Optimal Code Placement of Embedded Software for Instruction Caches

Hiroyuki Tomiyama Hiroto Yasuura
Department of Information Systems,
Interdisciplinary Graduate School of Engineering Sciences,
Kyushu University
6–1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan

Abstract

This paper presents a new code placement method for embedded software to maximize hit ratios of instruction caches. We formulate the code placement problem as an integer linear programming problem. One of the advantages of our method is that code can be moved beyond boundaries of functions, so that code placement is optimized globally. Experimental results show our method achieves 35% (max 45%) reduction of cache misses.

1 Introduction

In design of an embedded system, several design goals such as high performance, low cost, and low power consumption of the system must be achieved simultaneously. But these design goals are often mutually exclusive. Consider a system which consists of a processor core, main memories and cache memories. The performance of the system is expressed as the following formula:

$$Performance = \frac{1}{Execution\ time} = \frac{F}{IC \times (CPI + (1 - CHR) \times CMP)} \quad (1)$$

where F, IC, CPI, CHR and CMP denotes the clock frequency, the instruction count to be executed, clock cycles per instruction, the cache hit ratio, and the cache miss penalty respectively. To improve the performance, several approaches can be considered. One is to raise the clock frequency, but the power consumption is increased in proportion to the clock frequency. Raising the parallelism in the processor to make the CPI small is an attractive approach, but it leads the increase of chip area. High speed memories and buses with wide band width also require chip area and power. Reducing the instruction count to be executed is one of very effective approaches to make the high performance compatible with the low cost and the low power. Recently, a lot of code generation techniques for embedded processors have been proposed to minimize the number of executed instructions[4, 7]. Most of them do not require extra hardware cost, and some techniques target implementation of low power systems[11].

In this paper, we focus on another approach for performance improvement, reduction of cache misses. Let's consider an ideal processor whose CPI is 1 and the cache miss penalty is 10 cycles. If we can improve the cache hit ratio from 93% to 96%[1], the improvement produces over 20% enhancement of the performance of the system. It is widely recognized that cache hit ratios affect the performance gravely. Enlarging the cache size, increasing the associativity of caches and employing a better replace algorithm may reduce cache misses, but those approaches will also make the system more expensive. In code generation phase of embedded software design, code optimization for improving the cache hit ratio is very effective as well as reducing the instruction count and the power consumption.

In this paper, we propose a new optimization method for code placement of embedded software which minimizes the cache misses of instruction caches. Our approach makes it possible to accomplish the high performance of the system without extra hardware cost. It is also expected that the power consumption will be decreased by reducing cache misses.

In the following section, some related works are discussed. In section 3, we propose a new code placement method, and we formulate the code placement problem as an integer linear programming (ILP) problem in section 4. Experiments are presented in section 5, and we discuss some problems of the proposed method in section 6. Section 7 presents conclusions and addresses our future works.

2 Related Works

The code placement problem for instruction caches has been studied by many researchers in a field of computer architectures[6, 8, 9].

IMPACT-I C Compiler which was developed in Illinois University uses four techniques to maximize cache hit ratios[6]. (a) *Function inline expansion*: The function calls with high execution count are replaced with the function body if possible. (b) *Trace selection*: For each function, basic blocks which tend to execute in sequence are grouped into a trace. (c) *Function layout*: For each function, the trace of the function entrance is placed first, and then the most important descendent is selected to be placed after it. (d) *Global layout*: The functions which are executed close to each other in time are placed not to conflict in caches.

McFarling proposed a function placement technique for instruction caches in [8]. Dependencies among functions in the program are analyzed firstly. If it is found that

[1]Readers will notice in section 5 that we can improve the hit ratio of instruction cache from 93% to 96%.

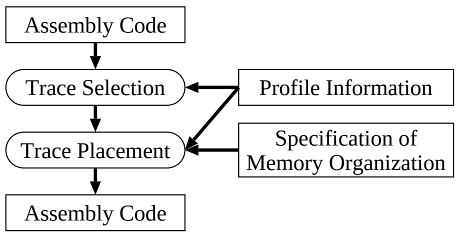


Figure 1: Overview of the code placement method

function F is called in F', the two functions are placed not to conflict each other. McFarling also proposed a technique determining which functions should be merged[9], but code placement inside functions is not considered.

