Minimizing Energy Consumption of Embedded Systems via Optimal Code Layout

Chen-Wei Huang and Shiao-Li Tsao, *Member*, *IEEE*

Abstract—Code repositioning is a well-known method of reducing inefficient off-chip memory accesses by streamlining cache behavior. Embedded systems with predetermined applications can achieve further improvement with the addition of fast and energy efficient scratchpad memory (SPM) on chip and moving frequent accesses code and/or data from main memory to SPM. While many researchers have attempted to either streamline cache accesses or improve the effectiveness of SPM, few studies focus on exploring their joint synergy. This study proposes integer linear programming (ILP) models that include both code repositioning and SPM code selection to identify the optimal code layout and reduce energy consumption in embedded systems with a cache and SPM. This study also proposes a two-stage metaheuristic algorithm. Experimental results reveal that 1) allocating a dedicated portion of the on-chip SRAM to the SPM is not always better than using a cache-only configuration and 2) it is not trivial to select code objects for the SPM. As much as 55 percent additional energy can be saved by applying both code repositioning and SPM code selection techniques.

Index Terms—Code layout, embedded systems, energy consumption, scratchpad memory.

1 Introduction

Power consumption is an important design consideration for embedded systems. Of all a system's hardware components, the memory system occupies a large portion of the energy budget and on-chip area [1]. Embedded systems typically use reduced instruction set computer (RISC) processors which have higher memory requirements than complex instruction set computer (CISC) processors due to the low code density and load-store architecture of RISC processors. To alleviate this problem, RISC processors are often equipped with large register sets for processor-memory data transfers. However, the instruction fetch path does not benefit from this approach. Hence, many studies aim to improve the performance and energy consumption of instruction memory hierarchy for RISC processors.

Instruction caches (I-cache) fill the gap between processors and memory. Since caches are meant to be "transparent," compilers and linkers by default do not rearrange the order of procedures. As a result, procedures with temporal proximity may accidentally overlap in the cache and cause frequent replacement of cache blocks. This notorious phenomenon is called cache thrashing [2]. To limit expensive instruction cache misses, many studies [3], [4], [5], [6], [7] explore the benefits of code repositioning at various granularities, and suggest not leaving the code layout problem to chance. The underlying concept of this approach is to place code objects with temporal proximity in suitable memory addresses to minimize cache thrashings. Instead of focusing on the granularity of the repositioning unit, this study seeks the combined benefit of exploiting both cache and scratchpad memory (SPM) in the instruction memory hierarchy, creating an approach that is orthogonal to all prior methods.

Embedded systems know their running software in advance and usually have predetermined instruction memory accesses. Therefore, embedded processors such as ARM11 [8], CELL [9], and ColdFire [10] often include fast and energy efficient on-chip scratchpad memory [11]. Compilers and programmers can move frequent accesses code or data blocks to the SPM to further improve the performance and energy efficiency of embedded systems. Since the characteristics of a hardware-controlled cache often complement those of a software-managed SPM, it is interesting to study the synergy creating by incorporating both code repositioning and SPM code selection techniques. This paper first presents generic integer linear programming (ILP) models for identifying the optimal code layout of embedded systems with cache and SPM. This paper also proposes a metaheuristic solution based on Tabu search, and compares the energy consumption of instruction fetches by applying different code layout schemes.

This paper is organized into six sections: Section 2 describes related research on code layout problems. Section 3 describes three integer linear programming models, which present the optimal layout for systems with different memory configurations. Section 4 presents the proposed two-stage metaheuristic algorithm. Section 5 presents experimental settings and results. Finally, Section 6 offers the conclusion.

2 Related Works

Panda et al. [12] demonstrated the benefits of using both cache and SPM in embedded systems. This approach partitions application data variables to disjoint address spaces occupied by SPM and main memory, making it possible to reap the combined benefits of both methods. To

• *The authors are with the Department of Computer Science, National Chiao Tung University, 1001 University Road, Hsinchu City 300, Taiwan. E-mail: {chenweih, sltsao}@cs.nctu.edu.tw.*

Manuscript received 2 Feb. 2011; revised 29 May 2011; accepted 19 June 2011; published online 8 July 2011.
Recommended for acceptance by E. Macii.
For information on obtaining reprints of this article, please send E-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-02-0079.
Digital Object Identifier no. 10.1109/TC.2011.122.

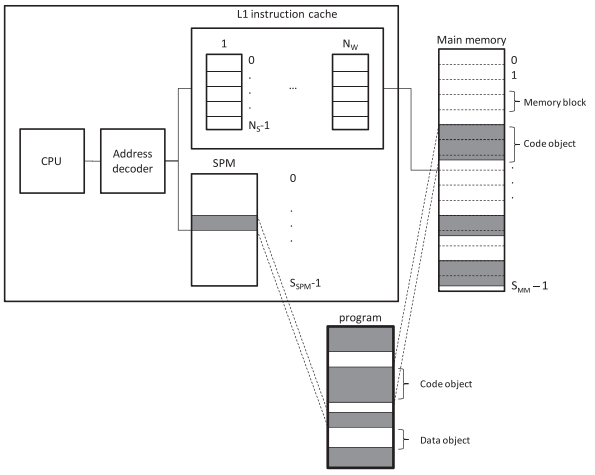


Fig. 1. Two-level instruction memory hierarchy.

improve the performance of dynamic binary translation in resource-constrained embedded systems, Baiocchi and Childers [13] proposed several SPM-aware policies for storing and replacing the translated code objects in the instruction cache. Although they mentioned the potential benefits of using SPM to ease the instruction cache conflicts, they did not investigate the interplay of SPM and cache. Ishitobi et al. [14] revealed the benefits of improving cache behavior by properly rearranging the order of program objects (code and data) in the memory space into cacheable, noncacheable, and scratchpad regions. However, their approach places program objects in compacted contiguous positions. This is an unnecessary constraint because the cache mapping rule is governed solely by memory positions, which determine whether two program objects compete for the same cache block. Thus, the generalized approach is to allow gaps in between the program objects in the main memory which can further ease cache conflicts. Verma et al. [15] assigned beneficial code objects to the SPM based on information extracted from instruction trace of a program. However, this method mandates unselected code objects to remain in the positions dictated by compilers and linkers. Hence, there are two major differences between previous studies and the proposed method: 1) this study investigates the combined benefits of employing both code repositioning and SPM code selection simultaneously; and 2) this study gives unselected code objects the freedom of noncontiguous placement.

3 Methodology

3.1 Terminology

This study adopts the Harvard architecture depicted in Fig. 1, assuming scratchpad memory locates on the same level as the set-associative level-1 (L1) instruction cache. While a program consists of a number of data and code objects, this study only considers the layout of code objects. A code object is allocated to either SPM or main memory depending on the code layout scheme. If a code object is allocated to the main memory, it occupies a number of contiguous memory blocks. Assume that the size of a memory block equals the size of a cache block. The proposed design always aligns the starting address of a code object with a memory block to develop ILP models. This alignment is not necessary for allocating code objects in the SPM. These assumptions are the same as adopted in [14], [15], and [16]. This study also develops ILP models based on the instruction trace, which is a trace log of the instructions that the processor has executed.

TABLE 1
Memory Configuration Parameters

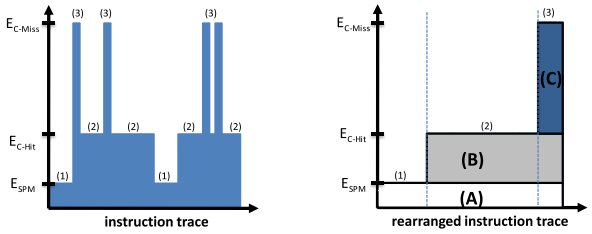
Symbol	Description
N_W	Associativity of the L1 I-cache
N_S	Number of sets in the L1 I-cache
S_{MM}	Size of the main memory in the number of memory blocks
S_{SPM}	Size of SPM in bytes

3.2 Objective Function

The goal of this study is to minimize the energy consumption for instruction fetches. As with parameters in Table 1, there are three possible cases for each instruction fetch: 1) code objects are allocated to the SPM, thus always appear in the SPM, 2) code objects are allocated to the main memory, and can be found in the cache, i.e., represent a cache hit, or 3) code objects are allocated to the main memory, but cannot be found in the cache, i.e., represent a cache miss. When the processor fetches the first instruction in a memory block which does not appear in the cache, the processor must spend E_{C-Miss} energy to bring the whole memory block to the cache. If the processor continues to execute an instruction in the same memory block already in the cache, the instructions are hits and the processor only spends E_{C-Hit} energy fetching the instruction. Table 2 lists the energy consumptions for the three possible cases. Fig. 2a illustrates that the energy consumed by CPU instruction fetches in chronological order. Therefore, the figure shows the energy per CPU instruction fetch versus instruction trace, instead of the conventional power versus time figure for a CPU. In other words, the total energy consumption is computed by summing the energy for all instruction fetches, rather than integrating of the power over time. Hence, the area under the energy envelope is the total instruction fetch energy to be minimized. For the sake of building mathematical models, rearrange the energy consumption for an instruction trace and organize the instruction accesses into

TABLE 2
Energy Consumption Parameters for Instruction Fetches

Symbol	Description
E_{SPM}	Energy consumption per instruction fetch from the SPM
E_{C-Hit}	Energy consumption per instruction fetch from the cache
E_{C-Miss}	Energy consumption per instruction cache miss



(a) Energy consumption for instruction fetches in an instruction trace.

