行政院國家科學委員會專題研究計畫　成果報告

性能可調適之即時系統：設計與實作

計畫類別：個別型計畫
計畫編號：NSC93-2218-E-002-140-
執行期間：93 年 08 月 01 日至 94 年 07 月 31 日
執行單位：國立臺灣大學資訊工程學系暨研究所

計畫主持人：郭大維

計畫參與人員：謝仁偉，張立平，羅習五，賴啟屏，郭錦福，盧永豐，陳建佳，
黃思偉，黃文彬，林俊仁，黃昱愷，劉淑萍

報告類型：精簡報告

處理方式：本計畫可公開查詢

中　華　民　國 94 年 10 月 27 日

行政院國家科學委員會專題研究計畫成果報告

計畫名稱：性能可調適之即時系統: 設計與實作

計畫編號：NSC 93-2218-E-002-140

執行期限：計畫自民國 93 年 08 月 01 日起至民國 94 年 07 月 31 日止

主持人：郭大維

計畫參與人員：謝仁偉，張立平，羅習五，賴啟屏，郭錦福，盧永豐，陳建佳，黃思偉，黃文彬，林俊仁，黃昱愷，劉淑萍

執行單位 ： 國立台灣大學資訊工程所

一、摘要

中文摘要：

在過去的數十年間，當多處理器的效能快速地提升、硬體效能變得可調節（包含多處理器甚至快閃記憶體），具有反應時間限制之電腦系統的設計及排程審核在即時系統中是非常重要的課題。本計畫中探討在可調節因素的處理器和儲存裝置上的即時程序的執行效能。同時，本計畫也會探討快閃記憶體的儲存系統與即時資源管理的實作影響。雖然同步多執行緒已經有許多成果在改善處理器的效率，不同於之前的許多結果，本計畫探討在同步多執行緒處理器（Simultaneous Multithreaded，SMT）上調節裝置速度的即時程序排程，這種行為將會相似於在多處理器上作排程，且允許各處理器執行速度不同以及資源共享（因為活動的邏輯處理器數量不同）。本計畫能決定獨立的週期性工作在即時同步多執行緒排程中是否能被排程，並提供「考慮工作是否能搬移」的以擬真為基礎的排程演算法，用以在不可調的同步多執行緒處理器上模擬出一個可調節速度的同步多執行緒處理器。同時，本研究也探討了快閃記憶體的儲存系統和即時資源管理的實作影響。在快閃記憶體儲存系統中，熱門資料辨別不止在快閃記憶體的資源回收系統中有很大的影響，也和快閃記憶體的存取效能和使用壽命息息相關。本研究更提供了一個非常有效率的方式在有限的空間裡即時辨別熱門資料，我們不但提供一個有效率的實作方式，也做了一連串的實驗來證明效率，在此我們已有令人振奮的結果。

Abstract：

In the past decades, as the performance of microprocessors increases so dramatically, and the performance of hardware components, including microprocessors or even flash memory, becomes adjustable, the design and the schedulability verification of computer systems with timing constraints is an important issue for real-time systems. In this research, we explore the performance in the real-time process with adjustable factors for the processors and storage devices. We shall also study the flash-memory storage systems and its real-time implementation impacts as parts of real-time resource management.

Although simultaneous multithreading (SMT) has been shown being an efficient technique to improve processor performance, little work has been done on real-time SMT scheduling. Different from many of the past results, we explore process scheduling on adjustable processor/device speeds to simultaneous multithreaded (SMT) processors that could behave like multiprocessors but with hardware sharing and different speeds (due to different numbers of activated logical processors). Real-time SMT scheduling for independent periodic task sets with schedulability guarantees is studied in this research. We propose emulation based scheduling algorithms with and without task migration to emulate an adjustable SMT processor over a nonadjustable SMT processor.

The flash-memory storage systems and its real-time implementation impacts as parts of real-time resource management are also studied in this research. Hot-data identification for flash-memory storage systems not only imposes great impacts on flash-memory garbage collection but also strongly affects the performance of flash-memory access and its life time (due to wear-levelling). We propose a highly efficient method for online hot-data identification with limited space requirements. We not only propose an efficient implementation of the proposed framework but also conduct a series of experiments to verify the performance of the proposed method, in which very encouraging results are presented.

關鍵詞：

SMT, real-time scheduling, flash memory, workload locality

二、前言

A lot of research work has been done on SMT to maximize the performance of an SMT processor and shown that it is a power-efficient design for embedded applications [8, 11, 12]. Dorai, et al. [2] studied the priority assignment problem in an SMT processor for tasks with different importance levels. Raasch et al. [7] proposed a dynamic priority assignment algorithm in which the system selects a time point called checkpoint, and the task with the farthest distance to the next checkpoint is assigned the highest priority. However, little work has been done on SMT scheduling for real-time systems. Gautham Thambidorai et al. [14] investigates resource allocation policies for SMT processors that preserve the performance of a “foreground” thread, while permitting the progress of background threads. The foreground thread could be used to serve real-time tasks. The VISA approach [15] assures the absolute performance guarantee of the foreground thread by continuously monitoring its progress. The two approaches assume that there is a good mixture of hard and soft real-time tasks (a hard realtime tasks can only be overlapped with soft real-time tasks) and require hardware modifications. Another pioneer work was done by Jain, et al. [6]. They studied the benefits in scheduling soft real-time tasks on SMT processors in order to minimize the number of deadline-missing.

We study real-time SMT scheduling for periodic task model [4]. Two SMT processor models are studied: The adjustable SMT processor model (a similar model could be found in the study [13]), and the non-adjustable SMT processor model (e.g., Pentium 4). Beginning with a task-set that is schedulable in 1-mode (disable the SMT technology), we try to maximize the degree of task overlap without violating tasks’ schedulability. It is important to note that this multithreading formalism does not provide any tangible performance benefit to the hard real-time tasks themselves. Nevertheless, the problem formulation is valuable, because the amount of idle time (no hard real-time tasks running) is increased with respect to purely serial (1- mode) execution. Idle time can be used for running soft real-time or non-real-time tasks (or saving power), without conflicting with hard real-time tasks.

Flash memory has become an excellent alternative for the design and implementations of storage systems, especially for embedded systems. With potentially very limited computing power from a flash-memory controller or an embedded-system microprocessor, it is of paramount importance to have efficient designs for space management methods. One critical example method is for hot-data identification, in which a given logical block address (LBA) is verified to see if it contains frequently accessed data (referred to

as hot data). Hot-data identification for flash-memory storage systems not only imposes great impacts on flash-memory garbage collection but also strongly affects the performance of flash-memory access and its life time.

