Abstract — This paper proposes a code placement 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 code layouts 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 loss compared to the best result achieved by the conventional approach.

I. 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.

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]. In addition to the scratchpad memory, our approach exploits the other non-cacheable memory region for bypassing streaming data which has a low temporal locality.

Figure 1. Overview of Our Code Layout

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. To the best of our knowledge, this is the first code placement algorithm which simultaneously finds the code layout for a cacheable region, a scratchpad region, and the other non-cacheable region.

The rest of the paper is organized as follows. In Section 2, previous work and our approach are summarized. Energy estimation formula and our algorithm for find best code placement presented in Section 3. Section 4 presents experimental results. The paper concludes in Section 5.

II. PREVIOUS WORK AND OUR APPROACH

A. Cache Conflict Miss Reduction

This section first explains 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 \((M/L) \mod C/L\). 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 [9-14].

### B. Code Placement for Scratchpad Memory

Unlike the cache memory, a scratchpad memory (SPM) does not need any hardware mechanisms for retaining least recently used lines or the tag search operations. Therefore, the SPM is more power efficient than the cache memory if programmers or compilers can optimally allocate code and data on the SPM. Banakar et al. proposed a technique for selecting an on-chip memory configuration from various sizes of cache and SPM [15]. Their experiments show that SPM-based compile-time memory outperforms cache-based run-time memory on almost all aspects. For example, the total energy consumption of SPM-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 on the SPM. Their experiments showed that the energy consumption of the memory system is reduced by 40% compared to the cache based approach [16]. However, their approach does not always work effectively for a processor with both SPM and cache memories.

### C. 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 [17]. By monitoring data access patterns, non-temporal data is detected and bypassed through the use of cache bypass buffers. Note that the non-temporal data represents data which has a low temporal locality. Rivers et al. proposed the NTS (non-temporal stream) cache for bypassing streaming data [18]. 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 use 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 it 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.

### D. 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.

![Figure 2. An Example of Code Placement](image)

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 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 of Figure 2, the cache conflict is resolved. The conventional code placement algorithm 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.

![Figure 3. An Example of Unoptimal Code Placement](image)
E. 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.

![Energy Characterization](image)

Applying Program Profiling Address Trace

Our Code Placement

Processor Memory Address Space

Figure 4. Code Layout Optimization Flow

III. PROBLEM DEFINITION

A. Code Placement for Scratchpad and Cache

Our code placement algorithm finds the locations of memory objects in a memory address space. The memory objects include functions, global variables, and constants. Each memory objects can be placed on a cacheable region, a scratchpad region or the other non-cacheable region. Since the location of the memory object in the cacheable region affects the number of cache misses as described in Section II.A, our algorithm finds the best order of the memory objects in the cacheable region.

B. Energy Estimation Model

The objective function of the problem is given by (2). Our algorithm tries to minimize $TE_{all}$ by changing the placements of memory objects. $TE_{all}$ is the total energy consumption of CPU and off-chip memory. $E_{cache}$ and $E_{cache}$ denote the energy consumptions for a cache-read and a cache-write, respectively. $E_{miss}$ and $E_{dmiss}$ represent the energy consumptions for a cache miss and a cache miss with cache write-back, respectively. $N_{cacher}$, $N_{cachew}$, $N_{miss}$ and $N_{miss}$ are the numbers of cache-reads, cache-writes, cache misses and cache misses with write back, respectively. $E_{pmem}$ and $E_{pmem}$ denote the energy dissipations for read and write accesses to the scratchpad memory, respectively. $N_{pmem}$ and $N_{pmem}$ are the numbers of read and write accesses to the scratchpad memory, respectively. $P_{logic}$ and $P_{logic}$ represent the average power consumption of logic port and the total execution time, respectively. $E_{off}$ and $E_{off}$ denote the energy consumptions for off-chip read and write, respectively. $P_{off}$ is the static power consumption in off-chip memory. $N_{off}$ and $N_{off}$ respectively denote the number of burst off-chip reads and writes. $N_{off}$ and $N_{off}$ are the numbers of single-reads and single-writes to the other non-cacheable region, respectively. $CN_{instart}$, $CN_{instart}$, $CN_{offbr}$ and $CN_{offsw}$ respectively denote the average numbers of cycles needed for executing an instruction, an off-chip burst-read, an off-chip burst-write, an off-chip single-read, and an off-chip single-write. $N_{const}$ is the number of instructions executed, and $CT$ is the clock cycle time.