In the above approaches, mobility of code is restricted by boundaries of functions. One of the advantages of our method is that code can be moved beyond boundaries of functions, so that code placement is optimized globally. An optimal placement can be obtained by solving an ILP problem. For a large program, an optimal placement can not be obtained in a practical time. Still, our method gives a great benefit, which means that in most cases, sub-optimal solutions are much better than ones to which previous methods lead. Furthermore, performance of the object code is much more important than compilation time in embedded software design because software programs will rarely be modified after they are stored in ROM. As it takes a longer time to solve the ILP problem, a better solution can be obtained which will improve the performance of the designed system.

3 Code Placement Method

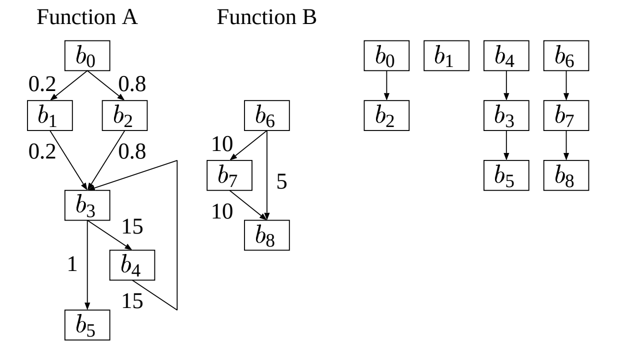
3.1 Overview

In this section, our code placement techniques are presented.

We suppose direct mapped caches and set associative caches which employ the least recently used (LRU) algorithm for replacement. We also assume that the target system has a Harvard architecture whose instruction caches and data caches are separated physically as well as logically. Since our objective is maximization of hit ratios of primary caches, behavior of secondary caches are not taken into account.

We show the overview of our code placement method in Fig.1. Inputs of our algorithm are assembly code, profile information of the application program, and specification of memory organization. The profile information is one or more sequence(s) of basic blocks which are accessed when typical input data is given to the program. For example, let's consider a program illustrated in Fig.2(a), which consists of two functions, A and B. Each node in the graph represents a basic block, and each directed edge represents a control dependency between basic blocks. A number associated with each edge denotes the ratio of times the edge is passed when the program is executed once. Here, we assume that function B is called in basic block b_4 in function A. The profile information for the program contains the following sequence,

$$(b_0, b_2, b_3, b_4, b_6, b_7, b_8, b_3, b_4, b_6, b_8, b_3, \cdots, b_3, b_5) \quad (2)$$



(a) Weighted control flow graph (b) Traces

Figure 2: Weighted control flow graph and traces

which means that basic block b_0 is executed first, next b_2, five basic blocks $\{b_3, b_4, b_6, b_7, b_8\}$ are executed iteratively, and finally, b_5 is executed. Specification of memory organization consists of the line size, the number of sets and ways of instruction caches, and the size of instruction memories. Output is assembly code of the programs whose cache hit ratio is maximized.

According to [5], cache miss factors are classified into three C's, first references (compulsory), capacity and conflicts. Concerning set associative caches with LRU replacement algorithm, however, we can not clearly distinguish cache misses caused by lack of the capacity and ones by cache conflicts, because insufficient capacity of caches causes cache conflicts. In this paper, we classify cache miss factors into the two, first references and cache conflicts.

The proposed method consists of two phases, *trace selection* and *trace placement*. The trace selection reduces cache misses caused by first reference, and the trace placement reduces cache conflicts. In section 3.2 and 3.3, we explain basic ideas and techniques of trace selection and trace placement respectively.

3.2 Trace Selection

Given assembly code of the program and profile information, we construct an weighted control flow graph firstly as shown in Fig.2(a). It is more probable that basic block b_2 will be executed after b_0 than b_1 will. In this case, b_2 should be placed just after b_0 because it is highly possible that the two basic blocks are on the same cache line. Due to the same reason, b_4 should be placed before b_3, b_5 after b_3, b_7 after b_6, and b_8 after b_7. As a result, the weighted control flow graph (a) is partitioned into four paths (linear subgraphs) shown in Fig.2(b). We call each path a *trace*.

The term *trace* is introduced in [2] which targets global microcode compaction, and the idea is extended in IMPACT-I C Compiler[6] to maximize cache hit ratios.

