Code and Data Placement for Embedded Processors with Scratchpad and Cache Memories

Ishitobi, Yuriko
Graduate School of Information Science and Electrical Engineering, Kyushu University

Ishihara, Tohru
System LSI Research Center, Kyushu University

Yasuura, Hiroto
Faculty of Information Science and Electrical Engineering, Kyushu University

Noname manuscript No.
(will be inserted by the editor)

Code and Data Placement for Embedded Processors with Scratchpad and Cache Memories

Yuriko ISHITOBI · Tohru ISHIHARA · Hiroto YASUURA

Received: date / Accepted: date

Abstract This paper proposes a code placement problem, its ILP formulation, and a heuristic algorithm for reducing the total energy consumption of embedded processor systems including a CPU core, on-chip and off-chip memories. Our approach exploits a non-cacheable memory region for an effective use of a cache memory and as a result, reduces the number of off-chip accesses. Our algorithm simultaneously finds a code layout for a cacheable region, a scratchpad region, and the other non-cacheable region of the address space so as to minimize the total energy consumption of the processor system. Experiments using a commercial embedded processor and an off-chip SDRAM demonstrate that our algorithm reduces the energy consumption of the processor system by 23% without any performance degradation compared to the best result achieved by the conventional approach

Keywords code placement · energy reduction · embeded processor

Yuriko ISHITOBI
Graduate School of Information Science and Electrical Engineering, Kyushu University
Motooka 744, Nishiku, Fukuoka-shi, 819-0395 Japan
E-mail: ishitobi@c.csce.kyushu-u.ac.jp

Tohru ISHIHARA
System LSI Research Center, Kyushu University
Momochihama 3-8-33, Sawara-ku, Fukuoka-shi, 814-0001 Japan
E-mail: ishihara@slrc.kyushu-u.ac.jp

Hiroto YASUURA
Faculty of Information Science and Electrical Engineering, Kyushu University
Motooka 744, Nishi-ku, Fukuoka-shi, 819-0395 Japan
E-mail: yasuura@c.csce.kyushu-u.ac.jp

1 Introduction

On-chip memories are one of the most power hungry components of today's microprocessors. For example, ARM920T microprocessor dissipates 43% of the power in its cache memories [1,2]. StrongARM SA-110 processor, which specifically targets low power applications, dissipates about 27% of the power in its instruction cache [3]. Many techniques have been proposed for optimizing cache configuration considering tradeoff between energy consumption of off-chip memory and cache memory [4–8]. All these approaches use the fact that while a bigger cache consumes more energy per access, it can reduce the number of cache misses and as a result can reduce the energy consumption of the off-chip memory.

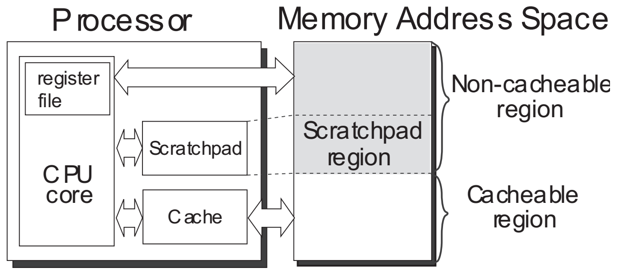

Fig. 1 Overview of Our Code Layout

One of the most effective approaches for reducing the energy consumption of off-chip memories without increasing a cache size is the code placement technique [9–14]. The idea is to modify the placement of basic blocks, procedures and/or data objects in the address space so that the number of cache conflict misses is minimized. This can significantly reduce the number of cache misses and improve a program's execution time. However, none of the previous methods takes the non-cacheable memory region into consideration in their methods. The non-cacheable memory region represents address spaces assigned for a scratchpad memory and the other memory sections which can be directly accessed from CPU core without caching the data. If we consider the non-cacheable section of address space, we can find better code placement which further reduces the energy consumption of processor systems.

Scratchpad memory can be used as a design alternative for the on-chip cache memory [15]. Current embedded processors particularly in the area of multimedia applications have on-chip scratchpad memories. In cache memory systems, the mapping of program elements is done during runtime, while in scratchpad memory systems this is done by the programmer or the compiler. Unlike the cache memory, the scratchpad memory does not need tag search operations and, as a result, it is more power efficient than the cache memory if programmers or compilers can optimally allocate code and data on the scratchpad memory [16]. Our approach also exploits a non-cacheable memory region for bypassing streaming data which has a low temporal locality.

This paper proposes a code placement technique which reduces the energy consumption of embedded processor systems including a CPU core, scratchpad, cache, and off-chip memories. This paper is an extension of our previous work [17]. In this paper, a code placement problem for minimizing the energy consumption of embedded processor systems is formally defined. An ILP formulation for the code placement problem is given as well. To the best of our knowledge, this is the first ILP formulation of the code placement problem in which the optimal code layouts for a cacheable region, a scratchpad region, and the other non-cacheable region are simultaneously determined.

The rest of the paper is organized as follows. Section 2 summarizes previous work and our approach. A formal definition of our code placement problem and its integer linear programming (ILP) formulation is presented in Section 3. Section 4 presents a heuristic algorithm. Experimental results are summarized in Section 5. The paper concludes in Section 6.

2 Previous Work and Our Approach

2.1 Cache Conflict Miss Reduction

We first explain the idea behind the conventional code placement technique. Consider a direct-mapped cache of size C ($= 2^m$ words) whose cache line size is L words, i.e., L consecutive words are fetched from the main memory on a cache read miss. In a direct-mapped cache, the cache line containing a word located at memory address M can be calculated by $(\lfloor M/L \rfloor mod \lfloor C/L \rfloor)$. Therefore, two memory locations M_i and M_j will be mapped onto the same cache line if the following condition holds,

$$\left(\left\lfloor \frac{M_i}{L} \right\rfloor - \left\lfloor \frac{M_j}{L} \right\rfloor \right) mod \frac{C}{L} = 0 \tag{1}$$

Several code placement techniques have used the above formula.

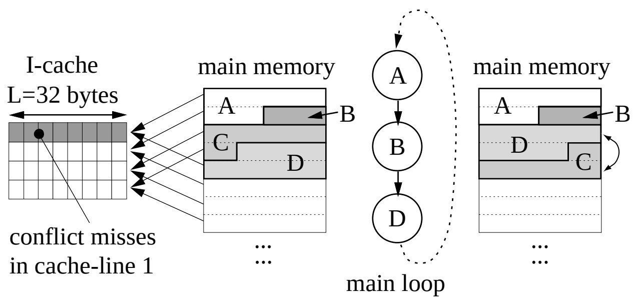

Fig. 2 An Example of Code Placement

Suppose we have a direct mapped cache with 4 cache-lines, where each cache-line is 32 bytes. Functions A, B, C and D are placed in the main memory as shown in the left side of Figure 2. If functions A, B, and D are accessed in a loop, conflict misses occur because A and D are mapped onto the same cache line. If the locations of C and D are swapped as shown in the right side of Figure 2, the cache conflict is resolved. Code placement modifies the placement of basic blocks or functions in the address space so that the total number of cache conflict misses is minimized [9–14]. We use this idea in our algorithm.

