A FAST PROCEDURE PLACEMENT ALGORITHM FOR OPTIMAL CACHE USE

Saverio Lorenzini Gabriele Luculli Cosimo Antonio Prete
Dipartimento di Ingegneria dell'Informazione, Università di Pisa
Via Diotisalvi, 2 - 56126 Pisa - Italy
saverio@aloha.iet.unipi.it luculli@sssup.it prete@iet.unipi.it *

ABSTRACT

This paper presents a procedure placement method for embedded applications. We use the trace-driven simulation to collect information on the use of the cache line and then a heuristic algorithm to perform the placement. The main features of our method are a short computation time and a strong reduction of miss ratio. Experimental results shows an average miss rate reduction of 32%, but better improvements are obtained depending on the specific application.

1. INTRODUCTION

The design of embedded systems [3] implies a trade–off solution to meet conflicting requirements: low cost, effective performance through small die size, low power consumption for some applications (e.g. wireless and portable products), high processing power and/or large memories for other ones (e.g. multimedia applications). The use of embedded processing components in electronic products with a wide consumer market gives additional crucial requirements such as the cost and the time–to–market.

The cache memory [11] is an architectural element often used to provide both the high processing power and the large main memory required by some applications. Cache memory keeps low the off–chip memory bandwidth. This allows the use of slower off-chip memories, lower cost and lower power consumption. But, a cache memory does not furnish processing services to an application and its presence and structure influence the cost and the power consumption. The tuning and optimization of such systems for embedded applications turn out to be a complex and crucial task [9]. Further, the size of on–chip cache is typically small with respect to the working set of the program, so that a large number of conflict misses [11] are expected. Conflict misses depend on both the software characteristics and the cache memory structure. An attempt towards an integrated view of the problem is to study how code allocation can be made for reducing the miss rate and for obtaining an optimal use of the cache.

The code allocation problem has been extensively studied. Researchers from the compiler community have proposed various approaches to improve the memory hierarchy performance: procedure placement, basic block reordering and function inline expansion are just few of the suggested techniques. These are commonly based on greedy algorithm, employing profile information from a specific data set, which fulfills the restructuring of the code at compile-time. Unfortunately, these optimizations are biased toward general-purpose code that means they assume no kind of run-time behavior of the program. In embedded systems the functionality does not change for the product lifetime and only a reduced set of different program behaviors is present. Exploiting such characteristics of embedded applications can result in more effective solutions and stronger optimizations, especially if a global tuning among cache design parameters and software code is tried.

In this paper we present some preliminary results on a fast method for procedure placement which can be efficiently used for optimal caching by embedded applications. At first we collect information about the access distribution for each cache line, then the number of conflict misses is estimated by a heuristic metric and finally, after the procedure placement, the actual cache performance is evaluated. The evaluation and data collecting phases are both based on trace-driven simulation. As we know, our method is the first approach that proposes a solution based on trace-driven simulation of the instruction cache, rather than on other profiling information. This is a main feature which allows an accurate evaluation of the true code behavior, instead of the average case that is commonly proposed, and it enables us to completely account for the cache effects. The high speed of the method can result in an extensive evaluation of several cache configurations.

2. RELATED WORKS

In the past the code placement problem has been extensively studied in order to reduce the execution time of general-purpose code in cache–based architectures [6, 8] or to improve the memory system performance [4]. Recently the attention has turned toward the optimization of embedded code [12]. The proposed methods use different allocation units and different algorithms to search for an optimal placement solution. All the algorithms make use of some description of the code structure but they consider just a very loose model of the cache. The cache size is often the only parameter that is accounted for [6, 13], so the final code layout is completely independent from the

*The Ministry of University and Scientific and Technological Research (MURST), Italy, and the University of Pisa, Italy, supported this work.

0-7803-3879-0 / 98 / $10.00

other cache parameters e.g. the replacement policy and the number of ways. It turns out that several proposed algorithm are not useful for embedded systems where strong HW/SW tuning and global optimization of the system are not an issue but a need.