(b) Energy consumption for the same instruction fetches in (a) which are rearranged based on their access categories.

Fig. 2. Energy consumption for instruction fetches.

the three aforementioned categories, indicated by vertical dotted lines, as Fig. 2b shows. The total energy consumption for an instruction trace is equivalent to the sum of the three horizontal strips labeled (A), (B), and (C). Since the area of strip (A), i.e., an SPM hit, is the sum of the base energy that each instruction must pay regardless of the code layout schemes, we can safely ignore this layout-invariant portion in the optimization process and add it back when calculating the total energy consumption. Hence, this study optimizes the instruction code layout to minimize the summation of the two horizontal strip areas (B) and (C).

Strip area (B) is easy to calculate because it represents the base energy that each instruction must pay if the instruction is fetched through cache-main memory path. It is only necessary to determine how many instructions in the code objects are allocated to the main memory, and then multiply this number by the height, i.e., $E_{C-Hit} - E_{SPM}$. To calculate the strip area (C), it is necessary to count the number of cache misses. Cache misses only occur when we fetch an instruction in a memory block that is not yet in the cache. Once the memory block has been brought into the cache, all subsequent accesses to instructions in the same memory block appear in the cache until replaced by another memory block. Therefore, identifying the memory block that each instruction belongs to makes it possible to treat the instruction trace as a sequence of memory block visits and preserve only the transitions. This extracted memory-block sequence is called the memory-block trace. With the memory-block trace, it is possible to examine the memory-block transitions in the instruction trace and determine the number of cache misses corresponding to code layout. Tables 3 and 4 describe the parameters and variables used in the ILP models.

Objective. Minimize the summation of strip areas (B) and (C) in Fig. 2b, where strip area (B) is the base energy consumption for fetching instructions of the code objects that are allocated to the main memory. The energy consumption of strip area (B) is

$$(E_{C-Hit} - E_{SPM}) \sum_{J=1}^{V} x_j I_j. \quad (O1)$$

Since strip area (C) represents the energy consumption due to cache misses, calculate this area by multiplying the number of cache misses and the strip height: $E_{C-Miss} - E_{C-Hit} - E_{SPM}$. Use the memory-block trace when calculating the cache misses. Cache misses occur either because the instruction is first seen by the processor (a cold miss) or it was replaced since last brought in to the cache (a conflict miss). Total misses include the cache misses under both conditions.

TABLE 3
Application- and Profile-Related Parameters

Symbol	Description
V	Number of code objects in the program
N_j	Size of code object j in the number of instructions
B_j	Number of memory blocks that code object j occupies
M_j^k	The k^{th} memory block of code object j
I_j	Number of the instructions that belong to code object j in the instruction trace
O_j^k	Number of occurrences of memory block M_j^k in the memory-block trace
A_j^k	Binary variable: 1, if memory block M_j^k appears in the memory-block trace
$\mathbb{D}_j^{ki}$	Set of distinct memory blocks appears in between $(i-1)^{th}$ and i^{th} occurrence of memory block M_j^k in the memory-block trace

1. *Cold misses.* If the code objects are allocated to the SPM, there is no cache miss. Otherwise, the first instruction access to every memory block of the code objects that are allocated to the main memory results in a cold cache miss. Assume code object j occupies B_j memory blocks. Denote the memory blocks of code object j as $M_j^1, \ldots, M_j^k, \ldots, M_j^{B_j}$. Use parameter A_j^k to indicate whether the memory block M_j^k appears in the memory-block trace or not. The number of cold misses is then

$$\sum_{j=1}^{V} x_j \sum_{k=1}^{B_j} A_j^k. \quad (O2a)$$

TABLE 4
Decision Variables

Symbol	Description
x_j	Binary variable: 1, if code object j is allocated to the main memory; 0, if it is allocated to the SPM
t_j	Address of starting memory block of code object j
y_{jp}	Binary variable: 1, if both code object j and p are allocated to the main memory
b_{jp}^{kq}	Binary variable: 1, if memory block M_j^k and memory block M_p^q are allocated to the addresses congruent modulo N_S
c_{jp}^{kq}	Natural number: Distance between M_j^k and M_p^q in integral unit of N_S
cf_{jp}^{kq}	Binary variable: 1, if M_j^k and M_p^q are mapped to same cache set, i.e. $cf_{jp}^{kq} = b_{jp}^{kq} \wedge y_{jp}$
cm_j^{ki}	Binary variable: 1, if the number of memory blocks in $\mathbb{D}_j^{ki}$ which are mapped to the same cache set as the memory block M_j^k exceeds the cache associativity, N_W

2. *Conflict misses.* Examine every memory block of all code objects that are allocated to the memory from their second access onward, and sum up all conflict misses for each memory block. In the memory-block trace, memory block M_j^k appears O_j^k times. $\mathbf{D}_j^{ki}$ denotes the set of distinct memory blocks appearing in between the $(i-1)$th and ith occurrence of memory block M_j^k in the memory-block trace. If the number of memory blocks in $\mathbf{D}_j^{ki}$ that are mapped to the same cache set as memory block M_j^k exceeds the cache associativity, N_W, memory block M_j^k must be evicted by some memory block between the $(i-1)$th and ith occurrence of M_j^k. In this case, use cm_j^{ki} to indicate a conflict miss occurred. Then, the total number of conflict misses can be calculated as

$$\sum_{j=1}^{V}\sum_{k=1}^{B_j}\sum_{i=2}^{O_j^k} cm_j^{ki}. \tag{O2b}$$

The following sections formulate ILP models according to whether code repositioning and SPM code selection techniques are applied.

3.3 Optimal Code Repositioning and SPM Code Selection

This section presents a generic ILP model incorporating both code repositioning and SPM code selection techniques. This model includes the following constraints:

1. *Early start.* It is necessary to ensure that every code object can fit in the main memory when assigning the address of starting memory block, defined as t_j, to each code object $j \in \{1, 2, \ldots, V\}$ containing N_j instructions and occupying B_j memory blocks.

$$0 \le t_j \le (S_{MM} - B_j)x_j,\ \forall j \in \{1, 2, \ldots, V\}. \tag{1}$$

The code objects selected into the SPM are governed by the SPM capacity constraint, and does not affect cache behavior. Therefore, these objects are assigned an arbitrary address of 0 for ease of modeling.

2. *SPM capacity.* The total size of all code objects allocated to the SPM must not exceed the SPM capacity:

$$\sum_{j=1}^{V} N_j(1 - x_j) \le S_{SPM},\ \forall j \in \{1, 2, \ldots, V\}. \tag{2}$$

3. *Nonoverlapping in the main memory.* For each pair of code objects $(j, p \in \{1, 2, \ldots, V\}, j < p)$ allocated to the main memory, the starting memory blocks must be separated sufficiently from each other to avoid overlapping. Use the binary variable y_{jp} to denote whether both code objects j and p are allocated to the main memory. This condition will also be used later when determining spatial conflicts

$$y_{jp} = x_j \wedge x_p \cong \min(x_j, x_p), \tag{3}$$

where y_{jp} can be linearized as

$$\begin{gathered} y_{jp} \le x_j \\ y_{jp} \le x_p \\ x_j + x_p - y_{jp} \le 1. \end{gathered}$$

Then, use z_{jp} to indicate if code object j is in front of code object p in the address space:

$$\begin{aligned} t_j - t_p + z_{jp}S_{MM} &\ge B_p y_{jp} \\ t_p - t_j + (1 - z_{jp})S_{MM} &\ge B_j y_{jp}. \end{aligned} \tag{4}$$

4. *Spatial conflict.* Two memory blocks M_j^k and M_p^q are mapped to the same cache set if and only if 1) both code object j and code object p are allocated to the main memory, i.e., $y_{jp} = 1$, and 2) M_j^k and M_p^q residing in locations that are congruent modulo to the number of sets in the cache, i.e., N_S [17].

The distance between any two memory blocks M_j^k and M_p^q residing in address $(t_j + k - 1)$ and $(t_p + q - 1)$ can be represented as

$$0 \le [(t_j + k - 1) - (t_p + q - 1)] - c_{jp}^{kq}N_S \le N_S - 1, \tag{5}$$

where c_{jp}^{kq} is a natural number.

