Minimizing Inter-Task Interferences in Scratch-Pad Memory Usage for Reducing the Energy Consumption of Multi-Task Systems

Gauthier, Lovic
System LSI Research Center, Kyushu University

Ishihara, Tohru
System LSI Research Center, Kyushu University

Takase, Hideki
Graduate School of Information Science, Nagoya University

Tomiyama, Hiroyuki
Department of VLSI System Design College of Science and Engineering, Ritsumeikan University

他

https://hdl.handle.net/2324/18607

出版情報：Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems, pp.157-166, 2010-10. ACM Press
バージョン：

Minimizing Inter-Task Interferences in Scratch-Pad Memory Usage for Reducing the Energy Consumption of Multi-Task Systems

Lovic Gauthier
System LSI Research Center
3rd Floor, Institute of System
LSI Design Industry, Fukuoka
3-8-33 Momochihama,
Sawara-ku, Fukuoka
814-0001 Japan
lovic@slrc.kyushu-u.ac.jp

Tohru Ishihara
System LSI Research Center
3rd Floor, Institute of System
LSI Design Industry, Fukuoka
3-8-33 Momochihama,
Sawara-ku, Fukuoka
814-0001 Japan
ishihara@slrc.kyushu-u.ac.jp

Hideki Takase
Graduate School of
Information Science,
Naogoya University
C3-1 (631), Furo-cho,
Chikusa-ku, Nagoya,
464-8603 Japan
takase@ertl.jp

Hiroyuki Tomiyama
Dept. of VLSI System Design
College of Science and
Engineering
Ritsumeikan University
1-1-1 Noji-Higashi
Kusatsu, Shiga 525-8577,
Japan
hiroyuki@acm.org

Hiroaki Takada
Graduate School of
Information Science,
Naogoya University
C3-1 (631), Furo-cho,
Chikusa-ku, Nagoya,
464-8603 Japan
hiro@ertl.jp

ABSTRACT

This paper presents a new technique for reducing the energy consumption of a multi-task system by sharing its scratch-pad memory (SPM) space among the tasks. With this technique, tasks can interfere by using common areas of the SPM. However, this requires to update these areas during context switches, which involves considerable overheads. Hence, an integer linear programming formulation is used at compile time for finding the best assignment of memory objects to the SPM and their respective locations inside it. Experiments show that the technique achieves up to 85% energy reduction with 8Kb of SPM and surpasses other sharing approaches.

Categories and Subject Descriptors

D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)* ; D.4.2 [**OPERATING SYSTEMS**]: Storage Management—*Storage hierarchies* ; C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: *Real-time and embedded systems* ; B.3.1 [**MEMORY STRUCTURES**]: Semiconductor Memories—*Dynamic memory (DRAM)* ; B.3.1 [**MEMORY STRUCTURES**]: Semiconductor Memories—*Static memory (SRAM)* ; B.3.2 [**MEMORY STRUCTURES**]: Design Styles—*Cache memories*

General Terms

Design, Performance

Keywords

Low energy, multi-task, scratch-pad memory

CASES'10, October 24–29, 2010, Scottsdale, Arizona, USA.
Copyright 2010 ACM 978-1-60558-903-9/10/10 ...$10.00.

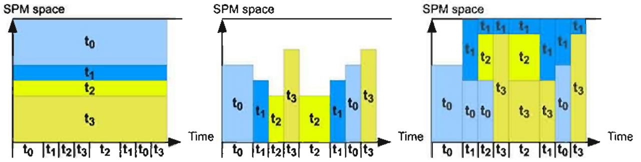
1. INTRODUCTION

Scratch-pad memories (SPM) are on-chip static random access memories (SRAM) which are often integrated with embedded processors. SPM are more area-expensive but faster and consume far less energy than external dynamic random access memories (DRAM). Hence, only SPM of small capacity can actually be used. Like caches, SPM are used as fast and low energy consuming buffers for frequently accessed code or data, but unlike them, they are not transparent to the software. Indeed, code and data are to be explicitly put into the SPM by the application. While more complex to use, SPM are faster than caches (in term of delay) and consume significantly less energy[1] (about 26% of what consumes the 4-way cache of the MeP processor we used in our experiments [9, 20]). Moreover, the behavior of SPM is much more predictable than the one of caches so that SPM are also preferred for real-time applications.

In a multi-task environment the SPM is to be shared among the tasks. The sharing can be done following two

[1] This is due to the additional tags and logic of the caches.

dimensions: spatial and temporal. For the first dimension, illustrated in figure 1a, the SPM space is divided into several areas, each of them being assigned to a single task (t_0, t_1, t_2 and t_3 in the figure) for placing its memory objects. This sharing is simple but it limits the space for each task to a small part of the SPM. For the second dimension, illustrated in figure 1b, the totality of the SPM space is provided to each task but when a task t_j is preempted by another task $t_{j'}$ the content of the SPM must be copied to the main memory (MM) if it has been modified. When t_i returns to the active state, this content must be copied back to the SPM (this second copy is always necessary). These additional copies consume time and energy and therefore limit the possible gain. When both sharing dimensions are used together, the SPM can be used more efficiently and lower energy consumption can be expected.


SPM space
t_0
t_1
t_2
t_3
Time
t_0 t_1 t_2 t_3 t_2 t_1 t_0 t_3
SPM space
Time
t_0 t_1 t_2 t_3 t_2 t_1 t_0 t_3
SPM space
Time
t_0 t_1 t_2 t_3 t_2 t_1 t_0 t_3

(a) Space (b) Time (c) Proposed

Figure 1: SPM sharing techniques

This paper presents a new fully software management of the SPM for multi-task preemptive real-time systems targeting minimal energy consumption related to the memory accesses. The approach uses both sharing dimensions and consists in maximizing the SPM space available to each task while keeping the required number of copies between the SPM and the MM at context switches as low as possible by minimizing the overlaps among the SPM blocks used by different tasks.

For that purpose, the technique selects at compile time for each task which memory objects are to be assigned to the SPM. A memory object can be a piece of code, for instance a function, or a piece of data, for instance a single variable or an array. At run-time, during the context switches, only the necessary number of bytes of the SPM are copied to the MM. These bytes correspond to the memory objects, or some parts of them, which have been modified and which will be overwritten be the next active task. Symmetrically, only the necessary number of bytes of the MM are copied back to the SPM. These bytes correspond to the memory objects, or some parts of them, which will be used by the next active task and which were previously overwritten by other tasks. In this paper, the operation which saves bytes from the SPM to the MM is called *store* and the one which restores bytes from the MM to the SPM is called *load*.

