Code Placement with Selective Cache Activity Minimization for Embedded Real-time Software Design

Junhyung Um
CAE Center and SoC R&D Center
System LSI Division
Samsung Electronics, KOREA

Taewhan Kim
Dept. of EECS and AITrc
KAIST, KOREA

Abstract – Many embedded system designs usually impose (hard) read-time constraints on tasks. Thus, computing a tight upper bound of the worst case execution time (WCET) of a software is a critically important, but difficult task. The difficulty arises particularly when the code is executed on processors with cache-based memory systems. In this paper, we propose a new code placement technique under cache activity consideration for real-time software design. Specifically, unlike the previous approaches which have tried to minimize total cache misses, which is not necessarily the best way to meet all timing constraints of tasks, we minimizes the cache misses in a selective way for tasks according to the degree of tightness (or urgency) of their timing constraints. Based on a concept of *selective cache activity minimization*, we propose a new approach which solves the code placement problem in two steps: (Step 1) We transform the code placement problem into so called an *interval selection* problem, which then we formulate into a 0-1 integer linear programming (ILP); (Step 2) We apply an efficient approximation algorithm, called Code-map, to solve the exact code placement formulation obtained in Step 1.

1 Introduction

Embedded systems usually impose (hard) real-time constraints that should be met for correct operations. Thus, improving the execution time of software and/or estimating a tight upper bound of the worst case execution time (WCET) of software are critically important in embedded system design. On the other hand, due to the growing speed gap between memory and processor, and the sophisticated designs of memory hierarchy, lots of research efforts have been devoted to the problems of improving and/or estimating memory access latency for the software programs. One of the most important, but hard, problems among them is an accurate estimation/reduction of WCETs for programs that run on a cache-based memory system. This is because there are no simple or direct ways to control the occurrences of cache misses (thus to control the cache reloading delays) to satisfy all timing constraints of the programs (or events) in real-time embedded systems. It is generally accepted that a careful placement of code to memory is one of effective ways in reducing or controlling cache misses.

There are extensive research works which have addressed the code placement problem in cache-based embedded real-time system. The authors in [1, 2] proposed partitioned-cache structures so that each partition assigned to a task can be used by the task exclusively. Their objective of cache partitioning is to meet a certain maximal cache miss count. Min and Hu [3] proposed a cache architecture, called *color-indexed, physically tagged* cache architecture, to minimize the number of cache misses. Bellas and Hajj [4] proposed the use of an additional (small) cache, called *L-cache*, between the cache and central processing unit to minimize the energy dissipation on cache. Li, Malik and Wolfe [5] proposed a method for accurate modeling of cache activity for a given code placement and computing a tight bound on WCET. They used an integer linear programming (ILP) for estimating WCET. Tomiyama and Yasuura [6] solved the code placement problem of minimizing the number of cache misses by formulating it as an ILP problem. Liveris *et al.* [7] proposed a code transformation technique based on cache and code structures to predict the number of cache misses. Parameswaran [8, 9] proposed an algorithm for placing code to cache, from which he derived a mapping of code to memory. The objective is to minimize the total latency of memory execution. Finally, Datta, Choudhury and Basu [11] solved the code placement problem to satisfy all timing constraints of tasks by formulating it into an 0-1 ILP. However, they did not consider the possibility of code sharing among the tasks.

In this paper, we propose a new code placement and cache activity optimization approach to embedded real-time software design. Specifically, unlike the existing approaches [3, 5, 6, 7, 8, 9, 11] which tried to meet the timing constraints by using an estimation of WCET with under/over-counted cache misses and/or by minimizing total number of cache misses, we perform the code placement so that the number of cache misses for each task is *selectively* minimized according to the degree of tightness of its timing constraint. The selective minimizations of cache misses does not always imply a minimal total number of cache misses, but will directly lead to meet all timing constraints. Based on a concept of *selective cache activity minimization*, we propose a new approach that solves the code placement problem in two steps: (Step 1) We formulate the code placement problem into a 0-1 integer linear programming (ILP) using a concept called *interval selection* problem; (Step 2) We then approximate the solution of the exact problem formulation obtained in Step 1 by using an relaxation algorithm, called Code-map;

