses becomes zero. Thus, amrmd should be allocated 64 MB and SimpleScalar 32 MB to minimize the capacity misses.

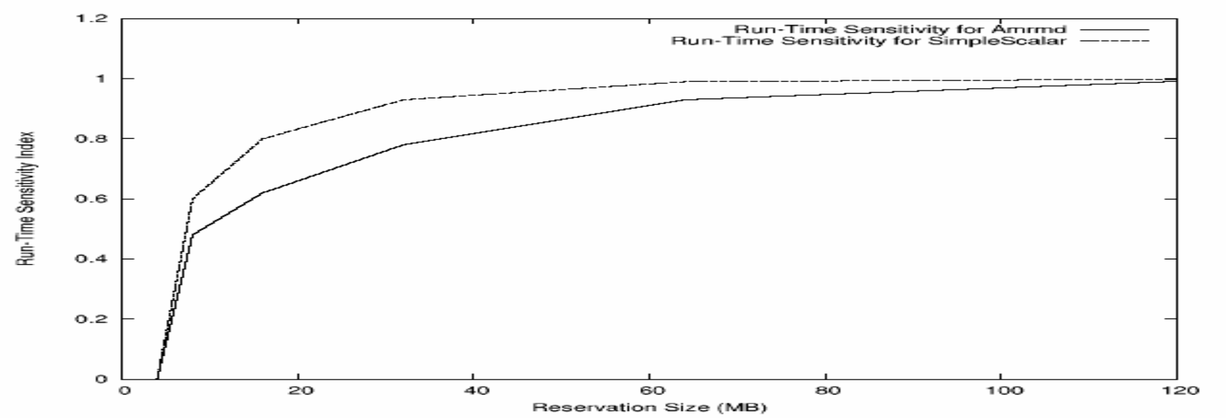


Figure 20: Run-Time Variation with Reservation Size

When the system memory resources are insufficient for optimal allocation to all tasks, we must apportion it efficiently. In particular, assuming a set of n tasks ($T_1..T_n$) in a system with R units of memory resources, each task T_i with K choices of reservation sizes $M_i = (M_i^1 .. M_i^K)$ and corresponding run-times ($C_i^1 .. C_i^K$) ,we would like to choose reservation sizes Ri for each task from M_i to maximize the sum of run-times insensitivity indices (C_{tot} = sum(Ci)) of all tasks in the system subject to the memory size constraints in the system(sum(R_i) < R).

3.7.1. Run-Time Minimization Algorithm

An algorithm to determine the optimal reservation sizes $R_{\underline{i}}$ for each application T_i to obtain C_{tot} is as follows:

- Let the current allocation of memory to T_i be R_i. Let the unallocated quantity of the memory be R^1. Compute the slopes corresponding to the rate of change in run-time with memory allocation for the task-set.
- Identify (a) the sub-collection of applications with the largest value of $C'_i(R_i)$, (b) the number of applications in that sub-collection (denoted by k), and (c) the application (denoted by j) with the second largest value of this quantity if any such application exists. If the largest value of $C'_i(R_i)$ is 0, then stop. No further allocation will increase system utility and spare resources are available.
- Increase R_i for each of the members of the sub-collection so that their values of $C'_i(R_i)$ decrease but continue to be equal until either (i) this value becomes equal to the second largest value or (ii) the additional resources added to this sub-collection equal R^l. In case (ii), stop as all resources have been optimally allocated.
- In case (i), one or more new applications should be added to the sub-collection. Return to Step 1.

This algorithm results in optimal reservation sizes (R_1 .. R_n) for the task-set such that the sum of their run-times is minimized. This algorithm uses the Karush Kuhn-Tucker condition and is very similar in spirit to the Q-RAM SRSD algorithm in [42]. The interested reader is referred to it for proof correctness.

3.8. Exploiting Memory Reservation

In the previous section, we proposed an algorithm to determine optimal memory reservation sizes for an embedded task-set. We now describe how memory reservation can be used productively by applications by customizing paging policy parameters so as to enhance performance gains.

3.8.1. QoS-Sensitive and Multimedia Applications

For multimedia applications such as mpeg players and videoconferencing tools, since page-faults generated due to capacity misses typically result in application-level jitter, we recommend that such applications be associated with firm reservations. For example, most mpeg players, due to their look-forward nature, do not need more than a memory space of 16 MB. Thus, we associate a firm reservation with these multimedia applications in order to guarantee jitter-free functioning. In an experiment, we associated a firm reservation of 16 MB with mplayer, an mpeg player. In parallel, we ran a malicious application that greedily consumed both memory and processing time and found that it did not affect the mpeg player's performance at all.

