Size-Constrained Code Placement for Cache Miss Rate Reduction [†]

Hiroyuki Tomiyama [‡] Hiroto Yasuura
Department of Computer Science and Communication Engineering
Graduate School of Information Science and Electrical Engineering
Kyushu University
6–1 Kasuga-koen, Kasuga, Fukuoka 816 Japan

Abstract

In design of an embedded system with a cache, it is important to minimize the cache miss rate to reduce the power consumption as well as to improve the performance of the system. We have previously proposed a code placement method which minimizes miss rates of instruction caches [10], but it makes code size larger. In most cases, code size is a tight design constraint. In this paper, we propose a size-constrained code placement method which minimizes cache miss rates under constraint on code size given by system designers. Experimental results show that the size-constrained code placement method achieves 36% decrease in cache misses with only 1.6% increase in code size compared with a naive placement, while the previous method proposed in [10] decreases 36% of cache misses with 25% increase in code size.

1. Introduction

Many recent mid- to high-range embedded systems such as communication and multimedia systems consist of RISC processors with an on-chip cache. In design of an embedded system with a cache, it is important to minimize the cache miss rate in terms of maximizing the system performance and minimizing power consumption. Cache misses make the execution speed of the program slow because they require extra-cycles to transfer code or data between the cache and the main memory. Furthermore, cache misses consume much power not only because the main memory is activated but also because off-chip busses are driven.

We have proposed a code placement method to minimize miss rates of instruction caches [10]. We have defined the code placement problem and formulated it as an integer linear programming (ILP) problem. An optimal code placement can be obtained by solving the ILP problem. Our experiments show that our method achieves about 35% (max 45%) decrease in cache misses.

Although the method proposed in [10] achieves great decrease in cache misses, we have two problems. One problem is increase in code size. Our experiments show that code size after optimization is 25–29% larger than before optimization. Increase in code size is a serious problem in embedded system design because larger main memory is required to store the code. In most cases of embedded software design, code size is the one of the most critical design constraints. Another problem is high computation cost for the optimization. Our method requires long computation time to find an optimal or quasi-optimal solution for the ILP problem. In embedded software design, the quality of object code is much more important than compilation time because embedded software is rarely modified after it is stored in ROM. However, this does not mean that very long compilation time for the optimization is acceptable. Quick estimation of the system performance is very important to optimize the whole system, but long compilation time makes it difficult.

In this paper, we propose a technique, called *trace merging*, which saves both increase in code size and compilation time. Trace merging is used to minimize the cache miss rates under constraint on code size given by system designers. The size-constrained code placement can improve the system performance and the power consumption without additional hardware.

The organization of this paper is as follows: Section 2 summarizes our code placement method previously proposed, and Section 3 proposes a size-constrained code placement method. Experiments are shown in Section 4.

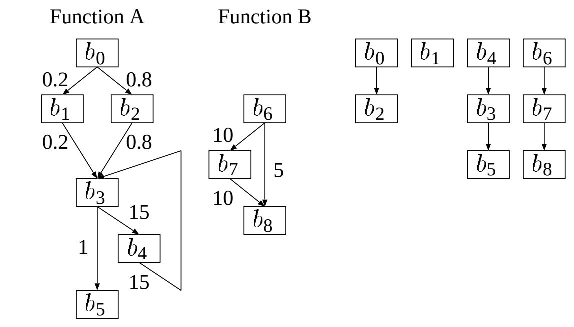
2. Previous Work

2.1. Overview

In this section, we summarize our code placement method proposed in [10].