2.2 Code Placement for Scratchpad Memory

Unlike the cache memory, the scratchpad memory does not need complicated hardware mechanisms for retaining LRU (Least Recently Used) lines and for the tag search operations. Therefore, scratchpad memory is more power efficient than the cache memory if programmers or compilers can optimally allocate code and data on the scratchpad memory. Banakar et al. proposed a technique for selecting an on-chip memory configuration from various sizes of cache and scratch pad memories [15]. Their experiments show that scratchpad based approach outperforms cache-based approach on almost all aspects. For example, the total energy consumption of scratchpad based systems is less than that of cache-based systems by 40% on an average. Steinke et al. proposed a compiler-oriented optimization technique for selecting program and data parts which should be placed in the scratchpad memory. Their experiments showed that the energy consumption of the memory system was reduced by 40% compared to the cache based approach [16]. However, their approach does not effectively work for a processor with both scratchpad and cache memories.

2.3 Cache Bypassing

The cache energy consumption can be reduced by simply not caching data which has a low temporal locality. This also leads to an effective use of the cache memory and, as a result, can reduce the number of cache misses. Johnson et al. perform a run-time analysis to detect spatial and temporal locality [18]. By monitoring data access patterns and bypass non-temporal data through the use of cache bypass buffers. Note that the *non-temporal* data denotes data which has a low temporal locality. Rivers et al. proposed the NTS (non-temporal stream) cache for bypassing streaming data [19]. Their primary focus is on reducing conflict misses in direct-mapped L1 caches by using a separate fully associative buffer, and selectively caching data in the main cache. Although the idea of cache bypassing is used in our approach, our approach does not need any additional hardware like the cache bypass buffers nor the stream buffers. Our approach identifies the non-temporal streaming data from profiling information and places the non-temporal data onto a non-cacheable memory region so that the non-temporal streaming data is not cached. To the best of our knowledge, this is the first compiler-based technique which finds non-temporal data objects bypassed without caching.

2.4 Motivational Example

The conventional code placement algorithm for a scratchpad memory does not necessarily find the optimal code and data placement if the target processor employs both scratchpad and cache memories. Suppose we have three functions and they are executed on a processor with 2KB scratchpad and 2KB cache memories as shown in Figure 3.

The code size of each function is 2KB as well. In this case, only one function can be placed in the scratchpad memory and the other two functions are allocated on a cache region in the address space. In the first half

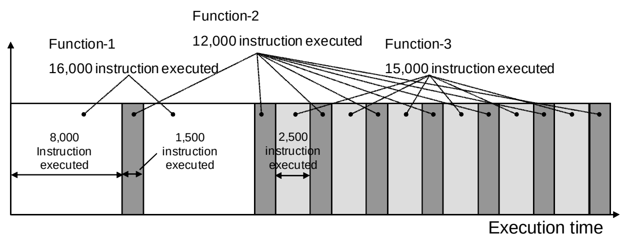
Function-1
16,000 instruction executed
Function-2
12,000 instruction executed
Function-3
15,000 instruction executed
8,000 Instruction executed
1,500 instruction executed
2,500 instruction executed
Execution time

Fig. 3 An Example of Program Execution Schedule

of the execution, Function-1 and 2 are alternately executed, and the later, Function-2 and 3 are alternately executed. If Function-1 which occupies the processor for the longest period of time is placed in the scratchpad, a lot of cache misses occur when Function-2 switches to Function-3 because these functions share the cache memory most of the time.

The number of cache misses can be drastically reduced by allocating Function-2 on the scratchpad memory, while the conventional technique allocates Function-1 to the scratchpad memory because it occupies the processor for the longest period of time. Even if there is no space left in the scratchpad memory, we still have a chance to reduce the number of cache misses by allocating Function-2 on the other non-cacheable memory regions. Our approach selectively finds program and data objects which should be placed in the scratchpad memory with taking the cache behavior into consideration.

2.5 Our Approach

As mentioned above, finding code placements separately for a cacheable region, a scratchpad region and the other non-cacheable region does not result in minimizing the total energy consumption of the embedded system. Our approach simultaneously finds code layouts for the cacheable region, the scratchpad region and the other non-cacheable region so as to minimize the total energy consumption of embedded processor systems. Note that the scratchpad memory is used as a compile-time memory in our approach. The overview of our technique for optimizing the code placement is shown in Figure 4. We first extract hardware dependent parameter values like the energy consumption and clock cycles required for a memory access using a netlist of the target processor. Instruction and data address traces for a target application program can be obtained using instruction-set simulator (ISS). Then, our code placement algorithm finds the optimal code layout using these previously obtained hardware and software characteristics.

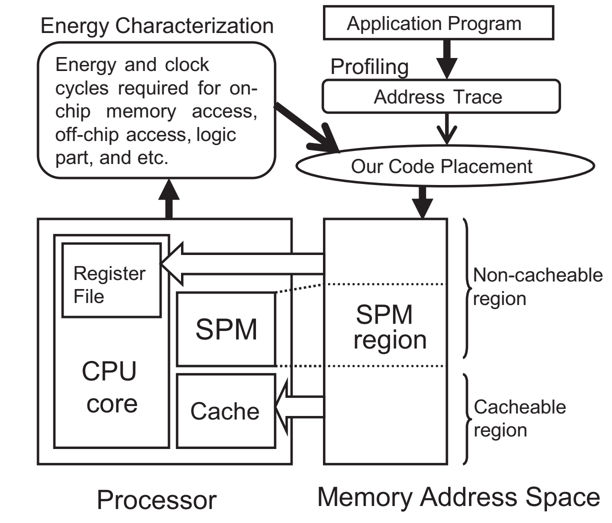

Fig. 4 Code Layout Optimization Flow

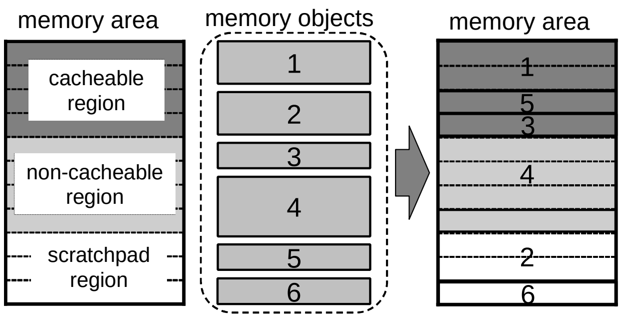

Fig. 5 An Example of Memory Object Placement

3 Problem Definition

3.1 Assumptions

This section gives assumptions for our code placement problem. Our code placement problem is designed for finding locations of memory objects in a memory address space as shown in Figure 5. The memory objects include functions, global variables, and constants. For the problem formulation, we assume all memory objects are aligned to a boundary of a memory block whose size is equal to the cache line size. Any memory object can be placed in a cacheable region, a scratchpad region or a non-cacheable region. In the rest of this paper, we call the other non-cacheable region which can be directly accessed from CPU core without caching as a non-cacheable region simply. Since the locations of memory objects in the cacheable region affects the number of cache misses as described in Section 2.1, the optimal order of memory objects in the cacheable region must be found for minimizing the energy consumption. This paper extends a code placement problem previously proposed by Tomiyama et al. in [11] which targets instruction codes only. Our code placement problem targets not only instruction codes but also global variables and constants. Therefore, in this paper, we formulate the number of write-backs from dirty cache lines to a main memory so that we can find the optimal order of data objects in the cacheable region.