This pair of memory blocks is mapped to the same cache set if they are apart in exact multiples of N_S. Use the binary variable b_{jp}^{kq} to indicate if memory block M_j^k and memory block M_p^q are allocated to the addresses congruent modulo in N_S. This relationship can be written in a linearized form as follows:

$$\begin{aligned} &(t_j + k - 1) - (t_p + q - 1) - c_{jp}^{kq}N_S \ge 1 - b_{jp}^{kq} \\ &(t_j + k - 1) - (t_p + q - 1) - c_{jp}^{kq}N_S \le (1 - b_{jp}^{kq})(N_S - 1). \end{aligned} \tag{6}$$

Use a variable

$$cf_{jp}^{kq} = y_{jp} \wedge b_{jp}^{kq} \cong \min\left(y_{jp}, b_{jp}^{kq}\right) \tag{7a}$$

to represent whether memory-block pairs M_j^k and M_p^q are mapped to the same cache set. cf_{jp}^{kq} can be linearized like y_{jp} in (2)

For memory blocks in the same code object, the intraobject cache set mapping conflict (i.e., $j = p$) is simply

$$b_{jj}^{kq} = \begin{cases} x_j, & \text{if } k \equiv q (\bmod N_S), \\ 0, & \text{otherwise.} \end{cases} \tag{7b}$$

5. *Temporal conflict miss.* Finally, determine whether a memory block M_j^k still resides in the cache set after the processor accesses the memory blocks between the $(i-1)$th and ith occurrence of memory block M_j^k. Use the binary variable cm_j^{ki}, summed in the objective function, to indicate whether the ith ($i \in \{2, 3, \ldots, o_j^k\}$) access to memory block M_j^k ($k \in \{1, 2, \ldots, B_j\}$) results in a cache conflict miss.

$$cm_j^{ki} = \begin{cases} 1, & \text{if } \sum_{M_p^q \in \mathbf{D}_j^{ki}} cf_{jp}^{kq} \ge N_W, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

This relationship can be linearized as

$$\begin{aligned} \sum_{M_p^q \in \mathbf{D}_j^{ki}} cf_{jp}^{kq} + \left(1 - cm_j^{ki}\right)U &\ge N_W \\ \sum_{M_p^q \in \mathbf{D}_j^{ki}} cf_{jp}^{kq} - cm_j^{ki}U + 1 &\le N_W, \end{aligned}$$

where U is a large number.

3.4 Optimal Code Repositioning for Cache-Only Configurations

An ILP model considering code layout for cache-only configurations can easily be obtained from the generic ILP model mentioned in the previous section. To do this,

simply remove the constraints (2 and 3) and parameters related to the decisions of the SPM code selection and slightly modify the model as described below:

1. Early Start, (1) reduces to

$$0 \le \mathrm{t_j} \le (\mathrm{S_{MM}} - \mathrm{B_j}),\ \forall \mathrm{j} \in \{1, 2, \ldots, \mathrm{V}\}. \tag{1'}$$

3. Nonoverlapping, by setting $\mathrm{y_{jp}} = 1$, (4) reduces to

$$\begin{aligned} &\mathrm{t_j} - \mathrm{t_p} + \mathrm{z_{jp}} \mathrm{S_{MM}} \ge \mathrm{B_p} \\ &\mathrm{t_p} - \mathrm{t_j} + (1 - \mathrm{z_{jp}}) \mathrm{S_{MM}} \ge \mathrm{B_j}. \end{aligned} \tag{4'}$$

4. Cache Set Mapping Conflict, (7a) reduces to

$$\mathrm{cf_{jp}^{kq}} = \mathrm{b_{jp}^{kq}}. \tag{7a'}$$

(7b) reduces to

$$\mathrm{b_{jj}^{kq}} = \begin{cases} 1, & \text{if } \mathrm{k} \equiv \mathrm{q} (\mathrm{mod}\ N_S), \\ 0, & \text{otherwise.} \end{cases} \tag{7b'}$$

With constraints (5, 6, and 8) intact, this leads to the ILP model of the code layout problem for cache-only systems.

3.5 Optimal SPM Code Selection Based on the Given Code Positions

An ILP model considering SPM code selection based on the code positions generated by the compiler/linker can also be obtained by adjusting the generic ILP model mentioned above. It is only necessary to modify the first constraint:

1. Early Start, (1) reduces to

$$\mathrm{t_j} = \overline{\mathrm{t_j}} \mathrm{x_j}, \forall \mathrm{j} \in \{1, 2, \ldots, \mathrm{V}\}, \tag{1''}$$

where $\overline{\mathrm{t_j}}$ is the starting memory block of code object j ($\mathrm{j} \in \{1, 2, \ldots, \mathrm{V}\}$) determined by the compiler/linker. Nonoverlapping constraints are automatically satisfied since the code objects are either assigned to the SPM or left intact in the compiled/linked positions. Therefore, the constraints ensuring nonoverlapping between any pair of code objects can be ignored. This approach is fundamentally similar to what Verma et al. proposed in [15], and the experiments in this study use it to compare the extra benefit of having freedom of code reposition.

3.6 ILP Model Complexity

For the ILP optimization process, the computation time complexity and memory requirement indeed depend on both the number of decision variables and the number of constraints in the model. The resource usage also varies by parameters used, such as the search strategies (DFS or BFS), backtracking, and branching strategies (0 first or 1 first). Unfortunately, there is no simple closed-form analytical equation relating the aforementioned factors to the computation time or memory requirement. Therefore, we have analyzed the asymptotic complexity of our ILP models in terms of the number of decision variables, the number of constraints, and code size to estimate the resource usage.

In our models, the number of binary variables is $\theta(\mathrm{V^2 B^2})$, and the number of constraints is $\theta(\mathrm{V^2 B^2} + \#\ \text{conflicting sets}\ \mathbf{D}_j^{\mathrm{ki}})$ $(\text{where } \mathrm{B} = \max\{\mathrm{B_j} | \text{code object j}\}$. In the experiments, to meet computation resource constraints, a number of numerous techniques can be used to obtain equivalent but smaller ILP models. We applied several techniques, mainly mathematical equivalent relationships and a hotspot phenomenon, as exhibited in the benchmarks. The number of binary variables reduced to $\theta(\mathrm{V^2 N_s^2})$, and the number of constraints to $\theta(\mathrm{V^2 N_s^2} + \#\ \text{of distinct conflicting sets})$.

4 Heurisitics

If we consider the SPM only, and ignore the cache, selecting suitable code objects into the SPM under the SPM capacity reduces to a 0-1 Knapsack problem. The code layout problem described above is thus NP-hard. A commercial ILP solver—CPLEX [19]—was used to solve the ILP models. However, due to the NP-hard nature of the problem, obtaining the optimal solutions within a reasonable time is often infeasible. In our experiments using synthesized benchmarks, the standard ILP searching process did not always yield the optimal solution after hours of computation using off-the-shelf PCs. Therefore, this study proposes a metaheuristic algorithm to obtain suboptimal solutions in a reasonable search time with the ability to avoid becoming trapped in a local optimum. The solution search includes two stages: "SPM code selection" and "main memory code reposition."

The input to the proposed algorithm is a list of code objects $\mathbf{N}$ sorted in descending order of access density. Access density is defined as the number of accesses to the code object divided by its code size. To avoid the bias toward frequently accessed code objects when selecting code objects into the SPM, the first "SPM code selection" stage uses the well-known partial enumeration skill [26]. Cut the sorted list $\mathbf{N}$ into two subsets: a "hotter" subset $\mathbf{H}$ and a "cooler" subset $\mathbf{C}$. Enumerate the hotter part of $\mathbf{N}$, i.e., $\mathbf{H}$, to produce the power set of $\mathbf{H}$. Besides the elements from the power set of $\mathbf{H}$ in the SPM, place code objects from $\mathbf{C}$ greedily into the slack space left in the SPM. This approach not only retains the engineering intuition of selecting code objects based on access density, but simultaneously gives less accessed code objects a chance to be selected. The second stage uses the Tabu search method to find good positions in the main memory for code objects that are not assigned to the SPM. Part of the reason for adopting a metaheuristic approach, such as Tabu search, is to avoid traps in the local optimum that occur in previous heuristic methods [14]. Experimental results show that these traps remain a serious problem, as Section 5 shows.

4.1 SPM Code Selection (Partial Enumeration)

Given a list of n code objects $\mathbf{N} = \{1, 2, \ldots, \mathrm{n}\}$ sorted in nonincreasing order of access density, cut the list $\mathbf{N}$ into two subsets: hotter subset $\mathbf{H} = \{1, 2, \ldots, \mathrm{h}\}$ and cooler subset $\mathbf{C} = \{\mathrm{h}+1, \mathrm{h}+2, \ldots, \mathrm{n}\}$, where

$$\mathrm{h} = \min\left\{ \mathrm{d} \,\middle|\, \sum_{\mathrm{j}=1}^{\mathrm{d}} \mathrm{N_j} \ge \mathrm{S_{SPM}} \right\}.$$

In other words, h is the point at which the SPM can no longer hold more code objects in $\mathbf{N}$ in a greedy fashion. For each element F from the power set of $\mathbf{H}$ that fits in the SPM (i.e., $\sum_{\mathrm{j} \in \mathrm{F}} \mathrm{N_j} \le \mathrm{S_{SPM}}$). Select code objects from $\mathbf{C}$ for the slack space of the SPM greedily. Thus, form the selected set $\mathbf{S}$ of code objects placed into the SPM.