2 Preliminaries

2.1 Target Cache Organization

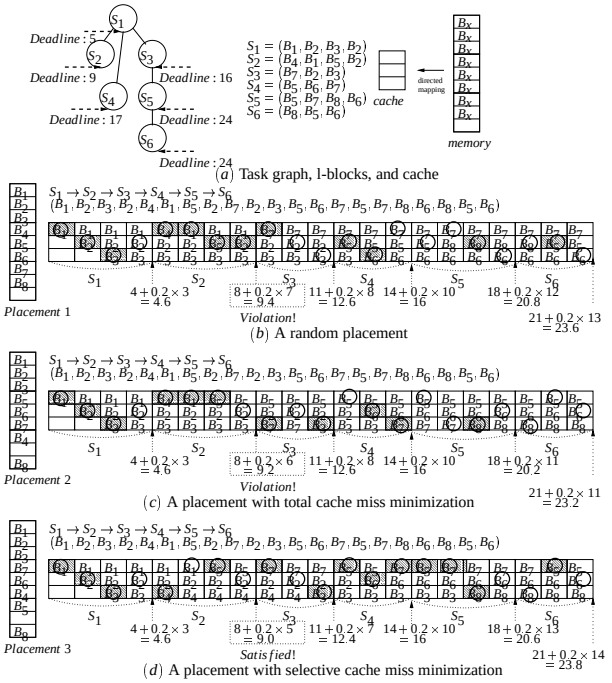
We support direct mapped caches and set associative caches. As an atomic structure to be used in cache activity analysis, we use *l-block* [5] that represents a contiguous sequence of instructions within the same block mapped to the same line in the instruction cache.[1] Each task (or job) consists of *l-blocks* each of which is a unit to be loaded from the memory to the cache if an instruction within it is to be executed. Consequently, all instructions within an *l-block* will have the same cache hit/miss counts. The *l-blocks* are of uniform size and their loading times from memory to cache are identical. Note that when an *l-block* is requested, it should be cached. That means it does not allow bypassing the caching. A caching algorithm is faced with a sequence of execution of tasks. Our objective is to take a full control of the behavior of instruction cache through careful assignments of *l-blocks* of tasks to memory,

[1] A line can contain one or multiple instructions, and its size is given according to the memory system used.

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

so that all the timing constraints of tasks are satisfied.[2]

For an N_{way}-way set-associative instruction cache, a cache that can store up to s instructions is equally divided into $m = \frac{s}{N_{way}}$ disjoint partitions (i.e., m blocks or lines) with each storing up to N_{way} instructions. Two addresses in memory are mapped to the identical block in cache only if their index fields[3] are identical. Note that $N_{way} = 1$ indicates a direct mapped cache and $N_{way} = s$ indicates a fully associative cache. To maintain a high speed of cache, N_{way} is usually a small number, and 1, 2 and 4-way caches are most commonly used.



(a) Task graph, l-blocks, and cache

(b) A random placement

(c) A placement with total cache miss minimization

(d) A placement with selective cache miss minimization

Figure 1: Motivating examples.

2.2 Motivating Examples

Before presenting our main idea of solving the code placement problem, we give a set of examples of code placement to illustrate how code placement affects the behavior of cache activity and thus the satisfiability of timing constraints of tasks. Suppose we want to execute an embedded software with six tasks S_1, S_2, $\cdots$, S_6 where the control/data dependencies among the tasks are shown on the left side of Figure 1(a). Note that each task has its timing constraint (i.e., deadline) to be met. For example, task S_1 should finish its execution by time 5. Further, each task consists of one or multiple blocks as shown in the middle of Figure 1(a). For example, S_1 is composed of three blocks B_1, B_2 and B_3 in which the execution order for S_1 is B_1, B_2, B_3, and B_2. Note that some blocks are shared among tasks. For example, B_1 is shared by S_1 and S_2. Now, suppose we have a direct mapping cache which can store up to three blocks and a memory with nine slots for storing the eight blocks, as shown on the right side of Figure 1(a). We assume the cache loading time is 0.2 unit of time.