3.2 Preliminaries

In the rest of this paper, the following definitions and notations are used.

N_{obj}	: The total number of memory objects
i	: A memory object ID number
B_i	: The number of memory blocks required for placing the memory object i
X_i	: The number of accesses to the memory object i
N_{set}	: The number of cache sets
N_{way}	: The number of cache ways
S_{spm}	: The number of memory blocks assigned to scratchpad memory

The values of above parameters are given as inputs of the code placement problem. The values for X_i can be obtained by using instruction set simulator. The locations of the memory object i are determined by 0-1 variables c_i, s_i, u_i, and $y_{i,i'}$ defined below.

$$c_i = \begin{cases} 1 : \text{if memory object } i \text{ is placed in} \\ \quad \text{a cacheable region} \\ 0 : \text{otherwise} \end{cases} \tag{2}$$

$$s_i = \begin{cases} 1 : \text{if memory object } i \text{ is placed in} \\ \quad \text{a scratchpad region} \\ 0 : \text{otherwise} \end{cases} \tag{3}$$

$$u_i = \begin{cases} 1 : \text{if memory object } i \text{ is placed in} \\ \quad \text{a non-cacheable region} \\ 0 : \text{otherwise} \end{cases} \tag{4}$$

$$y_{i,i'} = \begin{cases} 1 : \text{if } c_i = 1 \land c_{i'} = 1 \\ \quad \text{and memory object } i \text{ is placed at addresses} \\ \quad \text{higher than that of memory object } i' \\ 0 : \text{if } c_i = 0 \lor c_{i'} = 0 \\ \quad \text{or memory object } i \text{ is placed at addresses} \\ \quad \text{lower than that of memory object } i' \end{cases} \tag{5}$$

Suppose we have three memory objects 1, 5 and 3 as shown in Figure 6 (a). In this case, since memory object 1 is placed at addresses lower than those of memory objects 3 and 5 in cacheable region, $\{y_{1,3}, y_{1,5}\} = \{0, 0\}$. Similarly, since the memory object 5 is placed between memory objects 1 and 3, $\{y_{5,1}, y_{5,3}\} = \{1, 0\}$. If a memory object i' is placed to the scratchpad region or non-cacheable region and if a memory object i is placed to the cacheable region, any $y_{i,i'}$ should be 0. This is because the addresses for scratchpad and non-cacheable regions are assumed to be higher than those of cacheable region in our formulation. Therefore, if the memory object 5 is placed in the scratchpad region and others are placed in the cacheable region, any $y_{i,5}$ is set to 0. More specifically, $(y_{1,5}, y_{3,5}) = (0, 0)$ in this case as shown in Figure 6 (b).

The following formula (6) ensures that any memory object is placed in one of cacheable, scratchpad and non-cacheable regions.

$$c_i + s_i + u_i = 1 \tag{6}$$

3.3 Objective Function

The goal of our code placement is to minimize the total energy consumption required for executing a given application program with or without a constraint of total execution time. The objective function of the problem is given by (7). TE_{total} is the total energy consumption of CPU and off-chip memory. E_{cache} denotes the energy consumption for a cache access. E_{miss} and E_{wb} represent the energy consumptions for a cache miss and a cache write-back, respectively. N_{cache}, N_{miss} and N_{wb} are the numbers of cache accesses, cache misses and cache misses with write-back, respectively. E_{spm} denotes the energy dissipation required for an access to the scratchpad memory. N_{spm} is the number of accesses to the scratchpad memory. α is the number of off-chip memory accesses in case of a cache miss or a cache write-back, which can be determined by a specification of the target processor. P_{exe} and P_{stall} represent average power consumption of logic part when executing instructions and that of logic part when the pipeline is stalling, respectively. t_{exe} and t_{stall} represent the total time spent for executing instructions and that spent for the pipeline stall, respectively. E_{off} and P_{off} denote the energy consumption for an off-chip access and the static power consumption in off-chip memory. N_{offs} is the number of non-burst accesses to the non-cacheable region. CN_{inst}, CN_{miss}, CN_{wb} and CN_{offs} respectively denote the average numbers of clock cycles needed for executing an instruction, an off-chip burst-read in case of a cache miss, an off-chip burst-write for a cache write-back and an off-chip non-burst access. N_{inst} represents the total number of instructions executed for a given application program. CT is the clock cycle time of the target processor. E_{off} and P_{off} are obtained from a specification of the target memory device. The values of E_{cache}, E_{miss}, E_{wb} and E_{spm} can be estimated using a circuit simulator and a logic simulator. P_{exe} and P_{stall} can be estimated from results of logic level simulation. The values of CN_{miss}, CN_{wb}, CN_{offs} and α can be obtained from a specification of the target processor. The value of CN_{inst} is estimated from the result of cycle accurate simulation of each application program.

Our code placement problem is defined as follows; "For given B_i, X_i, N_{set}, N_{way}, S_{spm}, E_{cache}, E_{miss}, E_{wb}, E_{spm}, E_{off}, P_{exe},P_{stall} P_{off}, CT, CN_{inst}, CN_{miss}, CN_{wb}, CN_{off} and a memory footprint of the target application program TR, find values of c_i, s_i,u_i and $y_{i,i'}$ which minimize TE_{total}".

$$TE_{total} = TE_{cache} + TE_{spm} + TE_{logic} + TE_{off} \tag{7}$$

$$\begin{aligned}
TE_{cache} &= N_{cache} \cdot E_{cache} + N_{miss} \cdot E_{miss} + N_{wb} \cdot E_{wb} \\
TE_{spm} &= N_{spm} \cdot E_{spm} \\
TE_{logic} &= P_{exe} \cdot t_{exe} + P_{stall} \cdot t_{stall} \\
TE_{off} &= (\alpha N_{miss} + \alpha N_{wb} + N_{offsr})E_{off} \\
&\quad + P_{off} \cdot t_{all} \\
t_{all} &= t_{exe} + t_{stall} \qquad (8) \\
t_{exe} &= CT \cdot N_{inst} \cdot CN_{inst} \\
t_{stall} &= CT(N_{miss} \cdot CN_{miss} \\
&\quad + N_{offs} \cdot CN_{offs} + N_{wb} \cdot CN_{wb})
\end{aligned}$$