For multimedia tasks, we also provide the functionality to turn paging off completely so that no disk overhead will be incurred at the application layer since the total memory requirements of such multimedia tools easily fits within the system memory. For example, the total memory requirements of the popular mpeg player mplayer running an mp3 song of 7.32 minutes took 37,381 pages. For applications with predictable memory profiles such as multimedia players, it is possible to prefetch data pages into the memory reserve to counter the performance hit due to cold misses. Similarly, we also have extensions that allow the application to specify when to clean up the reservation rather than having the operating system do it implicitly.

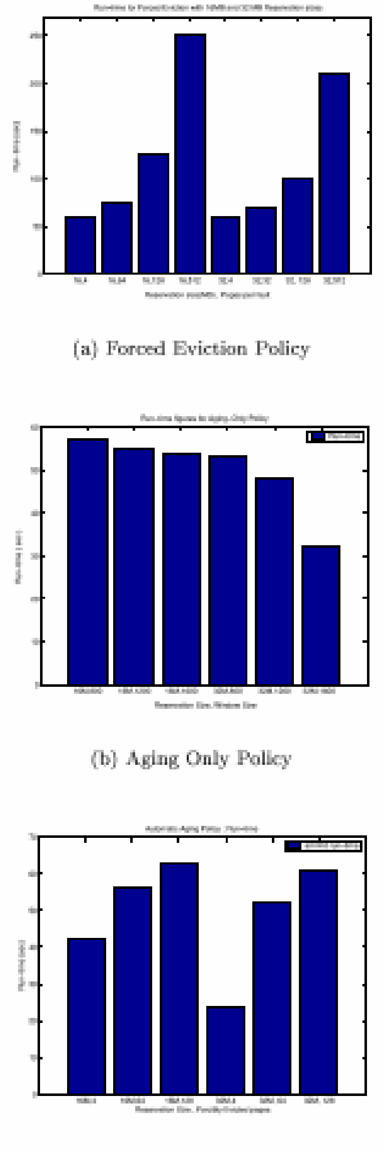
We note that in soft real-time multimedia systems, due to the capacity-insensitive nature of the applications, the benefits of determining optimal reservation sizes (see Section 3.7) far outweighs the impact of the paging policy used.

3.8.2. Application Customized Paging Policy

In this subsection, we explore the impact of the various “self-clocking” paging policies on the performance of typical applications. The application under concern was amrmd, which was chosen as a representative benchmark due to its high sensitivity to capacity misses. Capacity-insensitive applications are not affected much by policy.

3.8.2.1. Memory Policy for Capacity-Sensitive Applications

As far as page-faults are concerned, forced eviction performs well, especially for a low *Forcibly Evicted* Pages threshold, the parameter which signifies the number of pages that are forcefully evicted under memory pressure. However, since only a few pages can be evicted when the freelist is empty, multiple page flushes cannot be amortized over one disk access and hence there is considerable run-time overhead (Figure 21(a)). We found that the capacity misses increase by 23% when the reservation size is reduced from 32 MB to 16 MB.