The management of flash memory is carried out by either software on a host system (as a raw medium) or hardware circuits/firmware inside its device. In particular, Kawaguchi, et al. [22] proposed a flash-memory translation layer to provide a transparent way to access flash-memory through the emulating of a block device. Wu and Zwaenepoel [23] proposed to integrate a virtual memory mechanism with a nonvolatile storage system based on flash memory. Native flash-memory file systems were designed without imposing any disk-aware structures on the management of flash memory [24, 25]. Chang and Kuo focused on performance issues for flash-memory storage systems by considering an architectural improvement [17], an energy-aware scheduler [26], and a deterministic garbage collection mechanism [27]. Beside research efforts from the academics, many implementation designs and specifications were proposed from the industry, e.g., [28, 29, 30, 31]. While a number of excellent designs were proposed in the past years, many of the researchers, e.g., [17, 19, 22, 23], also pointed out that on-line access patterns would have a strong impact on the performance of flash-memory storage systems, due to garbage collection activities. Locality of data access were first explored by researchers, such as Kawaguchi, et al. [17, 19, 22, 23], where approaches were proposed to distribute hot data over flash-memory for wear levelling or to improve the performance of garbage collection and space allocation. Different from the past implementations, a multi-hash-function framework is proposed, in which multiple independent hash functions are adopted to reduce the chance of false identification of hot data and provide excellent performance for hot-data identification.

三、即時同步多執行緒排程

The performance issue in SMT system is very different from that in a multiprocessor (MP) system. The cost of task migration among the logical processors (LPs) normally is very low [5]. Changing the number of simultaneously running LPs can be done swiftly [5]. Operating systems (os) and applications may choose to take the advantages to have a higher flexibility in task scheduling. On the other hand, the LPs may compete with each other for functional units (e.g., ALU, load/store units) such that the execution times 1 of the running programs may be lengthened. The increased execution time could seriously affect the schedulability of a real-time system. It is the purpose of this paper to design an efficient scheduling algorithm to resolve this problem.

- ***Processor Model***

We consider the processing of independent periodic tasks in an SMT processor. A task τ_i is represented by a tuple (φ_i, c_i, p_i), where p_i and φ_i are the inter-arrival time and the first activation time of τ_i, respectively. c_i is the worst-case computation time of τ_i when τ_i is the only task being executed by the SMT processor. The relative deadline of τ_i is p_i. The *utilization* u_i of τ_i is derived as c_i p_i. An *activated logical processor* (a-LP) is an LP which is processing a task. An SMT processor is operating in the m-mode if it has m a-LPs. In the adjustable SMT processor model model, LP_j is associated with a *speed ratio* $RC_j \geq 0$. An activation of task τ_i executing in LP_j would finish the execution after $c_i \times (1/RC_j) \times (1/(1+\varepsilon))$ units of time where ε is the *speedup rate* in processing speed because of the SMT

technology. In the non-adjustable SMT processor model, the processing time needed by τ_i for each activation is $t \times m \times (1/1+\varepsilon)$ if the SMT processor always activates m LPs.

We assume availability of worst-case execution times of tasks (i.e., the assumption on ε). However, SMT processors inherit the complex architecture of superscalar processors, whose performance is virtually impossible to bound (at least not in a provable manner). So, the analytical model is an abstraction of an ideal SMT processor where the worst case execution time varies concavity with the number of activated logical processors. However, many techniques developed for superscalar processors are also suitable for SMT processors. For instance, cache conflict might be resolved efficiently by cache partitioning [9, 10]. In a safe critical systems, we can adopt certain hardware mechanisms to ensure that ε is always greater than 0. For example, in the VISA specification [15], the processor is continuously monitored. If a task tends to miss its deadline (i.e., $\varepsilon < 0$ for a period), the processor would be reconfigured to operate in *1-mode* and a conventional algorithms (e.g., EDF[4]) could be used to schedule tasks successfully.

We shall use the earliest deadline first (EDF) [4] to illustrate the scheduling problem in the non-adjustable SMT processor model. Consider two tasks τ_1 and τ_2 in an SMT processor serving by two LPs, as shown in Fig. 1. The height of a "task box" indicates the relative speed of its servicing LP. If the two LPs are activated concurrently to serve the two tasks, τ_2 may miss its deadline in some cases (e.g., when $\varepsilon = 0$), as shown in Fig. 1.(a). Since the deadline of τ_2 is earlier than that of τ_1, we may make LP1 inactive temporarily for some interval of time such that LP2 is the only a-LP to execute τ_2 at the full speed, as shown in Fig. 1.(b). The example underlines the motivation of this research. We aim at proposing an intelligent strategy to dynamically activated and deactivated LPs to meet task deadlines.

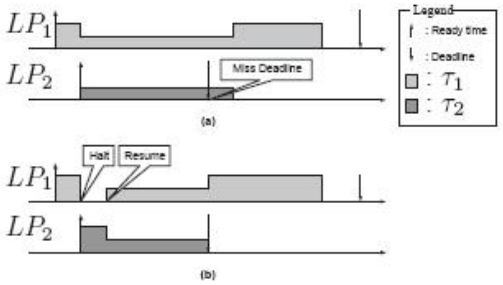

Figure 1. The SMT scheduling without/with consideration on timing constraints.

- ***Problem Definition***

Definition1: A "context switch" occurs when the set of running tasks executing by the SMT processor is changed. An "interval" of a schedule is a period of time in which no context switch occurs.

Definition2: The Multithreading Level (MTL) of a schedule is defined as:

$$\frac{\sum_{i=1}^{n}(\#\ of\ a\text{-}LPs\ in\ interval_i \times the\ length\ of\ interval_i)}{\sum_{i=1}^{n} the\ duration\ of\ interval_i}$$

where a schedule is a sequence of intervals with task assignments.

It can be seen that the MTL of a schedule is the average number of a-LPs. It is an approximate measure of the parallel degree of a schedule running over an SMT processor. Note that the rationale behind SMT is to maximize the utilization of physical execution resources by simultaneously processing. We adopt MTL as a primary performance metrics for the evaluation of SMT scheduling algorithms, provided that no deadline violation occurs.