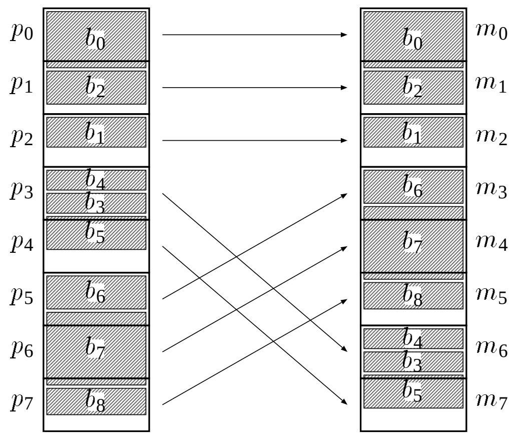
[†] This work has been partially supported by Grant in Aid for Scientific Research of the Ministry of Education, Science and Culture of Japan (No.9132).

[‡] A Research Fellow of the Japan Society for the Promotion of Science.



(a) Weighted control flow graph (b) Traces

Figure 1. Weighted control flow graph and traces



(a) Pseudo-memory (b) Main memory

Figure 2. Trace placement

We assume that the target system has a Harvard architecture whose instruction cache and data cache are separated each other physically as well as logically. We also assume that the associativity of the instruction cache is direct mapped or set associative with the least recently used (LRU) algorithm for replacement.

The code placement method uses profile information on the programs to be placed in a main memory. The profile information is one or more sequence(s) of basic blocks which are accessed when typical input data is given to the programs. The code placement method consists of two techniques, *trace selection* and *trace placement*. Trace selection is not new idea. It was proposed in [4]. After trace selection, trace placement is applied, which is the key of the method.

2.2. Trace Selection

For given assembly code of the programs and the profile information, a weighted control flow graph is constructed firstly. Figure 1 (a) shows a weighted control flow graph of a program with two functions, A and B. Each node in the graph, denoted by b_i, 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 frequency of edge traverse when the program is executed once, which is calculated from the profile information.

In Figure 1 (a), it is more probable that 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 Figure 1 (b). Each path is called a *trace* [1]

[1]The term *trace* was introduced by Fisher in [2] which addresses global microcode compaction.

Note that the last instruction of each trace is an unconditional jump operation or an exit of the function. This property enables us to place traces in arbitrary order.

2.3. Trace Placement

In [10], we have introduced a term *pseudo-memory* which means an imaginary main memory. We call a block in a pseudo-memory which is mapped onto a cache line a *pseudo-memory block*. Similarly, we call a block in the real main memory a *memory block*.

First, traces are placed in the pseudo-memory in arbitrary order, but different traces must be placed in different pseudo-memory blocks. In other words, each trace is placed at the top of a pseudo-memory block. This restruction requires a lot of fragments in the main memory where no instruction is placed. The fragments are unusable because they are very small. As a result, the code size increases.

We show an example of trace placement in a pseudo-memory in Figure 2 (a), where p_i's denote pseudo-memory blocks. Then, trace placement can be considered as a matching problem between pseudo-memory blocks and memory blocks which minimizes the cache miss count. Arrows in Figure 2 between pseudo-memory blocks p_i's and memory blocks m_i's show an example of a matching.

We have formulated the trace placement problem to minimize cache miss count as an ILP problem in [10]. Let x_i denote the index number of the memory block where p_i matches. The objective function is expressed by a function of x_i's which can be linearized. The constraints are expressed by linear equations of x_i's. In this paper, due to the limited space, we only explain the key of the ILP formulation.

After placing trace in the pseudo-memory, a sequence of basic blocks to be executed, which is given as profile information, is translated into a sequence of pseudo-memory blocks to be accessed. The sequence is called the *access sequence of pseudo-memory blocks* or simply *access sequence*. For example, let us use the program illustrated in Figure 1 and Figure 2 and assume that function B is called at the end of basic block b_4 in function A. The profile information on the program may contain the following sequence of basic blocks.

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

According to Figure 2 (a), the above sequence is translated into the following access sequence.