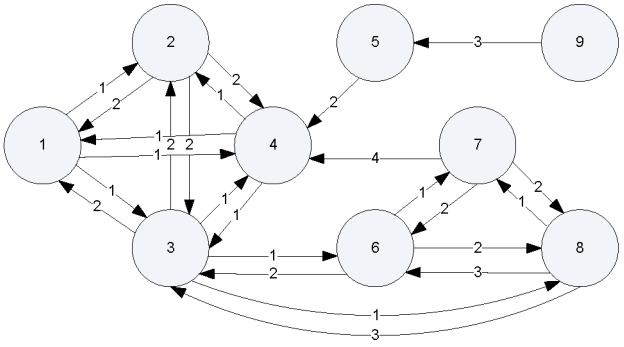


Fig. 3. Temporal conflict graph.

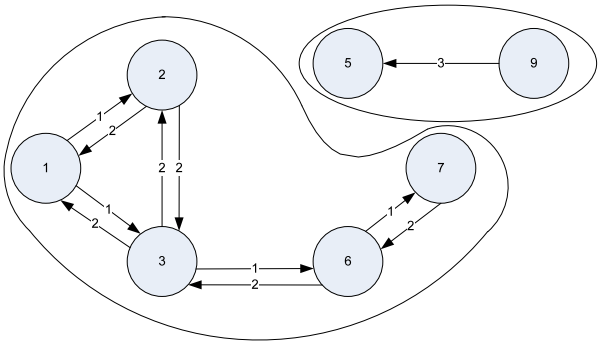


Fig. 4. Residual TCG.

4.2 Main Memory Code Reposition Based on Tabu Search

The positions of the remaining code objects $\mathbf{M}(\mathbf{M} = \mathbf{N}\backslash\mathbf{S})$ in the main memory can be found using the Tabu search method. For a particular set of code objects $\mathbf{M}$ remaining in the main memory, it is necessary to calculate three things:

1. Number of accesses in $\mathbf{M}$, which can be calculated by

$$\sum_{\mathrm{j}\in\mathbf{M}} I_i.$$

2. Number of cold misses in $\mathbf{M}$, which can be calculated by

$$\sum_{\mathrm{j}\in\mathbf{M}}\sum_{k=1}^{B_j} \mathrm{A}_{\mathrm{j}}^{\mathrm{k}}.$$

3. Number of conflict misses in $\mathbf{M}$.

To simplify the discussion, suppose there are $|\mathbf{M}| = m$ code objects in $\mathbf{M}$ and rename them as

$$\mathbf{M} = \{1, 2, \ldots, \mathrm{m}\}.$$

With the number of cache sets $\mathrm{N_S}$, the memory address can be viewed as $\mathrm{N_S}$ equivalent sets. Thus, each layout solution can be stated as $\mathrm{t} = (\mathrm{t_1}, \mathrm{t_2}, \ldots \mathrm{t_m})$ where $\mathrm{t_j}$ is the starting memory block address for code object $\mathrm{j} \in \mathbf{M}$, and $\mathrm{t_j} \in \{0, 1, \ldots, \mathrm{N_s} - 1\}$. For each layout solution t, the number of conflict misses is uniquely determined by the memory-block trace executed. Thus, this is the metric used in comparing solutions found by Tabu search.

The Tabu search heuristics include the following main components:

1. *Initial Solution, t^i, initial iteration $i = 0$.* For simplicity, use the zero vector $t^0 = (0, 0, \ldots, 0)$ to start with the search since for the metaheuristic such as Tabu search, as this point is relatively insensitive to the initial solution [24], [25], and any constructive heuristic such as [23] can be applied orthogonally to this method.

2. *Neighborhood, $\mathrm{N}(t^i)$.* Consider solution $\mathrm{t} \in \mathrm{N}(t^i)$ if it is within radius r of current center, t^i. In other words, $\mathrm{t} = (t^i + \mathrm{offset})\ mod\ N_S$ where $\mathrm{offset} = (o_1, o_2, \ldots, o_m)$ and $o_j \in \{0, 1, \ldots r\}$.

Since there are exponential $o(r^m)$ solutions in $\mathrm{N}(t^i)$, it is necessary to confine the search by intelligently sampling the neighborhood. Hence, this study introduces the following graph structure to capture the temporal conflict relationship between code objects in the memory-block trace.

The Temporal Conflict Graph, TCG = $(\mathbf{N}, \mathrm{E})$ is a directed weighted graph. Edge $\mathrm{e(p, j)} \in \mathrm{E}$, if $MO_p^q \in \mathbf{D}_j^{ki}$ for some q, k, and i. In other words, there is a potential conflict from code object p to code object j if any memory block belonging to code object p appears between two consecutive appearances of any memory blocks of code object j in the memory-block trace. The edge weight $\mathrm{w(p, j)}$ increments for each unique occurrence of such $MO_p^q \in \mathbf{D}_j^{ki}$ representing the possible conflicting severity of code object p to code object j. Because code repositioning cannot remove intraobject conflicts, it is not necessary to consider self-loops in TCG, i.e., there is no edge of type $\mathrm{e(j, j)}$.

For illustration, suppose the following memory-block trace with $\mathrm{n} = 9$:

11,21,22,21,31,32,43,11,22,25,31,43,51,52,43,71,72,73,74,
43,33,61,62,81,82,83,33,62,75,76,62,81,75,53,91,92,93,53.

Here, jk represents memory block k of code object j. We can construct TCG with $\mathbf{N} = \{1, 2, \ldots, 9\}$ as shown in Fig. 3.

Code objects selected for the SPM do not interfere with cache behavior. Thus, for each set of code object $\mathbf{S}$ selected for the SPM, pull away the constituent elements from TCG and all the adjacent edges forming the residual TCG, $\mathrm{rTCG} = (\mathbf{M}, E_{rTCG})$. Assume that $\mathbf{S} = \{4, 8\}$ is placed in the SPM, leaving its complement $\mathbf{M} = \{1, 2, 3, 5, 6, 7, 9\}$ in the main memory with corresponding rTCG as shown in Fig. 4.

This leads to two connected components {1,2,3,6,7} and {5,9}. Code objects can be positioned independently in the connected component since there is no intercomponent conflict. Without loss of generality, assume rTCG with one component and define the conflict impact factor of each code object $\mathrm{p} \in \mathbf{M}$ as

$$\sum_{(p,j)\in\mathrm{Ertcg}} c(p, j).$$

This term indicates how much trouble code object p may cause for its neighbors. A higher value means that this object should be settled first. Thus, the code objects in $\mathbf{M}$ are sorted by their conflict impact factor. When sampling the neighborhood $\mathrm{N}(t^i)$, similar to the partial enumeration spirit, the offset corresponding code objects with high impact factors are allowed to span all the possible values. For code objects with smaller impact factors, randomly choose an offset value from $\{0, 1, \ldots, \mathrm{r}\}$. As a result, a

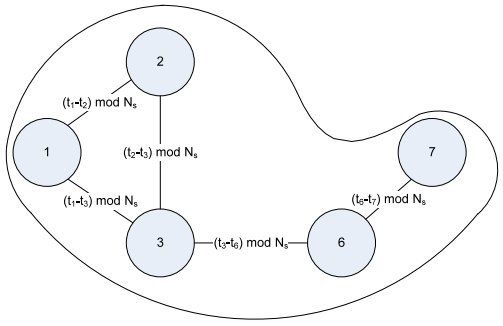


Fig. 5. Residual Tabu graph.

constant number of solutions from each neighborhood are sampled to find the best solution to be used as the center of the next neighborhood.

3. *Tabu Structure (Short term memory).* To avoid cycling, the Tabu search method remembers a small number of visited solutions. Therefore, it is necessary to devise a structure to represent previously visited solutions. This Tabu structure can be derived from rTCG by annotating each node j in rTCG with its starting address t_j, and annotating each edge (p, j) with $(t_p - t_j) mod\ N_s$ for $p < j$. This derived graph is called a residual Tabu graph, rTG as shown in Fig. 5. The number of conflict misses is the same for solutions with the same rTG structure. Hence, using rTG helps avoid cycling and makes it possible to skip nonprofitable solutions.

4. *Aspiration rule.* There is no need to devise an aspiration rule because there is no mis-kill using the proposed Tabu structure.

5. *Intensification rule.* Maintain a list of incumbent solutions and go back to explore their neighborhood exhaustively with a larger radius value.

6. *Diversification rule.* To better explore the solution space, it is necessary to escape from deeply searched areas. Again, use rTCG to aid the decision to pick a solution in a remote area in the solution space. For code object p in rTCG with the highest impact factor, add an offset value $\lceil \frac{N_S}{d} \rceil$ to its starting address t_p used in the last round. Randomly assign $\{0, 1, \ldots, N_S - 1\}$ for adjacent nodes to node p. Afterward, strip away node p with all its adjacent nodes from rTCG and continue until rTCG is empty. Note that d is the number of rounds devised to explore the solution space.

7. *Stopping rule.* Use the maximum computation time allowed or equivalently the maximum number of iterations as the stopping rule.