- ***Real-Time SMT Scheduling - Different Speed Ratios***

The propose of the *TASK-PARTITIONING*

algorithm (longest processing time first [3]) is to divide a set of tasks into M groups for an SMT processor having M LPs so as to evenly distribute workload over the M LPs. As shown in Fig. 2, the 5 tasks (i.e., $\{\tau_1, \ldots, \tau_5\}$) are partitioned into 3 groups (i.e., $\{T_1, T_2, T_3\}$) and each group is served by an LP (i.e., $\{LP_1, LP_2, LP_3\}$). We adopt the EDF scheduling algorithm [4] for the scheduling of the tasks in each LP. The reasons in choosing EDF is that the achievable utilization factor of EDF is 100%. Although an even distribution of workloads over M LPs provides a higher flexibility in functional units scheduling, any distribution of the tasks over the LPs always results in the same MTL, as long as all of the LPs are activated all the time.

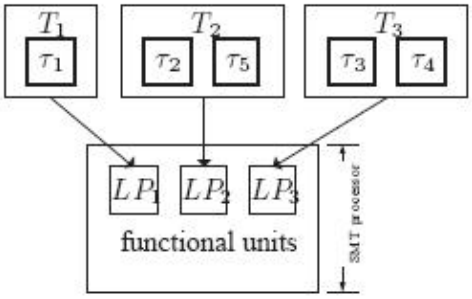

Figure 2. SMT scheduling with partitioning.

Algorithm TASK-PARTITIONING
Input: A set T of real-time tasks
A number M of logical processors
Output: A set of pair (T_y, RC_y) in which T_y is a subset of T and RC_y is the speed-ratio of LP_y

1: **set** $RC = 0$
2: **for all** logical processor LP_j **do**
3: **set** $RC_j = 0, T_j = \emptyset$
4: **while** $T \neq \emptyset$ **do**
5: Let $\tau_i \in T$ be the task with the highest utilization factor
6: **set** $T = T - \{\tau_i\}, RC = RC + u_i$
7: **if** $RC > 100\%$ **then return** failure
8: Let LP_x be the LP having the lightest workload
9: **set** $T_x = T_x \cup \{\tau_i\}, RC_x = RC_x + u_i$
10: **return** $\{(T_1, RC_1), (T_2, RC_2), \ldots, (T_M, RC_M)\}$

Theorem 1: Algorithm TASK-PARTITIONING can schedule any task set with a utilization no more than 100%, if ε is a non-negative value.

Theorem 2: If a task set T satisfies $u_i \leq \frac{\sum_{\forall \tau_i \in T} u_i}{M}$ for any task τ_i in T, where M is the maximal number of a-LPs that the SMT processor can support, the task partitioning of T divided by Algorithm TASK-PARTITIONING (i.e. T is partitioned into M subsets {(T_1,RC_1), (T_2,RC_2), . . . , (TM,RCM)}) would satisfy the following equation:

$$\frac{\max_{i=1}^{M} RC_i}{\min_{i=1}^{M} RC_i} \leq 2$$

- ***Emulation of Different Speed Ratios***

We extend the TASK-PARTITIONING algorithm for a non-adjustable SMT processor, where no task migration is allowed. A partition of tasks and the speed ratios (i.e., RC_j) are obtained from Algorithm *TASK-PARTITIONING*. The *EDF* algorithm is still adopted to schedule the tasks in each LP. The technical problem is when to activate and de-activate an LP such that all the LPs "virtually" operate at the originally assigned speeds as the LPs in an adjustable SMT processor and the schedulability of the tasks can be maintained.

DEFINITION 3: The MAX-MTL problem: Given two constants α and β, and a task set $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ running over a non-adjustable SMT machine with up to M logical processors, the goal of MAX-MTL is to find a schedule having the context switch number no more than α, and the MTL is no less than β.

Theorem 3: MAX-MTL problem is an NP-hard problem.

We call a time duration as an intensive-competition duration(ICD) if there are m' LPs with ready tasks, and the speed ratio, i.e., RC_j , of any of those m' LPs is higher than 1/m'. The idea of Algorithm SPEED-EMULATION is to operate all of the

LPs with ready tasks in each non-ICD (a duration which is not an ICD). It is because sufficient processing speed has been secured for each LP. During each ICD, the LPs could be activated and de-activated to guarantee the virtual speeds of the LPs respect to the assigned speed ratios of these LP.

Algorithm SPEED-EMULATION
Input: A number M of logical processors
A task partition $\{(T_1, RC_1), \ldots, (T_M, RC_M)\}$
The ending time $stop$
Output: A scheduling trace $ResSched$ of all logical processors before time $stop$

1: **set** $T = T_1 \cup T_2 \cup T_3 \ldots T_M$; $t_{now} = 0$; $ResSched = \emptyset$
2: **while** $t_{now} < stop$ **do**
3: Let LP' be the set of LPs having ready tasks at time t_{now}
4: **set** $t_{next} = NextSchedulingPoint(t_{now}, T)$
5: **set** m' = the cardinality of LP'
6: **if** $(\max_{\forall LP_j \in LP'} RC_j) < 1/m'$ **then**
7: **for all** $LP_j \in LP'$ **do**
8: Let τ_i be the highest priority ready-task in T_j
9: **set** $sched_i = sched_i \cup \{(t_{now}, t_{next})_j\}$
10: **else**
11: **set** IcdLength $= t_{next} - t_{now}$
12: **set** MinRc $= \min_{\forall LP_j \in LP'} RC_j$ {MinRc is the utilization factor of the lightest-loaded LP in LP'}
13: **set** SharedPL = IcdLength $\times$ MinRc $\times m'$
14: {schedule all tasks to run simultaneously in SharedP}
15: **for all** $LP_j \in LP'$ **do**
16: Let τ_i be the highest-priority ready task in T_j
17: **set** $sched_i = sched_i \cup \{(t_{now}, t_{now} + \text{SharedPL})_j\}$
18: **set** $t_{now} = t_{now} + \text{SharedPL}$
19: {schedule all tasks to run in "one by one" fashion in SoloP}
20: **for all** $LP_j \in LP'$ **do**
21: Let τ_i be the highest priority ready task in T_j
22: **set** $\text{SoloPL}_i = \text{IcdLength} \times (RC_j - \text{MinRc})$
23: **set** $sched_i = sched_i \cup \{(t_{now}, t_{now} + \text{SoloPL}_i)_j\}$
24: **set** $t_{now} = t_{now} + \text{SoloPL}_i$
25: **set** $t_{now} = t_{next}$
26: **for all** $\tau_i \in T$ **do**
27: **set** $ResSched = ResSched \cup \{sched_i\}$

Example 1 SPEED-EMULATION:

Consider a set of 4 tasks *T = {τ1 = (0, 1, 10), τ2 = (0, 1, 12), τ3 = (0, 2, 8), τ4 = (0, 3.15, 9)}*. The partition obtained from Algorithm *TASK-PARTITIONING* is *{(T1 = {τ1, τ2}, 18.33%), (T2 = {τ3}, 25%), (T3 = {τ4}, 35%)}*. Let the end time of the emulation be 10.91. The execution trace is illustrated in Fig. 3.