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

In the access sequence, the three pseudo-memory blocks p_5, p_6, and p_7 appear between the first and the second appearance of p_3. If p_3 is displaced by the three pseudo-memory blocks, a cache miss occurs at the second appearance of p_3. Let N_{set} and N_{way} denote the number of cache sets and cache ways, respectively. If $x_5 = x_3 \ (mod\ N_{set})$, p_5 is mapped onto the same cache set as p_3. If there are more than or equal to N_{way} pseudo-memory blocks p_i's in $\{p_5, p_6, p_7\}$ satisfying $x_i = x_3 \ (mod\ N_{set})$, p_3 is displaced from the cache. Hence, x_i's should satisfy the following inequality:

$$| \ \{ \ p_i \mid p_i \in \{p_5, p_6, p_7\} \text{ and } x_i = x_3 \ (mod\ N_{set}) \ \} \ | < N_{way} \qquad \text{(Eq.1)}$$

When p_i appears n_i times in the access sequence, there exist $\sum_i (n_i - 1)$ inequalities which x_i's should satisfy, such as (Eq.1). The objective of the ILP problem is to find x_i's satisfying as many inequalities as possible.

There are two kinds of constraints in the formulation.

1. All the pseudo-memory blocks must have matches.
2. If p_i and p_{i+1} have the same trace in themselves, $x_i = x_{i+1} - 1$.

Without the second constraint, one trace may be placed separately in the main memory.

For given programs and the profile information, an optimal placement can be obtained by solving the ILP problem.

2.4. Experimental Results and Problems

Experiments using two benchmark programs, GNU grep 2.0 and GNU sed 2.05, show that our code placement method achieves 35% (max 45%) decrease in cache misses compared with a naive placement [10]. The cache miss rate after code placement optimization is lower than the miss rate with the double size cache without optimization. In the experiments, a greedy algorithm is used for trace selection, and a local search algorithm is used for trace placement.

Experimental results also show that our method has two problems. One problem is increase in code size. Experiments show that code size after optimization is 25–29% larger than before optimization. Increase in code size is a serious problem in embedded system design. If the code size becomes larger than the main memory size, there exists no solution for the ILP problem and the designers must give up optimization of code placement. Alternatively, the designers decide to use larger main memory. In any case, the cost/performance of the designed system becomes worse.

The other problem is high computation cost for the optimization. Our implementation for trace placement requires 3–6 hours for GNU grep 2.0 and 10–30 hours for GNU sed 2.05 to output a locally optimal solution on SPARC Station 5. In embedded software design, the quality of object code is much more important than the compilation time because embedded software is rarely modified after stored in ROM. However, this does not mean that very long compilation time is acceptable. Quick estimation of the system performance is very important to optimize the whole system, but long compilation time makes it difficult.

The code placement method proposed in [10] requires a new technique to save both increase in code size and compilation time.

3. Size-Constrained Code Placement

In this section, we propose a technique, called *trace merging*, which saves increase in code size and compilation time. Next, we propose a size-constrained code placement method which minimizes cache miss rates under given constraint on code size.

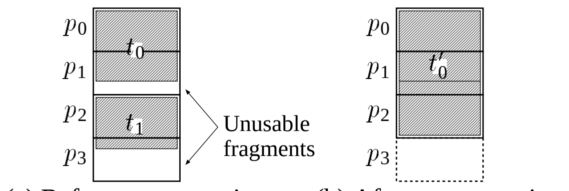
3.1. Trace Merging

As explained in Section 2.3, the code placement method proposed in [10] requires a lot of unusable fragments in a main memory, and makes code size larger. Let S_{line} and s_i denote the size of a cache line and the size of the *i*th trace, respectively. The total size of the unusable fragments S_{unused} is expressed by the following equations:

$$S_{unused} = \sum_i unused(s_i) \qquad \text{(Eq.2)}$$

$$unused(s_i) = \begin{cases} 0 & \text{if } res(s_i, S_{line}) = 0 \\ S_{line} - res(s_i, S_{line}) & \text{otherwise} \end{cases} \qquad \text{(Eq.3)}$$

where $res(x, y)$ is defined as the residue of x divided by y, namely $res(x, y) = z$ such that $0 \le z < y$ and $x = z \ (mod\ y)$. (Eq.2) and (Eq.3) suggest two approaches to reducing increase in code size.