The trace selection problem is defined as follows formally:

"For a given weighted control flow graph of the program, partition the graph into traces such that the sum of weights of all edges in the traces is maximized."

Since the trace selection problem is NP-complete unfortunately, in this paper, we use a greedy algorithm for the problem to obtain a quasi-optimal solution.

Note that the last instruction of each trace is an unconditional jump operation or an exit of the function. This property enables traces to be placed in arbitrary order. A trace is an atomic unit of machine instructions which can be placed without insertion of jump operations.

3.3 Trace Placement

After trace selection, we must determine in which order traces should be placed in a main memory space to minimize cache misses caused by cache conflicts.

We introduce a new concept, *pseudo-memory*, which is a virtual main memory. First, traces are placed in the pseudo-memory in arbitrary order, but we pose the restriction that different traces must be placed in different pseudo-memory blocks. Here, a memory block is a block in a main memory which are mapped onto one cache line, and a pseudo-memory block is a block in a pseudo-memory. We show an example of trace placement on a pseudo-memory in Fig.3(a). After that, a sequence of pseudo-memory blocks which are accessed in execution of the program is determined uniquely. Then, trace placement problem is defined as a matching problem between pseudo-memory blocks and real memory blocks. In the following section, we formulate the trace placement problem as an ILP problem. Here, we explain the effect of trace placement using the example shown in Fig.2 and Fig.3

According to Fig.3(a), the sequence of basic blocks (2) is translated into a sequence of pseudo-memory blocks as described below.

$$(p_0, p_1, p_3, p_5, p_6, p_7, p_3, p_5, p_7, p_3, \cdots, p_3, p_4) \quad (3)$$

In this paper, we call the sequence of pseudo-memory blocks to be accessed in execution of the program given a data(an input data to the program), the *access sequence of pseudo-memory blocks*, or simply the *access sequence*. The above access sequence tells that four pseudo-memory blocks $\{p_3, p_5, p_6, p_7\}$ are accessed iteratively. Assume a direct mapped cache whose number of cache line is 4. If we place traces in main memory in the same order as pseudo-memory, cache misses occur frequently due to the cache conflicts between p_3 and p_7. We show an optimal trace placement in Fig.3(b) where no cache conflicts can be seen.

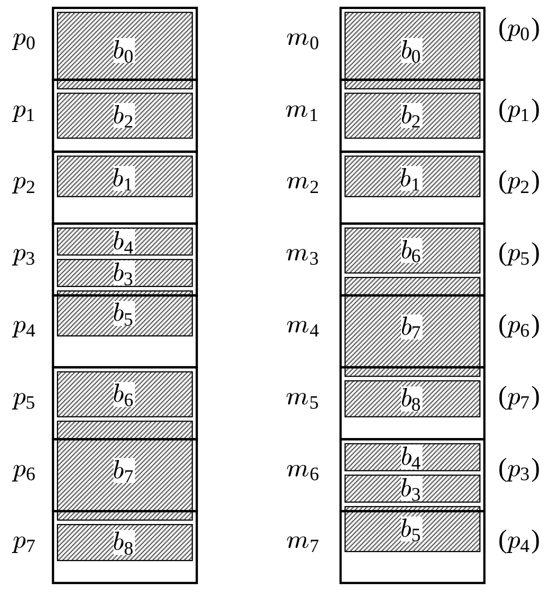
Because of the restriction that different traces must be placed in different pseudo-memory blocks, our method generates many redundant spaces in the instruction memory where no instructions reside. These redundant spaces are never used in execution of the program, and cause expansion of code size. We discuss the problem in section 6.2.

4 ILP Formulation of Trace Placement Problem

4.1 Preliminaries

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

N_{set}	The number of sets of an instruction cache
N_{way}	The number of ways of an instruction cache
N_{mem}	The number of blocks of an instruction memory
N_p	The number of blocks of a pseudo-memory



(a) Pseudo-memory (b) Main memory

Figure 3: Trace placement

p_i	A pseudo-memory block ($i = 0, \cdots, N_p - 1$)
P	A set of pseudo-memory blocks $P = \bigcup_i \{p_i\}$
x_i	The memory block where p_i is placed
X	A vector of x_i $X = (x_0, x_1, \cdots, x_{N_P})$
t_j	An ordered set of pseudo-memory blocks which construct jth trace $t_j \in P \times P \times \cdots$
T	A set of t_j $T = \bigcup_j \{t_j\}$