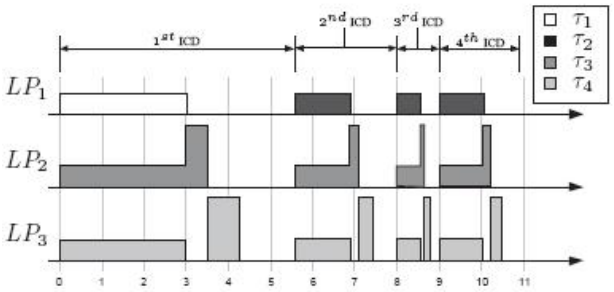

Figure 3. An illustration of speed emulation.

	1st ICD	2nd ICD	3rd ICD	4th ICD
Duration (t_{now}, t_{next})	(0, 5.45)	(5.45, 8)	(8, 9)	(9, 10.91)
SharedPL	3.00	1.4	0.55	1.05
SoloPL_1	0	0	0	0
SoloPL_2	0.36	0.17	0.07	0.13
SoloPL_3	0.91	0.43	0.16	0.32

Table 1. The schedule in Fig. 3.

Theorem 4: Any task set T_j assigned by Algorithm TASK-PARTITIONING to a logical processor LP_j is schedulable by EDF in LP_j with the speed ratio (i.e., RC_j) emulated by Algorithm SPEED-EMULATION.

Theorem 5: Given a task set T = {τ1, τ2, ... , τn} and a maximum number M of activated logical processors, Algorithm SPEED-EMULATION could achieve the following approximation bound if there is always at least one ready task in each task group T_j at any NextSchedulingPoint and $u_i \leq \frac{\sum_{\forall \tau_i \in T} u_i}{M}$ *for any task τi in T:*

$$\frac{MTL(\text{SPEED-EMULATION})}{MTL(OPT)} \geq \frac{M+1}{M \times 2}$$

where MTL(SPEED-EMULATION) and MTL(OPT) denote the MTL of a schedule derived by Algorithm SPEEDEMULATION and the maximum MTL of all schedules, respectively, and n ≥ M.

- ***Real-Time Scheduling with Task Migration***

Without task migration, it is possible that some

LPs are idle while some ready tasks are not being served by any LPs. Algorithm *SPEED-EMULATION-MIGRATION* is an extension of Algorithm *SPEED-EMULATION*. It includes a migration mechanism to move a ready task at an LP (i.e., an LP with more than one ready tasks) to another LP without any ready task. In Algorithm *SPEEDEMULATION-MIGRATION*, Steps 6 to 8 are to locate ready tasks that will not be dispatched by Algorithm *SPEEDEMULATION* for execution, i.e., the tasks in T'. Steps 11 to 13 are to migrate each task in T' to an LP that will be idle and de-activated by Algorithm *SPEED-EMULATION*. The While loop beginning at Step 3 is completed by executing Steps 7 to 26 of Algorithm *SPEED-EMULATION* that are for the emulation of the speed ratios of the LPs. Algorithm *SPEED-EMULATION-MIGRATION* then reports the final schedule by executing Steps 27 to 29 of Algorithm *SPEED-EMULATION*. The time complexity of Algorithm *PEED-EMULATION-MIGRATION* remains as that of *SPEED-EMULATION*.

EXAMPLE 2 SPEED-EMULATION-MIGRATION**:**

Consider the case in which the system has 6 tasks $T = \{\tau_1 = (0, 1, 6), \tau_2 = (0, 1, 6), \tau_3 = (2, 1, 6), \tau_4 = (2, 1, 6), \tau_5 = (4, 1, 6), \tau_6 = (4, 1, 6)\}$, the processor can support up to 3 LPs, and stop = 6. Let the resulted partition obtained from Algorithm *TASK-PARTITIONING* be $\{(T_1 = \{\tau_1, \tau_2\}, 0.33), (T_2 = \{\tau_3, \tau_4\}, 0.33), (T_3 = \{\tau_5, \tau_6\}, 0.33)\}$. The schedule derived from Algorithm *SPEED-EMULATION* is presented in Fig. 4.a, and the MTL of the schedule is 1. At time 0, Algorithm *SPEEDEMULATION-MIGRATION* discovers that there are 2 ready tasks, but there is only one task being dispatched for execution (Step 10). The algorithm migrates τ_2 from LP1 to LP2 during the interval (0, 2). The migration occurs again and again during the intervals (2, 4) and (4, 6), as shown in Fig. 4.b. The MTL of the resulted schedule is 2.

Algorithm SPEED-EMULATION-MIGRATION
Input and Output: The input and the output of this algorithm are as the same as those of Algorithm SPEED-EMULATION

1: **set** $T = T_1 \cup T_2 \cup T_3 \ldots T_M$
2: **set** $t_{now} = 0$
3: **while** $t_{now} < stop$ **do**
4: Let LP' be the set of logical processors having ready tasks at time t_{now}
5: Let T' be the set of ready tasks in T at time t_{now}
6: **for all** $LP_j \in LP'$ **do**
7: Select the task τ_i with the highest priority in T_j
8: **set** $T' = T' - \{\tau_i\}$
9: **set** m' = the cardinality of LP'
10: **while** $T' \neq \emptyset$ **and** $m' < M$ **do**
11: pick up a task τ_i in T' and **set** $T' = T' - \{\tau_i\}$
12: migrate τ_i to a logical processor $LP_j \in LP - LP'$ until next scheduling point (this value will be determined in Step 14).
13: **set** $LP' = LP' \cup \{LP_j\}$ and **set** $m' = m' + 1$
14: **set** $t_{next} = NextSchedulingPoint(t_{now}, T)$
15: Steps from 6 to 25 in Algorithm SPEED-EMULATION
16: Steps from 26 to 27 in Algorithm SPEED-EMULATION

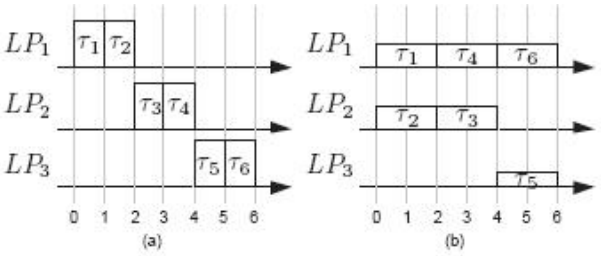

Figure 4. An illustration of task migration.