The more memory objects are assigned to the SPM the less the tasks consume energy for their memory accesses, but also the more energy is spent during the context switches for the stores and the loads. This last energy can however be reduced without changing which memory objects are assigned to the SPM if the sharing strategy reduces the overlaps among them when they are used by different tasks. For that purpose, the technique proposed in this paper assigns to each task one block in the SPM for placing its memory objects. The SPM update cost at the context switches is then reduced by selecting the addresses of the blocks which minimize the overlaps among them. Figure 1c illustrates such a sharing: as long as a task is active its SPM block is intact, but when the task is not active it can be partly or fully overlapped by the blocks of other tasks.

The optimizing problem is represented by an integer linear programming (ILP) formulation using profiling information for the memory accesses costs and the context switches rates. The objective of the formulation to minimize models the energy consumption related to the memory accesses and the solution indicates which memory objects are assigned to an SPM block and the addresses of such blocks. The relative positions of the memory objects within their respective blocks do not alter the resulting energy consumption and can therefore be fixed afterward.

To our knowledge, this is the first time a technique moves only the necessary bytes (i.e., from the overlaps) and not the totality of the memory objects of the SPM nor the full content of a memory object. This is also the first technique which uses the position of the blocks in the SPM as a parameter for reducing the overlaps' sizes and therefore the energy consumption of the stores and the loads to perform during the context switches.

The rest of the paper is organized as follows: the next section presents some related works. Section 3 explains how the technique works, then section 4 details the proposed ILP formulation and section 5 discusses about a possible refinement of the approach. Finally, section 6 presents some experimental results and section 7 concludes the paper.

2. RELATED WORKS

Optimizing the usage of the limited space of the SPM has been a subject of research for several years. A lot of work has been done for allocating memory objects of a single task to the SPM. Some of these approaches, like [12,14,15,25] decide the allocations at compile time, and others, like [2,6,11] do it at run time. Other works deal with dynamic memory objects like the stack [1,5,21] or arrays which are split and dynamically spread between the SPM and the MM [24].

While a majority of the works regarding the SPM consider only one task, several methods exist which use the SPM for multi-task applications, either for increasing the performances [16,17] or for reducing the energy consumption [3,13,18,19,23]. As presented in the introduction, there are two orthogonal ways to share the SPM space among the tasks: the spatial sharing and the temporal sharing. Papers like [18,19,23] explore both approaches and propose for each, ILP formulations meant to find at compile time the optimal sharing. Both approaches can also be merged for achieving better results: [18,23] splits the SPM area into two parts, one dedicated to the spatial sharing and one dedicated to the temporal sharing. [19] goes farther allowing higher priority tasks to use the SPM space of lower priority ones, but for the code memory objects only. The technique proposed in this paper has more freedom for moving the memory objects than the previous ones and it can also move parts of a memory object. [13] does propose a more general hybrid approach as any memory objects can be moved between the SPM and the MM at context switches, but only full memory objects can be moved. Furthermore, unlike the other approaches, it requires the full execution of the system to be fixed and known at compile time.

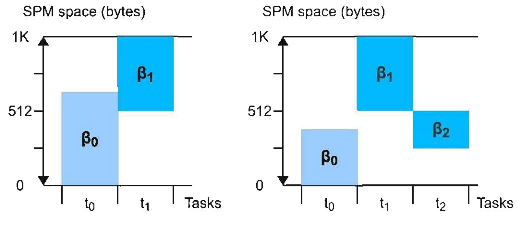
Some techniques are fully dynamic, i.e., they assign memory objects to the SPM at run time. For instance, [3] proposes to handle the SPM with a paging system, which requires the presence of an MMU in the target processor. Additionally to its compile-time hybrid approach mentioned earlier, [13] also proposes dynamic management methods which make use of lists of free blocks within the SPM.

Although they target multi-task systems, some approaches like [3, 23] do not use the scheduling as a parameter for optimizing the usage of the SPM. This induces suboptimal results, hence, the authors of [23] do consider the scheduling for their optimizations in [13]. Yet, as stated earlier, they assume that there is a global view of the entire execution flow of the system and propose to insert into this flow control points where to change the allocations of the SPM. Other approaches are more practical regarding the scheduling. For instance, [16, 17] are targeting systems with a static scheduling. A few other approaches target preemptive systems. For instance, [18, 19] optimizes the SPM usage while using the properties of a static priority-based real-time preemptive system. In this paper too, a static priority-based preemptive scheduling has been used for the experiments, but as explained in section 3.4, other policies are possible.

3. THE PROPOSED APPROACHES

3.1 Base idea

The proposed approach is based on the observation that if two tasks t_0 and t_1 do not occupy the totality of the SPM, the sizes of the corresponding loads and stores at context switches can be minimized by assigning the upper part of the SPM space to t_0 and the lower part to t_1, as it can be seen in figure 2a. In the figure, β_0 and β_1 are the parts of the SPM respectively used by t_0 and t_1. When a context switch occurs between t_0 and t_1, the proposed technique stores and loads only the bytes of the overlap between β_0 and β_1. Figure 2b shows a sharing for three tasks. Again, a good choice for the addresses of the blocks reduces the overlap, but unlike figure 2a, a block is located in the middle of the SPM for minimal overlaps.



(a) Two tasks (b) Three tasks

Figure 2: SPM shared between tasks (the horizontal axis does not express the time)

When more than two tasks are present, the energy consumption can be further reduced by using the fact that preemptions are usually not uniform. For instance, figure 3 shows a portion of scheduling with three tasks, t_0, t_1 and t_2. In the figure, each up arrow represents the firing of a task, each down arrow represents a preemption and each dashed line indicates the termination of a task. As seen in the figure, t_1 is preempted by t_0 two times while t_2 is preempted by t_0 only once. Provided that t_2 and t_1 have about the same requirement in SPM space, there would therefore be less copies between the SPM and the MM during the context switches if t_0 shares its SPM space with t_2 than with t_1 as it can be seen in figure 2b.

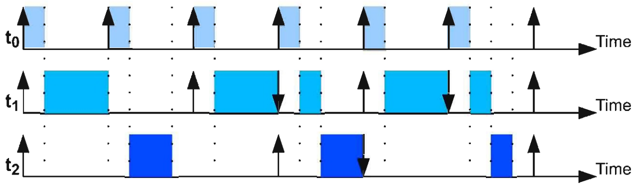


Figure 3: A portion of scheduling

In order to define more precisely the problem, the next subsection defines a few terms regarding areas of the SPM and the following one presents a few assumptions used in the paper. Then, details about the proposed technique are given by the subsequent sections.