In the example illustrated in Fig.3(a), trace (b_0, b_2) is placed in pseudo-memory blocks (p_0, p_1), (b_1) is in (p_2), (b_4, b_3, b_5) is in (p_3, p_4), and (b_6, b_7, b_8) is in (p_5, p_6, p_7). Then, T of this example is defined as follows:

$$t_0 = (p_0, p_1), t_1 = (p_2), t_2 = (p_3, p_4), t_3 = (p_5, p_6, p_7)$$
$$T = \{t_0, t_1, t_2, t_3\} \quad (4)$$

In the rest of this subsection, we define A_i which denotes profile information of pseudo-memory block p_i.

First, for each p_i, we define sets of pseudo-memory blocks $c_{i,l}$'s,

$$c_{i,l} = \{\ p_j \mid p_j\ (i \neq j) \text{ appears between the } l \text{ th appearance and } (l+1)\text{th appearance of } p_i \text{ in the access sequence of pseudo-memory blocks. } \} \quad (5)$$

In the access sequence, the same $c_{i,l}$ may appear more than twice. We define a set $C_i = \{c_{i,l}\}$ and denote content of C_i, $a_{i,k}$ $(k = 0, 1, \cdots, |C_i| - 1)$. Note that

$a_{i,k} \neq a_{i,k'}$ if $k \neq k'$. For example, we show how to define $a_{3,k}$'s. According to the access sequence of pseudo-memory blocks (3) described in section 3.3, there are two cases in the intervals between accesses to p_3. One is the case when three pseudo-blocks $\{p_5, p_6, p_7\}$ are accessed, and another is when two pseudo-blocks $\{p_5, p_7\}$ are accessed. Then $a_{3,k}$'s are defined as follows:

$$a_{3,0} = \{p_5, p_6, p_7\}, \;\; a_{3,1} = \{p_5, p_7\}$$

Next, we define $e_{i,k}$ for each $a_{i,k}$ as the times of appearance of $a_{i,k}$ in the access sequence of pseudo-memory blocks. In the above example, $e_{3,k}$'s are defined as follows:

$$e_{3,0} = 10, \;\; e_{3,1} = 5$$

Assume a n-way set associative cache. If more than $(n-1)$ pseudo-memory blocks in $a_{i,k}$ are mapped onto the same cache set as p_i, cache misses occur at least $e_{i,k}$ times.

We represent a tuple of $a_{i,k}$ and $e_{i,k}$ as $A_{i,k}$, i.e. $A_{i,k} = (a_{i,k}, e_{i,k})$, and define a set of $A_{i,k}$ as A_i.

$$A_i = \bigcup_k \{A_{i,k}\} \tag{6}$$

A_i's for all pseudo-memory blocks can be calculated from profile information of basic blocks described in section 3.1.

4.2 Problem Definition

Trace placement problem is defined as follows:

"For given N_{set}, N_{way}, N_{mem}, P, T, and A_i for all p_i, find X which minimizes the cache miss hit count."

This problem can be formulated as an integer linear programming problem. The objective function of the ILP problem is defined as formula (7). The value of $M(X)$ represents the number of cache misses when the programs are executed once.

$$\begin{aligned} M(X) &= \sum_{p_i \in P} \sum_{(e_{i,k}, a_{i,k}) \in A_i} e_{i,k} \times replaced(a_{i,k}) \\ &\quad + Constant \end{aligned} \tag{7}$$

Here, $Constant$ is the number of cache misses caused by first references. $Constant$ is determined by trace selection independent of trace placement. The value of function $replaced(a_{i,k})$ is 1 if the number of pseudo-memory blocks which are in $a_{i,k}$ and are mapped onto the same cache set as p_i is greater than or equal to N_{way}, that is the case where the pseudo-memory block p_i on the cache is displaced by other pseudo-memory block before p_i is executed again. Function $replaced(a_{i,k})$ is formulated as follows:

$$\begin{aligned} & replaced(a_{i,k}) \\ &= \begin{cases} 1 & if \sum_{p_{i'} \in a_{i,k}} conflict(x_i, x_{i'}) \geq N_{way} \\ 0 & otherwise \end{cases} \end{aligned} \tag{8}$$