The left side of Figure 1(b) shows an example of *l-block* placement to memory when the execution schedule of tasks is given as $S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_6$. (The corresponding sequence of block requests is also shown in the figure.) The table in Figure 1(b) shows the sequence of snapshots of cache corresponding to the sequence of block requests. The blocks in circle represent the requested blocks for execution and the blocks with shade indicate occurrences of cache miss. From the table, we can compute the time at which each task's execution is completed. For example, the completion time for S_2 is $1.0\times8+0.2\times7$ (=9.4) because there are 8 times of cache access in which 7 of them are cache misses. We can see that the code placement in Figure 1(b) is not successful due to the timing violation for task S_2. One intuitive way of satisfying the timing constraints is to minimize the total occurrences of cache misses [3, 6]. Figure 1(c) shows another sequence of snapshots of cache with minimum total number of cache misses for the same execution schedule of tasks in Figure 1(a). However, there is still a task with timing violation even though the latency of the software is reduced from 23.6 to 23.2. This example clearly indicates that minimizing the number of cache misses is effective in improving system's *overall* performance, but ineffective in real-time embedded computing systems.

On the other hand, Figure 1(d) shows a sequence of cache snapshots other than that in Figures 1(b) and (c). It meets all the timing constraints even with more cache misses than that of Figure 1(c). This example implies that a selective minimization of cache misses for tasks based on the degree of tightness (or urgency) of their timing constraints is essential to satisfy all timing constraints. (The proposed approach based on this observation is described in section 3.)

3 Code Placement for Selective Cache Activity Minimization

We propose an approach which solves the code placement problem in two steps: (Step 1) We transform the problem into so called an *interval selection* problem, which then we formulate into a 0-1 integer linear programming; (Step 2) We approximate the solutions to the exact formulation in Step 1 by proposing an algorithm called Code-map.

3.1 The Problem Formulation

We formulate the code placement problem based on a concept called *interval selection*. Note that for a sequence of *l-block* requests for execution, a distinct *l-block* in the sequence can be requested more than once. Let us examine, in the sequence, every interval between two consecutive requests of the same *l-block*. From the results of block placement to memory, we classify the intervals into two types: on-intervals are the intervals during which the corresponding *l-block* resides fully in the cache without any replacement by another block. The intervals other than on-intervals are called off-intervals. For example, the request intervals shown at the lower part of Figure 2 depict the on-intervals and off-intervals for the sequence of block requests and block placement shown at the upper part of Figure 2 where the symbols marked with 'H' represent the time of block request, and the dashed and dotted lines represent the on-intervals and off-intervals, respectively. It is clear that the more the number of on-intervals is the less the number of cache misses is. However, due to the limited cache capacity, a careful determination of on-/off-intervals is necessary to satisfy all timing constraints of tasks. We accomplish this by analyzing and describing the relations between the on/off-interval selection and code placement using a 0-1 integer linear programming formulation.

We define a number of notations and variables to be used in our ILP formulation:

- I_j^i : The j-th interval of block requests between two consecutive requests for block B_i. For example, in Figure 2, I_1^2 (= (2,4)) and I_2^2 (= (4,8)) represent the first and second intervals between consecutive requests for B_2, respectively.
- ℓ_j^i : The length of I_j^i, which is defined to the number of block requests within I_j^i except the first and the last requests of the interval. For example, in Figure 2, $\ell_1^2 = 1$ and $\ell_2^2 = 3$.

[2] In this paper, we exclude the consideration of the behavior of data cache. In addition, in the rest of paper we simply use terms *block* or *line* to refer to an *l-block*.

[3] The least significant bits of the address of a line in memory form an *index* field, and it can locate the partition of cache to which the line is supposed to be mapped. The remaining bits form a *tag* field.