At the finest-grain level, the allocation unit is the single basic block [1] (i.e. a sequence of instructions ended by a branch or return instruction). The goal is to find out a basic block ordering such that a sequence of basic block with better locality proprieties is obtained. The improvement of code locality always leads to a reduced number of cache conflicts. Such basic block reordering is usually performed within the procedure boundaries and results on a very effective reduction of misses, especially if the procedure completely maps inside the instruction cache. Instead at a coarse-grain level the allocation unit is the procedure. A single procedure is often called by several points inside the code, so the actual cache state at the procedure's call time depends on the specific calling point and it is completely different for each case. Usually it is not possible to find a procedure allocation that results in an improvement respect to all the call points; so the placement algorithm purpose is to reduce the global interference effect among procedures rather than improving the code locality. There are also some mixed approaches, which use the so-called trace as the allocation unit. A *trace* is a chain of basic blocks, which tends to execute in sequence, and each procedure is partitioned in a set of disjoint traces. In this context, the trace selection is related to the improvement of code locality and the trace placement to the reduction of cache conflicts.

From an algorithmic point of view almost all the proposed methods are based on greedy approaches [14, 4, 7, 10] whether just a single result is available using an integer linear programming formulation [12]. Code positioning is just one of a group of techniques that can be used to optimize the cache behavior at the software level. Procedure splitting [8], procedure merging [7] and function inline expansion [5, 2] are possible known alternatives, but useful during the code generation phase only. In the following paragraphs we give a short description of some research results.

McFarling [6] proposes an algorithm at the basic block level based on the idea of not caching infrequently used instructions. The algorithm constructs a control flow graph of the program, then tries to partition it in disjoint trees. If the height of each partitioned tree is less than the cache size, all nodes inside the tree can be directly mapped since they will not interfere with each other in the cache. The partitioning is guided by profile information (e.g. basic block count) on the control flow graph, in this way the most frequently used instructions are guaranteed not to interfere in cache, but that is not true for the other ones.

Pettis and Hansen [8] describe several techniques for improving code layout, starting from basic block reordering, through procedure splitting and up to procedure reordering. The main idea is the partitioning of the basic blocks of each procedure in two disjoint sets, the frequently executed and the infrequently executed. Such selection can be obtained by a top-down or bottom-up algorithm, nevertheless the most executed basic blocks are chained in a single trace and the infrequently executed one in another trace. All the second traces are grouped together in a single memory region; instead, each first trace remains as body of a procedure. In this way it is accomplished the procedure splitting. The final step is reordering all the procedures using a "closest is best" strategy on the call graph. They found that the reduction in execution time of the application when using procedure reordering is about 8%.

Hashemi, Kaeli and Calder [4] work out a suboptimal solution to the problem of procedure placement by determining a special ordering which guarantees no cache conflict misses between any two call–called procedures. They start with an annotated call graph of the program and attempt to *color* each node such that any two connected nodes does not share the same color set. The result is an exact solution, but in general the method is too pessimistic. They assume that the cache conflicts between call–called procedures is the major effect on the conflict miss rate, but that is not often true. The annotated call graph cannot completely account for the temporal locality of the code, but just for the effect of direct call between procedures. In particular it may happen that two nodes in the call graph, which share the same father node, are alternatively activated inside the father code. So cache conflict can arise between two such nodes and this could be a dominant effect on the miss rate, but their method is not able to account for such important effect. They report an average reduction of the cache miss rate by 45% over the original mapping and by 14% over the mapping algorithm of Pettis and Hansen.

Tomiyama and Yasuura [12] formulate the code placement problem as an integer linear programming problem. At first they use profile information to partition a set of weighted control flow graphs in disjoint traces by a greedy algorithm, and then they solve the trace placement problem by an ILP formulation. They assume to have a direct-mapped or set-associative instruction cache with LRU replacement, and the final solution depends directly on such assumption. Undoubtedly, this is an improvement respect to the other methods; however, their method requires a long computation time and it is infeasible to obtain optimal solutions for large programs in a practical time [12]. They achieve an average 35% reduction of cache miss for two benchmark programs.

3. OUR APPROACH

Programs are usually structured as a set of procedures; each procedure can be invoked from several call points and this results in an effective computation sharing of procedures. The instruction cache is shared among all the procedures and the main program. Typically the active working set of each procedure fills just a reduced set of cache lines at any time. Two or more procedures can be partially mapped to the same cache line, so cache misses can result from the alternative use of the same cache line. The procedure placement problem is to find a mapping for