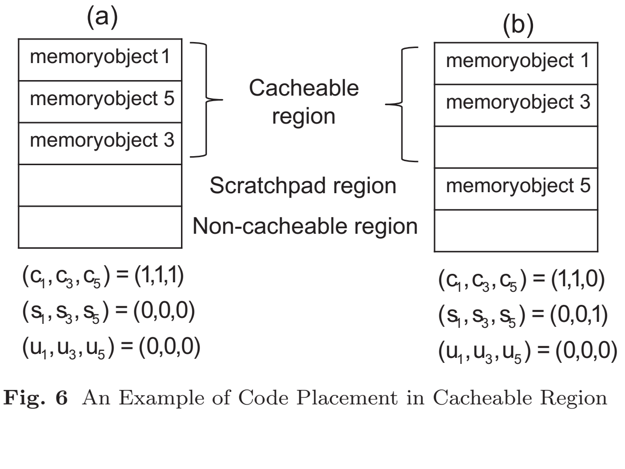

Fig. 6 An Example of Code Placement in Cacheable Region

3.4 Detailed Problem Formulation

The values of N_{cache}, N_{spm} and N_{offs} can be obtained from the following formulas.

$$N_{cache} = \sum_{i=1}^{N_{obj}} c_i \cdot X_i \quad (9)$$

$$N_{spm} = \sum_{i=1}^{N_{obj}} s_i \cdot X_i \quad (10)$$

$$N_{offs} = \sum_{i=1}^{N_{obj}} u_i \cdot X_i \quad (11)$$

Since the size of a scratchpad memory is limited and is usually much smaller than the size of object codes, the following constraint is introduced.

$$\sum_{i=1}^{N_{obj}} s_i \cdot B_i \leq S_{spm} \quad (12)$$

We calculate the values of N_{miss} and N_{wb} from a memory footprint TR. The following shows an example of TR.

$$p_{2,0}p_{4,0}p_{1,0}p_{1,1}p_{2,0}p_{4,0}p_{1,0}p_{1,1}p_{2,0}p_{4,1}p_{2,0}p_{2,1} \quad (13)$$

$p_{i,j}$ represents an access to a memory block j in a memory object i. The above TR shows that memory objects are accessed in the order shown in Figure 7.

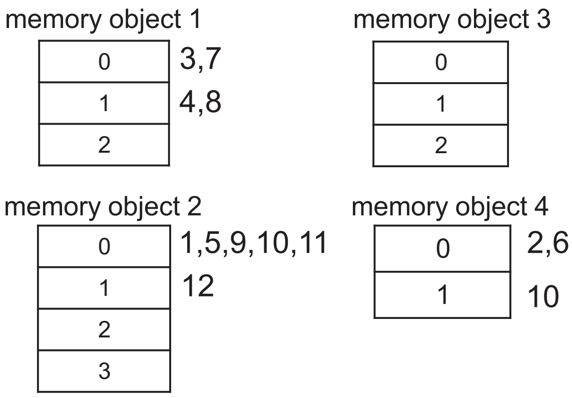

Fig. 7 An Example of Memory Footprints

Next, let us introduce a *block address* which represents an address of a memory block in a memory address space. The size of the memory block is equal to the cache line size as described in Section 3.1. The block address assigned to the memory block j in the memory object i is calculated by (14).

$$bn(i,j) = start\_address + \sum_{i'=1}^{N_{obj}} y_{i,i'} \cdot B_{i'} + j \quad (14)$$

where the $start\_address$ represents a base address of the cacheable region. This formula calculates the cumulative number of memory blocks placed at addresses lower than the address of the memory block j of the memory object i.

Next, we introduce $d_{i,j,l}$ and $g_{i,j,l}$ for calculating N_{miss} and N_{wb}, respectively. The $d_{i,j,l}$ represents a sequence of memory block accesses which appear between lth and $(l+1)$th appearances of $p_{i,j}$ in a given TR as follows.

$$d_{i,j,l} = \{p_{i',j'} | p_{i',j'} (i \neq i' or j \neq j') \text{ appear between the } l\text{th and the } (l+1)\text{th appearances of } p_{i,j} \text{ in } TR\}$$

For example in (13), $d_{1,0,0} = \{p_{1,1}, p_{2,0}, p_{4,0}\}$, $d_{2,0,0} = \{p_{4,0}, p_{1,0}, p_{1,1}\}$, $d_{2,0,1} = \{p_{4,0}, p_{1,0}, p_{1,1}\}$ and $d_{2,0,2} = \{p_{4,1}\}$. Since $d_{i,j,l}$ usually appears more than once in a typical memory access footprint, we newly introduce $a_{i,j,k}$ to eliminate repetitions of $d_{i,j,l}$ $(l = 0, 1, 2\ldots)$. Note that $a_{i,j,k}$ $(k = 0, 1, 2, \ldots)$ is not equal to $a_{i,j,k'}$ if $k \neq k'$. Since $d_{2,0,0} = d_{2,0,1}$, this repetition can be removed. As a result, $a_{2,0,k}$'s can be defined as follows;

$$a_{2,0,0} = \{p_{4,0}, p_{1,0}, p_{1,1}\}, a_{2,0,1} = \{p_{4,1}\}.$$

We next introduce $f_{i,j,k}$ which represents the number of appearances of $a_{i,j,k}$ in a memory footprint TR. For example in (13), $f_{2,0,k}$'s are defined as follows;

$$f_{2,0,0} = 2, f_{2,0,1} = 1$$

The $g_{i,j,l}$ represents a sequence of $a_{i,j,k}$ which appear between the lth appearance of $p_{i,j}$ and the next *write* access to the jth memory block of the memory object i in the TR.

$$g_{i,j,l} = \begin{cases} \{a_{i,j,k} | a_{i,j,k} \text{ appear between the } l\text{th appearance of } p_{i,j} \text{ and the next } \textit{write} \text{ access to the } j\text{th block of the memory object } i \text{ in a } TR & : l\text{th } p_{i,j} \text{ is write} \\ \phi & : l\text{th } p_{i,j} \text{ is read} \end{cases}$$

Suppose the first and the third appearances of $p_{2,0}$ in (13) are write accesses and only the second $p_{2,0}$ is a read access. In this case, $g_{2,0,0} = \{a_{2,0,0}, a_{2,0,1}\}$. Similar to $d_{i,j,l}$, $g_{i,j,l}$ may appear more than once in TR. We eliminate the repetitions of $g_{i,j,l}$ by introducing $w_{i,j,k}$ $(k = 0, 1, 2, \ldots)$. Note that $w_{i,j,k} \neq w_{i,j,k'}$ if $k \neq k'$. The number of appearances of $w_{i,j,k}$ in TR is defined by $e_{i,j,k}$.

$$N_{miss} = \sum_{\forall a_{i,j,k}} f_{i,j,k} \cdot replace(a_{i,j,k}) \quad (15)$$

$$N_{wb} = \sum_{\forall w_{i,j,k}} e_{i,j,k} \cdot \mathit{writeback}(w_{i,j,k}) \tag{16}$$

N_{miss} in (15) represents the number of cache conflict misses which occur in a given TR. N_{wb} in (16) represents the number of write-backs which transfer data from dirty cache lines to the main memory. The write-back occurs in case that the target cache line accessed is dirty.

$$\mathit{replace}(a_{i,j,k}) = \begin{cases} 1 : \sum_{p_{i',j'} \in a_{i,j,k}} \mathit{conflict}(p_{i,j}, p_{i',j'}) \geq N_{way} \\ 0 : \text{otherwise} \end{cases} \tag{17}$$

$$\mathit{writeback}(w_{i,j,k}) = \begin{cases} 1 : \sum_{a_{i,j,m} \in w_{i,j,k}} \mathit{replace}(a_{i,j,m}) > 0 \\ 0 : \text{otherwise} \end{cases} \tag{18}$$