3.2 Areas of the SPM

Various kind of contiguous areas (called *segments*) of the SPM are required for representing the problem, they are defined in this section.

First, tasks have access to the SPM through blocks which are defined as follows:

Block: it is a segment of the SPM used by a task for placing its memory objects. A block is noted β.

Considering several tasks, their sharing of same areas in the SPM is a set of overlaps defined as follows:

Overlap: it is a segment of the SPM which is the intersection of two blocks. An overlap is noted δ. When an overlap comes from more than two tasks, it is called multiple overlap.

The segments which are to be stored or loaded at context switches are all overlaps. However, as explained in section 4.5, not all the overlaps require to be stored or loaded. The following definition is therefore required:

Effective overlap: it is the contribution of an overlap to the number of bytes to be stored or loaded at some context switches.

With these definitions, the problem can be stated as maximizing the sizes of the blocks while minimizing the effective overlaps.

3.3 Preliminary assumptions

If a memory object is not modified by its task, it does not need to be stored. Hence its context switch cost for one byte is lower than the one for a memory object which is modified. In this paper, it is assumed that code is never modified and that data is always modified. Moreover, it is assumed that code and data are assigned to different parts of the memory architecture since it is often the case in practice. For instance, the MeP processor [9, 20] used for the experiments of this paper includes one SPM device for the code and another one for the data. If the target architecture includes only one SPM, this simplification can still be applied by splitting the SPM in two parts, one for the code memory objects and one for the data ones.

Another important point is about the consequence on the execution time of the proposed technique. Principally, store and load operations performed during the context switches do not only consume energy, they also consume time. For real-time systems this extra time must be taken into ac-

Table 1: Two preemption patterns

Pattern	Reference task	Executed tasks	Occurrences
p_0	t_1	$\{t_0\}$	2
p_1	t_2	$\{t_0,t_1\}$	1

count. Fortunately, if the worst case is computed while not using the SPM, optimizing the energy consumption by using this internal memory also improves the execution time even with the overhead of the stores and loads.

3.4 About the scheduling

When a task t_2 is preempted by a task t_1 the number of bytes to store (if data), and afterward to load, is the number of bytes of the SPM which are into the overlap between the block of t_2 and the one of t_1. If t_1 often preempts t_2 there will be as many stores and loads and the cost of the corresponding overlap will be high. By contrast, if a task t_0 never becomes active while t_2 is ready there will be no additional store nor load cost induced by the overlap between the block of t_2 and the one of t_0. Hence, optimizing the placement of the memory objects must take into account the scheduling for an efficient sharing of the SPM among the tasks.

Now let us assume that task t_2 is preempted by task t_1. Then task t_0 preempts t_1 before t_2 returns to execution. The overlap between t_2 and t_1 and the one between t_1 and t_0 are to be taken into account, but also the one between t_2 and t_0. Hence, direct preemptions are not enough for computing the total overlap cost related to a given task, the executions of the other tasks must also be taken into account.

Therefore, in order to compute the context switches costs, it is required to enumerate the patterns of preemption occurring along the execution of the system. In this, paper, these patterns are defined as follows:
Preemption set: it is a set of the tasks executed between two active states of the same instance of a given task. This latter task is called the *reference task* for this set[2]. A preemption set is noted P.

Preemption pattern: it is the association of a reference task with one of its preemption sets. Preemption patterns are also called patterns.

For the example given in figure 3 the patterns are given in table 1.

These preemptions patterns are used for determining the effective overlaps and how often the corresponding bytes are to be stored or loaded at some context switches. First, the number of bytes to store (if data) then to load for the reference task of a pattern is the size of the union of the overlap segments among this task's block and the ones of the preemption set's tasks. By union, it is meant that each byte to store or to load covered by one or more blocks is counted only once. Then, the energy consumption for updating the SPM during the context switches is computed by multiplying the cost corresponding to the overlaps of each pattern by its occurrence rate obtained from a set of profiling executions of the multi-task system.

Preemption patterns can be built for any preemptive systems. However, their possible contents vary depending on the scheduling policy: for a static priority-based policy, a pattern set cannot contain any task whose priority is lower than the one of the reference task whereas with policies like the earliest deadline first (EDF) this restriction does not apply.

3.5 Run-time Management of the SPM

The SPM management is performed during the context switches. In the case of a preemption, a function is called which stores the parts of the data blocks which overlap with the one of the coming task. If a previously preempted task returns to the active state, another function loads the parts of its block which have been overwritten while the task was inactive. For that purpose, the SPM space is split at compile time into several areas delimited by the starting and the ending addresses of the blocks. Areas are defined as follows:
Area: it is a part of the SPM delimited by two consecutive bounding (starting or ending) addresses of the blocks; an area is noted α.

Figure 4 illustrates such a splitting with three tasks (and three blocks). In the figure, α_0 is bounded by the starting addresses of β_0 and β_1, α_1 is bounded by the starting address of β_1 and the ending address of β_2 and so on.

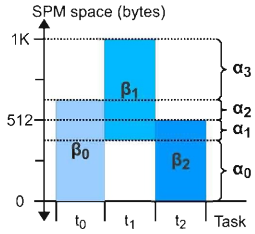


Figure 4: SPM split into areas following the blocks' bounds

When a context switch occurs, some of these areas are stored or loaded. Two tables are used for that purpose. The first table, built at compile time, is indexed by the blocks' identifiers and gives the set of the areas which are covered by each block. Table 3a is such a table which has been built from the example of figure 4. It can be seen from this table that the set of the areas covered by a given block is described by the starting and ending areas only. This is indeed enough as a single block is necessarily a range of contiguous areas.

Table 2: Tables for managing the areas at run time

Block	Covered areas	
	Start	End
β_0	α_0	α_2
β_1	α_1	α_3
β_2	α_0	α_1

(a) Blocks' covers

Area	Address	Size	Block
α_0	a0000000	384b	β_2
α_1	a0000180	128b	β_1
α_2	a0000200	128b	β_1
α_3	a0000280	384b	β_1

(b) Areas' states

The second table, initialized at compile time and updated at run time, is indexed by the areas' identifiers. Each of its entries gives the starting address and the size of the corresponding area, and the identifier of the block currently occupying it. When the system starts, this last field is empty. Table 3b is an instance of such a table at run time for the

[2] Only the first task of the preemption set actually preempts the reference task.

example of figure 4. The table corresponds to a state where task t_1 has preempted task t_2 and task t_0 is not ready.