(a) Before trace merging (b) After trace merging

Figure 3. An example of trace merging

1. To make the number of traces small. A half number of traces will result in a half increase in code size.
2. To make $res(s_i, S_{line})$ be zero or close to S_{line}. If the size of each trace is a multiple of the cache line size, code size does not increase.

The above approaches are realized by the following technique, called *trace merging*:

After trace selection, to merge a couple of traces such that the size of the newly created trace becomes a multiple of the cache line size.

An example of trace merging is illustrated in Figure 3. The two traces t_0 and t_1 are merged into a new trace t'_0. As a result, the number of memory blocks required by the program are reduced from 4 to 3.

As traces decreases, the cache miss rate may increase. This is because, in the ILP formulation, there is a constraint ensuring that a sequence of pseudo-memory blocks where a trace is placed must be placed consecutively (See Section 2.3). If two traces are merged into one, one more constraint equation for the ILP problem is derived. In usual, as constraints for an ILP problem increase, the quality of a solution for the problem becomes worse. Hence, we reach the following policy for trace merging:

There should be more traces in the program. In other words, the number of traces which are merged should be small.

3.2. Size-Constrained Code Placement Algorithm

Based on the discussions in Section 3.1, we propose an algorithm for the size-constrained code placement. The algorithm is shown in Figure 4. We use the following notations in the algorithm:

N_{mem}	The maximum number of memory blocks for the programs. N_{mem} is the constraint on code size given by system designers.
S_{inst}	The instruction length of the CPU. If the CPU has the variable instruction length, S_{inst} denotes the smallest one.
S_{line}	The cache line size.
N_{set}	The number of cache sets.
N_{way}	The number of cache ways.
T	A set of traces before trace merging.
T_{merged}	A set of traces after trace merging.
AP	Assembly code of programs to be placed.
PI	Profile information.

Function $Size_{with}(T)$ returns the total size of traces in T including the unusable fragments, and $Size_{without}(T)$ returns the total size of traces in T excluding the unusable fragments.

Here, we summarize the size-constrained code placement algorithm.

(1) Perform trace selection, and set T to a set of traces.

(2) Set T_{merged} to ϕ, move all traces with a multiple size of the cache line in T to T_{merged}, and set n to 2.

(3) If $Size_{with}(T \cup T_{merged}) \leq N_{mem}$, goto step (7).

(4) If there exists $T' \subseteq T$ such that $|T'| = n$ and that $Size_{without}(T')$ is a multiple of S_{line}, goto step (6).

(5) Increment n. If $n > S_{line}/S_{inst}$, merge all traces in T into a new trace t', join t' into T_{merged}, and goto step (8). Otherwise, goto step (3).

(6) Merge traces in T' and create a new trace t', remove all traces in T' from T, and join t' into T_{merged}, and goto step (3).

(7) Move all traces in T into T_{merged}.

(8) Perform trace placement.

One of the merits of the size-constraint code placement algorithm is that it can save not only the code size but also the compilation time. In the algorithm, the trace placement spends most of the total compilation time to solve the ILP problem. Since traces are decreased by trace merging, constraint equations in the ILP problem increase. As constraints in the ILP problem increase, the computation time becomes shorter. There is no need to solve the ILP problem more than once because the constraint on code size is satisfied before trace placement. Hence, the compilation time is saved.