Theorem 6: Any task set T_j assigned by Algorithm TASK-PARTITIONING to a logical processor LP_j is schedulable by EDF in LP_j with the speed ratio (i.e., RC_j) emulated by Algorithm SPEED-EMULATION-MIGRATION

.

- ***Performance Evaluation***

We compared the performance of Algorithm *SPEED-EMULATION* (referred to as MT-Emu), Algorithm *SPEED-EMULATION-MIGRATION*

(referred to as MT-Emu/Mig) with the Algorithm *EDF* without SMT (referred to as ST) and a soft real-time SMT scheduling algorithm called *GLOB-NOSYM-PLAIN* [6] (referred to as MT-SingQ). In ST, we applied EDF to schedule tasks one at a time without any MTL consideration. In MT-SingQ, we chose the M tasks with the earliest deadlines for processing in parallel over an SMT processor that could support up to M LPs. While the schedulability of the task sets were explored in the previous sections, we should evaluate the capability of the proposed algorithms on the maximizing of the performance of the SMT processor in this section.

The primary performance metric was the MTL. The secondary metric was the average duration of the idle time. The average duration of the idle time was used to assess the potential improvement on power and response time. The test data sets were generated with the following parameters: Task sets were randomly generated with periods ranged from 1 to 100 and utilization factors between 0% and 100%. The number of tasks in each task set for each experiment were 30. The summation of the utilization factors ranged from 70% to 130%. The maximum number of LPs in the experiments was 8. Each experiment was repeated for 10 times, and their resulted MTL were averaged. The simulation time was 1, 000, 000 time units. The speedup ratio of was based on a well-known model in [8], where the speedup ratio was a function of the number of a-LPs as shown in Table 2.

# of a-LPs	1	2	3	4	5	6	7	8
ϵ (%)	0	86	153	213	250	280	300	306

Table 2. Speedup ratios.

Fig. 5 shows the MTL of the four algorithms when the workload varied. The performance of MT-Emu/Mig and MT-SingQ was almost the same, and MT-Emu/Mig outperformed MT-Emu. It was shown that task migration did improve the MTL of schedules significantly. ST had the worst performance because no SMT benefit was obtained. When the system load exceeded 100%, all algorithms were running without any timing guarantee. In general, the performance improvement continued when the workload increased. Although EDF was adopted in the experiments and the algorithm designs, many observations obtained in this section would remain valid for fixed-priority scheduling algorithms (e.g., rate monotonic [4]). We must point out that MT-Emu/Mig achieved the same performance with an excellent soft real-time greedy algorithm such as MTSingQ even though one of the objectives in the design of MT-Emu/Mig was for the schedulability analysis of task sets. Moreover, the MTL improvement shown in Fig. 5.a and the idle time improvement shown in Fig. 5.b show the strong relationship between these two performance metrics.

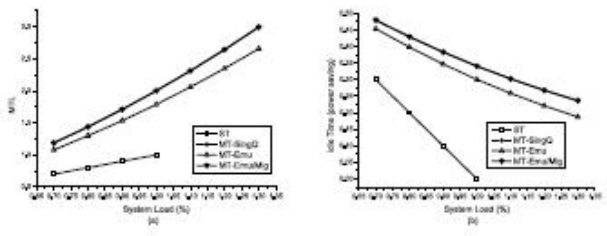

Figure 5. MTL versus System Loads

Fig. 6 shows the MTL of the algorithms when the number of LPs is varied. The performance

of MT-Emu/Mig and MT-SingQ remained the same, and they still outperformed MT-Emu. When the number of LPs increased, the performance gap between MT-Emu/Mig and MT-Emu decreased. It is because the speedup ratio of the SMT processor is increased when the number of LPs is increased. Consequently, the load of each LP becomes light. However, we surmised that the performance gap would remain while the number of LP is increased.

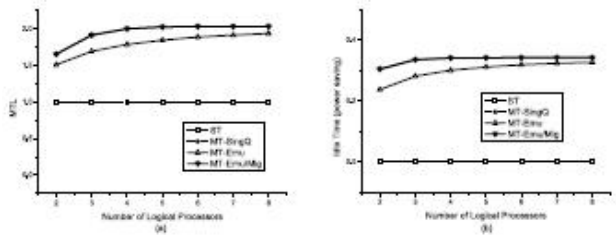

Figure 6. MTL versus the Number of LPs.

四、快閃記憶體有效即時辨別熱門資料

- ***Problem Definition***

Flash memory is usually accessed by embedded systems as a raw medium or indirectly through a block-oriented device. In other words, the management of flash memory is carried out by either software on a host system (as a raw medium) or hardware circuits/firmware inside its device.

A flash memory chip is usually partitioned into blocks of a fixed size, and each block is further partitioned into a fixed number of pages, where pages are basic write-operation units. A typical block size and a typical page size are 64KB and 512B, respectively. Flash memory has several unique characteristics that introduce challenges for the management issues: (1) write-once with bulk erases (2) wear-levelling. Data over flash memory must be written to free space. That is, when a page is written (/programmed), the space is no longer available unless it is erased. Out-place-updating is usually adopted to avoid erasing operations on every update. The effective (/latest) copy of data is considered as "live", and old versions of the data are invalidated and considered as "dead". Note that live and old versions of data might co-exist over flash memory simultaneously. Pages which store live data and dead data are called "live pages" and "dead pages", respectively. After the processing of a large number of page writes, the number of free pages on flash memory would be low. System activities (called garbage collection) are needed to reclaim dead pages scattered over blocks so that they could become free pages. As a result, a potentially large amount of live data might be copied to available space before a to-be-recycled block is erased. Since a flash-memory block has a limitation on the count of erases, a worn-out block could suffer from frequent write errors. "Wear-levelling" activities is thus needed to erase blocks on flash memory evenly.