The value of $conflict(x_i, x_{i'})$ is 1 if the two pseudo-memory blocks p_i and $p_{i'}$ are mapped onto the same cache line, otherwise 0. Function $conflict(x_i, x_{i'})$ is defined as follows formula:

$$\begin{aligned} & conflict(x_i, x_{i'}) \\ &= \begin{cases} 1 & if \; (x_i \, mod \, N_{set}) = (x_{i'} \, mod \, N_{set}) \\ 0 & otherwise \end{cases} \end{aligned} \tag{9}$$

Constraints are expressed by the following three formulas.

$$0 \leq x_i \leq N_{mem} - 1, \qquad 0 \leq i \leq N_p - 1 \tag{10}$$

$$i \neq i' \;\; \Rightarrow \;\; x_i \neq x_{i'} \tag{11}$$

$$(\cdots, \, p_i, \, p_{i'}, \, \cdots) \in T \;\; \Rightarrow \;\; x_i = x_{i'} - 1 \tag{12}$$

Formula (10) ensures that all the pseudo-memory blocks must be mapped to physical memory blocks. Formula (11) ensures that different pseudo-memory blocks must be placed in different memory blocks. Formula (12) ensures that a sequence of pseudo-memory blocks which construct a trace must be placed in sequence.

4.3 Linearization

$Replaced(a_{i,k})$ and $conflict(x_i, x_{i'})$ defined above are not linear functions. In this subsection, we explain how to linearize the two functions.

First, we prepare new variables $y_{i,i'}$'s and $z_{i,i'}$'s whose ranges are

$$y_{i,i'} \in \{0, 1\}, \;\; z_{i,i'} \in \mathbf{Z} \tag{13}$$

where $\mathbf{Z}$ is a set of integers. Intuitively, $y_{i,i'}$ holds the value of $conflict(x_i, x_{i'})$, and $z_{i,i'}$ holds the value of $(x_i - x_{i'}) \div N_{set}$. Then, formula (9) is replaced by formulas (13), (14), (15) and (16).

$$0 \leq (x_i - x_{i'}) - N_{set} \cdot z_{i,i'} < N_{set} \tag{14}$$

$$(x_i - x_{i'}) - N_{set} \cdot z_{i,i'} + y_{i,i'} \cdot U \;\; \neq \;\; 0 \tag{15}$$

$$(x_i - x_{i'}) - N_{set} \cdot z_{i,i'} - (1 - y_{i,i'}) \cdot U \;\; \leq \;\; 0 \tag{16}$$

Here, U is a large integer. Next, we prepare variables $w_{i,k}$'s whose ranges are

$$w_{i,k} \in \{0, 1\} \tag{17}$$

Intuitively, $w_{i,k}$ holds the value of $replaced(a_{i,k})$. Then, formula (8) is replaced by the three formulas (17), (18) and (19).

$$\sum_{p_{i'} \in a_{i,k}} y_{i,i'} + (1 - w_{i,k}) \cdot U \;\; \geq \;\; N_{way} \tag{18}$$

$$\sum_{p_{i'} \in a_{i,k}} y_{i,i'} - w_{i,k} \cdot U \;\; < \;\; N_{way} \tag{19}$$

The objective function $M(X)$ is re-defined as following formula:

$$\begin{aligned} M(X) &= \sum_{p_i \in P} \sum_{(e_{i,k}, a_{i,k}) \in A_i} e_{i,k} \cdot w_{i,k} \\ &\quad + Constant \end{aligned} \tag{20}$$

As a result, the code placement problem has been linearized, whose objective function is formula (20) and constraints are formulas (10)–(19).

Table 1: Benchmark programs

	GNU grep 2.0	GNU sed 2.05
#Size[†]	12436 lines	13544 lines
#Trace	1067 traces	1041 traces
#MemoryBlock	2673 blocks	2436 blocks
Description	print lines matching a pattern	stream editor

† Number of lines of the C program including comments

5 Experiments

In this section, effectiveness of our method is evaluated. We compare four code placement methods in terms of cache miss count and cache hit ratio when changing the organization of instruction caches.

Default: Benchmark programs are compiled with SunPro SPARCompiler C 3.0. No code placement techniques for instruction caches are applied.

Trace: After translating benchmark programs into assembly code, trace selection is performed. A greedy algorithm is used for trace selection.