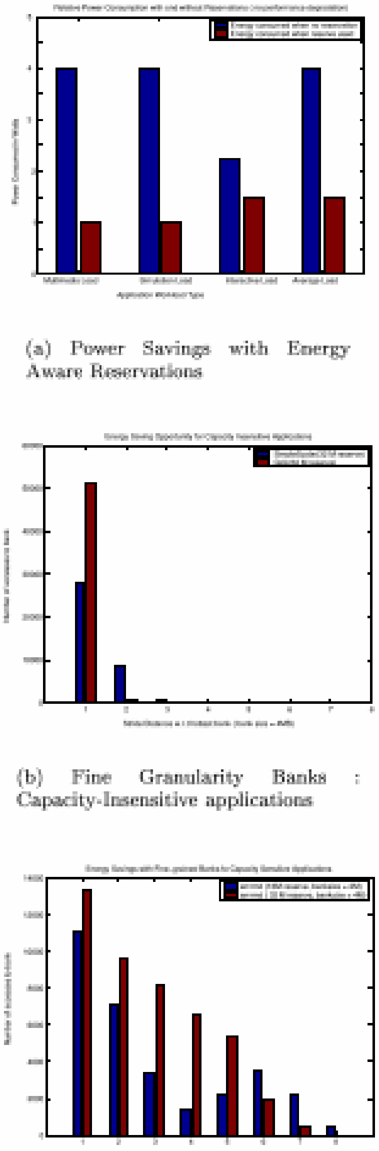


(a) Forced Eviction Policy

(b) Aging Only Policy

(c) Automatic Decay Policy

Figure 21: Self-clocking Paging Policy



(a) Power Savings with Energy Aware Reservations

(b) Fine Granularity Banks : Capacity-Insensitive applications

(c) Fine Granularity Banks : Capacity-Sensitive applications

Figure 22: Energy-aware Reservation

For the aging-only policy, the knob that can be varied is the window size. As the reservation size and/or the look-back window size increases, the number of capacity misses is reduced. This effect is evident in the run-time graph of Figure 21(b). With the same 32 MB reservation, different look-back window sizes lead to significantly different execution times, ranging from about 35 seconds for a 1600 page look-back window to 54 seconds for a 800 page look-back window. However, if the window size gets too large, there may be quick spurts of high page-fault rates leading to requests for free pages when the reservation freelist is empty. We found that for amrmd, it is not possible to make the window size more aggressive (larger) than 1600 (400*4) accesses.

The results indicate that the hybrid of the forced eviction and aging-only policies is worth exploring. The run-time results for a 16 MB reservation for the forcibly-evicted pages parameter choices of 4, 64 and 512 pages can be seen in Figure 21(c).As the number of pages evicted gets larger, the look-back window size is decreased and therefore the number of capacity misses increases. This behavior is repeated when the reservation size is 32 MB, except that the absolute number of capacity misses was found to be less, resulting in lower absolute run-times as can be seen. Forcibly evicting a small number of pages (4) under pressure leads to the best performance and using a 32 MB reservation yields better run-times as can be expected. In summary, when the number of forcibly evicted pages is small and the window size is large, the automatic-aging policy performs the best in terms of minimizing runtime.

We also evaluated the impact of interaction of multiple application types when attached to the same reservation. We experimentally verified that *multiple capacity-sensitive applications* could affect each other adversely when attached to the same reservation, causing up to 26% performance degradation. On the other hand, *capacity-insensitive applications* are insensitive to interaction effects when attached to the same reservation.

3.8.3. Evaluation of Energy-Aware Reservations

We now evaluate the effectiveness of energy-aware reservations in saving memory power by strategically turning unused banks off. Since there are documented hardware bugs in the RDRAM memory controller, we had to resort to indirect measurements based on kernel statistics rather than directly measure the memory energy consumption. Since the operating system has no way of ascertaining whether shared pages which exist in other banks are touched by a given process, we *pessimistically* assume that all currently in-use banks can be touched. Hence, by bin-packing the reservations into the minimum number of banks, we ensure that memory power is minimized with zero impact on performance. We compare our energy-aware reserves with a baseline system without reservations but where *unused banks are kept turned off.*

We classify application-workloads into multiple categories: *multimedia workloads* with media streaming and processing applications, *simulation workloads* like trace simulators, emulators etc, interactive workloads involving interactive programs such as the X server that consume a lot of memory pages with relatively infrequent user-propelled activity and *average workloads* with a typical mix of these tasks.

We found that the kernel and its allied threads occupy 3 banks of memory. Linux, as it stands, maps the kernel to the first few banks of RDRAM while user memory is allocated proceeding downward from the highest bank. We made changes to make physical memory allocation sequentially increasing so that the first user memory bank uses the remaining portion of the partially filled third kernel bank, thus saving us 1 bank.

For multimedia and simulation workloads, since the applications are capacity-insensitive applications, the working set size is small and hence the reservation sizes can be made small. In our example, we consider mplayer running in parallel with an mpeg tool Avi2Mpeg - the optimal reservation sizes for each is 16 MB. Given the RDRAM bank granularity, this directly translates to 1 bank. Figure 22(a) plots the memory power consumed by applications that use energy-aware reserves as compared to the naive turn-off policy discussed above. In operating systems without reserves, long video files soon result in the application occupying the entire memory space, thus requiring all banks to be eventually turned on. In comparison, only 4 banks need to be kept on in the reservation case as compared to 16 banks in the unreserved scenarios. Similarly, multiple threads of SimpleScalar were run in the simulation workload. This is shown in the first two pairs of bars of Figure 22(a). The kernel banks are always turned on since the application threads typically invoke kernel system calls such as file reads. This directly translates to energy savings of 3.0 watts. Each RDRAM device drains 313 mW of power in Attention mode, 225 mW in Standby mode , 11 mW in Nap and 7mW in Powerdown mode, for the banks that are turned on, we assume that an active bank is in Standby mode 50 percent of the time and in Attention mode the rest of the time which corresponds to 75 percent as compared to no reservation but with no performance impact.