When a context switch occurs, the block table is first looked up for finding the areas that will be used by the next active task. Then, for each found area, the area table is looked up for determining which block occupies it. If the block is from another task, the corresponding area of the new task's block is loaded from the MM (the former area is first stored if it is a data block). Finally, the area table is updated accordingly to its new state.

Note: Accessing and computing the evolution of the SPM state table implies additional costs which are taken into account into the average costs for storing and loading bytes between the SPM and the MM. For now, the tables are kept into the MM.

4. FORMULATION FOR THE TASK-GRAIN APPROACH

The goal of the ILP is to minimize the energy consumption related to the memory accesses including the tasks accesses and the stores and loads during the context switches. The objective function to minimize can therefore be decomposed into two parts as seen in equation (1): obj_{tasks} which accounts for the tasks' memory accesses costs, and $obj_{overlaps}$ which accounts for the costs of the loads and stores for overlapped parts within the SPM.

$$objective = obj_{tasks} + obj_{overlaps} \tag{1}$$

The parameters of the formulation include the tasks, the blocks, and the memory objects. Their relevant characteristics are described by the variables and the constants presented in the next subsection. When solved, the ILP gives the address of each block in the SPM and which memory objects are assigned to it.

Since it is assumed that the code and the data memory objects are on different SPM devices, there is actually one ILP for the code and one ILP for the data. As both ILP are similar (apart from the unit costs of the memory accesses and SPM updates during the context switches), no distinction is made between them in the rest of the paper unless it is relevant.

Note: In all the sections about the ILP, the numerated equations are part of the formulations used in the proposed technique while the unenumerated ones are there for explanation purpose only.

4.1 Main variables and constants

Several variables and constants are used in the ILP for representing the characteristics of the architecture, the system, the tasks, the blocks, and the memory objects. They are indexed by i for the preemption patterns, j for the tasks and the corresponding blocks, and k for the memory objects.

The variables are the following:

$x_{j,k}$: this binary variable is 1 if memory object k of task j is in the SPM when the task is active;

s_j : this integer variable gives the size of block j;

b_j : this integer variable gives the starting ("begin") address of block j;

e_j : this integer variable gives the ending ("end") address of block j;

$o_{j,j'}$: this integer variable is the size in bytes of the overlap between blocks j and j';

$o^{sel}_{j',j''}$: this binary variable is used for selecting the constraint defining the overlap $o_{j,j'}$;

$oe_{i,j,j'}$: this integer variable is the size in bytes of the effective overlap between blocks j and j' for pattern i;

$o^{min}_{j,j',j''}$: this binary variable is used for selecting the constraint defining the effective overlap $oe_{i,j,j'}$.

By default, the integer variables are greater or equal to 0.

The constants are the following:

R_i : it is the total number of occurrences for pattern i. It is obtained by profiling the multi-task application;

$Cspm^{tot}_{j,k}$: it is the energy cost of the total accesses to memory object k of task j if the object is in the SPM. This cost also includes the preliminary load required when a task is fired. It is obtained by profiling the multi-task application;

$Cmm^{tot}_{j,k}$: it is the energy cost of the total accesses to memory object k of task j if the object is in the MM. If there is a cache (e.g. k is a code memory object), this cost accounts for its accesses and the miss overhead, otherwise, it accounts for the direct accesses to the external DRAM. This cost also includes the lost of cache contents at context switches (this content being overwritten by the next active tasks). It is obtained by profiling the multi-task application;

$Ccxt$: it is the average energy spent in context switches for updating one byte of the SPM. In the case of code, it is the average cost for loading one byte, whereas in the case of data, it is the average cost for storing and loading one byte;

S_k : it is the size of the memory object k;

$S^{max}_{\beta j}$: it is the maximum size of block j. It is computed as the min of the size of the SPM and the sum of the sizes of all the memory objects of task j.

4.2 Tasks memory accesses formulation

The first part of the objective function is the sum of the energy consumed by each task while accessing its memory objects. For each memory object, the energy consumed depend on whether it is assigned to the MM or to the SPM. The $x_{j,k}$ variables are used for selecting the cost corresponding to the used memory, and obj_{tasks} is then computed as follows:

$$obj_{tasks} = \sum_{j,k} (Cspm_{j,k} - Cmm_{j,k}) * x_{j,k} \tag{2}$$

In the above equation, in order to keep the linearity of the objective, only the difference in energy consumptions between the accesses to the MM and the SPM is actually represented.

4.3 Constraints for the blocks

When a memory object is assigned to the SPM, it is put into the block of its corresponding task. The blocks are characterized by three variables: their size s_j, their starting address b_j and their ending address e_j. With them, blocks are constrained to be within the SPM as follow:

$$e_j < S_{spm} \tag{3}$$

The sizes of the blocks are linked to the beginning and ending addresses as follows:

$$s_j = e_j - b_j \tag{4}$$

Since by default, variables are bounded to be greater or equal to 0, e_j, b_j and s_j are never negative which implies that $e_j \geq b_j$.

The size of a memory block is the sum of the sizes of its memory objects:

$$s_j = \sum_k x_{j,k} * S_k \tag{5}$$

4.4 Overlaps formulation

The size of the overlap between blocks β_j and $\beta_{j'}$ can be expressed by considering each case of their relative positions:

$$o_{j,j'} = \begin{cases} e_j - b_j & \text{if } e_j \leq e_{j'} \text{ and } b_j \geq b_{j'} \\ e_j - b'_j & \text{if } e_j \leq e_{j'} \text{ and } b_{j'} > b_j \\ e'_j - b_j & \text{if } e_{j'} < e_j \text{ and } b_j \geq b_{j'} \\ e'_j - b'_j & \text{if } e_{j'} < e_j \text{ and } b_{j'} > b_j \\ 0 & \text{otherwise} \end{cases}$$

