Temporal Distribution Based Software Cache Partition To Reduce I-cache Misses

Xiaomi An Jiqiang Song Wendong Wang

SimpLight Nanoelectronics Ltd., Beijing China
100088
{xiaomi.an, jiqiang.song, wendong.wang}@simplnano.com

Abstract

As multimedia applications on mobile devices become more computationally demanding, embedded processors with one level I-cache become more prevalent, typically with a combined I-cache and SRAM of 32KB ~ 48KB total size. Code size reduction alone is no longer adequate for such applications since program sizes are much larger than the SRAM and I-cache combined. For such systems, a 3% I-cache miss rate could easily translate to more than 50% performance degradation. As such, code layout to minimize I-cache miss is essential to reduce the cycles lost.

In this paper, we propose a new code layout algorithm – temporal distribution based software cache partition with focus on multimedia code for mobile devices. This algorithm is built on top of Open64's [14] code reordering scheme. By characterizing code according to their temporal reference distribution characteristics, we partition the code and map them to logically different regions of the cache. Both capacity and conflict misses can be significantly reduced, and the cache is more effectively used. The algorithm has been implemented as a part of our tool-chain for our products.

We compare our results with previous works and show more efficacy in reducing I-cache misses with our approach, especially for applications suffering from capacity misses.

1. Observation and Motivation

As multi-core and multi-thread are being employed in embedded processors, instruction fetch efficiency is even more important to total system performance. Instruction cache performance becomes one of the most critical factors influencing the entire system design. For example, in our video codec processor, a cache miss rate of 3% will cause as much as 50% performance degradation for the H.264 encoder which is the most computationally demanding video encoding standard today.

Traditional code layout algorithms use both basic block and procedure as the unit for code positioning. Sometimes, new "procedures" are generated by procedure splitting before further positioning. In this work, we always split procedure into two or more sections as the unit for layout; called "code blocks". The record of code blocks' execution sequence at runtime is used to analyze the temporal characteristics of them for further code layout; we call the sequence "cb-trace".

By analyzing the runtime temporal characteristics of the code blocks, we observed two kinds of code blocks: code blocks which are uniformly distributed along the cb-trace and code blocks which exhibit a large skew in their distribution. For example, consider the following temporal sequence where each alphabet letter represents one code block:

$$ABCDEF(UV)^5ABCDEF(PQ)^5ABCDEF(XY)^5ABCDEF$$

In the above sequence, A,B,C,D,E,F have uniform distribution, while U,V,P,Q,X,Y do not have uniform distribution.

Code blocks exhibiting a large skew in the reference distribution have good temporal locality [13]. They are usually good candidates for traditional code layout algorithms, usually focuses on reducing I-cache conflict misses as they can be well placed to reduce this kind of misses very effectively. Consider the interleaved relationship between the pairs U and V, P and Q, X and Y; we need only two cache lines to hold the six code blocks, assuming each has the size of one cache line. We call them code blocks with good temporal locality.

On the other hand, code blocks exhibiting uniform reference distribution have little or no exploitable temporal locality. This is due to the following reasons:

1. They generally interleave pervasively with other code blocks and will cause many misses when sharing cache lines with other code blocks. E.g. we need at least six cache lines to hold A,B,C,D,E,F to avoid cache misses, since they are interleaved with all the other code blocks. To avoid cache misses, we practically have to let them hold cache lines exclusively.
2. They often have relatively long reuse distance and hence are prone to suffering from capacity miss. Because the traditional code layout algorithms are more effective on conflict misses, not much work has been done for programs with large capacity.

Patent pending

Since this kind of code blocks have temporally regular pattern, we call them code blocks with good temporal regularity.

From the above example, it can be seen that different kinds of code blocks need different layout policies. For code blocks with good temporal locality, multiple code blocks can share same cache lines and still incur no more cache misses. For code blocks with good temporal regularity, they had better hold cache lines as much as possible to avoid cache misses.