For interactive workloads, we consider the X-Server running a browser and spreadsheet application for 10 minutes. We found that a reservation size of 64MB (2 banks) was sufficient for smooth operation of X and the spreadsheet, thus requiring a total of 5 banks to be turned on. As compared to that, when X was launched without a reservation, along with typical X-based applications, X needed a steady state requirement of 6 banks of memory, thus requiring 9 banks in total to be kept turned on as shown in the third pair of bars of Figure 21. For interactive applications, we save up to 44.4 percent power.

When an “average” work-load with multiple types of applications, the combined savings is 2.5 Watts (62.5%) as shown in the final pair of bars of Figure 22(a). This is because the X server and SimpleScalar require just 3 banks while in the unreserved case, media applications and SimpleScalar alone occupied all 16 banks. For capacity-sensitive applications such as database indexing /searching programs, the look-back window is as large as the reservation size. Hence there is not much possibility of power savings for capacity-sensitive applications beyond carefully chosen reservation sizes without a performance hit.

Energy-aware reserves are *automatically* associated with soft reservations since the cost involved in keeping a disk spun-up due to additional paging activity outweighs the energy gained by turning banks off. Our approach, however, ensures that so long as there is an opportunity to save energy by judicious bank-alignment of pages, we save energy. Our results demonstrate that this is indeed the case in most practical systems with realistic workloads.

3.8.3.1. Finer-Granularity Memory Banks

We now explore whether applications can make use of banks with finer granularity than the 16MB/32MB currently supported by RAMBUS. When application memory reservations span multiple banks, the scheme discussed so far was to keep only the banks associated with the scheduled task on. However, when the application look-back window is small, there is no need to turn on all memory banks. Suppose the working bank as the memory bank that services most application requests in the last time-window (say 100 ms) of application execution. The formula for the *stride distance* of bank bk_{idx} w.r.t the working bank bk_{hot} when the number of banks in the reservation equals N is $(bk_{idx} - bk_{hot} + N)$mod N. Thus if a reservation spans 8 banks, the stride distance of bank 2 w.r.t bank 3 is 1 while the stride distance of bank 4 w.r.t bank 3 is 7. Since page allocation in a reserve is sequential, the stride distance signifies the time into the past when a particular bank was touched in the current interval.

We are interested in the way page faults are spread out among the banks - the wider they are spread out relative to the working bank, the less locality is exhibited by the application. This information can, in turn, be used to turn off banks in a reservation. Thus, these graphs can be used to determine the granularity of bank sizes that would be optimal for use. We do not consider memory bank sizes lower than 2 MB because at this point the bank sizes equal the size of the highest contiguous memory chunk that typical operating systems use.

Thus for the experiments, we collect kernel statistics for memory bank sizes of 4 MB. As the program executes, we periodically checkpoint the number of page faults w.r.t the working bank in the last 100 ms of program execution. Our results are shown in Figure 22(b). They show that capacity-insensitive applications can save an additional 88.5 percent energy for gzip compared to the savings got when bank sizes are coarse-grained (32 MB)! Similarly for SimpleScalar, the additional energy savings when bank sizes were reduced were 75 percent. This clearly shows that finer granularity banks can be used by some applications to augment power savings without a performance hit. However, it can be seen that for capacity-sensitive applications, there is no significant opportunity for additional power savings without degrading performance. In particular, for 16 MB reservation sizes, the savings is 0 percent while even for 32 MB reservation sizes, the maximum possible energy savings is only 18.75 percent as shown in Figure 22(c). We therefore conclude that finer granularity banks of at least around 2 MB or 4 MB can be leveraged by capacity-insensitive applications to save memory power consumption.

4. Conclusion

We have introduced a new power-aware, system-energy management scheme known as PowerTap. In normal operation, PowerTap optimizes the performance of a system within a given power budget. Alternately, PowerTap can achieve the best power efficiency while observing performance and other constraints associated with each application. Examples of such constraints include: timing or deadline constraints in real-time applications, video and audio quality constraints in multimedia applications, high disk bandwidth usage in video servers, application mission-time in the limited power system, etc. Within PowerTap, the operating system and other software tools collaborate with the hardware to manage and minimize power consumption while providing quality of service guarantees to applications as well as the overall system. System software and hardware monitor power levels for all critical system components, and adjust these levels dynamically to meet application timing, and system power/energy demands.