5 Experimental Results

This section evaluates the effectiveness of code repositioning, SPM code selection, and their combined advantages. The first section describes the simulation environment, while the second compares the energy reduction and memory access distribution of benchmark programs using the proposed ILP models, metaheuristics, and two existing methods [14], [15].

5.1 Experiment Setting

Fig. 6 illustrates the experimental flow. The first step is to identify the code objects. To reveal the real benefit of code layout, all the code (including initialization and libraries

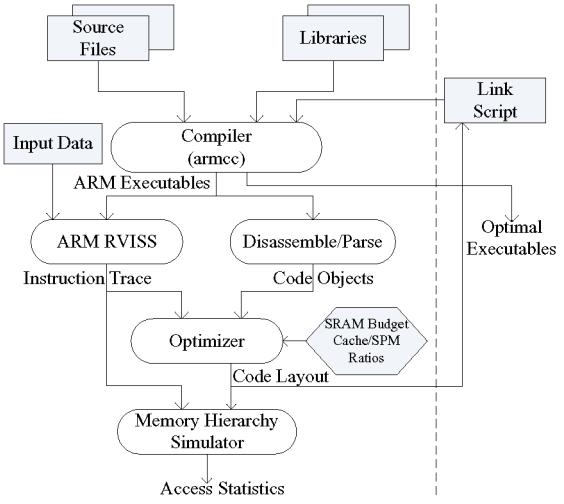


Fig. 6. Code layout optimization and experiment flow.

code) in the binary image was considered. For statically linked programs using precompiled libraries, high-level semantics are often inaccessible. Thus, this study parses the disassembled binary images and uses labels such as an unconditional jump or a return instruction to partition the program image into code objects. Program executable and input data were fed into ARM RealView Instruction Set Simulator (ARM RVISS) to collect instruction traces. Code object information and executed instruction trace were then fed into the pluggable optimizer modules. Five modules were implemented using a memory hierarchy simulator: 1) the proposed three ILP models coupled with CPLEX [19] optimization engine, 2) the proposed two-stage metaheuristic algorithm, and 3) a heuristic algorithm proposed in [14]. The results were then transformed into energy consumption values according to the energy weights.

The exact access energy of on-chip SRAM and off-chip SDRAM depends heavily on the detailed system configuration and manufacturing process, and inevitably lead to a trade-off. To demonstrate the benefits of code layout without delving into detailed design space exploration or competitive hardware architectures, this study uses simple relative energy weight of accessing SPM, cache, and SDRAM memory instead of using tools like CACTI [20]. Several studies indicate that the relative order of energies for per-word accessing to SPM, cache of the same size, and cache line fill energy from SDRAM memory is approximately 1:1.2:50 [15], [21].

Although the proposed methods are applicable to the general set-associative cache with various associativity, this study uses a direct-mapped cache as an example. Direct-mapped caches are attractive because of their good timing and energy performance, but may suffer severe cache conflicts. However, properly laying out the code should make it possible to alleviate this problem while retaining the desired cache behavior.

Fig. 6 also illustrates how to apply the proposed method. First, system designers can choose some common SRAM configurations, each representing a specific partition between the cache and SPM. The optimizer can assess these SRAM configurations, and select that which provides the minimal total instruction fetch energy. The optimizer

TABLE 5
Benchmarks

Name	# of Code Objects (Hot/All)	Exe. Time % (A)	Hot Code Size % (B)	Hotness (A)/(B)
Qsort†	(11/100)	91.9	38.53	2.385
Basicmath†	(11/101)	95.35	31.62	3.015
FFT‡	(14/147)	96.02	25.84	3.716
Dhrystone‡	(16/148)	96.82	15.28	6.336

generates the link script file that provides the layout information for the code objects. Thus, based on the link script file, the compiler can recompile the application, and the loader can preload the code objects destined in the SPM before the application starts.

5.2 Experimental Results

As mentioned in Section 4, the code layout problem is NP-hard. To compare the efficiency and effectiveness of layouts determined by ILP optimization models and heuristic results, it is necessary to speed up the ILP optimization process. Thanks to the "hotspot" phenomenon in most practical applications [18], it is possible to optimize the layout of a relatively small portion of hot codes. Thus, this section compares code layout schemes determined using five different methods:

1. Code repositioning for cache-only configurations.
2. SPM code selection based on the positions that compiler/linker generates.
3. Code repositioning and SPM code selection.
4. Contiguous layout heuristic proposed in [14].
5. The proposed two-stage metaheuristic.

Among these methods, the first three methods were solved using an ILP optimization process, whereas the last two were heuristic approaches.

The ILP optimization process uses the optimal solution obtained in Section 3.5 to serve as a starting feasible solution for improvement. Empirically, noninferior code layouts are obtained within one hour and may be proved optimal with more time. The proposed two-stage heuristic method can parallelize the search for each set of selected code objects, and only requires a few minutes to determine the solution using a PC. The experiments in this study included the

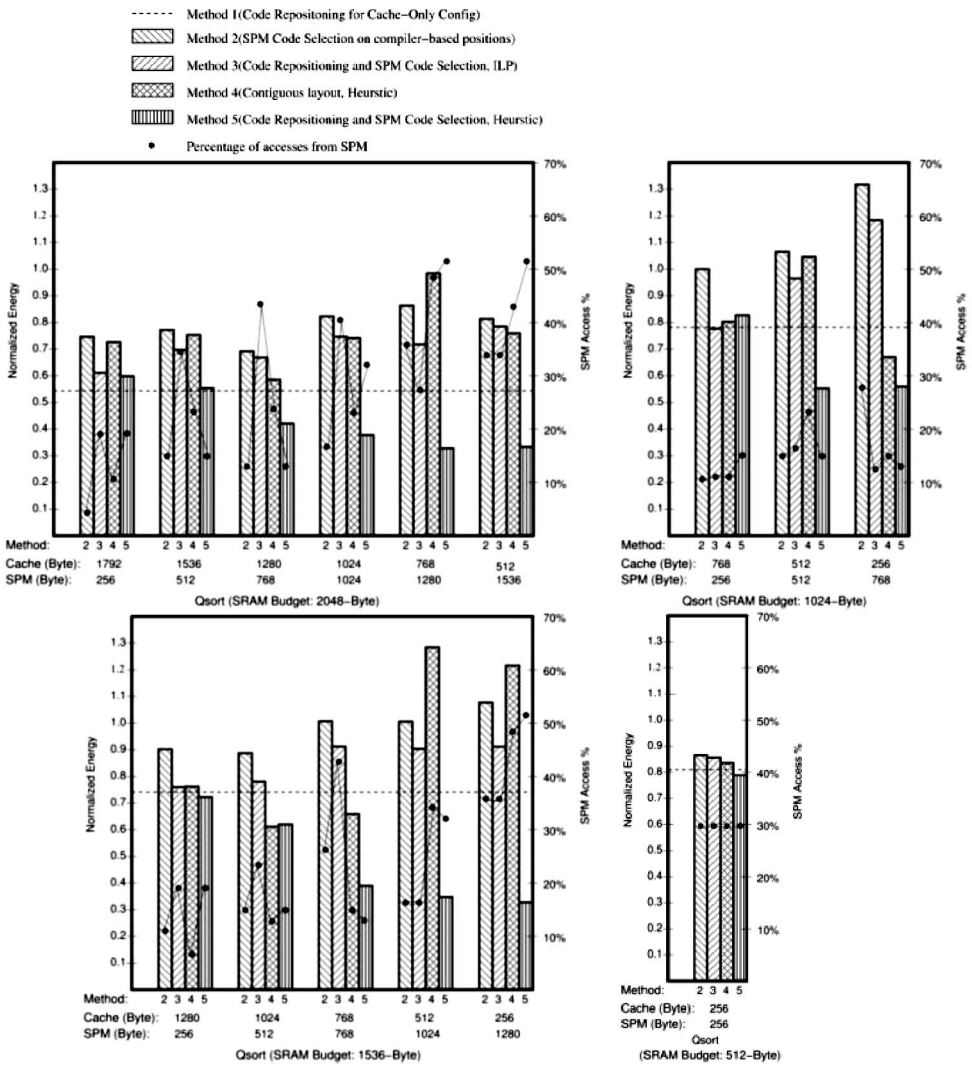


Fig. 7. Normalized energy consumption (Qsort).