$\mathit{conflict}(p_{i,j}, p_{i',j'})$ in (17) denotes whether or not $p_{i,j}$ and $p_{i',j'}$ are mapped on the same cache line. If these two are mapped on the same cache line, the value of $\mathit{conflict}(p_{i,j}, p_{i',j'})$ is set to 1 and otherwise it is set to 0 as shown in (19).

$$\mathit{conflict}(p_{i,j}, p_{i',j'}) = \begin{cases} 1 : \text{if } c_i \neq 0,\ c_{i'} \neq 0 \text{ and} \\ \quad (bn(i,j) \bmod N_{set}) \\ \quad = (bn(i',j') \bmod N_{set}) \\ 0 : \text{otherwise} \end{cases} \tag{19}$$

The value of TE_{total} can be calculated using (9)-(12), (14)-(19), TR, c_i, s_i, u_i, $y_{i,i'}$ and the other hardware dependent parameters shown in 3.2.

3.5 ILP Formulation

In formulations presented in Sections 3.2, 3.3 and 3.4, the objective function and several constraints are not expressed in a linear formula. This section shows linearizations for these function and constraints.

First, we linearize (5). The function $y_{i,i'}$ defined in (5) can be modified into the following linear inequalities.

$$\begin{gathered} -y_{i,i'} + c_{i'} \geq 0 \\ y_{i,i'} + y_{i',i} + c_i + c_{i'} < 4 \\ y_{i,i'} + y_{i',i} - c_i - c_{i'} \geq -1 \\ 0 \leq y_{i,i'} + y_{i',i''} - y_{i,i''} \leq 1 \\ y_{i,i'} \in \{0,1\} \end{gathered} \tag{20}$$

The first inequality in (20) sets $y_{i,i'}$ to 0 in case that the memory object i' is placed in the scratchpad or the non-cacheable region. The second and the third inequalities set one of $y_{i,i'}$ and $y_{i',i}$ to 1 and the other one to 0 if both of memory objects i and i' are placed in the cacheable region. The fourth inequality prevents memory objects from being in a three cornered deadlock. Suppose the memory object i is placed at addresses higher than that of memory object i' and the memory object i' is placed at addresses higher than that of memory object i''. In this case the memory object i must be placed at addresses higher than that of memory object i''.

Next, we linearize (19). A large integer number U is used in this linearization. We introduce new variables $q_{(i,j),(i',j')}$ and $z_{(i,j),(i',j')}$. These variables intuitively hold the values of $\mathit{conflict}(p_{i,j}, p_{i',j'})$ and $(bn(i,j) - bn(i',j'))/N_{set}$, respectively.

$$\begin{gathered} 0 \leq (bn(i,j) - bn(i',j')) - N_{set} \cdot z_{(i,j),(i',j')} < N_{set} \\ 0 < (bn(i,j) - bn(i',j')) - N_{set} \cdot z_{(i,j),(i',j')} \\ +q_{(i,j),(i',j')} + (1 - c_i) + (1 - c_{i'}) \leq N_{set} \\ (bn(i,j) - bn(i',j')) - N_{set} \cdot z_{(i,j),(i',j')} \\ -(1 - q_{(i,j),(i',j')}) \cdot U \leq 0 \\ q_{(i,j),(i',j')} \in \{0,1\} \\ z_{(i,j),(i',j')} \in \mathbf{Z} \end{gathered} \tag{21}$$

The value of $q_{(i,j),(i',j')}$ is 1 if the memory objects i and i' are placed in two different memory blocks which are mapped on to the same cache line. New variables $v_{i,j,k}$ are introduced to linearize (17), which intuitively hold the values of $\mathit{replace}(a_{i,j,k})$. Using $v_{i,j,k}$, (17) is replaced by the following linear expressions.

$$\begin{gathered} \sum_{p_{i',j'} \in a_{i,j,k}} q_{(i,j),(i',j')} + (1 - v_{i,j,k}) \cdot U \geq N_{way} \\ \sum_{p_{i',j'} \in a_{i,j,k}} q_{(i,j),(i',j')} - v_{i,j,k} \cdot U < N_{way} \\ v_{i,j,k} \in \{0,1\} \end{gathered} \tag{22}$$

At last, we linearize (18). We introduce variables $x_{i,j,k}$ which hold the values of $\mathit{writeback}(w_{i,j,k})$ intuitively. The following linear expressions replace (18).

$$\begin{gathered} \sum_{a_{i,j,m} \in w_{i,j,k}} v_{i,j,m} - x_{i,j,k} \cdot U \leq 0 \\ \sum_{a_{i,j,m} \in w_{i,j,k}} v_{i,j,m} + (1 - x_{i,j,k}) \cdot U > 0 \\ x_{i,j,k} \in \{0,1\} \end{gathered} \tag{23}$$

Finally, the code placement problem is successfully linearized. The variables to be determined in this problem are c_i, s_i, u_i, $y_{i,i'}$, $q_{(i,j),(i',j')}$, $z_{(i,j),(i',j')}$, $v_{i,j,k}$ and $x_{i,j,k}$.

4 Heuristics

The number of variables to be determined in our code placement problem defined in the previous section is more than 300,000 for a typical application program. Therefore, it is infeasible to use an ILP solver to find exact solution. Instead, in this section, we introduce heuristics for finding near optimal solutions. In our heuristics, the assumption for sizes of memory objects explained in Section3.1 is not necessary.

The heuristic algorithm consists of a main routine (*Main*) and three subroutines (*Explore scratchpad region*, *Swap locations of memory objects* and *Explore non-cacheable region*). As an input of our algorithm, a list of memory objects F, where the elements are sorted by descending order of access ratio is used. The access ratio is defined as the number of accesses to the memory object divided by its code size. Our algorithm, at the first of *Main*, calculates the total execution time for the original object code and save it to t_{const}. The main loop of the algorithm starts from the original code where all the memory objects are placed in a cacheable region. Then the optimal location of each memory object is found in the address space. This is done by choosing a single memory object o from top of F and changing the placement of the memory object in the address space. *Main* calls three subroutines and search the optimal location of each memory object. If the algorithm finds a new location for a memory object, where the energy consumption can be reduced by allocating the memory object to the location, $Location[*]$ is updated. The $Location[*]$ keeps locations of every memory objects.

```
Main
Input: $TR$,$F$,$S_{spm}$,$E_{cache}$,$E_{spm}$,$E_{off}$,$E_{miss}$,$E_{wb}$
$P_{logic}$,$P_{off}$,$CT$,$CN_{inst}$,$CN_{offb}$,$CN_{offs}$
Output: location of memory objects in optimized object
    code
$t_{const} = t_{all}$;  $TE_{min}$=$t_{min}$=infinity;
$S_{rest} = S_{spm}$;
repeat
  for($t = 0; t < |F|; t++$)do
    $o = F[t]$;
    $TE_{org} = TE_{min}$;  $t_{org} = t_{min}$;
    go Explore scratchpad region;
    go Swap locations of memory objects;
    go Explore non-cacheable region;
end of search:
    if($t_{min} \leq t_{const}$)then
      Update $Location[*]$;
    end if
  end for
until $TE_{min}$ stops decreasing
Output locations of memory objects
```