\[
TE_{all} = TE_{cache} + TE_{pmem} + TE_{log} + TE_{off} \tag{2}
\]

\[
TE_{cache} = N_{cacher} E_{cacher} + N_{cachew} E_{cachew} + N_{miss} E_{miss} + N_{miss} E_{miss}
\]

\[
TE_{off} = N_{pmem} E_{pmem} + N_{pmem} E_{pmem}
\]

\[
TE_{log} = P_{logic} CT
\]

\[
TE_{off} = (N_{offr} E_{offr} + (N_{offr} + N_{offw}) E_{offw}) + P_{off} \cdot t
\]

\[
t = CT \cdot (N_{inst} CN_{offbr} + N_{inst} CN_{offsw} + N_{inst} CN_{offsw})
\]

Cacheable region

Non-cacheable region

Scratchpad region

CPU core

register file

Scratchpad

Cache

Figure 4. Code Layout Optimization Flow

C. Algorithm

Our algorithm uses a list of memory objects $F$, where the elements are sorted by descending order of access frequency. The access frequency is defined as the number of accesses to the memory object divided by its code size. Our algorithm, at first, calculates the total execution time for the original object code and save it to $T_{mem}$. The main loop of the algorithm starts from the original object 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 $p$ from top of $F$ and changing the placement of the memory object in the address space. The algorithm first checks whether or not it can be placed on a scratchpad region. This is done by comparing the space left in the scratchpad memory ($SPM_{red}$) and the size of $p$ ($size[p]$). If there is enough space left in the scratchpad memory, $p$ is placed...
there. Even if there is no space left for \( p \) in the scratchpad memory, our algorithm places \( p \) in the scratchpad memory by evicting \( p' \) from the scratchpad memory if this leads to an energy reduction. Otherwise, our algorithm tries to find the best location for \( p \) in a cacheable region by changing the order of memory objects. After that, the algorithm tries to present in the previous sub-section.

\[
\text{Calculate} \ t_{\text{const}} = t_{\text{lat}}; \quad TE_{\text{min}} = t_{\text{min}} = \infty; \quad \text{SPM}_{\text{rest}} = \text{SPS};
\]

 repeat

 for \( t = 0; t < |F|; t++ \) do

 \( p = F[t] \);

 \( \text{BEST}_{\text{loc}} = p; \)

 if (SPM_{\text{rest}} ≥ SIZE(p)) then

 Place memory object \( p \) to scratchpad region;

 \( \text{SPM}_{\text{rest}} = \text{SPM}_{\text{rest}} - \text{SIZE}[p]; \)

 next;

 end if

 for each \( p' \in F \) and \( p' ≠ p \) do

 if \( p' \) resides in scratchpad region &&

 \( \text{SIZE}[p] - \text{SIZE}[p'] ≤ \text{SPM}_{\text{rest}} \) then

 Evict \( p' \) from scratchpad region to cacheable region and place \( p \) to scratchpad region;

 Update \( \text{TR} \) according to new location and calculate \( TE_{\text{all}} \) and \( t_{\text{lat}} \);

 if \( (TE_{\text{all}} = TE_{\text{min}}) \) then

 save present placement of \( p \) as \( \text{BEST}_{\text{loc}} \):

 \( TE_{\text{min}} = TE_{\text{all}}; \quad t_{\text{min}} = t_{\text{lat}} \);

 end if

 end if

 Insert memory object \( p \) in the place of \( p' \);

 Update \( \text{TR} \) according to the new location and calculate \( TE_{\text{all}} \) and \( t_{\text{lat}} \);

 if \( (TE_{\text{all}} ≤ TE_{\text{min}}) \) then

 Save the placement of \( p \) as \( \text{BEST}_{\text{loc}} \):