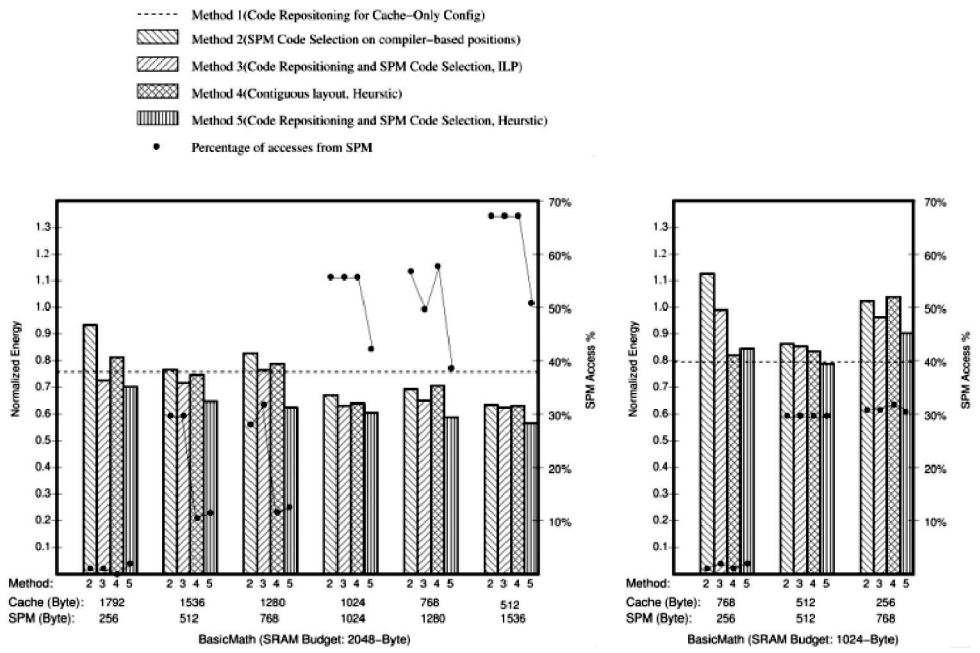


Fig. 8. Normalized energy consumption (BasicMath).

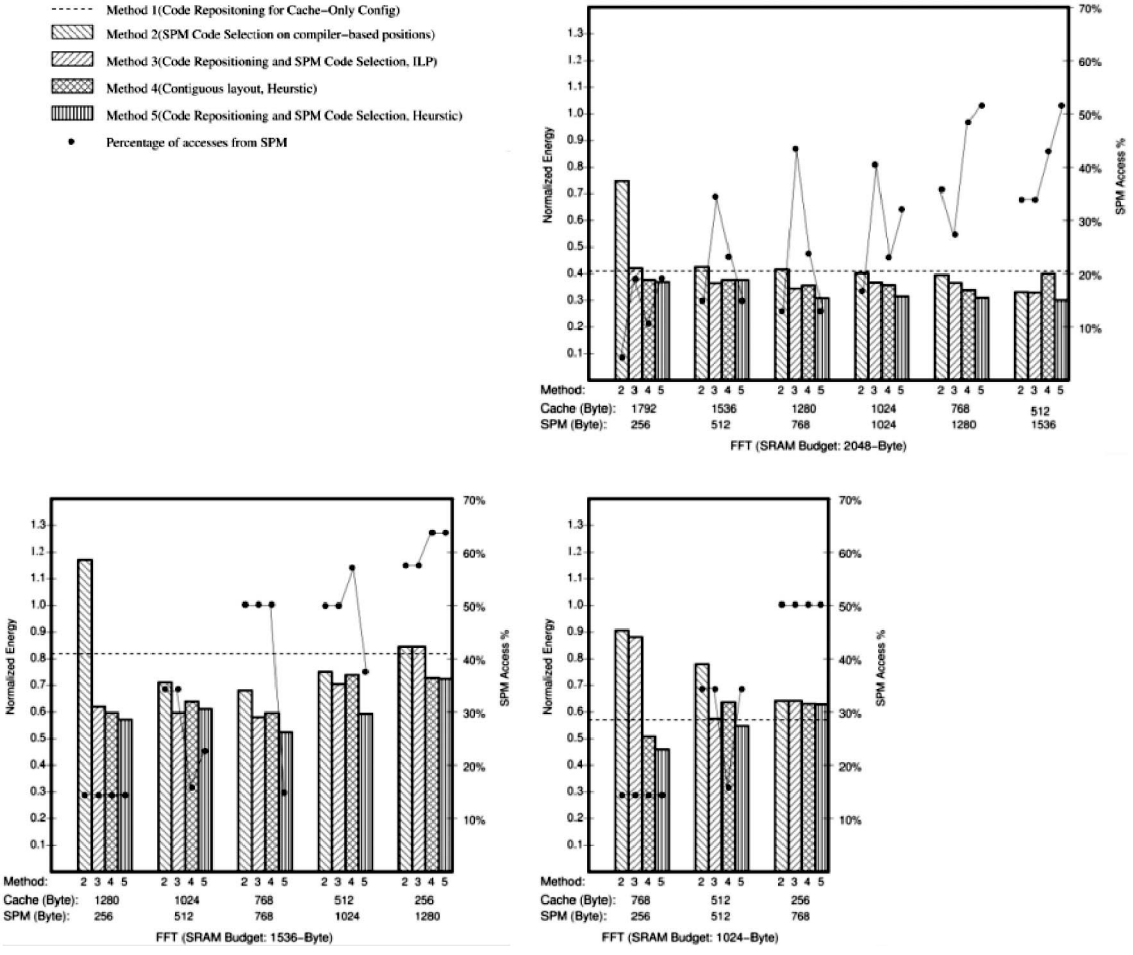


Fig. 9. Normalized energy consumption (FFT).

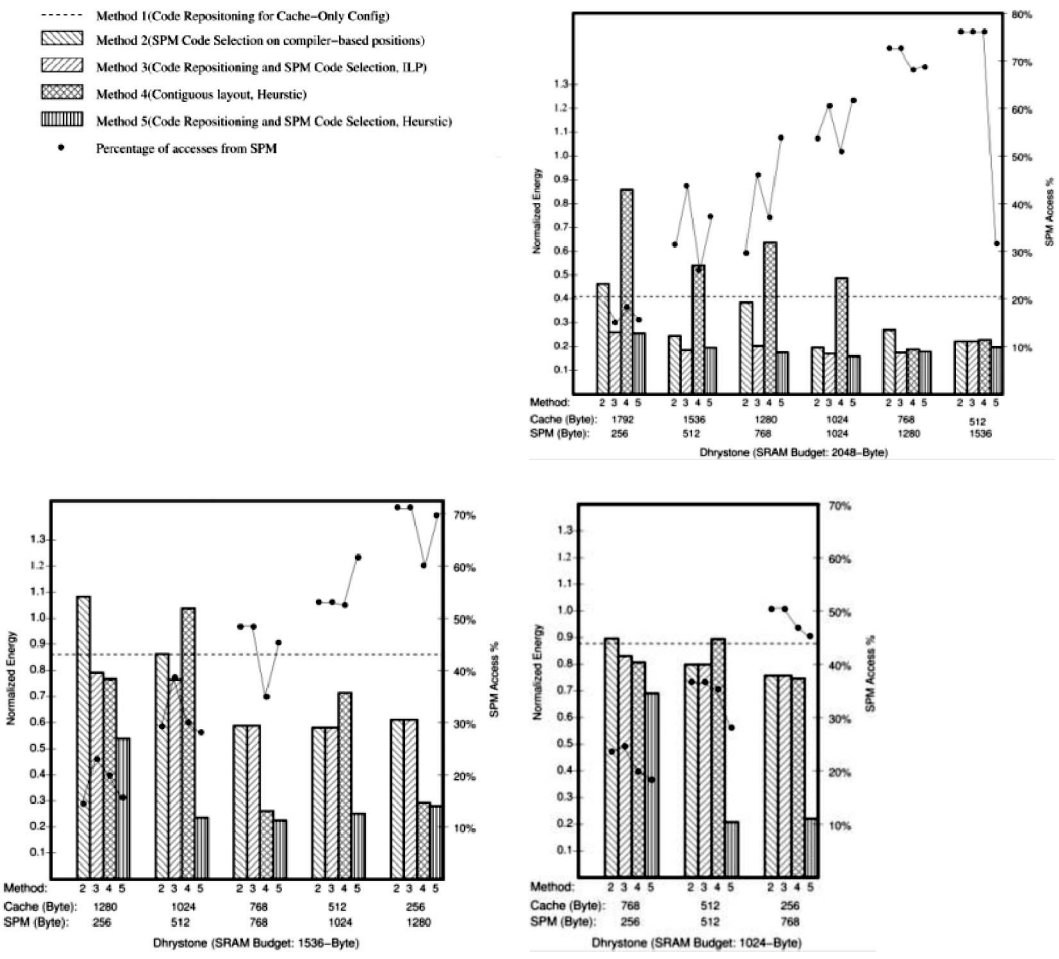


Fig. 10. Normalized energy consumption (Dhrystone).

heuristic proposed by Ishitobi et al. in [14] to test the potential benefit of a noncontiguous layout.

Table 5 lists benchmarks selected from MiBench† [22] and ARM RealView Development Suite‡ [8]. Code objects were considered hot if they were executed more than one percent of total executed instruction count. Table 5 shows that over 90 percent of total execution time falls in about 10 percent of all the code objects occupying 15-40 percent of the original code size.

The total on-chip SRAM budget was decided by setting the cache hit ratio from 90-95 percent for the original unoptimized trace of each benchmark. Moreover, various SRAM budget splits between cache and SPM were tested to assess the sensitivity of different methods toward energy consumption. Energy consumption levels were normalized to those of the corresponding original trace for each benchmark in which the entire on-chip SRAM was used as cache to reveal the effects of considering only the hot code objects. If the normalized energy is higher than 1, it indicates that more code objects shall be considered in the layout process.

