Partitioning and Allocation of Scratch-Pad Memory for Priority-Based Preemptive Multi-Task Systems

Hideki Takase[†*], Hiroyuki Tomiyama[†] and Hiroaki Takada[†]
[†]Graduate School of Information Science, Nagoya University
C3-1 (631), Furo-cho, Chikusa-ku, Nagoya, 464-8603 Japan
Email: {takase, tomiyama, hiro}@ertl.jp
[*]Japan Society for the Promotion of Science

***Abstract*—Scratch-pad memory has been employed as a partial or entire replacement for cache memory due to its better energy efficiency. In this paper, we propose scratch-pad memory management techniques for priority-based preemptive multi-task systems. Our techniques are applicable to a real-time environment. The three methods which we propose, *i.e.*, spatial, temporal, and hybrid methods, bring about effective usage of the scratch-pad memory space, and achieve energy reduction in the instruction memory subsystems. We formulate each method as an integer programming problem that simultaneously determines (1) partitioning of scratch-pad memory space for the tasks, and (2) allocation of program code to scratch-pad memory space for each task. It is remarkable that periods and priorities of tasks are considered in the formulas. Additionally, we implement an RTOS-hardware cooperative support mechanism for a runtime code allocation to the scratch-pad memory space. We have made the experiments with the fully functional real-time operating system. The experimental results with four task sets have demonstrated the effectiveness of our techniques. Up to 73 % energy reduction compared to a standard method was achieved.**

I. Introduction

One of the most serious considerations in the modern embedded real-time systems is the excessive energy consumption. Minimizing energy consumption contributes to decrease cooling and operating costs of chips, to extend battery lifetime in portable systems, and to increase system reliability. These days, cache memories have become employed not only in general-purpose processors but also in embedded processors. Caches can improve average performance of processors by exploiting the locality of memory references in a program. Caches also contribute to energy reduction by decreasing the amount of accesses to off-chip memory. However, cache becomes one of the most energy-hungry components in the embedded processors. For example, the ARM920T processor dissipates 43 % of its power in caches [1]. More recently, scratch-pad memory (SPM) has attracted attention due to its better energy efficiency than cache.

A number of techniques have been proposed so far for effective usage of SPM in terms of energy consumption and/or performance. SPM consists of only decoding circuits, data arrays, and output units. Unlike in caches, it does not require tag comparison on SPM. Due to its simplified architecture, SPM is more energy/area efficient than cache. SPM also contributes to better predictability of real-time performance since an unanticipated access miss (like a cache miss) does not occur. On the other hand, programmers or compilers have to decide about the allocation to the SPM space since SPM does not have a hardware mechanism for code/data replacement.

As the scale and complexity of the embedded systems increase, the embedded processors are generally required to execute multiple tasks concurrently. Embedded multi-task systems are classified as soft real-time or hard real-time systems depending on the importance of deadline constraints. Soft real-time systems are typically used where the real-time performance is not so important. Task deadline miss causes in degraded quality, but the system can continue to operate the execution of a task. In hard real-time systems, on the other hand, high responsiveness is indispensable for systems to work correctly. The completion of a processing after its deadline is considered useless. To guarantee task deadline constraints, a priority-based preemptive scheduling algorithm is employed in the hard real-time systems.

In this paper, we propose three methods for SPM partitioning and code allocation, named spatial, temporal, and hybrid methods. Spatial method means that each task has its exclusive space in SPM. Temporal means that a running task can use the entire SPM space. The content of SPM is swapped under the cooperative support of the real-time operating system (RTOS) and hardware module on the task context switch. The hybrid method is basically spatial method but a higher priority task can temporarily use the space of lower priority tasks. Each method is formulated as an integer programming problem that can determine the energy efficient partitioning and allocation of the SPM space. The contribution of this work is that energy efficient usage of SPM is confirmed in the priority-based preemptive multi-task systems. It is also remarkable that our methods consider an attribution of the task scheduling for minimizing energy consumption on the instruction memory subsystems.

The rest of this paper is organized as follows. Section II provides a brief survey on related works. Section III describes our SPM partitioning and code allocation techniques in detail. Section IV presents our experimental setup and results. Finally, Section V summarizes the contributions of this paper.