Explore scratchpad region checks whether or not the memory object o can be placed in a scratchpad region. This is done by comparing the space left in the scratchpad memory (S_{rest}) and the size of o ($SIZE[o]$). If there is enough space left in the scratchpad memory and if the energy consumption can be reduced by allocating o into the scratchpad region, o is placed there and the $Location[*]$ is updated accordingly. In this case, the algorithm skips *Swap locations of memory objects* and *Explore non-cacheable region.* In *Swap the locations of memory objects*, the location of memory object o is exchanged with the other memory object o'. Even if there is no space left for o in the scratchpad memory, *Swap the locations of memory objects* places o in the scratchpad memory by evicting o' from the scratchpad memory if this leads to an energy reduction. The o' evicted from the scratchpad memory is placed to the position where the o is formerly placed. Even if both of o and o' reside in the cacheable region, our algorithm swaps the locations of o and o' if this leads to an energy reduction. Locations of all memory objects which correspond to the optimal value of the object function are saved to $Location[*]$ in *Swap the locations of memory objects. Explore non-cacheable region* tentatively places the memory object o to the non-cacheable region and compares the energy consumption with that corresponding to the previous locations of o.

If there is an energy reduction, $Location[*]$ is updated to the locations where o is placed to the non-cacheable region. For each placement of o, the algorithm updates the memory address trace (TR) according to the location of memory objects and calculates the value of the objective function and the value of the execution time (t_{all}) using equations (7) and (8) presented in the section 3.3. TE_{min} keeps a tentatively minimized energy value and $Location[*]$ is updated.

After executing three subroutines, $Location[*]$ is updated if the execution time is less than or equal to t_{const}. The main loop continues as long as the total energy consumption (TE_{total}) reduces.

```
Explore scratchpad region
if($S_{rest} \geq SIZE[o]$)then
  place memory object $o$ to scratchpad region;
  $S_{rest} = S_{rest} - SIZE[o]$
  Update $TR$ according to new location;
  Calculate $TE_{total}$ and $t_{all}$;
  if($TE_{total} \leq TE_{min}$)
    $TE_{min} = TE_{total}$;  $t_{min} = t_{all}$;
    Update $Location[*]$;
    go end of search
  end if
end if
```

```
*Swap locations of memory objects*
for each $o' \in F$ and $o' \neq o$ do
  if($o'$ resides in scratchpad &&
    $SIZE[o] - SIZE[o'] \leq S_{rest}$)then
    Evict $o'$ from scratchpad region to cacheable
    region and place $o$ to scratchpad region;
    Update $TR$ according to new location;
    Calculate $TE_{total}$ and $t_{all}$;
  else if($o'$ placed in the cacheable region) then
      Swap the placement memory object $o$ for $o'$;
      Update $TR$ according to new location;
      Calculate $TE_{total}$ and $t_{all}$;
    end if
  end if
  if($TE_{total} < TE_{min}$) then
    $TE_{min} = TE_{total}$;  $t_{min} = t_{all}$;
    Update $Location[*]$;
  end if
end for
```

```
*Explore non-cacheable region*
Place $o$ to non-cacheable region;
Update $TR$ according to new locations;
Calculate $TE_{total}$ and $t_{all}$;
if($TE_{total} < TE_{min}$)then
  $TE_{min} = TE_{total}$;  $t_{min} = t_{all}$;
  Update $Location[*]$;
end if
```

5 Experimental Results

5.1 Target System

We target a system which consists of a CPU core, on-chip cache and scratchpad memories, and SDRAM as an off-chip main memory as shown in Figure 4.

A Micron's SDRAM DDR-II [20] and an SH3-DSP processor are used for our experiments. Our SH3-DSP processor design has a CPU core, a DSP core, an unified instruction and data cache, and a scratchpad memory. The processor is synthesized with a $0.18\mu m$ CMOS standard cell library and an SRAM module library. The power consumed in the logic part of the processor is estimated by gate-level simulation. First, we generate the Switching Activity Interchange Format (SAIF) file through gate-level simulation using $NCVerilog^{TM}$ from Cadence design systems. Then, the power consumption, P_{logic} is calculated using $PowerCompiler^{TM}$, a gate-level power calculation tool from SYNOPSYS. For calculating the memory access delays and the energy consumptions for the SRAM modules, NanoSim from SYNOPSYS is used. Specifications of the SRAM modules are described in Table 1. We choose supply voltages for the modules so that the access delay of each module is equal to or less than 954 [p sec] which is the read access time for the 4KB scratchpad memory. Note that the normal and the maximum supply voltages for the target process technology are 1.80V and 2.50V, respectively. In [21], Panwer et al. showed that cache-tag access and tag comparison do not need to be performed for all instruction fetches. Consider an instruction j executed immediately after an instruction i. If i is non-branch instruction and is not located at the end of the cache line, it is easy to detect that j resides in the same cache-way as i. Therefore, there is no need to perform a tag lookup for instruction j [21,22]. In this case, only a single cache way has to be activated. We refer to this case as single-way access. The energy consumption corresponding to the single-way access is shown in the third column of TABLE 1. On the other hand, a tag search operation is required for a non-sequential fetch such as a branch or a sequential fetch across a cache line boundary. In this case all cache ways including tag arrays and data arrays should be activated. We refer to this access as full-way access. The energy consumption for the full-way access is shown in the fourth column of TABLE 1.

Table 1 Specification of SRAM models

Memory Size	Supply Voltage	Single-way access energy	Full-way access energy
8kB4-way 128-set cache	1.70V	420.308pJ	2209.34pJ
16kB4-way 256-set cache	1.80V	573.696pJ	2946.672pJ
4kB scratchpad	1.75V	520.896pJ	
8kB scratchpad	2.25V	1381.440pJ	
16kB scratchpad	2.50V	2382.240pJ	

5.2 Benchmark Program

Three benchmark programs are used in our experiment; adpcm, compress, JPEG encoder, and MPEG2 encoder. All of the programs are compiled with "-O3" option. The GNU C compiler and debugger for SH3-DSP architecture are used for generating address traces. The trace of each benchmark program is one million instructions long. Table 2 shows the code size in byte and the number of memory objects for each benchmark program. The code size is the total size of all memory objects. The memory objects include functions, global variables and constants.

5.3 Results of ILP Solution

We use $CPLEX$, an ILP solver of ILOG, for obtaining solutions of our ILP problem. The objective of the ILP formulation is to evaluate the quality of our heuristics with respect to both computational time and values of

Table 2 Specification of benchmark programs

Benchmark	Code size	# of Memory objects
adpcm	1,380,608byte	48
compress	36,778byte	130
JPEG encoder	138,334byte	427
MPEG2 encoder	133,998byte	477

Table 3 Platform Specification

	CPU	Memory
Platform1	Intel(R) Xeon(R) CPU 3.0GHz	16GB
Platform2	Intel(R) Xeon(R) CPU 3.0GHz	8GB

objective function. An Intel Xeon quad CPU computer running Linux at 3GHz with 16GB or 8GB memory is used for the evaluation as shown in Table3. Platform1 is used for experiments of *compress* $MPEG2$ and $JPEG$ benchmarks. Platform2 is used for the experiment of *adpcm* benchmark.