An important component of PowerTap is the **Power-Aware Real-Time Operating System** (PARTOS). PARTOS is a morphable real-time operating system that manages power in the system hardware resources. Such resources include disks, memory, and network interface cards. PARTOS also manages power levels for periodic activities such as interrupt handlers, and for system software services such as TCP/IP stacks. Furthermore, PARTOS tunes the system software computational requirements. Since PARTOS is morphable, it can adapt its power management policies depending on the available power source (battery, or AC). Under battery power, it balances the discharging rates and the recharging rates of the batteries. Under AC power, the low average power rate is focused.

Carnegie Mellon University has previously developed the Quality of Service (QoS)-Resource Allocation Model (Q-RAM). PowerTap exploits an extension of Q-RAM called the **Power-Aware Resource Allocation Model** (PAQ-RAM). Applications running under PARTOS have various quality of service requirements with respect to their usage of finite system resources. PAQ-RAM allows PARTOS to balance these multiple quality of service requirements and maximize the utility (users' satisfaction) of the overall system. Portable, battery operated systems have a restricted energy usage requirement (the discharge rate of the battery has to be lower than the charge rate to maintain stability). Airplanes and other mobile vehicles also fall into this category. PAQ-RAM takes this energy budget into account with other resources and adjusts the quality of service of individual application based on its criticality and the amount of resources available.

In conclusion, PowerTap is analogous to a water tap, allowing applications to draw as much power and performance as they need, but no more. This approach economizes power and extends the mission life of energy-poor systems such as battery-operated and energy-harvesting platforms. PowerTap will see significant use in military and space-borne applications.

5. References

[1] D. d. Niz, L. Abeni, S. Saewong, and R. Rajkumar, "Resource Sharing in Reservation-Based Systems," presented at the IEEE Real-Time Systems Symposium, 2001.

[2] S. Saewong and R. Rajkumar, "Practical Voltage-Scaling for Fixed-Priority RT-Systems," presented at the 9th IEEE Real-Time and Embedded Techonology and Applications Symposium, 2003.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," presented at the IEEE Real-Time Systems Symposium, London, UK, 2001.

[4] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable voltage core-based systems," presented at the 36th Design Automation Conference, 1998.

[5] P. Pillai and K. G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," presented at the 18th ACM Symposium on Operating Systems Principles, SOSP'01, 2001.

[6] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. D. Micheli, "Dynamic voltage scaling and power management for portable systems," presented at the 38th Design Automation Conference, 2001.

[7] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," presented at the 36th Annual Symposium on Foundations of Computer Science, 1995.

[8] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar, "Critical Power Slope: Understanding the runtime effects of frequency scaling," presented at the 16th Annual ACM International Conference on Supercomputing, 2002.

[9] Compaq Computer Corporation, "Compaq iPAQ h3600 hardware design specification version 0.2f," http://www.handhelds.org/Compaq/iPAQH3600

[10] ADI Engineering Inc., "BRH reference platform," http://www.adiengineering.com/productsBRH.html

[11] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource Kernels: A Resource-Centric Approach for Real-Time Systems," presented at the SPIE Conference on Multimedia Computing and Networking, 1998.

[12] S. Saewong and R. Rajkumar, "Analysis of Hierarchical Fixed-Priority Scheduling," presented at the IEEE Euromicro Conference on Real-Time Systems, 2002.

[13] C. L. Liu and J. Layland, "Scheduling algorithms for multiple programming in a hard real-time environment," *Journal of the ACM*, Vol. 20(1), 1973.

[14] Maxim Integrated Products Inc., "Maxim 1855 Evaluation Kit Reference," http://pdfserve.maxim-ic.com/arpdf/MAX1716EVKIT-MAX1855EVKIT.pdf

[15] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," *IEEE Transactions on Software Engineering*, Vol. 23, 1997.

[16] F. H. P. Fitzek and M. Reisslein, "MPEG-4 and H.263 video traces for network performance evaluation (extended version)," Technical Report TKN-00-06, Technical University of Berlin, Germany 2000.

[17] G. Quan and X. S. Hu, "Enhanced fixed-priority scheduling with (m, k)-firm guarantee," presented at IEEE Real-Time System Symposium, 2000.