II. Related Works and Our Strategy

Considerable amount of researches on SPM have been conducted so far for energy and/or performance optimization.

Banakar et al. proposed an allocation technique by selecting an on-chip memory configuration from various size of cache

and SPM [2]. The authors of [3] proposed a compiler-oriented optimization approach to allocate data to SPM for performance improvement. The authors of [4] formulated the energy optimal code/data allocation to SPM as a 0/1 integer programming problem based on the size of SPM and the energy consumption of each function. Since the 0/1 integer programming problem is an NP-hard problem, several heuristic algorithms were proposed. The authors of [5] proposed the data allocation method based on the possibility of data-cache conflicts. In [6], a dynamic programming algorithm for deciding energy efficient code/data allocation of the SPM space was studied. The authors of [7] introduced a hardware mechanism for efficient code allocation to the SPM space at runtime. In [8], the customized instructions for the runtime code allocation were proposed. [9] proposed a hardware/software approach to manage the contents of SPM. However, these previous techniques are only applicable in the single-task systems.

Several techniques to improve performance or energy were appeared for the multi-task systems. Verma et al. proposed the SPM partitioning scheme among multiple tasks [10]. Each task uses a fixed amount of the SPM space for the purpose of energy minimization. [11] and [12] proposed RTOS-centric approaches to utilize the SPM space in the multi-task systems. These authors implemented the runtime SPM management mechanism in a tiny kernel, but the functionality of their kernel is limited. More noteworthy thing is that these above approaches assume the soft real-time environments where tasks are scheduled by the time-sharing round-robin manner. Applying these techniques to the hard real-time systems will bring in non-optimal energy savings.

In the case of hard real-time systems, priority-based scheduling is generally employed for the real-time performance. We have studied energy efficient SPM partitioning approaches to a priority-based multi-task systems in [13]. However, the work of [13] focused on the non-preemptive multi-task systems. If CPU utilization rate is high, the schedulability is hard to guarantee by non-preemptive scheduling. In this paper, we extend prior work to be applicable to preemptive multi-task environments. This work achieves energy efficient utilization of SPM in the hard real-time systems.

III. Partitioning and Allocation Techniques

This section describes our SPM partitioning and code allocation techniques in detail. Each method is formulated as the integer programming problem. It should be noted that our techniques use the attribution of the scheduling policy for the flexible utilization of the SPM space. Also, this work focuses on the energy reduction for instruction memory access, and code allocation is performed at the function-level granularity[1].

A. Target System Organization and Definition of Symbols

We focus on an environment where multiple tasks are executed on a single processor. The tasks take either of dormant, ready, and running states. The task scheduling algorithm is based on the rate-monotonic scheduling [14], that is, the fixed-priority based preemptive scheduling. If more than one ready task exists, the highest priority task will be in the running state, and the others in the ready state. Furthermore, a higher priority task can preempt the CPU when a lower priority one is running. All tasks are cyclically activated, and the periods and priorities of tasks are statically given. Also, there is neither inter-task communication nor synchronization.

TABLE I
Definitions of Symbols

Names	Definitions
$task_i$	The i-th task. $1 \leq i \leq M$
$period_i$	The arrival interval of $task_i$.
$priority_i$	The execution priority of $task_i$.
$func_{i,j}$	The j-th function in $task_i$. $1 \leq j \leq N$
$fetch_{i,j}$	The total number of executed instructions in $func_{i,j}$ per execution of $task_i$.
$size_{i,j}$	The code size of $func_{i,j}$.
$Esaving_{i,j}$	The energy reduction if $func_{i,j}$ is allocated to SPM.
$Eoverhead_{i,j}$	The energy consumption for transferring $func_{i,j}$ from the main memory to SPM.
$E_{SPMgain}$	The difference of energy consumption between SPM and cache on read access.
E_{C_read}	The energy on read access to cache.
E_{S_read}	The energy on read access to SPM.
E_{S_write}	The energy on write access to SPM.
E_{MM_read}	The energy on read access to the main memory.
$SPMsize$	The total size of SPM.
$hyperperiod$	The least common multiple of task periods.

Table I indicates the definitions of symbols used in our integer programming problems[2]. All parameters of symbols shown in Table I can be obtained statically.