 \( TE_{\text{min}} = TE_{\text{all}}; \quad t_{\text{min}} = t_{\text{lat}} \);

 end if

 end for

 Place \( p \) to non-cacheable region;

 Update \( \text{TR} \) according to the new location and calculate \( TE_{\text{all}} \) and \( t_{\text{lat}} \);

 if \( (TE_{\text{all}} ≤ TE_{\text{min}}) \) then

 Save the placement of \( p \) as \( \text{BEST}_{\text{loc}} \):

 \( TE_{\text{min}} = TE_{\text{all}}; \quad t_{\text{min}} = t_{\text{lat}} \);

 end if

 if \( (t_{\text{min}} ≤ t_{\text{const}}) \) then

 Fix the location of \( p \) as \( \text{BEST}_{\text{loc}} \);

 end if

 end for

 until \( (TE_{\text{min}} \) stops decreasing)

 Output \locations of memory objects

end Procedure

\[ TE_{\text{min}} \] keeps a temporarily minimized energy value. The execution time is temporarily saved in \( t_{\text{min}} \). The location which minimizes the total energy consumption under a time constraint \( (t_{\text{const}}) \) is selected. The algorithm continues as long as the total energy consumption \( (TE_{\text{all}}) \) reduces. The computation time of the algorithm is quadratic in terms of the number of memory objects in the application program.

IV. EXPERIMENTAL RESULTS

A. 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 5.

![Figure 5. Target System](image)

A Micron’s SDRAM DDR-II [19] 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 0.18μm 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 NC-Verilog\textsuperscript{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 I. 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 [20], Panwer et al. showed that cache-tag access and tag comparison do not need to be performed for all instruction fetches. Consider an instruction \( i \) executed immediately after an instruction \( j \). 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 \( j \) [20][21]. 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 I. 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 I.

TABLE I. SPECIFICATION OF SRAM MODULES

<table>
<thead>
<tr>
<th>Memory Size</th>
<th>Supply Voltage</th>
<th>Single-way access energy</th>
<th>Full-way access energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>8kB 4-way 128-set cache</td>
<td>1.70V</td>
<td>420.308 pJ</td>
<td>2209.34 pJ</td>
</tr>
<tr>
<td>16kB 4-way 256-set cache</td>
<td>1.80V</td>
<td>573.696 pJ</td>
<td>2946.672 pJ</td>
</tr>
<tr>
<td>4kB scratchpad</td>
<td>1.75V</td>
<td>520.896 pJ</td>
<td></td>
</tr>
<tr>
<td>8kB scratchpad</td>
<td>2.25V</td>
<td>1381.440 pJ</td>
<td></td>
</tr>
<tr>
<td>16kB scratchpad</td>
<td>2.50V</td>
<td>2382.240 pJ</td>
<td></td>
</tr>
</tbody>
</table>

B. Benchmark Programs

Three benchmark programs are used in our experiment; 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 II shows the code size in byte and the number of memory objects for each benchmark program. The memory objects include functions, global variables and constants.

TABLE II. SPECIFICATION OF BENCHMARK PROGRAMS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Code Size</th>
<th># Memory Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>36,778 byte</td>
<td>130</td>
</tr>
<tr>
<td>JPEG encoder</td>
<td>138,334 byte</td>
<td>427</td>
</tr>
<tr>
<td>MPEG2 encoder</td>
<td>133,998 byte</td>
<td>477</td>
</tr>
</tbody>
</table>

C. Results

Figure 6, 7, and 8 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 III.

As one can see from Figure 6, the energy consumption of any program in TABLE II 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-based compile-time memory is less energy efficient than the cache-based run-time memory if the scratchpad memory size is larger than the cache memory size. 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 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 18% with a 1% 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.

V. CONCLUSION

In this paper, a code placement algorithm for reducing the energy consumption of embedded processor systems is 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 program of JST and Grant-in-Aid for Scientific Research (A) 19200004.

REFERENCES