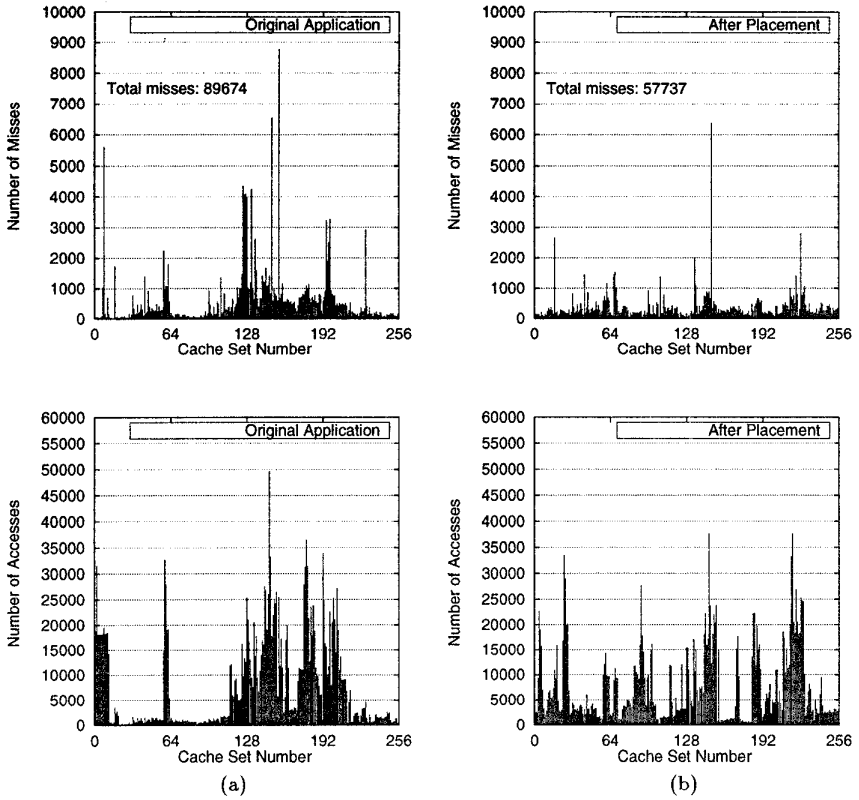

Figure 1. Misses and references distributions of the original application and after the procedure placement. The miss rate changes from 4.5% to 2.8% (4K–byte direct-mapped cache, 16–byte line size, cjpeg application).

each procedure in order to obtain reduced effects of interference in the cache.

The number of misses generated at run–time depends on both the characteristics of spatial–temporal locality of each procedure and the exact sequence of the calls to them. The execution of the same procedure several times, in a short interval of time, is another kind of code temporal locality that can be efficiently exploited by a cache. A subsequent call to the same procedure can experience a small number of cache misses if useful cache lines are kept in cache from the previous execution. Unfortunately conflicts can arise among subsequent procedures sharing some cache lines and useful lines can be swapped out the cache. Such interference strongly depends on the procedure mapping to the memory, so a good procedure placement should result in a reduction of interferences in cache.

Our code placement method is based on two phases. *At first we try to reduce the misses generated by each single procedure.* We define the *program line* as a code portion that is exactly mapped to a single cache line. A procedure consists of a set of program lines, which have a one-to-one mapping to the cache lines. At run-time it may happen that not all instructions inside a program line are executed, because of a branch before the end of the program line. Nevertheless the execution of such program line will result in a cache miss if it is not in cache. So we could pay a full miss penalty to execute a small number of instructions in such a program line. In the first phase of our method we shift the starting address of each procedure by an offset smaller than the cache line size, in order to minimize the number of program lines that could be possibly executed. The best offset is found by evaluating all possible offsets, and choosing the one that results in the minimum number of executed program lines. At the end of this first phase, all the procedures of the application could be no more contiguous in main memory rather there are some holes without useful code between them.

In the second phase we perform the procedure placement. The main idea of our algorithm is to balance the load (i.e. the number of references to) of each cache line, so that the instruction references spread all over the cache rather than over a reduced set of cache lines. The reduction in the number of misses results from the reduction of the number of conflicts caused by the overlapping of the procedure code in the cache. In general it is not possible to remove all the overlapped regions but this is not useful neither hope-

Table 1. Details of our benchmark applications.

Application Description	Size	Trace ref.
BFIR: FIR filter with 120 taps	29K bytes	1.6 Mil.
PCM: Encoder for pulse code mod.	17K bytes	1.1 Mil.
DPCM: Decoder pcm	17K bytes	1.5 Mil.
CJPEG: Picture compressor	161K bytes	1.4 Mil.

ful, rather we need to reduce the overlapping in the shared cache lines which have an high number of conflict misses. In order to make clear the main idea Figure 1 reports an example. It shows the number of references and misses for an original application (a) and after the procedure placement (b). The improvement is proved by the loss of several spikes after the procedure placement and the reduction of the overall miss number, as well as a clear balance of the load is shown in the distribution of cache references. In the following paragraphs we give more details on the second phase of our algorithm. The input data are the execution trace, which is obtained by the trace driven simulation, the range of the memory region that is addressed by each procedure (i.e. the starting and ending addresses of each procedure in main memory) and the cache parameters.