These cases are finally formulated as four constraints using four mutually exclusive binary variables noted $o^{sel}_{j,j}$, $o^{sel}_{j,j'}$, $o^{sel}_{j',j}$, $o^{sel}_{j',j'}$:

$$o_{j,j'} \geq e_j - b_j - M_{j,j} * (1 - o^{sel}_{j,j}) \tag{6a}$$

$$o_{j,j'} \geq e_j - b_{j'} - M_{j,j'} * (1 - o^{sel}_{j,j'}) \tag{6b}$$

$$o_{j,j'} \geq e_{j'} - b_j - M_{j',j} * (1 - o^{sel}_{j',j}) \tag{6c}$$

$$o_{j,j'} \geq e_{j'} - b_{j'} - M_{j',j'} * (1 - o^{sel}_{j',j'}) \tag{6d}$$

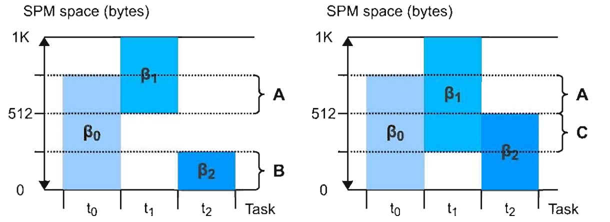
$$o^{sel}_{j,j} + o^{sel}_{j',j} + o^{sel}_{j',j} + o^{sel}_{j',j'} = 1 \tag{7}$$

In equations (6) the constants M are used as "big M" [8]: they have to be large enough to ensure the equality to be correct when the corresponding o^{sel} variable is 0, but short solving time also requires them to be as small as possible. Here, $M_{j,j}$ and $M_{j',j'}$ can be the maximum size of the respective blocks j and j' (i.e., $S^{max}_{\beta j}$ and $S^{max}_{\beta j'}$) while $M_{j,j'}$ and $M_{j',j'}$ are the size of the SPM. Equation (7) ensures that exactly one o^{sel} is 1.

When there is no overlap, these equations are still valid as then, either $e_j - b_{j'}$ or $e_{j'} - b_j$ is negative so that $o_{j,j'}$ will be set to 0.

4.5 Effective overlaps

Figure 5 illustrates the necessity to distinguish between the overlaps and the effective overlaps. In the figure, parts A of the SPM are used by t_0 and by t_1, part B is used by t_0 and t_2, and part C is used by t_2, t_1 and t_0. In the case of figure 5a, the total overlap cost for task t_2 is the sum of the overlap costs of β_2 with β_0 and the one of β_2 with β_1, that is to say:



(a) (b)

Figure 5: Cases of overlaps between three blocks

$$o_{2,0} * Ccxt + o_{2,1} * Ccxt = o_{2,0} * Ccxt$$

Yet, the full overlap cost can be more complicated to compute: some parts of the SPM can be used by more than two tasks as it is the case of part C in figure 5b which is used by t_0, t_1 and t_2. When t_2 is preempted by t_1, this part is stored so that it is not necessary to store it again when t_0 preempts t_1. The same goes for the loads.

As soon as an overlap is included into another overlap built from the same pattern, it should not be take into account for the context switches cost and its corresponding effective overlap is 0. Let $oe_{i,j,j'}$ (effective overlap) be the number of bytes of the block of task t_j which will be overwritten at a context switch because of task $t_{j'}$ during pattern i[3]. When $t_{j'}$ is the only task of preemption set, $oe_{i,j,j'}$ is simply equal to $o_{j,j'}$. Otherwise, this effective overlap is expressed as follows:

$$oe_{i,j,j'} = \begin{cases} 0, & \text{if } \exists j'' \in P_i - \{j'\} \qquad \delta_{j,j'} \subset \delta_{j,j''} \\ o_{j,j'}, & \text{otherwise} \end{cases}$$

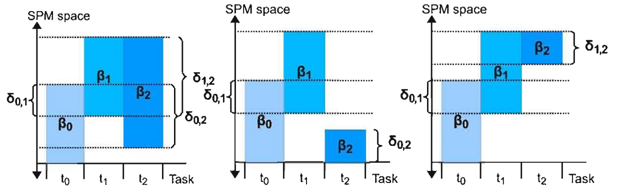
In the equation, $\delta_{j,j'}$ represents the overlap segment between the block of task j and the one of task j', and P_i is the preemption set i. For instance, in figure 6a, is t_0 is the reference task, $\delta_{0,1}$ is not effective since it is included into $\delta_{0,2}$.

The actual computation of an effective overlaps is then deduced from the fact that a multiple overlap, if present, is also one of the single overlaps made of two of its blocks, the smallest of them. Therefore, when an overlap is included into another overlap, it is the smallest overlap of the three obtained from the concerned memory blocks, hence the overlap segments inclusion can be tested as follows:

$$\delta_{j,j'} \subset \delta_{j,j''} \Leftrightarrow o_{j,j'} = min(o_{j,j'}, o_{j,j''}, o_{j',j''})$$

Figure 6 illustrates this by giving the three possible cases: in the case 6a, the overlap $\delta_{0,1}$ is really included into the overlap $\delta_{0,2}$ (hence, $o_{0,1}$ is not effective). It is also included into the overlap $\delta_{1,2}$, which is necessary due to the definition of the overlap segments. This first case also shows symmetrically that $\delta_{0,2}$ is not included into $\delta_{0,1}$ and is not the smallest of the three. In the case 6b of the figure, β_2 does not overlap with β_1, therefore the corresponding overlap is 0 which is the minimum of the three. In the case 6c β_2 does not overlap with β_0 and the corresponding overlap is 0.

[3] t_j is thence the reference task of P_i.



(a) (b) (c)

Figure 6: Cases of overlaps inclusion and non-inclusion

The linear formulation of each effective overlap $oe_{i,j,j'}$ is then expressed using a binary variable $o^{min}_{j,j',j''}$ which is 1 when $\delta_{j,j'}$ is included into a $\delta_{j,j''}$ where j'' is from $P_i - \{j'\}$:

$$(1 - o^{min}_{j,j',j''}) * min(S^{max}_{\beta j}, S^{max}_{\beta j'}, S^{max}_{\beta j''}) \geq \\ o_{j,j'} - o_{j,j''} + \epsilon_{j,j'} - \epsilon_{j,j''} \quad (8a)$$

$$(1 - o^{min}_{j,j',j''}) * min(S^{max}_{\beta j}, S^{max}_{\beta j'}, S^{max}_{\beta j''}) \geq \\ o_{j,j'} - o_{j',j''} + \epsilon_{j,j'} - \epsilon_{j',j''} \quad (8b)$$

$$oe_{i,j,j'} \geq o_{j,j'} - min(S^{max}_{\beta j}, S^{max}_{\beta j'}) * \sum_{j'' \in P_i - \{j'\}} o^{min}_{j,j',j''} \quad (9)$$

In the above equations, the constants computed with the min of the maximum possible sizes of the blocks β_j, $\beta_{j'}$ or $\beta_{j''}$ are the tightest which ensure that $o^{min}_{j,j',j''}$ and $oe_{i,j,j'}$ will be forced to 0 when required. Additionally, the $\epsilon_{j,j'}$ constants are there to ensure that at least one effective overlap is not set to 0 when several overlaps are equal. These positive constants are strictly smaller than 1 and are all different from one another. This artificially forces the subtraction of overlaps of equations (8) to be different from 0.