Func: After trace selection, functions are sorted according to their execution counts. This technique is helpful to avoid cache conflicts among functions which are frequently executed.

Ours: The method proposed in this paper is applied. We use a local search algorithm for the ILP problem.

We use SPARC instruction set as a target architecture, and use GNU grep 2.0 and GNU sed 2.05 as benchmark programs(See Table 1). For simplification, we assume that no C library functions are called in the programs.

First, we change the associativity of instruction cache from 1 to 4 and calculate cache miss counts and cache hit ratios for each of the four methods. The cache size and the line size are fixed to 1K bytes and 32 bytes respectively. The experimental result is shown in Table 2. Next, we change the cache size from 512 bytes to 2K bytes and also calculate cache miss counts and cache hit ratios. Here, we assume direct mapped caches whose associativity is 1, and the cache line size is fixed to 32 bytes. We show the result in Table 3.

These results show that the proposed method (**Ours**) obtains the highest cache hit ratio in any condition. **Ours** achieves 35% reduction of cache misses on average (max 45%) as compared with **Default**. Table 3 indicates that if system designers want to improve the performance, they should apply our method before doubling the cache size. **Trace** achieves 16% reduction of cache misses, which confirms the effectiveness of trace selection. But **Ours** is much better owing to trace placement without limitation on function boundaries.

6 Discussions

6.1 Computation Time

While our method achieves drastic reduction of cache misses, it requires a long computation time to solve the ILP problem. It is impossible to obtain optimal solutions for large programs in a practical time. We have implemented a solver of the ILP problem which employs a local search algorithm. The solver required 3–6 hours for GNU grep and 10–38 hours for GNU sed to generate a locally optimal solution on Sparc Station 5 (microSPARC-II, 85MHz, 32MB, Solaris 2.4).

We illustrate the relation between computation time and cache miss count in Fig.4. Cache miss count using the best solution at the time is plotted. Our experiments show that 70–93% of cache miss reduction are gained at the first 1 hour of the computation time for GNU grep, and 63–78% of reduction for GNU sed.

There is a tradeoff between the above computation time and the cache miss count. Embedded software designers had better optimize code placement as long as the design time permits. It is highly possible that better algorithms and implementations for the ILP problem can give better solutions in a shorter time.

6.2 Code Size

As mentioned in section 3.3, our method generates a lot of redundant spaces in instruction memories, which

Table 2: Cache miss counts and cache hit ratios when changing associativity

	GNU grep 2.0					
	1-way		2-way		4-way	
	Misses	Ratio	Misses	Ratio	Misses	Ratio
Default	1071	.9319	1025	.9348	1039	.9339
Trace	895	.9435	859	.9458	847	.9465
Func	848	.9465	786	.9504	790	.9501
Ours	584	.9631	636	.9598	631	.9601

	GNU sed 2.05					
	1-way		2-way		4-way	
	Misses	Ratio	Misses	Ratio	Misses	Ratio
Default	2999	.9288	2922	.9306	2887	.9315
Trace	2395	.9432	2365	.9439	2367	.9439
Func	2488	.9410	2430	.9424	2383	.9435
Ours	2021	.9521	2124	.9497	2172	.9485

Table 3: Cache miss counts and cache hit ratios when changing cache size

	GNU grep 2.0					
	512 bytes		1024 bytes		2048 bytes	
	Misses	Ratio	Misses	Ratio	Misses	Ratio
Default	1355	.9138	1071	.9319	693	.9559
Trace	1136	.9283	895	.9435	740	.9533
Func	1113	.9297	848	.9465	644	.9593
Ours	865	.9454	584	.9631	438	.9723

	GNU sed 2.05					
	512 bytes		1024 bytes		2048 bytes	
	Misses	Ratio	Misses	Ratio	Misses	Ratio
Default	3386	.9196	2999	.9288	2563	.9392
Trace	2801	.9336	2395	.9432	1996	.9527
Func	2819	.9332	2488	.9410	1991	.9528
Ours	2414	.9428	2021	.9521	1493	.9646