However, traditional code layout algorithms do not distinguish the difference between these two kinds of code blocks. When the cache capacity is large enough, we can do a proper layout using the traditional algorithm. Suppose we have more than eight cache lines. A traditional code layout algorithm (e.g. TRG based algorithm) can generate a placement to avoid all the conflict misses, as shown by the following:

A	B	C	D	E	F	U/P/X	V/Q/Y

However, when there is not enough cache capacity, say, only six cache lines, different layout methods will generate different number of cache misses E.g. M1 will have 24 misses and M2 have only 18 misses as shown by the following.

M1:

A/E	B/F	C	D	U/P/X	V/Q/Y

M2:

A	B	C	D	U/P/X/E	V/Q/Y/F

The key point here is to prevent code blocks with good temporal regularity from sharing cache lines among themselves as much as possible (since it incurs cache line thrashing too easily), and to let them hold cache lines exclusively or share cache lines only with code blocks with good temporal locality if needed.

For example, when A and E share the same cache line, A and E will suffer from cache misses each time they are referenced. However, when E shares a cache line with U, P and X, only references to E incur more cache misses due to the good temporal locality of U, P and X.

Based on the above observation, we devised a temporal distribution based software cache partition algorithm to do code layout. Firstly, we characterize the code blocks by temporal distribution and classify them according to good temporal locality and good temporal regularity, respectively. Secondly, we partition the cache into two regions to hold these two types of code blocks.

2. Solutions and Methodology

Like traditional code layout algorithms, our partition based algorithm is also heuristic based. Since the code placement policy depends heavily on the program characteristics, it is important to design an adaptive algorithm. Since our processor is targeted for multimedia, we focus our design and evaluation on multimedia applications only.

Our layout process includes five steps: 1) code block formation, including basic-block level (bb-level) reorder and procedure splitting optimization, 2) execution and cb-trace generation, 3) cb-trace analysis and temporal distribution calculation, 4) iterative partition of cache and code blocks, and 5) layout and placement generation. The following is the summary of our solutions:

1. Characterize the temporal distribution of a cb-trace, in terms of temporal regularity and locality. We use statistical analysis of positions in cb-traces to do this.
2. Since we want code blocks with good temporal regularity to hold cache lines exclusively, we only select the ones whose cache misses are critical for total performance. Good candidates of these code blocks should have the following characteristics:
   a) They should be hot code blocks and directly affect application performance.
   b) They should be “dense” code blocks, that is, code blocks with few branches. We use instruction density (dynamic instruction count of the code block divided by its size) to evaluate the denseness. This is beneficial to improve spatial usage of cache lines.
3. Because different code sections inside one procedure may exhibit different characteristics, hotness, and density, the various parts of a program procedure may be completely different from each other, we do bb-level reorder and procedure splitting before code layout. This improves the uniformity of the hotness and density of code blocks generated. Then, we generate distinct placement for these split code blocks. The algorithm and implementation of bb-level reorder and procedure splitting are based on Open64 and will be discussed in detail in section 3.1.
4. To make the partition applicable to different program characteristics, we developed an iterative cache partition algorithm, which makes the algorithm more flexible and easy to use.
5. Finally, the TRG algorithm (which will be explained in detail in section 3.2) is used to further place code blocks inside each partitioned cache region.

The rest of the paper is organized as the following. Section 3 reviews the related work. Section 4 gives the equations to calculate temporal distribution of code blocks and classify the code blocks. Section 5 describes the iterative partition and layout algorithm. Section 6 and 7 evaluate the code layout algorithm by some typical multimedia embedded applications, including four video and one audio program. We conclude in Section 8.

3. Related Work

Much work has been done on code layout algorithms to reduce cache misses. McFarling[2] repositioned programs so that a direct-mapped cache behaves like a full-associative cache. Since it is implemented by reordering basic blocks in object files, portability, debuggability, and certification are issues concerning this approach. Ramirez[4] used maximization of the sequentiality of instructions. However, when capacity misses dominate, neither full-associative nor maximized instruction sequence can help.