Sometimes, effective overlaps are also required to be subtracted when computing the context switches costs, they are called *subtractive* in this paper. For instance, in figure 6a, if the reference task is $t2$, both $\delta_{0,2}$ and $\delta_{1,2}$ are effective as none is included into another overlap, but they still intersect. This intersection, which is $\delta_{1,2}$ should be counted only once instead of twice. For that purpose, the corresponding effective overlap is to subtract once. Such a subtractive effective overlap corresponds to overlaps between tasks of the pattern, while the previous effective overlaps correspond to overlaps between the reference task and one task of the pattern. Subtractive effective overlap $oe_{i,j',j''}$ is non zero only if the corresponding overlap segment is counted twice, that is to say when both $oe_{i,j,j'}$ and $oe_{i,j',j''}$ are non-zero. Moreover, it is to be counted only if it is not included into another subtractive effective overlap (as for the previously formulated additive effective overlaps). Hence the formulation for $oe_{i,j',j''}$ is close in principle to the previous one, but with the differences that it is removed from the cost so that the solver will try to maximize it and that it can be non zero only if both effective overlaps $oe_{i,j,j'}$ and $oe_{i,j',j''}$ are non zero. Equations (10) ensure this last property, and equation (11) performs their actual computation.

$$oe_{i,j',j''} \leq min(S^{max}_{\beta j'}, S^{max}_{\beta j''}) * oe^{use}_{i,j,j'} \quad (10a)$$
$$oe_{i,j',j''} \leq min(S^{max}_{\beta j'}, S^{max}_{\beta j''}) * oe^{use}_{i,j,j''} \quad (10b)$$

In the above equation, the binary variable $oe^{use}_{i,j,j'}$ is one when $oe_{i,j,j'}$ is non zero and is formulated straight forwardly.

$$oe_{i,j',j''} \leq o_{j',j''} \quad (11a)$$

$$\forall j''' \in P_i - \{j', j''\} \quad oe_{i,j',j''} \leq \\ min(S^{max}_{\beta j'}, S^{max}_{\beta j''}) * (2 - o^{min}_{j',j'',j'''} - oe^{use}_{i,j,j'''}) \quad (11b)$$

In equations (11b), $oe^{use}_{i,j,j'''}$ is there for eliminating the cases where the subtractive overlap coming from j''' is not used so that $oe_{i,j',j''}$ is not subtract along with $oe_{i,j',j''}$ even if $o_{j',j''} \in o_{j'',j'''}$.

These effective overlap variables are finally used for computing the second part of the objective function (j_i is the index of the reference task for pattern i):

$$obj_{overlaps} = \\ Ccxt * \sum_i \Big(R_i * (\sum_{j' \in P_i} oe_{i,j_i,j'} - \sum_{j',j'' \in P_i} oe_{i,j',j''})\Big) \quad (12)$$

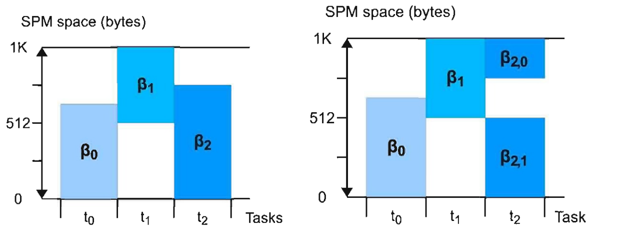
While not necessary for the completeness of the model, a final set of constraints is added for reducing the solving time. They set the minimum sizes for the overlap variables depending on the sizes of their corresponding blocks as follows:

$$o_{j,j'} \geq s_j + s_{j'} - S_{spm} \quad (13)$$

These last constraints allow the LP-solver to cut off several continuous solutions which do not correspond to any good integer solutions [8].

5. REFINING THE TECHNIQUE

The technique can be refined by assigning more than one block per task in order to give more opportunities for reducing the overlaps. For instance, in figure 7, three tasks, t_0, t_1 and t_2 are sharing the SPM. It is assumed that t_2 is preempted by t_0 and t_1, and that t_1 is preempted by t_0. In figure 7a, there is one block per task and the corresponding overlap sizes are given in table 4a. But if task t_2 has two blocks which can be placed as shown in figure 7b, the overlap size between t_2 and t_0 is reduced while not increasing the other overlaps. Table 4b gives the sizes of the overlaps for this second case.



(a) One block per task (b) Two blocks for t_2

Figure 7: Less overlap when using more than one block per task

Thence, better solutions should be found by increasing the number of blocks. Ideally, the true optimal solution could be reached if enough blocks are added. However, the solving time quickly grows with the number of blocks. Moreover,

Table 3: Overlap sizes corresponding to figure 7

Overlap	Size (bytes)
$\beta_0 \cap \beta_1$	128
$\beta_0 \cap \beta_2$	640
$\beta_1 \cap \beta_2$	256

(a) One block per task

Overlap	size (bytes)
$\beta_0 \cap \beta_1$	128
$\beta_0 \cap \beta_2$	512
$\beta_1 \cap \beta_2$	256

(b) Two blocks for t_2

experiments will show that the solution is often hardly better with additional blocks.

Nevertheless, the formulation is easily modified for supporting several SPM blocks per task. First, variables and constants are extended with an additional index which refers to the corresponding memory block. This index is noted l and is now different from j, the task index. The equations of the formulation are all updated with the extended variables and constants. When sums iterates on the tasks' indexes j, they are extended for also iterating on the blocks' indexes l.

New constraints are also required for forbidding two blocks of a same task to overlap with each other (as they are to be used simultaneously). Since the order of the blocks within the SPM is of no importance, it is enough to impose the starting address of each block $l+1$ of a given task to be greater or equal to the ending address of block l (Nb_j is the number of blocks for task j):

$$\forall l < Nb_j \qquad b_{j,l+1} \geq e_{j,l} \tag{14}$$

6. EXPERIMENTS

6.1 Experimental environment

We applied our technique on a MeP [9, 20] processor configuration including an instruction SPM, a 4-way instruction cache and a data SPM. We used the Toshiba's MeP Integrator (MPI) tool chain [10] for compiling and simulating the applications. Energy characteristics of this architecture are given in table 4. Compilations were performed with the $-O2$ level of optimization. The multi-task was scheduled using static priorities and rate monotonic conditions.