Figs. 7, 8, 9, and 10 show the normalized energy consumptions with the percentage of accesses residing in the SPM by applying five different layout schemes under various splits of total on-chip SRAM budgets. To test the benefit of dedicating a portion of the SRAM budget to the SPM, the normalized energy consumption of

1. Method 1 corresponds to code repositioning in cache-only configuration served as the horizontal watershed (represented as a dashed line in each figure). If the normalized energy consumption of some SRAM budget split with applied method is higher than this watershed, it means that the method under this split is not beneficial. If all methods result in higher values, it indicates that splitting out SPM is not beneficial.
2. Method 2 corresponds to SPM code selection based on the positions that the compiler/linker generates.
3. Method 3 corresponds to code repositioning and SPM code selection.
4. Method 4 corresponds to contiguous layout heuristic proposed in [14].
5. Method 5 corresponds to the proposed two-stage metaheuristic.

The normalized energy for various cache-SPM configurations appears as bars, while dots represent the percentage of accesses residing in the SPM.

The experiments in this study yielded seven interesting observations:

1. Repositioning hot code objects can reduce instruction fetch energy by 11-60 percent for the tested benchmarks, based on 58 different SRAM budget splitting scenarios. The actual effect depends on individual

benchmark and compiler characteristics. However, code repositioning is generally beneficial to cache behavior, agreeing with numerous prior studies.

2. Incorporating SPM into the memory hierarchy is mostly, but not always, beneficial. Of the 58 tested splitting scenarios, only six showed that there is no way to reduce energy consumption comparing to that of applying code repositioning to the cache-only configuration. These six splitting scenarios share no obvious features. Benchmark CacheSize_SPMSize notation was used to indicate the benchmark, the cache, and SPM sizes. In two cases, the SRAM budget was large and the SPM split was comparatively small (Qsort 1792_256 and 1536_512). The SPM share was comparatively large in three cases (BasicMath 256_1280 and 256_768; FFT 256_768). In one case, the SRAM budget was small (BasicMath 256_256).
3. It is difficult to select code objects for the SPM. As Verma et al. [15] indicated in their cache-aware SPM code selection research, greedily selecting the most-executed code objects into the SPM may not be the best strategy. Selecting code for the SPM becomes even more complicated if code objects are allowed to be repositioned. A comparison of the results of method 2 (SPM Code Selection) and method 3 (Code Repositioning and SPM Code Selection) reveals the effects of code repositioning freedom. In five cases, selecting cooler code objects for the SPM is beneficial. There are 31 cases where same code objects were assigned to SPM indicating the improvement comes from code repositioning solely. In 22 cases, hotter code objects were able to be selected, leading to better SPM utilization given code repositioning freedom. This suggests that the design space of optimal split is not smooth.
4. The benefit of having a noncontiguous layout over contiguous restriction is significant. Comparing heuristic methods 4 and 5 shows that noncontiguous layouts achieve better performance in 49 of 58 scenarios. In theory, a contiguous layout is a special case of noncontiguous layout, so these results are not surprising. In the 49 cases, we outperformed at maximum 90 percent and 23 percent in average whereas for the nine loosing cases, we only loose by maximum of six percent and 1.6 percent in average. The reason why the nine noncontiguous layout cases performed worse than a contiguous layout is that the proposed heuristic chose hotter code objects for the SPM when cooler code objects lead to better energy consumption. Since the first stage of the proposed two-stage metaheuristic chooses code object fundamentally based on their access density, objects in **C** were picked greedily into the slack space which may block cooler objects from being selected for the SPM when they should have. Thus, by exploring the solution space partially, there is chance of missing better solutions. Thus, enlarging the enumeration set makes it possible to achieve better results, with a trade-off in computation cost.
5. Heuristic algorithms may be trapped in local optima. An obvious case of this problem appeared in BasicMath 1792_256. There is no access residing in the SPM indicating no code object was assigned to the SPM. However, it is clear that having any code objects with size smaller than the SPM capacity is beneficial both in terms of less access energy and reducing conflicts with other code objects. Thus, this study proposes a metaheuristic such as Tabu search to avoid these traps.
6. The result of swapping and compacting code sequence is sometimes worse than not reordering code object positions and leaves the code layout noncontiguous. A comparison of the results obtained using method 2 (SPM Code Selection using compiler/linker given position) and method 4 (Contiguous layout by swapping code objects) reveals 14 cases in which using compiler/linker given position is better than swapping code objects around by maximum of 39 percent and average of 15.4 percent. This suggests not overlooking the benefit of noncontiguous layout.
7. The ILP optimization process is time consuming and may halt progress indefinitely. The experiments in this study show that the ILP optimizing solution discovery duration increases annoyingly and often halts progress completely after running for a while. This is probably due to difficulty of finding a feasible solution in the branch-and-cut process given the enormous number of constraints. However, heuristics can explicitly enumerate the solution and only require enough computation time to evaluate the quality of the solution, which is a much easier approach. Targeting only on the few hot code objects using heuristic algorithms can successfully identify better quality solutions than the ILP optimization process.

Finally, Table 6 summarizes 1) the benefit of having a dedicated SPM over the cache-only configuration and 2) the additional energy saving of having the freedom to reposition code objects over using compiler/linker given positions. For FFT and Dhrystone, an SRAM budget of 512 bytes (marked as NA) is too small to have meaningful improvement. Thus, the SRAM budget started at 1,024 bytes. Tables 6a and 6b compare the net benefit obtained using methods 3 and 5 (italicized in parenthesis) with that of methods 1 and 2, respectively.

Table 6a demonstrates whether partitioning out SPM is favorable over using the entire SRAM as cache using code repositioning (method 1). For SRAM budgets with positive maximum value entries, having SPM is beneficial in most cases. However, for configurations with negative improvement range entries, such as 512 bytes, there is no benefit of partitioning SPM even if code repositioning and code selection are used. Table 6b demonstrates the importance of considering both code repositioning and SPM code selection at the same time, as opposed to considering only SPM code selection using compiler-given positions (method 2). The energy saving headroom available is usually more than 10 percent, which is quite significant, and can be as large as 55.1 percent for FFT with an SRAM budget of 1,536 bytes and 75.7 percent using method 5 for Qsort with an SRAM budget of 1,024 bytes.

The discussion above shows that selecting the best code object for the SPM is not easy, and giving code objects the freedom to rearrange themselves can significantly improve performance. Hence, the layout choice of repositioning and SPM code selection should be considered simultaneously instead of in separate stages.

TABLE 6
Normalized Energy Consumption (Percent)

(a) AVG & MAX SAVING OVER CODE RE-POSITIONING FOR CACHE-ONLY.

On-Chip SRAM	Qsort	Basicmath	FFT	Dhrystone
2048 Bytes	-16.1; **-6.7**	7.27; **13.3**	4.46; **8.2**	20.68; **23.8**
	*(10.76; **21.7**)*	*(13.6; **19.3**)*	*(8.2; **11.2**)*	(21.41; **24.6**)
1536 Bytes	-11.2; **-2**	-0.2; **8.3**	14.85; **23.7**	19.47; **28.2**
	*(25.92; **41.3**)*	*(5.91; **11.56**)*	*(21.27; **28.6**)*	(55.56; **63.7**)
1024 Bytes	-19.34; **0.5**	-14.1; **-6**	-12.85; **-0.5**	8.24; **12.1**
	*(13.50; **23.4**)*	*(-5.1; **0.7**)*	*(2.55; **11**)*	(50.48; **67.2**)
512 Bytes	-20.3; **-20.3**	-22; **-22**	NA	NA
	*(-18.9; **-18.9**)*	*(-11; **-11**)*		
Summary	-16.7; **0.5**	-7.25; **13.3**	2.15; **23.7**	16.13; **28.2**
	*(7.82; **41.3**)*	*(0.8525; **19.3**)*	*(10.67; **28.6**)*	(42.48; **67.2**)

Note: In each entry, the first row shows data obtained by ILP solver and second row shows data obtained by Heuristics. In each row, average data is shown first and maximal data is shown by bold face.

(b) AVG & MAX ENERGY SAVING HEADROOM OVER SPM CODE SELECTION.