B. Spatial Method

The spatial method partitions the SPM space for tasks statically. Each task exclusively uses a given SPM space while executing a task set. Fig. 1(a) shows an example for partitioning of the SPM space among three tasks. The amount of SPM partitioned to each task is decided depending on the energy savings of its functions. We consider the task periods for utilizing the SPM space as efficiently as possible. A short period task is likely to be given a large SPM space since the short period means high execution frequency. For example, Task1 in Fig. 1(a) occupies the large SPM space.

We formulate the partitioning and allocation of the spatial method as the following integer programming problem.

$$\textbf{Maximize}: Esaving = \sum_i \sum_j Esaving\_spt_{i,j} \cdot x_{i,j} \quad (1)$$

$$Esaving\_spt_{i,j} = fetch_{i,j} \cdot \frac{hyperperiod}{period_i} \cdot E_{gain} \quad (2)$$

$$E_{gain} = E_{C\_read} - E_{S\_read} \quad (3)$$

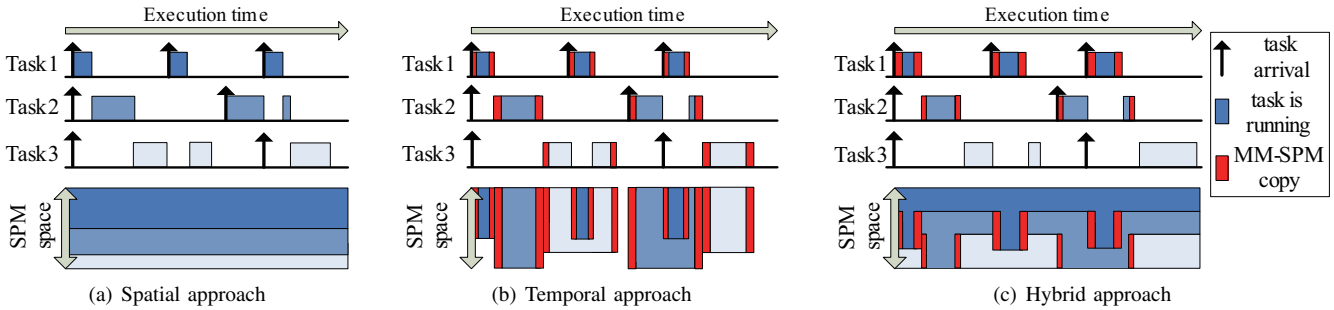
$$\textbf{s.t.}: \sum_i SPMsize\_spt_i \leq SPMsize \quad (4)$$

$$SPMsize\_spt_i = \sum_j size_{i,j} \cdot x_{i,j} \quad (5)$$

$$\textbf{s.t.}: \ \forall i, \forall j. \ \ x_{i,j} \leq 1 \quad (6)$$

[1]This work assumes that programs are written in a C language, and term "function" denotes a function of C program. Also, tasks call several functions.

[2]Note that the values of $fetch_{i,j}$ are obtained by instruction-level simulation in our work. However, $fetch_{i,j}$ may be varied depending on input data to the task. In that case, their expected values are obtained by giving a set of representative input data and running the task multiple times.



(a) Spatial approach (b) Temporal approach (c) Hybrid approach

Fig. 1. The SPM partitioning and code allocation approaches.

It aims at maximizing the total energy reduction $Esaving$. Here in the problem, $x_{i,j}$ denotes a binary variable whose value is 1 if $func_{i,j}$ is allocated to SPM, otherwise 0. $SPMsize\_spt_i$ denotes the size of SPM that is partitioned to $task_i$ by the spatial method. By finding these values, our integer programming problem simultaneously determines partitioning of the SPM space for the tasks and allocation of functions to the SPM space for each task.

C. Temporal Method

As shown in Fig. 1(b), the whole SPM space is assigned to currently running tasks in the temporal method. The functions frequently fetched in a running task are allocated to SPM. Instead, it is necessary to transfer program code of the task from the main memory to SPM twice per task activate interval.

1) When a task transits the running state for the first time at its arrival interval. The frequently accessed functions in a task are transferred to the SPM space.
2) When a task terminates and transits to the dormant state. The contents of the SPM are reallocated before its task transits the running state.