At the start we compute the miss distribution on the cache lines and the number of conflict misses suffered–produced by the procedures. Then we position in main memory the procedure with the greatest quantity of conflicts. The exact placement of this procedure does not matter since it is the first one, so it cannot conflict with any previous allocated procedures. All the other procedures are placed according to a heuristic metric of the contentions in the shared cache lines between the actual procedure and the already positioned procedures. The metric is a simple cost function which tries to evaluate the cache interference effect as the conflict miss overhead due to the actual procedure. It is a heuristic metric because it cannot account for the exact number of conflicts that result from the mapping of the actual procedure. Such exact evaluation is possible but could be too time-consuming, therefore not feasible for practical approach. Nevertheless the experiments show that our metric is useful to choose a good procedure placement, almost every procedure takes advantage of the new mapping in terms of a reduce number of conflict misses.

4. RESULTS ANALYSIS

In this section, we propose some preliminary results on the improvement of the cache miss rate after the procedure placement. Our target architecture is based on the embedded processors of the ARM family. The ARM processors are 32–bit RISC architectures with on-chip unified cache and they are typically used in embedded products. To evaluate the performance of the instruction cache we use an advanced version of ChARM [9] trace–driven cache simulator. We have considered four embedded applications (see Table 1) typically used in the areas of speech/image processing and some different input patterns for some of them.

Table 2. Instruction cache miss rates (in percentages) both of default layout (Def.) and after the procedure placement (Map.). A 4K–byte cache size with various set–associativity is evaluated.

Application	1-Way		2-Way		4-Way	
	Def.	Map.	Def.	Map.	Def.	Map.
BFIR1	4.40	2.54	3.64	1.82	2.73	2.02
BFIR2	4.66	2.30	2.91	1.13	1.59	0.35
BFIR3	4.51	2.04	3.51	2.20	1.69	1.27
PCM	3.18	0.31	0.38	0.27	0.29	0.27
DPCM	0.76	0.34	0.30	0.28	0.29	0.27
CJPEG	4.48	2.89	2.39	2.25	2.18	2.17

First, we assume a 4K–byte cache with 16–byte cache lines, and we vary the set–associativity. Experimental results are shown in Table 2 and partially reported in Figure 2 with an extended number of ways up to 64 (pictures on the first row). Overall, there is a strong reduction of the miss rate. The filtering application is more sensitive to the procedure placement than the other ones, but in any case there is an average cache misses reduction of about 32% (max 78% for BFIR, max 90% for the other applications). Next, we assume a direct–mapped cache with 16–byte line, and we vary the cache size. Just few results are reported in Figure 2 (second row) because of limitations in space. Note the Figure 2(c), after the procedure placement it is possible to use a 1K–byte direct–mapped cache and to obtain a better performance of a 4K–byte cache without procedure placement. Designers can fruitful take advantage of such effect and to obtain better cache designs with reduced on–chip area, reduced power consumption and good cache performance.

Finally, our procedure placement algorithm is very fast, the computation time was less than one second in any reported case (for our experiments we use an HP9000 Model 725/100 workstation with 64M bytes of ram memory). The cache performance improvement cost an expansion of the code size, but usually such increasing is less than 40% of the original size and in practical case it is less than 10%.

5. CONCLUSIONS

In this paper a procedure placement method for embedded applications has been proposed. The main features of the method are short computation time, strong reduction of the miss rate and the fully usability with any kind of instruction cache. The algorithm exploits the cache by balancing the load in each cache line. These result in the reduction of the miss rate and in the optimal use of the cache.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers:Principles, Techniqeus and Tools.* Addison-Wesley Publishing Company, 1986.

[2] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu. The effect of code expanding optimizations on instruction cache design. *IEEE Transaction on Computers*, 42(9):1045–1057, September 1993.