Layered designs are usually adopted for the implementations of flash-memory storage systems, regardless of hardware or software implementations of certain layers. The Memory Technology Device (MTD) driver and the Flash Translation Layer (FTL) driver are the two major layers for flash-memory management, as shown in Figure 7. The MTD driver provides lower-level functionalities of a storage medium, such as read, write, and erase. Based on these services, higher-level management algorithms, such as wear-levelling, garbage collection, and physical/logical address translation, are implemented in the FTL driver. The objective of the FTL driver is to provide transparent services for user applications and file systems to access flash memory as a block-oriented device. An alternative approach is to combine the functionalities of an FTL and a file system to realize a native flash-memory

file system, such as JFFS [24]. Regardless of which approach is taken, how to provide an efficient FTL implementation is always a challenging and critical issue for flash-memory storage/file systems.

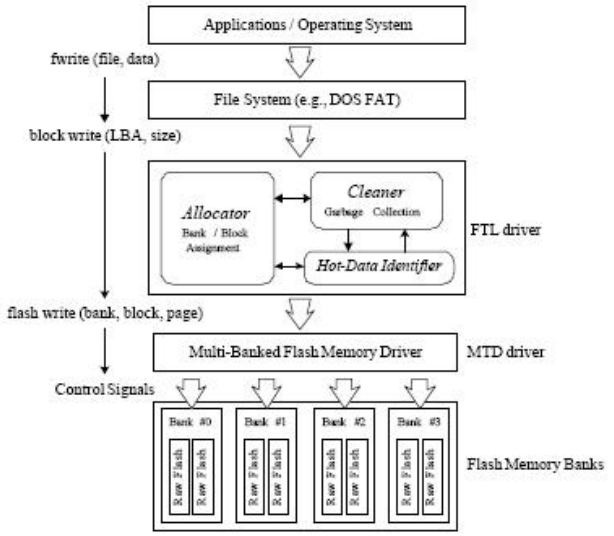

Figure7: A typical system architecture for flash-memory storage systems

The implementation of an FTL driver could consist of an allocator and a cleaner. The allocator is responsible to the finding of proper pages on flash memory to dispatch writes, and the cleaner is responsible to the reclaiming of pages with invalidated data, where space reclaiming is referred to as garbage collection. One important implementation issue for flash-memory management is wear-levelling, which is to evenly distribute the number of erasing for each block (because of the limitation on the number of erasing for blocks, e.g., 106). A proper design for the allocator and the cleaner could not only improve the performance of a flash-memory storage system but also increase its life time.

- ***On-line Locality Tracking***

We propose to adopt K independent hash functions to hash a given LBA into multiple entries of a M-entry hash table to track the write number of the LBA, where each entry is associated with a counter of C bits (Please refer to Table 1 for the definition of symbols). Whenever a write is issued to the FTL, the corresponding LBA is hashed simultaneously by K given hash functions. Each counter corresponding to the K hashed values (in the hash table) is incremented by one to reflect the fact that the LBA is written again. If a counter reaches its maximum value, it is left unchanged. Note that we do not increase any counter for a read because there is no invalidation of any page for a read. For every given number of sectors have been written, called the "decay period" of the write numbers, the values of all counters are divided by 2 in terms of a right shifting of their bits. It is an aging mechanism to exponentially decay the values of all write numbers as time goes on. Whenever an LBA is to be verified as a location for hot data, the LBA is also hashed simultaneously by the K hash functions. We say that the LBA contains hot data if the H most significant bits of every counter of the K hashed values contain a non-zero bit value.

System Model Parameters	Notation
Number of Hash Functions	K
Size of Counter	C
Write Counts for an LBA to Become Hot	$2^{(C-H)}$
Number of Counters in a Hash Table	M
Number of Write References	N
Ratio of Hot Data in All Data ($< 50\%$)	R

Table3: Notations of the system model parameters

Figure 3.(a) shows the increment of the counters that correspond to the hashed values of K hash functions for a given LBA, where there are four given independent hash functions, and each counter is of four bits. Figure 3.(b) shows the hot-data identification of an LBA, where only the first two most significant bits of each counter is considered to verify whether the LBA corresponds to hot data. The rationale behind the adopting of K independent hash functions is to reduce the chance for the false

identification of hot data. Because hashing tends to randomly maps a large address space into a small one, it is possible to falsely identify a given LBA as a location for hot data. With multiple hash functions adopted in the proposed framework, the chance of false identification might be reduced. In addition to this idea, the adopting of multiple independent hash functions also helps in the reducing of the hash table space, as indicated by Bloom [16].

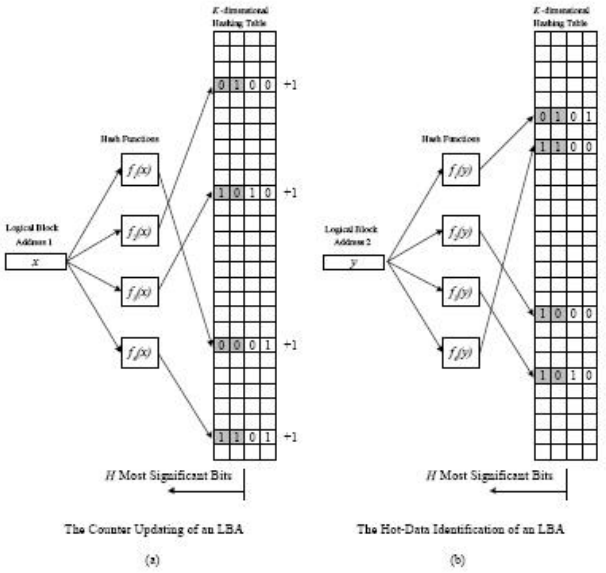

Figure8: The counter updating and the hot-data identification of an LBA, where C = 4, K = 4, and H = 2.

Instead of enlarging the hash table to improve false identification, we propose to increase only counters of the *K* hashed values that have the minimum value to improve false identification (Please refer to Table 1 for the definition of symbols). The rationale behind the counter-increment policy is as follows: The reason for false identification is because counters of the K hash values of a non-hot LBA are increased by non-hot data writes, due to hashing collision. If an LBA is for hot data, then the policy in the increasing of small counters for its writes would still let all of the K counters corresponding to the LBA go over $2^{(C-H)}$ (because other writes would make up the loss in counter increasing). However, if an LBA is for non-hot data, then the policy would reduce the chance of false identification because a less number of counters will be falsely increased due to collision. We shall show in the experiments how much performance improvement could be obtained, compared to the basic framework proposed in Section 3.1.

The revised policy in counter increasing would introduce extra time complexity in the hot-data verification of each LBA because of the locating of counters with the minimum value. The revised policy would certainly increase the implementation difficulty of the algorithm with a certain degree, regardless of whether this algorithm is implemented in software, firmware, or even hardware.