4. Experiments

We have applied the size-constrained code placement method to a benchmark program. We vary the constraint on code size, and calculate cache miss counts, miss rates, and compilation time, for various cache organization. The constraint on code size is given in terms of the number of memory blocks for the program, denoted by N_{mem} in the previous section. The benchmark program is GNU grep

Table 1. Miss rate and computation time varying associativity

(A) 1-way set associative cache (direct mapped)

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	584	3.68%	17,382 sec	1,067
(b) 2,271 (72.7KB)	590	3.72%	7,121 sec	665
(c) 2,166 (69.3KB)	592	3.73%	6,159 sec	554
(d) — (68.2KB)	1,071	6.80%	— sec	—

(B) 2-way set associative cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	636	4.01%	11,923 sec	1,067
(b) 2,271 (72.7KB)	617	3.89%	4,992 sec	665
(c) 2,166 (69.3KB)	616	3.88%	5,040 sec	554
(d) — (68.2KB)	1,025	6.51%	— sec	—

(C) 4-way set associative cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	631	3.98%	20,187 sec	1,067
(b) 2,271 (72.7KB)	638	4.02%	5,216 sec	665
(c) 2,166 (69.3KB)	649	4.09%	3,806 sec	554
(d) — (68.2KB)	1,039	6.60%	— sec	—

(D) 8-way set associative cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	696	4.39%	22,612 sec	1,067
(b) 2,271 (72.7KB)	693	4.37%	3,197 sec	665
(c) 2,166 (69.3KB)	687	4.33%	6,101 sec	554
(d) — (68.2KB)	1,032	6.55%	— sec	—

Table 2. Miss rate and computation time varying cache size

(A) 256 byte cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	1,198	7.55%	5,522 sec	1,067
(b) 2,271 (72.7KB)	1,203	7.58%	3,031 sec	665
(c) 2,166 (69.3KB)	1,206	7.60%	2,118 sec	554
(d) — (68.2KB)	1,701	10.81%	— sec	—

(B) 512 byte cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	865	5.45%	13,078 sec	1,067
(b) 2,271 (72.7KB)	880	5.55%	5,402 sec	665
(c) 2,166 (69.3KB)	890	5.61%	4,611 sec	554
(d) — (68.2KB)	1,355	8.61%	— sec	—

(C) 1,024 byte cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	584	3.68%	17,382 sec	1,067
(b) 2,271 (72.7KB)	590	3.72%	7,121 sec	665
(c) 2,166 (69.3KB)	592	3.73%	6,159 sec	554
(d) — (68.2KB)	1,071	6.80%	— sec	—

(D) 2,048 byte cache

N_{mem}	Misses	Rate	CPU time	Traces
(a) 2,673 (85.5KB)	438	2.76%	14,414 sec	1,067
(b) 2,271 (72.7KB)	436	2.75%	7,994 sec	665
(c) 2,166 (69.3KB)	432	2.72%	5,627 sec	554
(d) — (68.2KB)	693	4.40%	— sec	—

2.0[2], and the target architecture is the SPARC architecture. The experiments are performed on SPARCstation 5 (microSPARC-II, 85MHz, 32MB, Solaris 2.4).

First, we assume a 1K byte cache with 32 byte cache lines, and vary the constraint on code size and the cache associativity. Experimental results are shown in Table 1. For each associativity, N_{mem} is set to 2,673, 2,271, and 2,166. When case (a), trace merging is not applied. This is the case the previous code placement method proposed in [10] is applied. When case (b), the range of variable n in the size-constrained code placement algorithm is $1 \leq n \leq 2$. In other words, only two traces are merged into one at a time. When case (c), code size after optimization is minimum. The case (d) shows the results of a naive placement. This is the case no code placement optimization is applied. A number in parentheses in the first column gives the code size in terms of kilo bytes. The forth column in each table shows the CPU time required to find a locally optimal solution for the ILP problem in trace placement.

Next, we assume a direct mapped cache with 32 byte cache lines, and vary the constraint on code size and the cache size. Experimental results are shown in Table 2.

In Table 1 and Table 2, there is no significant difference on cache miss count caused by the constraint on code size. Under any constraint on code size, the size-constrained code placement achieves about 36% decrease in cache misses on average, compared with the naive placement. Table 2 shows that the cache miss rate after code placement optimization is lower than the miss rate with the double size cache without optimization. On the other hand, as constraint on code size is tighter, the CPU time becomes shorter because traces are decreased.

Our experimental results show that trace merging can reduce both code size and computation time keeping the miss rate low. Further experiments using various embedded software are required to validate our method.