Table 4: Average energy consumptions of memory accesses for a MeP processor

Memory access type	Energy (nJ per word)
SPM (8Kb)/code fetch	0.3774
SDRAM/code fetch	62.541
cache hit/code fetch	1.433
cache miss/code fetch	57.897
SPM (8Kb)/data read	0.3774
SPM (8Kb)/data write	0.5053
SDRAM/data read	42.6466
SDRAM/data write	18.3986

Five sets of tasks have been used. Tasks were taken from the EEMBC [4] and the MiBench [22] benchmark suites. Table 5 describes the used sets. For each set, tasks were configured to perform one round (e.g., one frame for the mp3) when fired, and for their executions, the total load of the processor was set to about 60%.

Profiling was performed in two steps. First, each task has been profiled individually for enumerating its accesses to each of its memory objects and the number of cache misses of the code fetches. Then, the scheduling of the multi-task application is profiled for enumerating the context switches and deducing the preemption patterns and their occurrences.

For purpose of comparison, a space sharing (*Space*), a time sharing (*Time*) and hybrid space and time sharing (*Hybrid*) approaches were also implemented. For all of them, the scheduling has been taken into account for optimizing the usage of the SPM. Two approaches were actually used for Hybrid: [19] was used for the code and [18] was used for the data. The approach proposed in this paper is experimented into its original form with one block per task (*One*) and into a refined version with two blocks per task (*Two*).

Table 5: Tasks-sets used for the experiments

Set	Tasks	Nb of memory objects	
		Code	Data
Set 0	fft, mp3 decode, mp3 decode, string search	147	52
Set 1	patricia, patricia, string search, string search 2	56	32
Set 2	cubic, fft, patricia, qsort, string search	108	28
Set 3	cubic, qsort, rad2deg, adpcm encode, adpcm decode	64	20
Set 4	cubic, patricia, qsort, rad2deg, adpcm encode, adpcm decode, string search, string search 2	206	82
Set 5	cubic, fft, mad, mpeg decode, mp3 decode, patricia, qsort, rad2deg, adpcm encode, adpcm decode, string search	333	196

6.2 Results

The comparison of the different approaches is given in figure 8 and figure 9. In the figures, the energy consumption related to the memory accesses for each approach and each task set is given normalized to the energy consumed when the code and the data are in the MM. It is important to notice that when in the MM, the code is accessed through the cache whereas the data is not. For figure 9 the results of Two for the code SPM are not given as the solving time exceeded 8 hours while not achieving a good solution.

As seen in the figures, both One and Two perform better than the other approaches. For the code, with an 8Kb SPM, Space achieves on average an energy reduction of 31%, Time 35%, Hybrid 39% and One 51%. The average of Two is 45% but set 5 is not taken into account. With the same SPM, the respective maximum energy reductions are 47%, 75%, 75% and 77%. For the data, with an 8Kb SPM, Space achieves on average an energy reduction of 60%, Time 52%, Hybrid 60%, One and Two 63%. With the same SPM, the respective maximum energy reductions are 79%, 67%, 79% and 85%.

Remarkably, Two hardly achieves better than One. Actually the gain for Two is visible for set 2 only. This can be explained by the fact that having several blocks per task is beneficial only if for several tasks some memory objects are better left into the SPM all the time, or if preemptions are irregular so that some overlaps are significantly more expensive than others. Moreover, the more the tasks are present, the more these irregularities tend to dilute.

When comparing the code to the data cases, higher energy reduction should be expected while using the SPM since the data is accessed directly in the external DRAM whereas the code is accessed through the cache. This is indeed the case provided the SPM is large enough to include enough data (e.g. for set 3).

When solving the ILP formulations, the worst solving time for One was about 10 minutes for set 5 which contains 12 tasks, 10 seconds with set 4 which contains 8 tasks and less than one second with the other sets. Actually the solving time depends significantly on the number of tasks but not much on the number of memory objects. This is because the solver we used [7] deals very efficiently with the knapsack constraints which can be deduced from the size of the blocks. Solving Two is much slower, indeed, good solutions could not be found with set 5 after several hours. This time, no knapsack constraint could be extracted because for each memory objects, there are two possible blocks.

7. CONCLUSION

This paper presented a technique for optimizing the sharing of scratch-pad memories among several tasks targeting the minimization of the energy consumption regarding the memory accesses. The technique maximizes the number of accesses to the SPM by tasks while minimizing the number of copies between the SPM and the MM which are required during the context switches for updating the areas of the SPM used by several different tasks. For that purpose, the technique selects which memory objects are to be assigned to which blocks and the location of these blocks within the SPM. The SPM allocation is obtained by solving an ILP whose complexity can be tuned by choosing the number of blocks available to each task. Experimental results have shown that even when tuned for the fastest answer, the proposed technique achieves better results than other recent approaches and up to 85% energy reduction with an SPM size of 8Kb.

As future work, we plan to improve the formulation to allow blocks which are modified and blocks which are not, to be located in a same part of the SPM. Another planned improvement is to bring support of memory objects shared among several tasks. Also, while our technique with one block per task did achieve better than hybrid space and time approaches in our experiments, it does not supersets them. Therefore, cases may happen where an hybrid approach does perform better. However, when there is at least two blocks per task, it does supersets the hybrid techniques, but the solving time can be prohibitively long. A possible faster compromise to explore could then be to merge our technique using one block per task with the space sharing. Finally, several techniques exist for reducing the energy consumption through the management of the SPM at the task level. Some of them handle efficiently the stack, and others pre-load parts of large arrays. Greater energy reduction would be achieved if those task-level techniques could be merged with the multi-task-level technique presented in the paper.