- ***Performance Evaluation***

The proposed multi-hash-function framework, the direct address method, and the two-level LRU list method were evaluated over an Intel Pentium4 2.40GHz platform with 248MB RAM. The hot-data-LBA and candidate-LBA lists of the two-level LRU list method could have up to 512 and 1024 nodes, respectively. Two hash functions were adopted for the proposed multi-hash-function framework. Each counter for a hash-table entry was of 4-bits, and the number of hash-table entries ranged from 2048 to 10240.

The trace of data access for performance evaluation was collected over a mobile PC with a 20GB hard disk, 384MB RAM, and an Intel Pentium-III 800MHz processor. The operating system was Windows XP, and the hard disk was formatted as NTFS. In order to emulate a 512MB flash memory storage system, whose only LBAs within a range of 512MB in the trace was extracted. The hot ratio *R* of the

workload was set as 20%. Since $N \leq M / (1-R)$, the number of writes for each decay was set as 5117 for a 4096-entry hash table (Please refer to Table 3 for the definition of symbols). The same number of writes for each decay was adopted for other hash-table sizes for comparisons. Figure 9 shows the ratio of hot data to all data with respect to the number of writes that had been executed. The figure was derived based on the direct address method (because there was no false hot-data identification, due to hash collision). As shown in the figure, the ratio of hot data to all data varied between 10% and 30%, and the ratio remained around 20% most of time. Note that the ratio dropped to a very low number at the end of the trace. We would address the impacts on the proposed framework later.

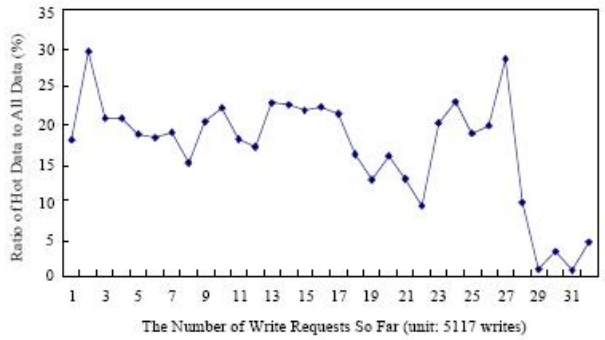

Figure 9: The locality in data access (decaying period: 5117 writes, hot-data threshold on the number of writes to an LBA: 4)

Figure 10 shows the ratio of false hot-data identification for the multi-hash-function framework (denoted as basic in the figure) and the framework with an enhanced counter update policy (denoted as enhanced in the figure), compared to the direct address method. Let X be the number of LBA's being identified for non-hot data by the direct address method but being identified for hot data by the (basic/enhanced) multi-hash-function framework for every 5117 writes. Y was 5117. The ratio of false hot-data identification for the (basic/enhanced) multi-hash-function framework was defined as *(X/Y)*. As shown in Figure 10, the enhanced multi-hash-function framework outperformed the basic multi-hash-function framework. Note that there were some peaks for lines in Figure 10. It was because the ratio of hot data to all data varied in Figure 9. Note that when the ratio of hot data to all data dropped significantly (e.g., when the number of writes was around (22 x 5117 = 112574), the ratio of false identification increased. However, as the values of counters in the hash table were decayed, false identifications of hot data were gradually reduced.

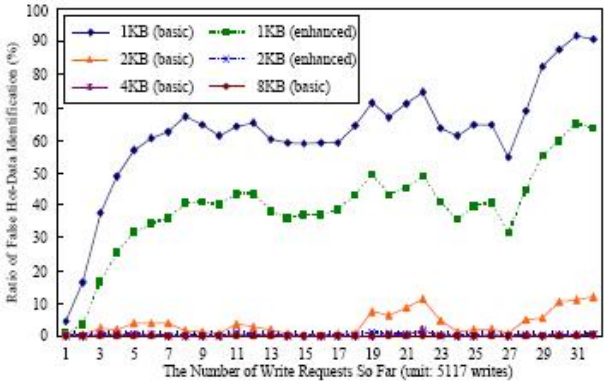

Figure 10: Ratio of false identification for various hash-table sizes

Figure 11 shows the performance gap achieved by the framework and the direct address method, when the decay period ranged from twice of the original setup to 1/4 of the original setup. We should point out that when the decay period was too large, the chance of false hot-data identification might increase more than expected because the results of "incorrect" counter increments would be accumulated. If

we had to set the decay period as a unreasonably large number, then we should have a large hash table!

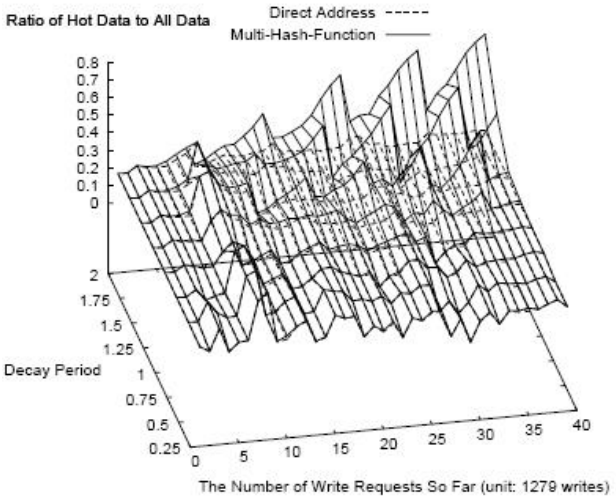

Figure 11: The performance gap achieved by the multi-hash-function framework and the direct address method

	Multi-Hash-Function Framework		Two-Level LRU List [2]	
	Average	Standard Deviation	Average	Standard Deviation
Checkup	2431.358	97.98981	4126.353	2328.367
Status-Update	1537.848	45.09809	12301.75	11453.72
Decay	3565	90.7671	N/A	N/A

Table 4: CPU cycles per operation (Unit: CPU cycles)

The third part of experiments was to evaluate the runtime overheads of the (basic) multi-hash-function framework, compared with a two-level LRU list method [17]. RDTSC (read time-stamp counter), that was an Intel supported instruction [21], was used to measure the required CPU cycles. In the experiments, the multi-hash-function framework adopted a 2KB hash table with 4096 entries. Table 2 shows the run-time overheads for each operation of the experimented methods, where "Checkup" means the verification of whether an LBA is for hot data, "Status-Update" means the updating of the status of an LBA, and "Decay" means the decaying of all counters. The "Status-Update" operation of the two-level LRU list method was the insertion of the LBA into the two lists. The "Status-Update" operation of the multi-hash-function framework was the increments of counters. It was shown that the "Checkup" overheads of the multi-hash-function framework was about 1/2 of that of the two-level LRU list method. The "Status-Update" overheads of the multi-hash-function framework was about 1/8 of that of the two-level LRU list method. We must point out that the standard deviation of the run-time overheads for the multi-hash-function framework was much smaller, compared with that for the two-level LRU list method. Beside the reducing of run-time overheads, the "Decay" overheads of the multi-hash-function framework was only slightly more than 2 times of that for the "Status-Update" overheads.