Pettis and Hansen[3] presented a profile-guided algorithm based on “closest is best” which can be applied on both the bb-level and procedure level. Hashemi[1] kept track of the cache lines (colors) occupied by each mapped procedure and used it to guide procedure mapping. Instead of using the weighted call graph, Gloy[5] developed the

temporal relationship graph (TRG) by gathering temporal profile information representing the interleaving of procedures in a program trace. Although these methods differ from our approach since they focus only on conflict misses while ours deal with both capacity and conflict misses, we still benefit from these traditional technologies, especially bb-level reorder, procedure-splitting, and TRG. We will give a brief introduction in section 3.1 and 3.2.

In the embedded world, Chiou[11] presented a hardware mechanism named column caching by which software can dynamically partition the cache and map data regions to a specific set of cache "columns". Sanghai [6] presented a framework which took temporal locality into account and partitioned the codes to map them onto SRAM and I-cache separately. However, both of these methods are not simple and cannot adapt to different applications dynamically because they either need support from hardware or need to be configured statically and thus lack flexibility.

3.1 Basic-block Level Reorder and Procedure Splitting

Basic block reordering of functions into hot and cold portions is now common in most compilers. This improves static branch prediction rate and code locality for better instruction cache usage. The bb-level reorder algorithm implemented in the Open64 compiler is based on Pattis and Hansen's algorithm in which consecutive basic blocks will form a bb-chain. Bb-chains formed by hot basic blocks can be identified by profiling and are enclosed in a hot region. Likewise, the Open64 compiler groups the set of cold basic blocks together; they are chained to form a cold region. Then distinct sections (we termed them code blocks throughout this paper) are generated for the hot and the cold regions respectively, which are further laid out by our layout algorithm.

Our code layout algorithm benefits from bb-level reorder in two folds: 1) to increase the spatial locality of programs so as to improve the instruction cache performance and 2) to perform procedural splitting (pu-split).

We benefit from pu-split in three folds: 1) to make generated code blocks more uniform in hotness, regularity, locality and density, 2) to enable finer grain control for code layout heuristics by reducing sizes of objects to be laid out, and 3) by separating HOT/COLD code regions, we can focus on the HOT part for more precise code layout and use COLD parts as pads to fill in the "holes" generated by the layout of hot objects.

3.2 Temporal Relation Graph

The temporal relation graph (TRG) is a powerful tool to precisely express the temporal interleaved relationship (called temporal profile information) between code blocks. We use it to calculate the desired instruction cache size needed and to map the code blocks onto the cache region while reducing conflict misses as much as possible.

The temporal profile information is formally defined as the following. Given a cb-trace of code block references, for two code blocks P and Q, let R(P,Q) be the number of times that two consecutive occurrences of P are interleaved with at least one reference Q, or vice versa. It can be calculated by maintaining a stack of references and increasing R(P,Q) between P and Q when interleaved references occur. R(P,Q) is recorded in TRG as the weight of edge (P, Q).

4. Weighted Temporal Distribution and Code Block Classification

In this section, we will first illustrate the method to identify good candidates for code blocks with good temporal regularity in terms of temporal distribution. Then we will introduce the process of code block classification.

4.1 Weighted Temporal Distribution

To identify good candidates for code blocks with good temporal regularity, we calculate weighted temporal distribution (WTD), which is measurement of whether a code block is a good candidate in terms of temporal distribution. Both hotness and distribution uniformity are considered in its calculation.

To characterize distribution uniformity, we combine the variance of reuse distance with live range information of code blocks. The code blocks with long live range and small variance of reuse distance will be classified to have uniform distribution. The formula is illustrated below.

Assume we have a cb-trace of code blocks such as "…A…A…", in which A is a code block and the total length of the cb-trace is L.

We define P_A to be the array of positions where A appears in the cb-trace. Assuming the total number of A appearances is NA and P_A(i) is the position number of i-th appearance of A in cb-trace, then by definition $P_A(N_A)$ is the position number of A's last appearance, and $P_A(1)$ is the position number of A's first appearance. The live range of A can be calculated as $LR_A = P_A(N_A) - P_A(1)$. A has a longer live range if LRA is larger.