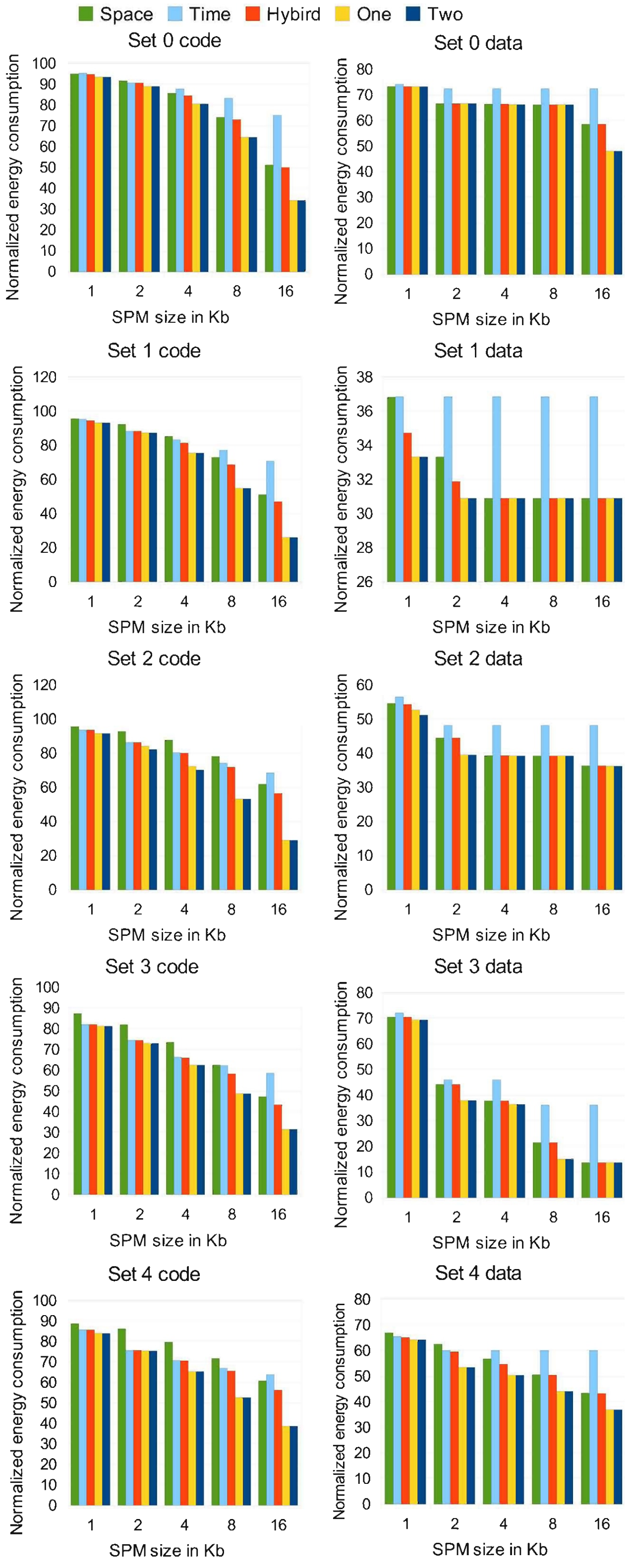


Figure 8: Energy consumption related to the memory accesses normalized to the case where the SPM is not used (percentage)

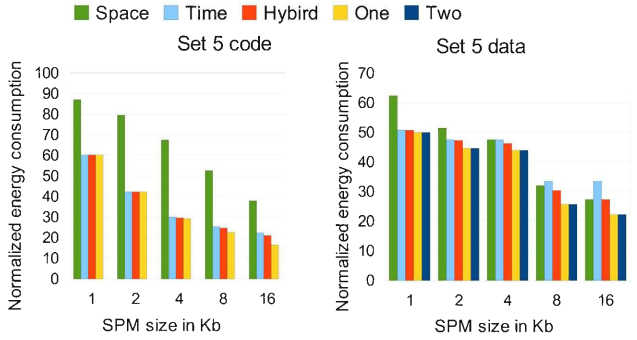


Figure 9: Energy consumption related to the memory accesses normalized to the case where the SPM is not used (percentage)

Acknowledgements

This work is supported by Toshiba and the CREST ULP program of JST.

8. REFERENCES

[1] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, 2002.

[2] B. Egger, J. Lee, and H. Shin. Dynamic scratchpad memory management for code in portable systems with an mmu. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–38, 2008.

[3] B. Egger, J. Lee, and H. Shin. Scratchpad memory management in a multitasking environment. In *EMSOFT '08*, pages 265–274, New York, NY, USA, 2008. ACM.

[4] Embedded Microprocessor Benchmark Consortium. EEMBC benchmark suite. `http://www.eembc.org/home.php`.

[5] L. Gauthier and T. Ishihara. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *ESTIMedia '09*, Grenoble, France, 2009.

[6] E. G. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. *SIGARCH Comput. Archit. News*, 28(2):107–116, 2000.

[7] ILOG. CPLEX LP solver. `http://www.ilog.com/products/cplex`.

[8] ILOG. ILOG CPLEX user's manual, 2009. `ftp://public.dhe.ibm.com/software/websphere/ilog/docs/optimization/cplex/ps_usrmancplex.pdf`.

[9] T. Ishihara, S. Yamaguchi, Y. Ishitobi, T. Matsumura, Y. Kunitake, Y. Oyama, Y. Kaneda, M. Muroyama, and T. Sato. Ample: An adaptive multi-performance processor for low-energy embedded applications. In *SASP '08*, pages 83–88, 2008.

[10] A. Mizuno, H. Uetani, and H. Eichel. Design methodology and system for a configurable media embedded processor extensible to vliw architecture. In *ICCD '02*, page 2, Washington, DC, USA, 2002. IEEE Computer Society.

[11] C. A. Moritz, M. Frank, and S. P. Amarasinghe. Flexcache: A framework for flexible compiler generated data caching. In *IMS '00*, pages 135–146, London, UK, 2001. Springer-Verlag.

[12] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97*, page 7, Washington, DC, USA, 1997. IEEE Computer Society.

[13] R. Pyka, C. Fassbach, M. Verma, H. Falk, and P. Marwedel. Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications. In *SCOPES '07*, pages 41–50, New York, NY, USA, 2007. ACM.

[14] J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *CASES '01*, pages 15–23, New York, NY, USA, 2001. ACM.

[15] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[16] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *CASES '06*, pages 401–410, New York, NY, USA, 2006. ACM.

[17] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. In *CODES+ISSS '08*, pages 37–42, New York, NY, USA, 2008. ACM.

[18] 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*, 2:180–188, 2009.

[19] H. Takase, H. Tomiyama, and H. Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *DATE '10*, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Toshiba. MeP processor. `http://www.semicon.toshiba.co.jp/eng/product/micro/mep/document/index.html`.

[21] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[22] University of Michigan. MiBench benchmark suite. `http://www.eecs.umich.edu/mibench/`.

[23] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In *Estimedia '05*, volume 0, pages 115–120, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[24] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. In *ASP-DAC '03*, pages 77–83, New York, NY, USA, 2003. ACM.

[25] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *WMPI '04*, pages 114–120, New York, NY, USA, 2004. ACM.