'MM-SPM copy' in Fig. 1(b) refers these transfers.

The integer programming formulation is as follows.

$$\textbf{Maximize}: Esaving = \sum_i \sum_j Esaving\_tmp_{i,j} \cdot y_{i,j} \quad (7)$$

$$Esaving\_tmp_{i,j} = fetch_{i,j} \cdot E_{gain} - Eoverhead_{i,j} \cdot 2 \quad (8)$$

$$E_{gain} = E_{C\_read} - E_{S\_read} \quad (9)$$

$$Eoverhead_{i,j} = size_{i,j} \cdot (E_{S\_write} + E_{MM\_read}) \quad (10)$$

$$\textbf{s.t.}: \ \forall i. \ \ SPMsize\_tmp_i \leq SPMsize \quad (11)$$

$$SPMsize\_tmp_i = \sum_j size_{i,j} \cdot y_{i,j} \quad (12)$$

$$\textbf{s.t.}: \ \forall i, \forall j. \ \ y_{i,j} \leq 1 \quad (13)$$

Here, $SPMsize\_tmp_i$ denotes the size of SPM that is partitioned to $task_i$ by the temporal method. $Eoverhead_{i,j}$ denotes the energy consumption for transferring $func_{i,j}$. The functions allocated to SPM are decided considering the amount of energy consumption on these transfers. The energy overhead for transferring a function is proportional to its code size. For this reason, functions with not only frequent access but also small size are likely to be allocated to the SPM space.

In the temporal method, the functions are dynamically allocated to SPM. Compared with the spatial method, the temporal method is effective when the locality of memory references in a task is high. Meanwhile, the temporal method is not effective if energy overhead on the code transfer is large.

D. Hybrid Method

The hybrid method is a mixture of the previous two methods. More energy reduction can be achieved by integrating the spatial and temporal methods. As shown in Fig. 1(c), the SPM space is basically partitioned to each task by the spatial method, but a higher priority task can temporarily use the space where lower priority tasks statically occupy. In other words, a higher priority task *preempts* not only the CPU but also the SPM space used by lower priority tasks. For example, Task1, which takes the highest priority in Fig. 1(c), uses both its own spatial space and preempted temporal space of Task2 and Task3 only when Task1 is in the running state.

In the integer programming problem of the hybrid method, the task priorities are positively utilized in addition to the task periods. The amount of the SPM space where a task uses is the sum of (i) the statically space partitioned by the spatial method and (ii) the temporary space where lower priority tasks use. The amounts of (i) and (ii) are determined so that the total energy reduction $Esaving$ becomes the largest. We formulated the hybrid method as follows.

$$\textbf{Maximize}: Esaving = Esaving\_spt + Esaving\_tmp \quad (14)$$

$$Esaving\_spt = \sum_i \sum_j Esaving\_spt_{i,j} \cdot x_{i,j} \quad (15)$$

$$Esaving\_spt_{i,j} = fetch_{i,j} \cdot \frac{hyperperiod}{period_i} \cdot E_{gain} \quad (16)$$

$$Esaving\_tmp = \sum_i \sum_j Esaving\_tmp_{i,j} \cdot y_{i,j} \quad (17)$$

$$Esaving\_tmp_{i,j} = (fetch_{i,j} \cdot E_{gain} - Eoverhead_{i,j} \cdot 2) \cdot \frac{hyperperiod}{period_i} \quad (18)$$

$$E_{gain} = E_{C\_read} - E_{S\_read} \quad (19)$$

$$Eoverhead_{i,j} = size_{i,j} \cdot (E_{S\_write} + E_{MM\_read}) \quad (20)$$

$$\textbf{s.t.}: \sum_i SPMsize\_spt_i \leq SPMsize \quad (21)$$

$$SPMsize\_spt_i = \sum_j size_{i,j} \cdot x_{i,j} \quad (22)$$

$$\textbf{s.t.}: \ \forall i. \ \sum_j size_{i,j} \cdot y_{i,j} \leq SPMsize\_tmp_i \quad (23)$$

$$\exists k, \ priority_k > priority_i. \quad SPMsize\_tmp_i = SPMsize - \sum_k SPMsize\_spt_k \quad (24)$$

$$\textbf{s.t.}: \ \forall i, \forall j. \ \ x_{i,j} + y_{i,j} \leq 1 \quad (25)$$