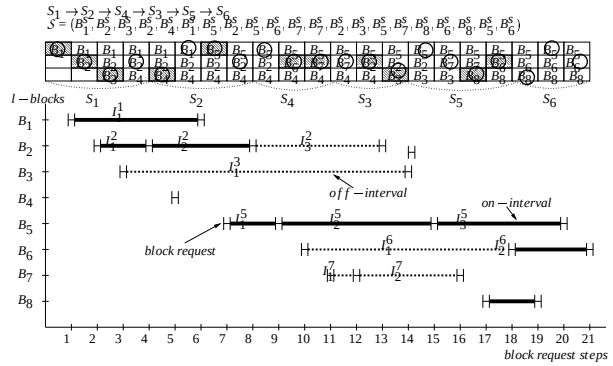


Figure 2: The derivation of on- and off-intervals.

- s_j^i : The step number of the first block request performed immediately before the execution of the first block request in I_j^i.
- e_j^i : The step number of the first block request performed immediately after the execution of the last block request in I_j^i.
- x_{ij} : $x_{ij} = 1$ iff I_j^i is an on-interval, and $x_{ij} = 0$ iff I_j^i is an off-interval.
- y_{ik} : $y_{ik} = 1$ iff block B_i is placed to the memory location of address k.
- N_{mem}, N_{cache}, N_{block}, and $N_{request}$: Memory, cache sizes (in terms of blocks), the number of distinct *l-blocks* to be placed, and the total number of block requests.

The ILP formulation consists of three types of constraints: *placement constraint*, *cache size constraint* and *deadline constraint*.

Placement constraint: The following two inequalities ensure that every distinct *l-block* should be placed in memory, and every block location of memory at address k can contain at most one *l-block*.

For each (distinct) block B_i, $i = 1, 2, \cdots N_{block}$, $\sum_{k=1}^{N_{mem}} y_{ik} = 1$,

for each memory address $k = 1, 2, \cdots N_{mem}$, $\sum_{i=1}^{N_{block}} y_{ik} \leq 1$.

Cache size constraint: For each time of block requests in the execution sequence of tasks, the number of blocks placed in cache at that time is constrained by the cache size. We formulate the constraint by using a concept of setting on-/off-intervals. For the t-th block request in the execution sequence, let $A(t) = \{I_j^i | s_j^i \leq t \leq e_j^i\}$. That is, $A(t)$ is the set of all intervals within which the t-th block request occurs. Then, we define $I\_Set(A(t), N)$ be the collection of subsets of $A(t)$ where the size of each subset is exactly N. For example, in Figure 2, $A(5)$=$\{I_1^1, I_2^2, I_1^3\}$, $I\_Set(A(5), 3) = \{\{I_1^1, I_2^2, I_1^3\}\}$, and $I\_Set(A(5), 2) = \{\{I_1^1, I_2^2\}, \{I_1^1, I_2^3\}, \{I_1^1, I_1^3\}\}$.

Suppose that we use N_{way}-set associative cache mapping. Then, at each t-th block request, $t = 1, 2, \cdots$, the number of intervals in $A(t)$ that can be on-interval and whose corresponding blocks can be placed to the memory location of the same set-address must be no more than N_{way}. This constraint is expressed as follows: For each t-th block request, $t = 1, 2, \cdots N_{request}$, for each memory's set-address $s = 1, 2, \cdots \lceil \frac{N_{cache}}{N_{way}} \rceil$, and for each element $R \in I\_Set(A(t), N_{way}+1)$,

$$\sum_{I_j^i \in R} (x_{ij} + \sum_{\forall k:\ k \equiv s \bmod \lceil \frac{N_{cache}}{N_{way}} \rceil} y_{ik}) \ \leq \ 2 \cdot N_{way} + 1$$

where the inequality violation occurs when the blocks corresponding to the intervals in R (i.e., all $N_{way}+1$ intervals, thus $N_{way}+1$ distinct blocks) were all in cache at the time when t-th block request occurs and further they were mapped to the same set-address in cache.