On-Chip SRAM	Qsort	Basicmath	FFT	Dhrystone
2048 Bytes	8.08; **14.6**	-6.87; **20.7**	7.8; **32.7**	9.94; **20.2**
	*(34.93; **53.5**)*	*(13.2; **47.37**)*	*(11.6; **37.7**)*	*(10.17; **20.9**)*
1536 Bytes	12.2; **16.6**	4.33; **13.9**	16.22; **55.1**	7.84; **29.3**
	*(49.42; **65.8**)*	*(11.63; **22.9**)*	*(22.64; **59.8**)*	*(43.94; **62.7**)*
1024 Bytes	15.22; **22.2**	6.81; **13.5**	7.6; **20.4**	2.18; **6.6**
	*(48.66; **75.7**)*	*(15.9; **27.9**)*	*(23.2; **44.6**)*	*(44.42; **59.1**)*
512 Bytes	11.0; **11.0**	0.8; **0.8**	NA	NA
	*(12.6; **12.6**)*	*(11.8; **11.8**)*		
Summary	11.63; **16.1**	4.7; **12.23**	10.54; **36.06**	6.65; **18.7**
	*(36.4; **51.9**)*	*(13.13; **21.43**)*	*(19.15; **47.37**)*	*(32.84; **47.57**)*

From both the average and maximal data listed in Table 6a, we can see that Qsort performs the lowest, whereas Dhrystone performs the highest when we use the ILP solver. This characteristic can be explained by observing the "Hotness" column in Table 5 which equals to "Exe. Time" divided by "Hot Code Size." We sorted the benchmarks according to this ratio. Qsort has the least degree of hotspot behavior in the four tested benchmarks, having 91.9 percent of the execution time spread over 38.53 percent of the code. By contrast, Dhrystone consumed 96.82 percent of the execution time over a much smaller portion (15.28 percent) of the code, signifying that benchmarks exhibiting a higher hotspot or repeated pattern can be optimized more easily by the ILP solver.

This finding justifies the characteristic of the ILP solver that attempts to prune as much solution space as possible, thus often performing searches and backtracking in greedy fashion. Conversely, our heuristic method conducts the search in the solution space more thoroughly with a guided sampling; thus, it is not restricted by such a factor. Therefore, solutions found by our heuristic method depend more on the granularity of code objects, compiler options, and memory configuration, which are the design space exploration issues discussed below.

From the experimental results, we observed that the improvement depends on the characteristics of the target application, SRAM partition between the cache and SPM, and granularity of code objects. However, due to limited knowledge at present, we did not delve further into this design space exploration issue and did not optimize SRAM partitions and code object sizes for a specific target application. Design space exploration may very well be our next goal. In this study, we focused on optimizing the code layout of embedded systems, and illustrated the possible improvement under a given cache and SPM configuration.

6 CONCLUSIONS

This paper presents an optimal code layout for embedded systems with a cache and SPM. The proposed ILP models address layout problems under different memory configurations. Results demonstrate that layout determined using code repositioning and SPM code selection simultaneously offers better results than that using only the SPM code selection technique. The benefit of energy saving is manifested by either capturing the conflicting sets that occur relatively frequently in the SPM or rearranging the conflicting objects into less contended cache sets. The proposed methods minimize energy consumption by reducing expensive cache misses, which also helps improve performance.

ACKNOWLEDGMENTS

The authors would like to thank MediaTek Inc. and National Science Council of the Republic of China for financially supporting this research under Contract Nos. 100-2219-E-009-022-, 100-2220-E-009-038-, and 99-2220-E-009-045-.

REFERENCES

[1] D. Keitel-Schulz and N. Wehn, "Embedded DRAM Development: Technology, Physical Design, and Application Issues," *IEEE Design and Test of Computers*, vol. 18, no. 3, pp. 7-15, May 2001.
[2] J. Handy, *The Cache Memory Book*, second ed., pp. 22-23. Academic Press Professional, Inc., 1998.
[3] A.D. Samples and P.N. Hilfinger, "Code Reorganization for Instruction Caches," Technical Report UCB/CSD-88-447, EECS Dept., Univ. of California, Berkeley, Oct. 1988.
[4] W.W. Hwu and P.P. Chang, "Achieving High Instruction Cache Performance with an Optimizing Compiler," *Proc. 16th Ann. Int'l Symp. Computer Architecture (ISCA '89)*, pp. 242-251, 1989.
[5] S. McFarling, "Program Optimization for Instruction Caches," *SIGARCH Computer Architecture News*, vol. 17, no. 2, pp. 183-191, 1989.
[6] W.Y. Chen, P.P. Chang, T.M. Conte, and W.W. Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design," *IEEE Trans. Computers*, vol. 42, no. 9, pp. 1045-1057, Sept. 1993.
[7] N. Gloy, M. Smith, and C. Young, "Performance Issues in Correlated Branch Prediction Schemes," *Proc. 28th Ann. Int'l Symp. Microarchitecture (Micro '95)*, pp. 3-14, Nov./Dec. 1995.
[8] ARM, http://www.arm.com/products/processors/classic/arm11/index.php, 2011.
[9] IBM, http://www.research.ibm.com/cell/, 2011.
[10] Freescale, http://www.freescale.com/webapp/sps/site/homepage.jsp?code=PC68KCF, 2011.
[11] 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," *Proc. 10th Int'l Symp. Hardware/Software Codesign (CODES '02)*, pp. 73-78, 2002.
[12] P.R. Panda, N.D. Dutt, and A. Nicolau, "On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems," *ACM Trans. Design Automation of Electronic Systems*, vol. 5, no. 3, pp. 682-704, 2000.
[13] J.A. Baiocchi and B.R. Childers, "Heterogeneous Code Cache: Using Scratchpad and Main Memory in Dynamic Binary Translators," *Proc. 46th Ann. Design Automation Conf. (DAC '09)*, pp. 744-749, 2009.

[14] Y. Ishitobi, T. Ishihara, and H. Yasuura, "Code and Data Placement for Embedded Processors with Scratchpad and Cache Memories," *J. Signal Processing Systems,* vol. 60, pp. 221-224, Aug. 2010.
[15] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-Aware Scratchpad Allocation Algorithms for Energy-Constrained Embedded Systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 25, no. 10, pp. 2035-2051, Oct. 2006.
[16] H. Tomiyama and H. Yasuura, "Optimal Code Placement of Embedded Software for Instruction Caches," *Proc. European Design and Test Conf.,* pp. 96-101, 1996.
[17] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* fourth ed. Morgan Kaufmann, 2006.
[18] M.C. Merten, A.R. Trick, C.N. George, J.C. Gyllenhaal, and W.W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization," *Proc. 26th Ann. Int'l Symp. Computer Architecture (ISCA '99),* pp. 136-147, 1999.
[19] CPLEX, http://www01.ibm.com/software/integration/optimization/cplex-optimizer/, 2011.
[20] J.H. Ahn, S. Thoziyoor, N. Muralimanohar, and N.P. Jouppi, "Cacti 5.1," technical report, HP Laboratories, 2008.
[21] H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU," *ACM SIGPLAN Notices,* vol. 42, no. 7, pp. 195-206, 2007.
[22] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, Commercially Representative Embedded Benchmark Suite," *Proc. IEEE Int'l Workshop Workload Characterization (WWC '01),* pp. 3-14, Dec. 2001.
[23] K. Pettis and R. Hansen, "Profile Guided Code Positioning," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 16-27, June 1990.
[24] F. Glover, "Tabu Search—Part I," *ORSA J. Computing,* vol. 1, no. 3, pp. 190-206, 1989.
[25] F. Glover, "Tabu Search—Part II," *ORSA J. Computing,* vol. 2, no. 1, pp. 4-32, 1990.
[26] E. Verlind, G. Jong, and B. Lin, "Efficient Partial Enumeration for Timing Analysis of Asynchronous Systems," *Proc. 33rd Ann. Design Automation Conf.,* pp. 55-58, 1996.

Chen-Wei Huang received the BS degree in chemical engineering with minor in mathematics and electrical engineering from National Tsing Hua University, Taiwan, in 1996 and dual MS degrees in environmental engineering and computer science in 1998 and 2007 both from National Chiao Tung University, Taiwan, respectively. He is currently working toward the PhD degree at National Chiao Tung University. He worked as a research engineer in Chung-Shan Institute of Science and Technology from 1999 to 2004. He received the academical paper award from the Chinese Institute of Environmental Engineering in 2007. His research interests include embedded systems analysis and optimization, and wireless network.

Shiao-Li Tsao received the PhD degree in engineering science from National Cheng Kung University, Taiwan, in 1999. He was a visiting scholar at Bell Labs, Lucent technologies, US, in the summer of 1998, a visiting professor at the Department of Electrical and Computer Engineering, University of Waterloo, Canada, in the summer of 2007, and the Department of Computer Science, ETH Zurich, Switzerland, in the summer of 2010. From 1999 to 2003, he joined Computers and Communications Research Labs (CCL) of Industrial Technology Research Institute (ITRI) as a researcher and a section manager. He is currently an associate professor at the Department of Computer Science of National Chiao Tung University, Taiwan. His research interests include embedded software and system, and mobile communication and wireless network. He has published more than 75 international journal and conference papers, and has held or applied 16 US patents. He received the Research Achievement Awards of ITRI in 2000 and 2004, Highly Cited Patent Award of ITRI in 2007, Outstanding Project Award of Ministry of Economic Affairs (MOEA) in 2003, and Advanced Technologies Award of MOEA in 2003. He also received the Young Engineer Award from the Chinese Institute of Electrical Engineering in 2007, Outstanding Teaching Award of National Chiao Tung University, and K.T. Li Outstanding Young Scholar Award from ACM Taipei/Taiwan chapter in 2008. He is a member of the IEEE.