There are three constraints in the above formulation. First constraint determines the partitioning and allocation of the SPM space for the tasks in the spatial method. Second determines the partitioning and allocation of the SPM space where a higher priority task can temporarily use. The last restricts both two methods allocate same function to the SPM space. The decision variables are $x_{i,j}$, $y_{i,j}$, $SPMsize\_spt_i$, and $SPMsize\_tmp_i$. Their optimal values simultaneously achieve the partitioning of the spatial space for the tasks, the temporal space used by the higher priority task, and the allocation of functions to the SPM space for each task.

The hybrid method can use both the other two methods as the situation demands. Both inter-task and intra-task optimizations are simultaneously performed. The higher priority tasks predominately use the temporal method. On the contrary, the lower priority tasks mainly use the spatial method. Additionally, the more amount of program code can be allocated to the SPM space since the hybrid method partly employs the spatial method that does not need the dynamic code transfer. The temporal method is also used for dynamic code allocation in the hybrid method. Nevertheless, the total amount of energy overhead on the code transfer can be kept to the minimum to utilize the spatial space at the same time. Therefore, the hybrid method is the most stable method.

E. Runtime SPM Management

As described in the previous subsections, the temporal and hybrid methods have to transfer program code to the SPM space at the proper timing. To perform efficient transfer, we implement a runtime SPM manager to evaluative systems. Fig. 2 shows the workflow of our mechanism. This workflow consists of system design (left part of Fig. 2) and runtime SPM management (right part) phases.

We employed the SkyEye–1.2.6_rc1 [15] simulator as the target environment. The SkyEye is an instruction-set simulator that emulates various embedded processors. We chose the ARM920T [16] core in this work.

At first of the system design phase, an in-house profiler collects input parameters shown in Table I based on the instruction-level simulation. The profiling unit is just a task for deriving these pieces of information. Then, the optimal SPM partitioning and code allocation are determined statically at the compile time. We used the glpsol 4.23 [17] to solve our integer programming problems. In our environment, all integer programming problems could solved within one seconds. The GNU arm-elf environment [18] was used for cross development. The functions of tasks that are determined in the spatial space are allocated at the system design phase.

In the runtime SPM management phase, the functions are dynamically managed under the cooperative support of an RTOS and hardware. The RTOS notifies the ID number of the target task to the DMA controller on hardware, and then, the DMA controller operates the runtime code allocation appropriately. It should be noted that programmers do not have to make any modification of the program code of tasks since our framework automatically operates the SPM management.

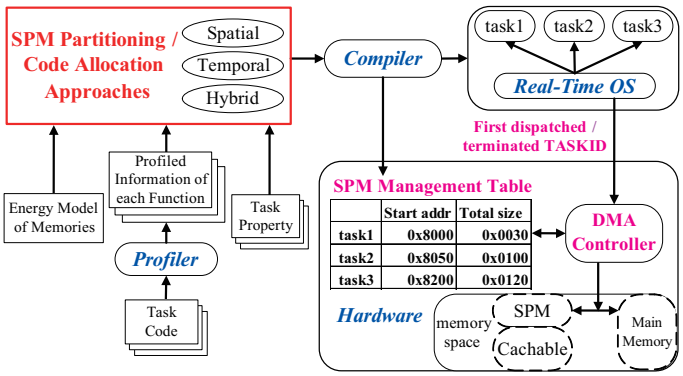


Fig. 2. The workflow of runtime SPM manager.

The RTOS detects the proper timing of code transfer on the task scheduling. Note that there are only two notification per one task's activate interval described in Section III-C. When the DMA controller receives the ID number, the required program code is transferred from the main memory to SPM along with the reference of a SPM management table. This table is generated statically at the compile time to assist the runtime SPM management. A key of this table is the ID number of tasks, and values of the key are the start address of code group and the total size of its functions to be allocated to the SPM space. The SPM management table is implemented to the SkyEye simulator as a dedicated module. The runtime SPM management becomes simplified by referring the table.