Furthermore, if we consider the case that the block (say B_v) requested at the t-th block request is to be placed to a memory location of set-address s, the number of intervals in $A(t)$ that can be on-interval and whose corresponding blocks can be placed to the memory locations of set-address s must be no more than $N_{way} - 1$. This constraint is expressed by: For each t-th block request, $t = 1, 2, \cdots N_{request}$, for each set-address $s = 1, 2, \cdots \lceil \frac{N_{cache}}{N_{way}} \rceil$, and for each element $R \in I\_Set(A(t), N_{way})$,

$$\sum_{I_j^i \in R} (x_{ij} + \sum_{\forall k:\ k \equiv s \bmod \lceil \frac{N_{cache}}{N_{way}} \rceil} y_{ik}) + \sum_{\forall k:\ k \equiv s \bmod \lceil \frac{N_{cache}}{N_{way}} \rceil} y_{vk} \ \leq \ 2 \cdot N_{way}.$$

where the inequality violation occurs when the blocks corresponding to the intervals in R (i.e., all N_{way} intervals, thus N_{way} distinct blocks) were mapped to the same set-address in cache and further they were all in cache at an instance of time after when the t-th requested block is load into cache.

Delay constraint: For the simplicity of description, we assume that we have scaled the block access time to 1 unit of time, the block reloading time to α unit of time, and the deadline of task S_k to D_k units of time. Then, for each task S_k, $k = 1, 2, \cdots$, the total block reloading time executed immediately before the completion of task S_k (which is $\alpha \cdot$ *#cache_misses*) should not exceed $D_k - C_k$. This constraint is expressed by: For each task S_k, $k = 1, 2, \ \cdots$,

$$\alpha \cdot (\sum_{I_j^i \in \{I_{j'}^{i'} : e_{j'}^{i'} < C_k\}} (1 - x_{ij})) \ \leq \ D_k - C_k.$$

3.2 The Proposed Approximation Technique

The optimization problem is, for the set of intervals I_i^j derived from a given sequence of block requests, to determine which intervals should be on-interval and which intervals should be off-interval so that the total number of off-intervals is as small as possible while meeting all timing constraints. Solving the problem is then equivalent to solving the problem of minimizing the quantity of $\sum_{I_j^i}(1 - x_{ij})$ and satisfying all the inequalities for the placement, cache size and delay constraints in our 0-1 ILP formulation.

Our key idea in our approximation technique, called Code-map summarized in Figure 3, is that we relax the constraint that all variables (i.e., x_{ij} and y_{ik}) in the ILP formulation should be either 0 or 1 and nothing else. By relaxation, we mean the variables to have any fractional value in between 0 and 1, in addition to 0 or 1. Consequently, Code-map uses an efficient LP (linear programming) formulation to approximate the exact solution of the ILP formulation. From the LP solution obtained, which might be far from feasible solutions, we derive a solution, which is much less far from feasible solutions, by selecting and setting the most "promising" variable to 0 or 1. Then, in the next iteration we generate another LP formulation for the code placement with the preservation of the values of the variables that were set in the prior iterations, and select the next "promising" variable from the solution of LP formulation and set it to 0 or 1. We repeat this process until all the variables are determined to either 0 or 1.

The variable to be selected and set with 0 or 1 at each iteration is the one with the largest value among

$$(x_{ij} - \bar{x}) / \sum_{\forall x_{ij}} (x_{ij} - \bar{x})^2, \quad (y_{ik} - \bar{y}) / \sum_{\forall y_{ik}} (y_{ik} - \bar{y})^2 \qquad (1)$$