With LR_A, the average reuse distance of A can be calculated as $ARD_A = LR_A/(N_A - 1)$.Then, $Var(RD_A)$, the normalized variance of A's reuse distance, can be calculated by:

$$Var(RD_A) = \frac{\sum_{i=2}^{N_A}\left(\frac{P_A(i)-P_A(i-1)}{ARD_A}-1\right)^2}{N_A-1}$$

The value of $Var(RD_A)$ reflects whether A is uniformly distributed within its live range, that is, from the first time it appears to the last time it appears.

To characterize whether A is uniformly distributed along the whole cb-trace, we define TD(A), the temporal distribution of A, by:

$$TD(A) = \frac{(LR_A / L)^2}{(1+Var(RD_A))^2}$$

We can see that the larger the value of TD(A), the more uniformly A is distributed along the cb-trace.

Considering hotness, we define weighted temporal distribution of A WTD(A) as:

$$WTD(A) = N_A * TD(A)$$

The value of WTD(A) reflects whether A is a good candidate in terms of temporal distribution. The larger the value of WTD(A), the better candidate A is. When N_A is equal to one, we define WTD(A) to be zero. However we did not include the denseness factor in calculating WTD, since WTD only focuses on the temporal characteristics while denseness deals will space. We will consider denseness in the partition process.

To illustrate the process of WTD calculation in detail, we give an example in the following, using the sequence ABCDEF(UV)5ABCDEF(PQ)5ABCDEF(XY)5ABCDEF from Section 1.

We can see that the length of the cb-trace L=54. For code block A, we get position array P_A={1, 16, 31, 46}. The total number of A appearances N_A=4. We can then calculate *WTD*(A) using the above formulas:

Live range of A, LR_A=46-1=45

Average reuse distance of A, ARD_A=45/(4-1)=15

Variance of reuse distance of A, $Var(RD_A)$=0

Temporal distribution of A, $TD(A)=(45/54)^2/(1+0)^2 = 0.83$.

Weighted temporal distribution of A, *WTD*(A)=4*0.83= 3.33 .

Similarly, we can calculate *WTD*(U)=0.11. We can see that *WTD*(A)>>*WTD*(U), and thus A will be a better candidate for code block with good temporal regularity than U in terms of temporal distribution. This is consistent with the observation in Section 1.

4.2 Code Block Classification

Three classes of code blocks are used in our approach:

1. Cold code blocks, which is used as a pad to fill the "holes" generated by code placement. They are identified first by counting the number of appearances in the cb-trace.
2. Hot code blocks with good temporal regularity. For simplicity, we call them "regular code blocks". They are identified by *WTD* calculation. When the value of *WTD* of one code block is greater than a predefined threshold, we classify it as regular code block. In all our experiments presented later in this paper, we set it to be 4.
3. Leftover code blocks. They belong to the hot code blocks but with good temporal locality (or irregular distribution). We call them "irregular code blocks".

The set of regular and irregular code blocks will be further classified in the following partition phase by characterizing instruction density.

5. Iterative Partition and Layout

In this section, we present the method to partition the cache into two distinct regions, to adjust the code blocks between regular and irregular classes when needed, and to layout each class inside its corresponding region. After that, regular code blocks are guaranteed to have little cache misses while the irregular ones may incur some cache misses.

We use a heuristic based algorithm which calculates the needed cache size for each class iteratively while adjusting the node from regular class to irregular, according to instruction density.

We first calculate the needed size of the each class, that is, `RB_SIZE` (size needed by regular code blocks) and `IRB_SIZE` (size needed by irregular code blocks). The method to calculate `RB_SIZE` is based on TRG, which is the smallest size needed to avoid cache misses for regular code blocks. `IRB_SIZE` is evaluated by giving the product of the maximal size of all the irregular code blocks and a coefficient set in advance, that is, `N` in the following pseudo code. The value of the `N` will help control the size of `IRB_SIZE`, and thus will help tune layout results for different applications. In all our experiments presented later in this paper, we set `N` to be 1.