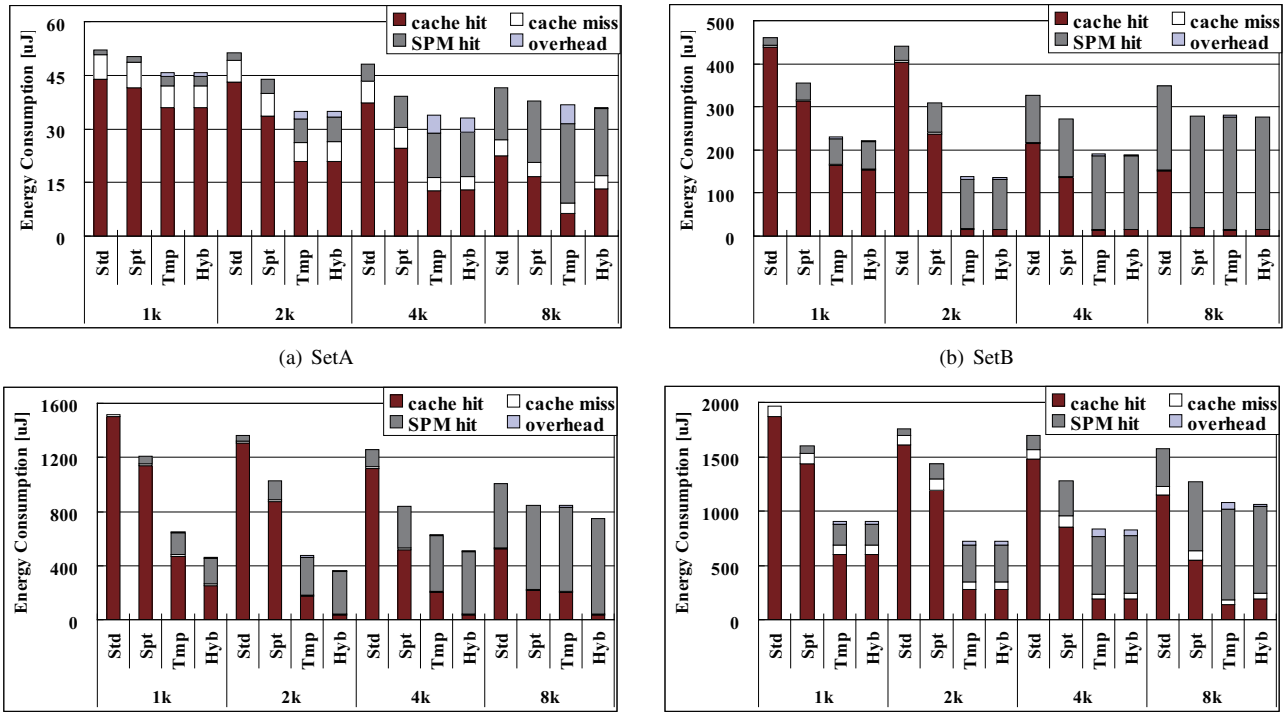
We implemented the function of our runtime SPM support mechanism to TOPPERS/ASP kernel (Release 1.3.2) [19]. TOPPERS/ASP kernel has the full functionality required for the construction of hard real-time systems, and conducts the priority-based preemptive task scheduler [20]. We added the detection mechanism into the dispatcher of TOPPERS/ASP kernel with only 35 lines in the assembler language.

IV. Evaluations

A. Experimental Setup

The proposed techniques were evaluated on the assorted task sets as presented in Table II. Lacking a benchmark suite consisting of multi-task applications, we selected 16 programs as tasks from the EEMBC benchmark suite [21] to assemble representative multi-task applications. In our experiments, a task with shorter period was given to a higher priority. The periods of tasks were set according to be proportional to their execution times, and the total CPU utilization rate of the task set was set about 50 %. Note that the total CPU utilization does not affect the effectiveness (in terms of energy saving ratio) of our techniques. The task set was executed based on the scheduling of TOPPERS/ASP kernel. We derived the total energy consumption on the instruction memory subsystem while a task set is executing in its hyperperiod. Also, we do not consider static energy consumption since applying our techniques does not affect the amount of static energy.