[18] S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proprotionate progress: A notion of fairness in resource allocation," *Algorithmica*, Vol. 15, pp. 600-625, 1996.

[19] Head of the Telecommunication Networks (TKN) Group, "MPEG-4 and H.263 video traces for network performance evaluation," http://www-tkn.ee.tu-berlin.de/research/trace/trace.html

[20] A. Bavier, B. Montz, and L. Peterson, "Predicting MPEG execution times," presented at SIGMETRICS/PERFORMANCE' 98, International Conference on Measurement and Modeling of Computer Systems, 1998.

[21] M. Hamdaoui and P. Ramanathan, "Evaluating dynamic failure probability for streams with (m, k)-firm deadlines," *IEEE Transactions on Computers*, Vol. 46, 1997.

[22] Andrew W. Appel and Kai Li, “Virtual memory primitives for user programs,” *In Proceedings of ASPLOS* , 1991.

[23] Kieran Harty, David Cheriton, “Application-controlled physical memory using external page-cache management,” *In Proceedings of ASPLOS,* 1992.

[24] P. Cao, E. Felten, and K. Li, “Application-controlled file caching policies,” *In Proceedings of USENIX,* 1994.

[25] D Engler, M. Kaashoek, and J. O'Toole Jr., “Exokernel: An operating system architecture for application-level resource management,” *In Proceedings of 15 ACM Symposium on Operating Systems Principles*, 1995.

[26] R. E. Kessler, Mark D. Hill, “Page Placement Algorithms for Large Real-Indexed Caches,” *ACM Transactions on Computer Systems,* 1992.

[27] Gideon Glass, Pei Cao, “Adaptive Page Replacement Based on Memory Reference Behavior. Measurement and Modeling of Computer Systems,”1997.

[28] Steven M. Hand, “Self-Paging in the Nemesis Operating System,” *In Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[29] T. Nakayama and H. Tezuka, “Virtual memory management for interactive continuous media applications,” *In Proceedings of Intl. Conf. on Multimedia Computing and Systems*, June 1997.

[30] Sourav Ghosh, Ragunathan Rajkumar, Jeffery P. Hansen, John P. Lehoczky, “Scalable Resource Allocation for Multi-Processor QoS Optimization,” *In Proceedings of ICDCS*, 2003.

[31] Todd M. Austin, Eric Larson, Dan Ernst, “SimpleScalar: An Infrastructure for Computer System Modeling,” *IEEE Computer 35(2): 59-67*, 2002.

[32] H. Tokuda,T. Nakajima, and P. Rao, “Real-Time Mach: Towards a predictable real-time system,” *In Proceedings of Usenix Mach Workshop*, Oct. 1990.

[33] Raj Rajkumar, Kanaka Juvva, Anastasio Molano and Shui Oikawa, “Resource Kernels: A Resource-Centric Approach to Real-Time Systems,” *In Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking,* January 1998.

[34] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, Carla Schlatter Ellis, “Power Aware Page Allocation,” *In Proceedings of ASPLOS*, 2000.

[35] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramniam, and M. J. Irwin, “Hardware and software techniques for controlling DRAM power modes,” *IEEE Transactions on Computers*, 2001.

[36] H. Huang, P. Pillai, and K. G. Shin, “Design and Implementation of Power-Aware Virtual Memory,” *In Proceedings of USENIX Technical Conference*, June 2003.

[37] N. AbouGhazaleh, D. Mosse, B. Childers, et al, “Collaborative Operating System and Compiler Power Management for Real-Time Applications,” *In IEEE Real-Time Embedded Technology and Applications Symposium (RTAS)*, 2003.

[38] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, “Influence of compiler optimizations on system power,” *In Proceedings of Design Automation Conference*, 2000.

[39] U. Kremer, J. Hicks, and J. Rehg, “A compilation framework for power and energy management on mobile computers,” *In International Workshop on Languages and Compilers for Parallel Computing*, 2001.

[40] X Li , Z Li , F David et al, “Performance directed energy management for main memory and disks,” *In Proceedings of ASPLOS,* 2004.

[41] Anand Eswaran, “Energy-aware Memory Reservation Support for Applications,” *Masters Thesis submitted to the Department of ECE , Carnegie Mellon,* July 2004.

[42] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, “A Resource Allocation Model for QoS Management,” *In Proceedings of 18th Real-Time Systems Symposium*, December 1997.

[43] Jerry Hom and Uli Kremer, “Energy Management of Virtual Memory on Diskless Devices,” *In Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, September 2001.