If the sum of `RB_SIZE` and `IRB_SIZE` is less than or equal to cache size, the partition ends. Otherwise, we adjust one code block in the regular class into irregular class. The code block with minimal density in regular class will be selected to be adjusted. Then, `RB_SIZE` and `IRB_SIZE` are recalculated. The process of adjusting and recalculating may repeat multiple times until the sum of calculated `RB_SIZE` and `IRB_SIZE` is no more than cache size.

```
Input:  CACHE_SIZE: actual cache size
        N: coefficient set to help evaluate IRB
        RB: regular code block set
        IRB: irregular code block set
Output: RB: regular code block set
        IRB: irregular code block set
        RB_SIZE: size needed by RB
        IRB_SIZE: size needed by IRB
/* Given two code block set, RB and IRB, partition
/* the cache into two parts and calculate size of each part */
Func Partition ( RB, IRB ) {
    Sort the nodes in RB by instruction density
   // highest instruction density first
    RB_SIZE = Calc_rb_size ( RB )
    IRB_SIZE = Calc_irb_size ( IRB )
    While ( RB_SIZE + IRB_SIZE > CACHE_SIZE ) {
          Adjust ( RB, IRB )
          RB_SIZE = Calc_rb_size ( RB )
          IRB_SIZE = Calc_irb_size( IRB )
    }
}
/* Adjust node with minimal instruction density*/
/* from RB into IRB */
Func Adjust ( RB, IRB ) {
    Remove tail node from RB and insert it to IRB.
}
/* Calculate the needed size of RB */
Func Calc_rb_size ( RB ) {
    RB_SIZE = 0
    For each node P in BS {
       If (P fits into allocated cache lines)
       // P can be mapped onto the allocated cache lines
       // without incurring cache misses
            Continue
       Else
       // Allocate cache lines for P
            RB_SIZE = RB_SIZE + size ( P )
          // size ( P ) is text size of code block P
       End if
    }
   Return BS_SIZE
}
/* calculate the needed size of IRB */
Func Calc_irb_size( IRB ) {
    IRB_SIZE = 0
    max_node_size = max size of all the nodes in IRB
   Return N* max_node_size
}
```

Figure 1. Pseudo code for iterative partition algorithm

The pseudo code for the partitioning is illustrated in figure 1.

In addition, there may exist very large irregular code blocks which will incur waste of cache area (since regular code blocks may not get enough space), or even cause the while loop in function Partition to loop infinitely. To prevent this, we extract this type of code blocks in the partition process and map them aligned with the start of the

cache region for irregular code blocks.

After partitioning, we lay out code blocks of each class within their own cache region respectively using the TRG algorithm and generate placement for each code block.

6. Experiments

To evaluate the effectiveness of our algorithm, an execution trace is used to quantify the instruction cache miss rate and the performance. We used two simulators, a function simulator which accepts the executable file as input and outputs the instruction trace file and a performance simulator which accepts the instruction trace file and outputs performance data. Our layout tool accepts the instruction trace file generated by our function simulator as input, and outputs a link script which specifies the desired memory layout. The linker produces the final binary based on the linker script. The complete process is shown in figure 2. All the tools mentioned above are part of our toolchain for our product.

A set of video and audio codec applications are used to evaluate the performance of our algorithm, including H264 video encoder (Baseline Profile), H264 video decoder (Baseline Profile), AVS-M video decoder, MPEG4-ASP video decoder, and G.729.a voice coder, ordered according to their computation complexity from high to low. The computation complexity of H264 encoder is up to four times that of the H264 decoder while the H264 decoder has computation complexity up to 50% more than that of the MPEG4 decoder. It is important that the processor can deliver the required performance of the most demanding application. Hence, the optimization effect on the H264 encoder is of highest importance. The cache configuration of all experiments is assumed to be two-way set-associative, with 32 byte cache line size.