The instruction memory subsystem was assumed to consist of cache and SPM as the on-chip memory, and SDRAM as the off-chip main memory. The cache organization is 8 KB in size



(a) SetA

(b) SetB

(c) SetC

(d) SetD

Fig. 3. Experimental results.

TABLE II
THE TASK SETS

	#	Tasks included
SetA	5	aifftr, basefp, bitmnp, cacheb, idctrn
SetB	7	bezier, dither, ospf, pktflow, rotate, routelookup, text
SetC	11	conven, rgbcmy, rgbyiq, viterb, and SetB
SetD	16	the combination of SetA and SetC

and 4-way in associativity. The size of SPM is varied from 1, 2, 4, and 8 KB. We assumed that references to SDRAM are performed by 4-words burst access. The energy model of each memory access is based on CACTI 5.3 [22].

To evaluate the benefits of proposed methods, we brought a straightforward method as a baseline for comparison. This is because that a previous technique to utilize SPM for priority-based preemptive multi-task systems does not exist. In this method, the partitioning of the SPM space is not considered. At first of this method, the SPM space is partitioned equally for the tasks. Then, the allocation of functions to the SPM space is statically decided using a knapsack problem presented in [4].

B. Results and Discussion

Fig. 3 shows the experimental results. The bars in figures indicate the energy consumption of the instruction memory subsystem in μJ. The amount of energy consumption is analyzed into four factors; access energy of cache hits, that of cache misses, that of SPM hits, and that of the code transfer from the main memory to SPM. The values of "cache miss" and "overhead" include access energy of SDRAM. '*x*k' in the x-axis denotes the size of SPM. 'Std', 'Spt', 'Tmp', and 'Hyb' denote the energy consumption of the straightforward, spatial, temporal, and hybrid methods, respectively.

From these figures, the effectiveness of the proposed techniques has been demonstrated. Energy savings were achieved in all cases. At maximum, 33 % of energy reduction by the spatial method, 69 % by the temporal method, and 73 % by the hybrid method were achieved compared with the straightforward method. On average, 18 % of energy consumption by the spatial method, 38 % by the temporal method, and 41 % by the hybrid one were reduced.

Next, we focus on the comparison among proposed three methods. Experimental results showed that both the temporal and hybrid methods reduced more total energy consumption than the spatial method. This tendency indicates that the code replacement was effectively performed in our environment. Note that the total energy consumption of the code transfer is not so trivial. Results also showed that there was a slight overhead in terms of performance (there is no more than 5 % of the execution time on the task set). Moreover, the hybrid method becomes the most effective in almost all situations[3]. Up to 64 % and 28 % of energy consumption were reduced compared with spatial and temporal methods, respectively. This indicates that the hybrid method makes full advantage of the attribution of the scheduling policy. As described in Section III-D, a higher priority task preempt not only the CPU but also the space of SPM used by lower priority tasks in the hybrid method. We suggest that the hybrid method is the most suitable technique for the hard real-time systems.

[3]In the case of 1K SPM of SetD, the temporal method achieves slightly smaller energy consumption than the hybrid method. This is due to the difference of energy on cache misses, and the point of view of energy reduction in on-chip memory is larger.

In Figs. 3(b) and 3(c), our partitioning and allocation methods became more beneficial on energy savings. By contrary, the effectiveness of our methods was smaller in Fig. 3(a). The reason for this difference is the characteristics of a task set. Note that the number of tasks did not affect the effectiveness of our techniques so much. The SetA consists of automotive control programs. These have characteristics that a processing is almost straightforward. Meanwhile, the others include media processing programs. There are many loop iterations in those programs, and the locality of memory references is high. The effectiveness of our methods comes under the influence of the locality in a task.

Moreover, experimental results indicated that enlarging the size of SPM did not always reduce energy consumption. For example, 4 KB in SPM size achieved the least energy consumption in the SetA. The reason of this tendency is that the size of SPM is plentiful. Note that increasing the size of SPM raises extra energy on not only MM–SPM copy but also each read access to SPM especially when code allocation was enough to perform on smaller SPM size. Our integer programming problems have treated the size of SPM as a constant parameter. This implies the possibility that the size of SPM should be decided more appropriately for the purpose of energy minimization.