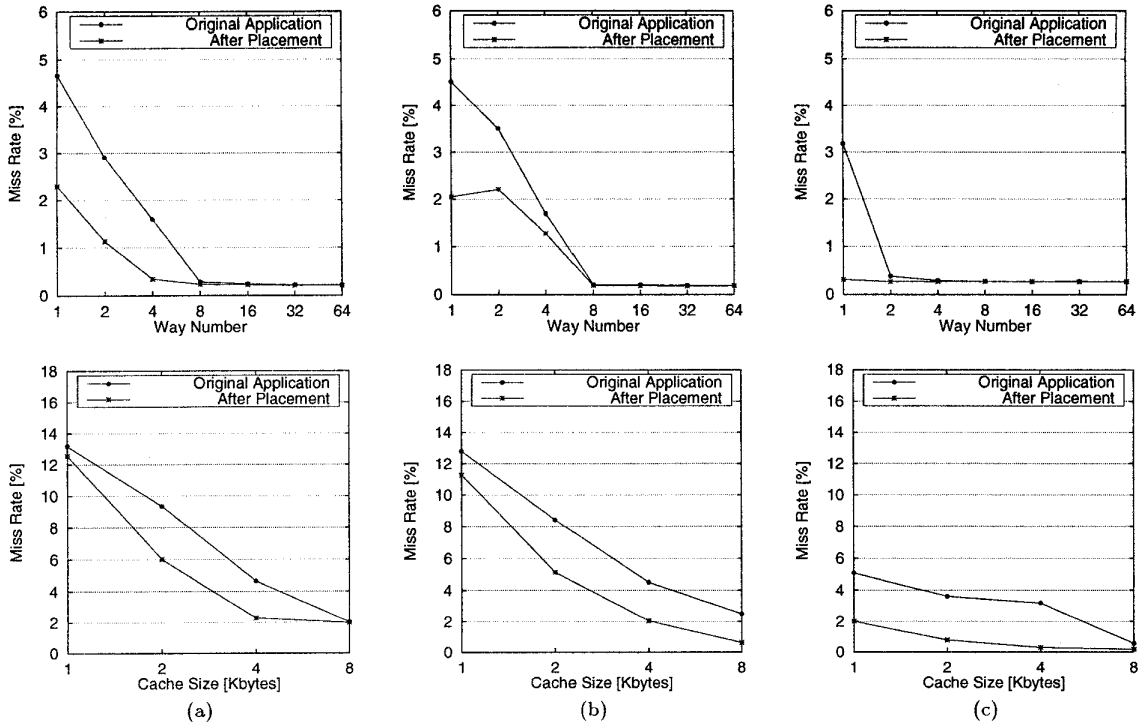

Figure 2. Miss rates for benchmarks (a) BFIR2, (b) BFIR3, (c) PCM when varying the number of ways for a 4K–byte cache or the cache size for a direct–mapped cache.

[3] D.D. Gajski and F. Vahid. Specification and design of embedded software-hardware systems. *IEEE Design Test of Computer*, 12(1), Spring 1995.

[4] Amir H. Hashemi, David R. Kelly, and Brad Calder. Efficient procedure mapping using cache line coloring. *ACM SIGPLAN Notices*, 32(5):171–182, May 1997.

[5] W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In Michael Yoeli and Gabriel Silberman, editors, *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, Jerusalem, Israel, June 1989. IEEE Computer Society Press.

[6] S. McFarling. Program optimization for instruction caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 24, pages 183–193, New York, NY, May 1989. ACM Press.

[7] Scott McFarling. Procedure merging with instruction caches. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 71–79, Toronto, Canada, June 1991.

[8] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 16–27, White Plains, NY, USA, June 1990. ACM Press.

[9] Cosimo Antonio Prete, Marco Graziano, and Francesco Lazzarini. The ChARM tool for tuning embedded systems. *IEEE Micro*, 17(4):67–75, July/ August 1997.

[10] A. D. Samples and P. N. Hilfinger. Code reorganization for instruction caches. Tech. Report UCB-CSD-88-447, Univ. of California, Berkeley, 1988.

[11] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[12] H. Tomiyama and H. Yasuura. Optimal code placement fo embedded software for instruction caches. In *Proceedings of European Design&Test Conference*, pages 96–101. IEEE Computer Society Press, 1996.

[13] Josep Torrellas, Chun Xia, and Russell Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 360–369, Raleigh, North Carolina, January 22–25, 1995. IEEE Computer Society TCCA.

[14] Chun Xia and Josep Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *Proceedings of the 23rd Annual International Symposium on Computer Architecure*, pages 271–282, New York, May22–24 1006. ACM Press.