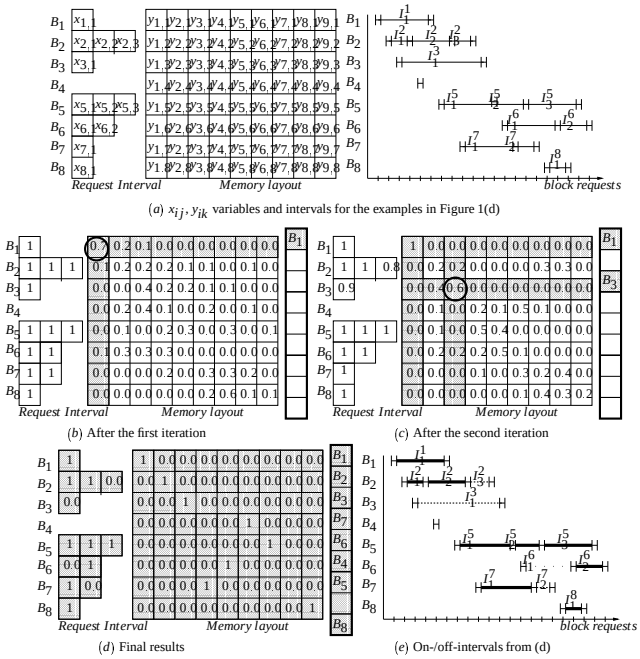
where $\bar{x}$ and $\bar{y}$ are the averages of the values of x_{ij} and y_{ik} variables in the LP solution that have not been set to 0 or 1 during the previous iterations.

The example in Figure 4 shows the steps of procedure by Code-map for the task schedule in Figure 1(a). Figure 4(a) shows the variables x_{ij} and y_{ik}, and the intervals corresponding to the task schedule. Figure 1(b) shows the values of variables obtained from the LP formulation. Note that the values of variables x_{ij} were all 1

Code-map: *Code mapping for selective cache miss minimization*
repeat {
- Generate the exact 0-1 ILP and approximate it using the proposed relaxed LP;
- Select a variable among x_{ij}, y_{ik} with the largest value of *Eq.(1)*;
- Set the value of the selected variable to 0 or 1 by rounding;

} **until** (all variables are set to 0 or 1)

Figure 3: The procedure of Code-map.



(*a*) x_{ij}, y_{ik} variables and intervals for the examples in Figure 1(d)

(*b*) After the first iteration

(*c*) After the second iteration

(*d*) Final results

(*e*) On-/off-intervals from (d)

Figure 4: An example illustrating the procedure by Code-map.

to satisfy the objective in LP formulation: $min.\ \sum_{I_j^i}(1-x_{ij})$. Then, according to the *Eq.(1)*, we select variable $y_{1,1}$ and set to 1, which tells that block B_1 is placed to the memory location of address 1, as indicated in Figure 1(b). The result for the LP formulation of the second iteration of Code-map is shown in Figure 1(c) where $y_{3,3}$ is selected and set to 1. By repeating this process, we have the result for the final LP formulation, as in Figure 1(d), from which we determine that the on-intervals are I_1^1, I_1^2, I_2^2, $\cdots$ I_1^7 and I_1^8 (i.e., $x_{1,1} = x_{2,1} = x_{2,2} = \cdots = x_{7,1} = x_{8,1} = 1$), and the rest are off-intervals.

4 Experimental Results

We tested our algorithm on a set of randomly generated testcases and SPEC95 benchmark programs to demonstrate the effectiveness of the strategy of selective cache-miss minimization. We compiled the benchmarks using the MIPS-like PISA processor instruction set in SimpleScalar 2.0 [12] tool set. We generated traces for the benchmarks using SimpleScalar 2.0 [12] with default configuration on SunUltra II-server(version 2.6). In addition, we partition them considering dependency between traces to solve this by LP/ILP solver. We perform the cache simulation using a Pentium-4 500MHz Linux machine with 1 GB Memory. Further, we used ILOG CPLEX 7.0 [13] to solve the LP formulations. We set the delay of accessing a block from cache is 0.2 time unit and the delay of cache reloading is 0.04 time unit. To impose tight deadlines to tasks, we set the deadline of each task to $N_{tot} \cdot (1+\delta)$ where N_{tot} is the total number of blocks in tasks and δ is a random number in between 0.01 and 0.1.