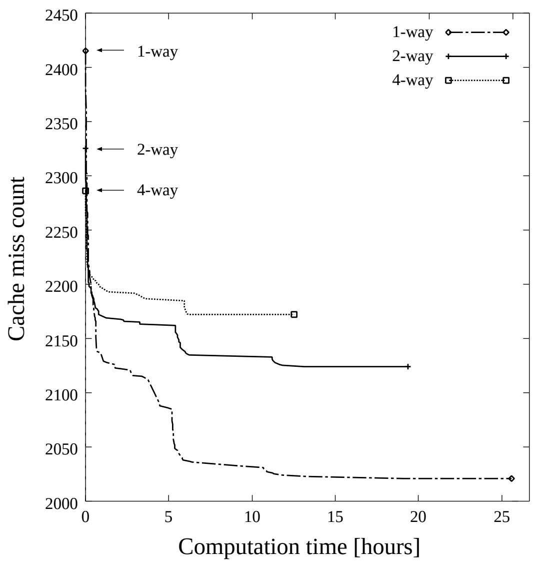


Figure 4: Computation time versus cache miss count: Benchmark program is GNU sed 2.05. Cache size is 1K bytes.

make the code size large. Experiments show that code size including empty words becomes 25–29% larger than before optimization(See Table 4). Since, in usual, the memory size varies discretely e.g. 512K bytes, 1M bytes, 2M bytes, and so on, a small amount of code size expansion may be absorbed by quantumization of the memory size. But in most cases of embedded system design, code size expansion is a serious problem especially for on-chip memory. Designers want to save the memory size to reduce the cost of designed systems. We take account of the main memory size as a constraint of the optimization in formula (10). In the experiments in the previous section, we assume that the main memory and pseudo-memory have the same capacity. If the main memory size is larger than the pseudo-memory size, higher cache hit ratio is expected. Otherwise, no solutions of the ILP problem exist. In this case, we need some techniques to eliminate the memory space redundancy.

One technique to reduce the redundancy of memory spaces is with extra hardware. Since there is no need to implement the redundant spaces physically, redundancies can be removed by re-customizing the address decoder of instruction memory at a little expense of hardware cost. This approach may be effective for a system with large production volumes and the program is completely fixed.

Another technique is to merge a couple of traces into one trace so that the size of the merged trace becomes a multiple of the cache line size. While this technique can reduce the redundancy without modification of hardware, the quality of solutions of the ILP problem may become worse. This is because decrease of the number of traces causes increase of the constraints (12) for the ILP problem.

7 Conclusions

In this paper, a new code placement technique for embedded software to maximize cache hit ratios has been presented. We have formulated the code placement problem as an integer linear programming problem. Although our formulation is very simple, experiments prove the effectiveness of our approach. We have implemented local search algorithm for ILP problem, but it takes a long computation time. Efficient algorithms and implementations to solve the ILP problem is required for practical applications.

In embedded system design, determining the organization of memory system such as cache size, cache line size, associativity, bandwidth between main memory and cache memory, is one of the most important tasks. Incorporating our method with memory system design will achieve a great success in the embedded system design. Optimization of memory system organization is one of our future works.

Table 4: Comparison of code size

	GNU grep 2.0	GNU sed 2.05
Default	68.2K bytes	60.2K bytes
Ours	85.5K bytes	78.0K bytes

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addition-Wesley, 1986.

[2] J. A. Fisher. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Trans. Computers*, C–30(7):478–490, July 1981.

[3] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[4] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man. "An Efficient Microcode Compiler for Application Specific DSP Processors". *IEEE Trans. CAD/ICAS*, 9(9):925–937, September 1990.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[6] W. W. Hwu and P. P. Chang. "Achieving High Instruction Cache Performance with an Optimizing Compiler". In *Proc. of 16th Int'l Symp. on Computer Architecture*, pages 242–251, 1989.

[7] C. Liem, T. May, and P. Paulin. "Instruction-Set Matching and Selection for DSP and ASIP Code Generation". In *Proc. of ED&TC94*, pages 31–37, 1994.

[8] S. McFarling. "Program Optimization for Instruction Caches". In *Proc. of 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, 1989.

[9] S. McFarling. "Procedure Merging with Instruction Caches". In *Proc. of Programming Language Design and Implementation*, pages 71–79, 1991.

[10] SPARC International, Inc. *The SPARC Architecture Manual Version 8*, 1992.

[11] V. Tiwari, S. Malik, and A. Wolfe. "Power Analysis of Embedded Software: A First Step towards Software Power Minimization". In *Proc. of ICCAD-94*, pages 384–390, 1994.