V. Conclusions

In this paper, we proposed SPM management techniques for a fixed-priority based preemptive multi-task systems. Our techniques give the benefit of energy reduction in the instruction memory subsystems. We proposed three methods named spatial, temporal, and hybrid methods. These are applicable to a hard real-time environment. It is remarkable that task periods and priorities are utilized in our methods for effective usage of the SPM space. Each method is formulated as an integer programming problem for the energy efficient SPM partitioning and code allocation. Our techniques achieve both inter-task and intra-task optimizations in terms of energy consumption simultaneously. Additional contribution of this work is that dynamic code allocation performs efficiently under the cooperative support of RTOS and hardware.

Experimental results conducted the effectiveness of our approaches. It should be noted that a fully functional RTOS was operated to achieve the hard real-time in our environment. Compared with a straightforward method, up to 73 % energy reduction was achieved. The hybrid method, where a higher priority task can preempt the SPM space of lower priority tasks and utilize it temporary, achieved the best result. In future, we are planning to extend these techniques for the data memory subsystems and the multiprocessor systems.

Acknowledgment

We thank Prof. Tohru Ishihara, Dr. Gang Zeng, and Dr. Tetsuo Yokoyama for suggesting our experiments. This work is supported in part by Core Research for Evolutional Science and Technology (CREST) from Japan Science and Technology Agency.

References

[1] S. Segars, "Low power design techniques for microprocessors," presented at IEEE International Solid-State Circuits Conference (Tutorial), Feb. 2001.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proceedings of International Symposium on Hardware/Software Codesign (CODES)*, Colorado, USA, May 2002, pp. 73–78.

[3] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Transaction on Embedded Computing Systems (TECS)*, vol. 1, no. 1, pp. 6–26, 2002.

[4] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, Mar. 2002, pp. 409–415.

[5] P. R. Panda, A. Nicolau, and N. Dutt, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1998.

[6] F. Angiolini, L. Benini, and A. Caprara, "An efficient profile-based algorithm for scratchpad memory partitioning," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1660–1676, 2005.

[7] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Exploiting statistical information for implementation of instruction scratchpad memory in embedded system," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 8, pp. 816–829, 2006.

[8] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, "Reducing energy consumption by dynamic copying of instructions onto onchip memory," in *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*, Kyoto, Japan, 2002, pp. 213–218.

[9] A. Janapsatya, S. Parameswaran, and A. Ignjatovic, "Hardware/software managed scratchpad memory for embedded system," in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, New Jersey, USA, Nov. 2004, pp. 370–377.

[10] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: A first approach," in *Proceedings of IEEE 3rd Workshop on Embedded System for Real-Time Multimedia (ESTIMedia)*, Jersey City, USA, Sep. 2005, pp. 115–200.

[11] R. Pyka, C. Faßbach, M. Verma, H. Falk, and P. Marwedel, "Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications," in *Proceedings of 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, Nice, France, 2007, pp. 41–50.

[12] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management in a multitasking environment," in *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT)*, Atlanta, USA, Dec. 2008, pp. 265–274.

[13] H. Takase, H. Tomiyama, and H. Takada, "Partitioning and allocation of scratch-pad memory in priority-based multi-task systems," *IPSJ Transactions on System LSI Design Methodology*, vol. 2 (2009), pp. 180–188, Aug 2009.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[15] SkyEye - Open Source Simulator. [Online]. Available: http://www.skyeye.org/

[16] ARM920T - ARM Processor. [Online]. Available: http://www.arm.com/products/CPUs/ARM920T.html

[17] GLPK (GNU Linear Programming Kit). [Online]. Available: http://www.gnu.org/software/glpk/

[18] GCC, the GNU Compiler Collection. [Online]. Available: http://gcc.gnu.org/

[19] TOPPERS Project. [Online]. Available: http://toppers.jp/en/

[20] μITRON4.0 Specification. [Online]. Available: http://www.ertl.jp/ITRON/SPEC/mitron4-e.html

[21] EEMBC – The Embedded Microprocessor Benchmark Consortium. [Online]. Available: http://www.eembc.org/

[22] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "CACTI 5.0," HP Laboratories, Tech. Rep., 2006.