Table 1 shows comparisons, in terms of the number of tasks with timing violation, of the code placements produced by cache-miss-min (the total cache miss minimization by [6]) and our code-map (the selective cache miss minimization). We use 16 Kbyte 2-way/4-way associative instruction cache with 32-byte lines. For a cache replacement policy in cache-miss-min, we used LRU strategy in the experiments. In addition, we partitioned each benchmark program to roughly 5000 tasks. Overall, code-map reduces the number of tasks with timing violations by 16.7% and 24.5% over that by [6] when 2-way and 4-way associative caches, respectively.

program	#tasks with timing violation				improv.(%) over cache-miss-min	
	cache-miss-min [6]		code-map			
	2-way	4-way	2-way	4-way	2-way	4-way
COMPRESS95	713	603	629	553	11.8	8.3
GCC	3419	3241	3008	2901	12.0	10.5
GO	3688	2302	2335	1034	36.7	55.1
LI	3425	1629	3007	1026	12.2	37.0
M88KSIM	2310	1997	2055	1593	11.0	20.2
PERL	2593	2190	3459	1839	48.7	16.0
avg.					**16.7**	**24.5**

Table 1: Comparisons of results for benchmark examples.

5 Conclusion

In this paper, we proposed a new code placement approach that is suitable for embedded hard read-time software design with cache-based memory system. Our key feature is that, unlike the previous approaches which have tried to minimize total cache misses, which is not necessarily the best way to meet all timing constraints of tasks, we minimizes the cache misses in a selective way for tasks according to the degree of tightness or urgency of their timing constraints. Based on a concept of the *selective cache activity minimization*, we solved the code placement problem by decomposing it into two subproblems: (1) A translation of the problem into an *interval selection* problem and an exact problem formulation by a 0-1 ILP; (2) An efficient approximation of the ILP, called Code-map.

Acknowledgment This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc).

REFERENCES

[1] D. B. Kirk, "SMART (Strategic Memory Allocation for Real-Time Cache Design," *Proc. of 10th Real-Time System Symp.*, 1989.

[2] J. Liedtke, H. Hartig, and M. Hohmuth, "OS-Controlled Cache Predictability for Real-Time Systems," *Proc. of Real-Time Technology and Application Symposium*, 1997.

[3] R. Min and Y. Hu, "Improving Performance of Large Physically Indexed Caches by Decoupling Memory Addresses from Cache Addresses," *IEEE Trans. on Computers*, Vol.50, No.11, 2001.

[4] N. E. Bellas and I. N. Hajj, "Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors," *IEEE Trans. on VLSI Systems*, Vol8, No.3, 2000.

[5] Y.-S. Li, S. Malik and A. Wolfe, "Performance Estimation of Embedded Software with Instruction Cache Modeling," *Proc. ICCAD*, 1995.

[6] H. Tomiyama and H. Yasuura, "Optimal Code Placement of Embedded Software for Instruction Cache," *Proc. EDAC*, 1996.

[7] N. Liveris, N. D. Zervas, D. Soudris, and C. E. Goutis, "A Code Transformation-Based Methodology for Improving I-Cache Performance of DSP Applications," *Proc. DATE*, 2002.

[8] S. Parameswaran, "Code Placement in Hardware Software Co-synthesis to Improve Performance and Reduce Cost," *Proc. DATE*, 2001.

[9] S. Parameswaran and J. Henkel, "I-CoPES: Fast Instruction Code Placement for Embedded Systems to Improve Performance and Energy Efficiency," *Proc. ICCAD*, 2001.

[10] P. Jain, S. Devadas, D. Engels, and L. Rudolph, "Software-assisted Cache Replacement Mechanisms for Embedded Systems", *Proc. ICCAD*, 2001.

[11] A. Datta, S. Choudhury and A. Basu, "Using Randomized Rounding to Satisfy Timing Constraints of Real-Time Preemptive Tasks," *Proc. ASPDAC*, 2002.

[12] D. Burger, T.M. Austin, The SimpleScalar Tool Set, Version 2.0, TR No. 1342, University of Wisconsin-Madison CSD, June 1997.

[13] *http://www.ilog.com*