Figure8, Figure9 and Table4 show results of *compress* and *adpcm* obtained by $CPLEX$ and our heuristics. **ITE** and **HTE** shown in Figure8 and Figure 9 respectively represent the total energy consumption obtained by $CPLEX$ and that obtained by our heuristic algorithm. **IST** and **HST** respectively represent the CPU time spent for $CPLEX$ and that for our heuristic algorithm. As shown in Figure8 and Figure9, differences between the energy consumption results obtained by the ILP solution and those obtained by our heuristics are negligible. Regarding the CPU time, our heuristics largely outperforms the ILP solution.

5.4 Result Using Our Heuristics

This subsection focuses on the results of energy consumption and performance of benchmark programs run on the SH3-DSP processor. Figure 10, 11, and 12 show the energy consumption and performance results for compress, JPEG encoder, and MPEG encoder, respectively. Left and right sides of each figure show results for 16kB and 8kB 4-way caches, respectively. 16kB, 8kB and 4kB scratchpad memories are examined as well. Vertical bar charts and straight lines represent the energy consumption and the number of cycles executed, respectively. The following five approaches are compared.

- ORG: Given benchmark programs are compiled with -O3 option. Every functions and data objects resides in a cacheable region in this case.
- CHE: Locations of functions and data objects are optimized using the technique proposed in [9,12]. Scratchpad memory is not used in this case.
- SPM: Functions and data objects are relocated to a scratchpad memory using the technique proposed in [16].
- CBN: Locations of functions and data objects are optimized by applying CHE just after applying SPM.
- OUR: Locations of functions and data objects are optimized by our algorithm presented in Section 4.

As one can see from Figure 10, 11, and 12, the energy consumption of any program in TABLE 2 optimized with our approach, OUR, is always the smallest of all. If we employ a large scratchpad memory on a chip, the object code optimized with SPM or CBN consumes higher energy than that optimized with CHE. This is because the scratchpad memory can be less energy efficient than the cache memory if the energy per access for the scratchpad memory is much larger than that for the cache memory. In this case, CHE outperforms SPM and CBN in terms of energy consumption. However, the object code optimized with CHE needs more execution time than that optimized with SPM or CBN. In

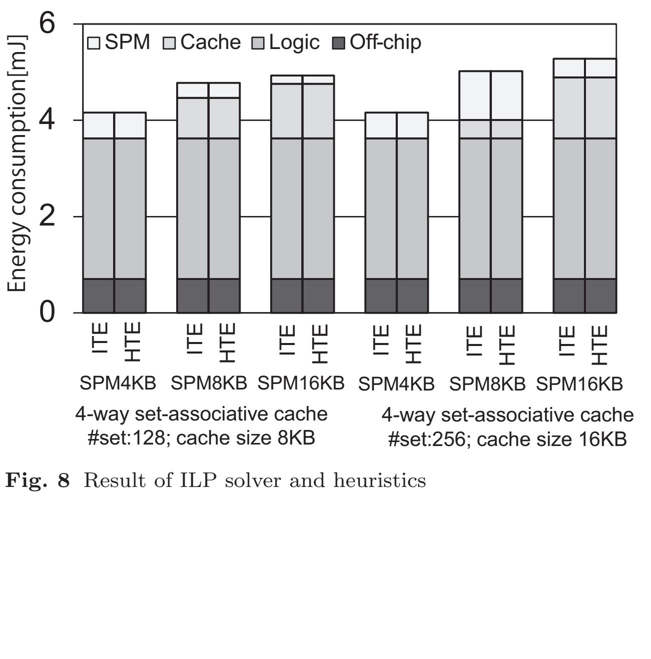

Fig. 8 Result of ILP solver and heuristics

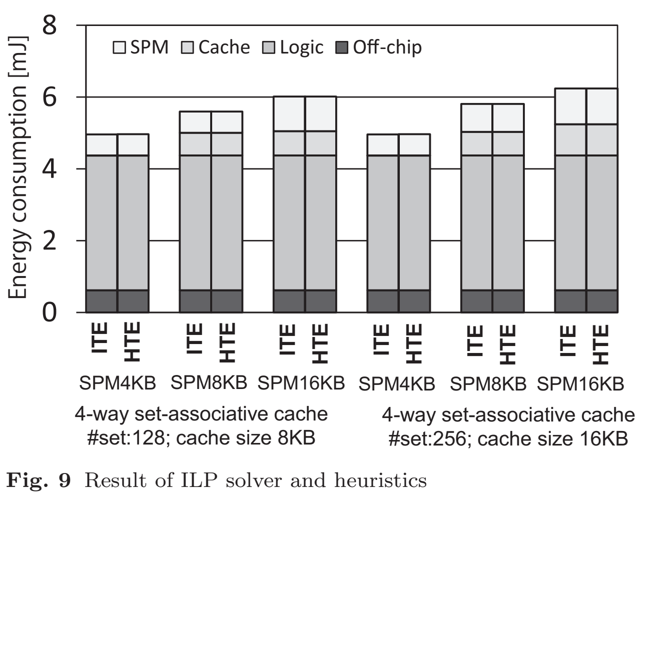

Fig. 9 Result of ILP solver and heuristics

Cache		4way #set128			4way #set:256		
SPM		4KB	8KB	16KB	4KB	8KB	16KB
compress	IST [sec]	47.3	120.1	1220.8	44.0	43.3	11711.2
	HST [sec]	6.5	56.1	39.6	6.1	18.0	58.7
adpcm	IST [sec]	1387.9	1139.2	1622.2	363.5	613.1	471.9
	HST [sec]	5.3	13.3	7.8	5.6	8.5	16.0
MPEG2	IST [sec]	over 300,000					
	HST [sec]	177.6	598.8	613.8	163.8	294.3	333.8
JPEG	IST [sec]	over 300,000					
	HST [sec]	1268.4	898.9	2108.0	2540.3	1444.9	2474.2

Table 4 The result of Solution Times

many cases, our approach is better than the best result obtained with the other approaches in terms of both energy consumption and execution time.

For compress, our approach outperforms on almost all aspects. For example, for the processor with 8KB 4-way set-associative cache and 16KB scratchpad memories, the object code optimized with OUR is 23% smaller in energy consumption and 2% faster in execution time compared to CBN. Even if the processor employs smaller scratchpad memory, our approach works effectively. For the processor with 8KB 4-way set-associative cache and 4KB scratchpad memories, the object code optimized with OUR is 10% smaller in energy consumption and 6% faster in execution time compared to CBN. For JPEG encoder, our approach works well for a processor with a 16-KB 4-way set-associative cache. For the processor with 16KB cache and 16KB scratchpad memories, the energy consumption can be reduced by 16% with a 0.6% improvement in execution time compared with CBN. The results obtained by our approach do not have obvious improvement for MPEG2 encoder compared to the results obtained by the other approaches. This is because only a few basic blocks are frequently executed in this program and these basic blocks can reside in cache or scratchpad memory in any memory configuration. However, important observation is that our approach always minimizes the energy consumption with only 0.1% performance loss compared to the best result achieved by the other approaches.