五、結果與討論

In this research, we explore the performance in the real-time process with adjustable factors for the processors and storage devices. We first propose a real-time SMT scheduling algorithm for real-time tasks running on adjustable SMT processors. We then propose a speed emulation algorithm to emulate LPs with different speed ratios over those with equal speed ratio. The schedulability analysis and approximation bound are presented. The speed emulation algorithm is further improved in the maximization of the number of activated logical processors. The performance of the proposed algorithms is evaluated by comparing the algorithms with a good soft real-time SMT scheduling algorithm, such as [21]. Hot data identification has been an important issue in the performance study for flash-memory storage

systems. It not only imposes great impacts on garbage collection but also could significantly affect the performance of flash-memory access and its life time (due to wear-levelling). In this research, we propose a highly efficient method for on-line hot-data identification with limited space requirements. Different from the past implementations, a multi-hash-function framework is proposed, in which multiple independent hash functions are adopted to reduce the chance of false identification of hot data and provide predictable and excellent performance for hot-data identification. A series of experiments was conducted to verify the performance of the proposed method, it was shown that the proposed framework with very limited RAM space could perform closely to an nearly ideal method.

六、參考文獻

[1] Z. Deng and J. W.-S. Liu. "Scheduling Real-Time Applications in an Open Environment," In Proc. of the 18th IEEE Real-Time Systems Symposium, IEEE Computer Society Press, Dec. 1997.

[2] G. K. Dorai and D. Yeung and S. Choi. "Optimizing SMT Processors for High Single-Thread Performance," Journal of Instruction-Level Parallelism, Vol.5, pp. 1–35, Apr. 2003.

[3] M. R. Garey and D. S. Johnson. "Computers and intractability: A guide to the theory of NP-completeness," W.H. Freeman and Co, 1979.

[4] C.L. Liu and Layland. "Scheduling algorithms for multiprogramming in a hard real-time environment," J. Assoc. Comput. Mach., vol. 20, pp.46–61, 1973.

[5] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, M. Upton. "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, pp. 4–15, Feb. 2002.

[6] R. Jain, C. Hughes, and S. Adve. "Soft real-time scheduling on simultaneous multithreaded processors," In Proc. of the 23rd IEEE Real-Time Systems Symposium, pp. 134–145, Dec. 2002.

[7] S. E. Raasch and S.K. Reinhardt. "Applications of Thread Prioritization in SMT Processors," In Proc. of the Workshop on Multithreaded Execution and Compilation, 1999

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," In Proc. of the 23rd Annual International Symposium on Computer Architecture, pp. 191–202, May 22–24, 1996.

[9] J. V. Busquets-Mataix and J. J. Serrano and A. Wellings. "Hybrid Instruction Cache Partitioning for Preemptive Real- Time Systems," In Proc. of the 9th Euromicro Workshop on Real Time Systems, Jun. 1997.

[10] D. B. Kirk. "SMART (Strategic Memory Allocation for Real-Time Cache Design)," In Proc. 10th Real-Time Systems Symp., pp. 229–237, Dec. 1989.

[11] S. Kaxiras, G. Narlikar, A. D. Berenbaum, and Z. Hu. "Comparing Power Consumption of an SMT and a CMP DSP for Mobile PhoneWorkloads," In Proc. of the Intl. Conf. on Compilers, Arch. and Synthesis for Embedded Systems, 2001.

[12] J. Seng, D. Tullsen, and G. Z. N. Cai. "Power-Sensitive Multithreaded Architecture," In Proc. of the 7th ASPLOS, 1996.

[13] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernandez, Alex Ramirez, Mateo Valero. "Predictable performance in SMT processors," In Conf. Computing Frontiers 2004.

[14] Gautham Thambidorai, Donald Yeung, Seungryul Choi. "Optimizing SMT Processors for High Single-Thread Performance," Jornal of Instruction-Level Parallelism, vol. 5, 2003.

[15] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil Rotenberg, E. Mueller, F. "Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems," In Proc. of the 30th

Annual International Symposium on Computer Architecture, pp. 350–361, 9–11 June 2003.

[16] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," Communications of the ACM, Vol. 13, No. 7, July 1970, pp. 422-426.

[17] L. P. Chang and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," 8th IEEE RTAS, September 2002, pp. 187-196.

[18] L. P. Chang and T. W. Kuo, "An Efficient Management Scheme for Large-Scale Flash-Memory Storage Systems," ACM SAC, 2004.

[19] M. L. Chiang, Paul C. H. Lee, and R. C. Chang, "Managing Flash Memory in Personal Communication Devices," Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97), Singapore, Dec. 1997, pp. 177-182.

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms, Second Edition" The MIT Press, 2001.

[21] I. Coorporation, "Using the RDTSC Instruction for Performance Monitoring," tech. rep., Intel Coorporation, 1997.

[22] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory based File System," Proceedings of the 1995 USENIX Technical Conference, January 1995, pp. 155-164.

[23] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile Main Memory Storage System," Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, 1994, pp. 86-97.

[24] D. Woodhouse, Red Hat, Inc. "JFFS: The Journaling Flash File System".

[25] Aleph One Company, "Yet Another Flash Filing System".

[26] L. P. Chang, and T. W. Kuo, "A Dynamic-Voltage-Adjustment Mechanism in Reducing the Power Consumption of Flash Memory for Portable Devices," IEEE Conference on Consumer Electronic (ICCE 2001), LA. USA, June 2001.

[27] L. P. Chang, T. W. Kuo,"A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems," The 8th International Conference on Real-Time Computing Systems and Applications (RTCSA 2002), 2002.

[28] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".

[29] SSFDC Forum, "SmartMediaTM Specification", 1999.

[30] Compact Flash Association, "CompactF lashTM 1.4 Specification," 1998.

[31] M-Systems, Flash-memory Translation Layer for NAND flash (NFTL).