For video codecs, the experimental results shown in Section 7 correspond to encoding (or decoding) one P frame, the typical frame type in video sequences. Since each frame has 396 macroblocks, each frame represents 396 different variations that the program will run through. For the G.729.a encoder, the results correspond to encoding three audio frames. It is worth noting that the codec sources we tested have been optimized for two different processors. The H.264 encoder, H.264 decoder, and AVS-M decoder are optimized on a multi-threaded 4-issue processor with powerful 16-wide SIMD engine and multimedia instruction extensions, while MPEG4 decoder and G.729.a encoder are optimized on a single-threaded 2-issue RISC processor. This explains why H.264 and AVS-M codec have much better performance numbers than MPEG4 decoder and G.729.a encoder in Section 7.

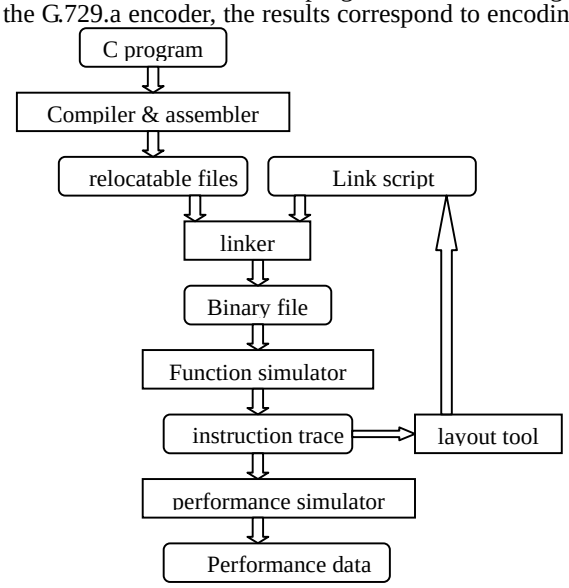

Figure 2. Code layout process

To compare the performance of our layout algorithm (TD) with other approaches, we also implemented the PH and TRG algorithm (the type of TRG for procedures, not the TRG for procedure chunks) and compared the instruction cache results of the three methods. To reflect the potential of different layout methods, all comparisons are based on the same fully optimized binary.

Although our algorithm is profile-based, it is well adaptive to different inputs. This is because our algorithm, in theory, gives higher priority to regular code blocks, which tend to be more stable than the irregular ones. Another set of experiments was performed to evaluate and compare the adaptability of TD, PH and TRG to different input streams.

7. Results

Table 1 shows the performance data of our algorithm. To help understand the contribution of each type of code layout optimization, instruction cache miss rate and total cycle count after each optimization phase are presented. Original programs use the standard link order specified in the Makefile. Bb-level reorder inside the procedure reduced the instruction cache miss rate by an average of 19%. Our layout tool alone reduced the average instruction cache miss rate by 26% without pu-split and by 39% with pu-split. It should be noted that pu-split in Open64 helped increase the effect of layout tool significantly.

Table 2 and Figure 3 compare the instruction cache efficiency between our TD algorithm and the two classic algorithms, PH and TRG. TD achieved the best performance in almost all cases. Especially for H264 encoder, TD outperforms PH and TRG by about 35%. We attribute this to the majority of cache misses being capacity miss, which is not reduced by considering the affinity relationship between code blocks alone, as most previous code layout algorithms did.

The distribution of different kinds of cache misses is shown in Table 3. We can see that TD significantly reduced the number of capacity misses for all the test cases. The minor reduction of compulsory misses is due to better alignment.

Table 4 and Figure 4 give the I-cache miss reduction with different input data by TD, PH and TRG algorithm, respectively. In Figure 4, “HE” represents H264 encoder and “HD” represents H264 decoder. For both the H.264 encoder and decoder, we used “stenfan” as input for profile training, which is a video clip of a person playing tennis. We also used another two set of inputs, “akiyo” and “football”, for further performance evaluation. “Akiyo” is a video clip of typical news report program with an anchor person sitting, almost still. “Football” is a video clip of playing football with fast camera and object motions. We can see that TD produced generally good results and was approaching the best performance for all the inputs.