When trace merging is not applied, code size is increased by 25% compared with the naive placement. Even when N_{mem} is the smallest, the size-constrained placement makes code size 1.6% larger. In some cases, trace selection makes code size larger. Let us consider the program shown in Figure 5 (a). The naive placement places basic blocks in the main memory in the order shown in Figure 5 (b). In this case, no jump operation is required. But our method select traces as shown in Figure 5 (c). In this case, a jump operation is required at the exit of b_1. If only a little increase in code size is not permitted, we should select traces such

[2]The program is written in C with 12,436 lines including comments.

that the number of traces is minimized, and apply trace merging and trace placement.

5. Conclusion

In this paper, we have proposed a size-constrained code placement method which minimizes miss rates of instruction caches under constraint on code size given by system designers. Experimental results show that our method achieves 36% decrease in cache misses with only 1.6% increase in code size. Our method can improve both the system performance and the power consumption without additional hardware cost.

In our experiments, there is no significant trade-off between the code size and the cache miss rate. In future, we will continue experiments using various embedded software.

Many embedded systems are real-time systems. In design of real-time systems, it is important to minimize the worst case execution time of programs. Our current code placement method takes no account of the worst case execution time. Development of a code placement method for real-time systems is also one of our future works.

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] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2nd edition, 1996.

[4] 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.

[5] Y.-T. S. Li, S. Malik, and A. Wolfe. “Performance Estimation of Embedded Software with Instruction Cache Modeling”. In *Proc. of ICCAD-95*, pages 380–387, 1995.

[6] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[7] 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.

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

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

[10] H. Tomiyama and H. Yasuura. “Optimal Code Placement of Embedded Software for Instruction Caches”. In *Proc. of ED&TC96*, pages 96–101, 1996.

[11] A. Wolfe. “Software-Based Cache Partitioning for Real-Time Applications”. *Journal of Computer & Software Engineering*, 1(3):315–327, 1994.

```
/* Size constrained code placement */
$SCCP(AP, PI, N_{mem}, S_{inst}, S_{line}, N_{set}, N_{way})$
{
  $T = TraceSelection(AP, PI)$;
  $T_{merged} = SCTM(N_{mem}, S_{inst}, S_{line}, T)$;
  $TracePlacement(AP, PI, T_{merged}, N_{mem}, N_{set},$
     $N_{way})$;
}

/* Size constrained trace merging */
subroutine $SCTM(N_{mem}, S_{inst}, S_{line}, T)$
{
  $T_{merged} = \phi$;  $n = 1$;
  while ($Size_{with}(T \cup T_{marged}) \leq N_{mem}$) {
    $T' = Select(n, T, S_{line})$;
    if ($T' = \phi$) {
      $n = n + 1$;
      if ($n > (S_{line}/S_{inst})$) {
        Merge all traces in $T$ and create a new trace $t'$;
        $T_{merged} = T_{merged} \cup \{t'\}$;
        return ($T_{merged}$);
      }
    } else {
      Merge all traces in $T'$ and create a new trace $t'$;
      $T = T - T'$;  $T_{merged} = T_{merged} \cup \{t'\}$;
    }
  }
  $T_{merged} = T_{merged} \cup T$;
  return ($T_{merged}$);
}

subroutine $Select(n, T, S_{line})$
{
  $T'$ = a subset of $T$ such that $|T'| = n$
    and that $Size_{without}(T')$ is a multiple of $S_{line}$;
  return($T'$);
}
```

Figure 4. Size-constrained code placement algorithm

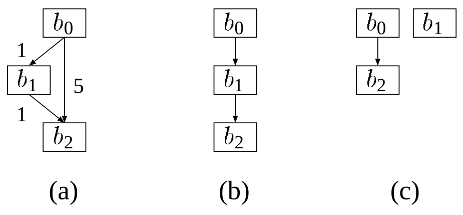


Figure 5. An example of trace selection which makes code size larger