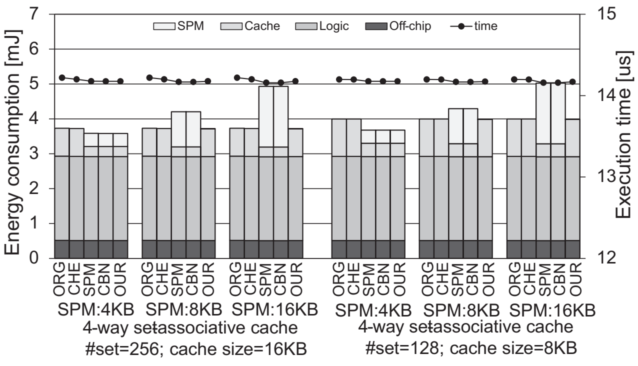

Fig. 12 Result for MPEG2 encoder

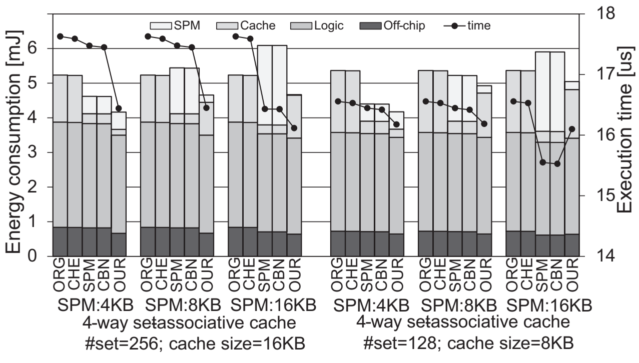

Fig. 10 Result for compress

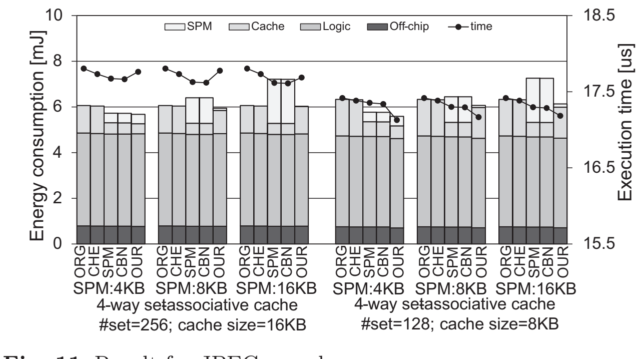

Fig. 11 Result for JPEG encoder

6 Conclusion

In this paper, a code placement problem, its ILP formulation, and a heuristic algorithm for reducing the total energy consumption of embedded processor systems are proposed. Our approach exploits a non-cacheable memory region for an effective use of a cache memory and as a result, reduces the total energy consumption of a processor system. Experiments using a commercial embedded processor and an off-chip SDRAM demonstrate that our algorithm reduces the energy consumption of the processor system by 23% without any performance loss compared to the best result achieved by the conventional approach. In the other case, the result of our approach is 10% smaller in energy consumption and 6% faster in execution time compared to the best result obtained by the conventional approach. Our future work will be devoted to extend our current algorithm to find a memory configuration and the best code layout for them concurrently.

Acknowledgment

This work is supported by VDEC, the Univ. of Tokyo with the collaboration of Renesas Technology Corp., ROHM Co., Ltd., Toppan Printing Co., Ltd., Synopsys, Inc. and Cadence Design Systems, Inc. This work is also supported by CREST ULP program of JST and Grant-in-Aid for Scientific Research (A) 19200004.

References

1. S. Segars, "Low Power Design Techniques for Microprocessors", ISSCC Tutorial note, Feb., 2001.
2. ARM Ltd., "ARM Processor Core Overview", http://www.arm.com/products/CPUs/
3. J. Montanaro et al., "A 160 MHz, 32b 0.5W CMOS RISC Microprocessor", In Proc. of ISSCC, Feb., 1996.
4. C. Su and A. Despain, "Cache Design Trade-offs for Power and Performance Optimization: A Case Study", In Proc. of ISLPED, pp.63-68, Aug., 1995.
5. P. Hicks, M. Walnock, and R. M. Owens, "Analysis of Power Consumption in Memory Hierarchies", In Proc. of ISLPED, pp.239-242, Aug., 1997.
6. Y. Li, and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems", In Proc. of DAC, pp.188-193, June, 1998.
7. W. T. Shine, and C. Chacrabarti, "Memory Exploration for Low Power, Embedded Systems", In Proc. of DAC, pp.140-145, June, 1999.
8. A. Malik, B. Moyer and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility", In Proc. of ISLPED, pp.241-243, July 2000.
9. S. McFarling, "Program Optimization for Instruction Caches", In Proc. of Int'l Conference on Architecture Support for Programming Languages and Operating Systems, pp.183-191, April 1989.
10. W. W. Hwu and P. P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler", In Proc. of ISCA, pp.242-251, May 1989.
11. H. Tomiyama and H. Yasuura, "Optimal Code Placement of Embedded Software for Instruction Caches", In Proc. of European Design and Test Conference, pp.96-101, March, 1996.
12. P. Panda, N. Dutt, and A. Nicolau, "Memory Organization for Improved Data Cache Performance in Embedded Processors", in Proc. of ISSS, pp.90-95, Nov. 1996.
13. A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient Procedure Mapping Using Cache Line Coloring", in Proc. of Programming Language Design and Implementation, pp.171-182, June, 1997.
14. S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior", ACM Trans. on Programming Languages and Systems, vol.21, no.4, pp.703-746, July, 1999.
15. 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 Proc. of CODES, pp.73-78, May, 2002
16. S. Stenke, L. Wehmeyer, B. Lee, P Marwedel, "Assigning Program and Data Objects to Scratchpad for Energy Reduction", in Proc. of DATE, pp.409-415, Mar. 2002.
17. Y. Ishitobi, T. Ishihara, H. Yasuura, "Code Placement for Reducing the Energy Consumption of Embedded Processors with Scratchpad and Cache Memories", in Proc. of ESTIMedia, pp.13-18, Mar. 2007.
18. T. L. Johnson, M. C. Merten, and W. W. Hwu, "Run-Time Spatial Locality Detection and Optimization", in Proc. of the 30th Int'l Symposium on Microarchitecture, pp. 57-64, Dec., 1997.
19. J. A. Rivers and E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design", in Proc. of the 25th Int'l Conference on Parallel Processing, pp.154-163, Aug., 1996.
20. "The Micron System Power Calculator", http://www.micron.com/support/designsupport/tools/powercalc/powercalc
21. R. Panwar, and D. Rennels, "Reducing the Frequency of Tag Compares for Low Power I-Cache Design", In Proc. of ISLPED, pp.57-62, August 1995.
22. M. Mullar, "Power Efficiency & Low Cost: The ARM6 Family", In Proc. of Hot Chips IV, August 1992.