Test Environment	32K instruction cache, I-cache miss penalty: 12					
	Application	H264 enc	H264 dec	AVSM dec	MPEG4	G729.A
Original	I$ miss rate	3.15 %	1.41 %	1.15 %	0.12 %	0.034 %
	Cycle count	7,394,114	2,547,316	2,348,833	6,741,494	2,950,998
BB reorder	I$ miss rate	2.78 %	1.33 %	0.73 %	0.07 %	0.034 %
	Cycle count	6,938,883	2,458,446	2,202,098	6,694,694	2,643,767
BB reorder and layout	I$ miss rate	2.47 %	0.80 %	0.21 %	0.04 %	0.029 %
	Cycle count	6,583,374	2,322,348	2,074,742	6,672,417	2,640,573
BB reorder, pu-split, layout	I$ miss rate	1.74 %	0.47 %	0.13 %	0.03 %	0.028 %
	Cycle count	5,936,792	2,235,330	2,058,496	6,660,513	2,638,241
Reduction of I$ miss rate		44.8 %	66. 6 %	88.7 %	72.5 %	17.6 %
Reduction of cycle count		19.7 %	8.8 %	12.4 %	1.2 %	10.6 %

Table 1. I-cache miss rate and total cycles w.r.t. various optimizations and combinations.

Test Environment	32K instruction cache				
Application	H264 enc	H264 dec	AVSM dec	MPEG4 dec	G729.A
Original	3.15%	1.41%	1.15%	0.12%	0.034%
PH	2.74 %	0.49 %	0.26 %	0.040 %	0.030 %
TRG	2.64 %	0.86 %	0.24 %	0.037 %	0.028 %
TD	1.74 %	0.47 %	0.13 %	0.033 %	0.028 %

Table 2. I-cache miss rate of TD, PH, and TRG algorithms.

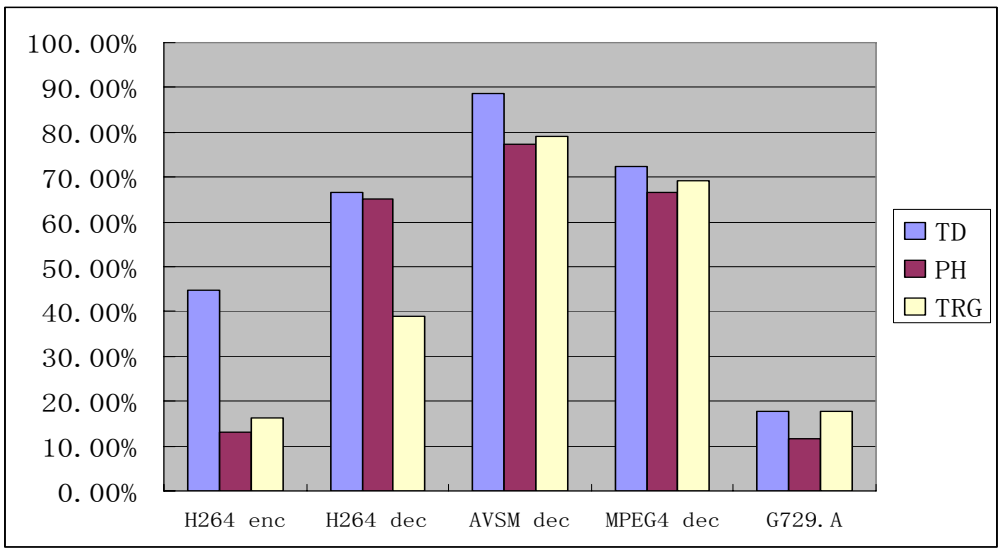

Figure 3. Reduction of I-cache misses by TD, PH and TRG.

Test Environment	32K instruction cache					
Application	H264 enc			H264 dec		
Input data	stenfan	akiyo	football	stenfan	akiyo	football
Original	3.15 %	2.11 %	3.35 %	1.41 %	1.02 %	1.91 %
PH	2.74 %	2.02 %	2.97 %	0.49 %	0.26 %	1.72 %
TRG	2.64 %	1.49 %	2.87 %	0.86 %	0.45 %	1.55 %
TD	1.74 %	0.86 %	2.04 %	0.47 %	0.29 %	1.60 %

Table 4. I-cache miss rate of TD, PH, and TRG algorithms.

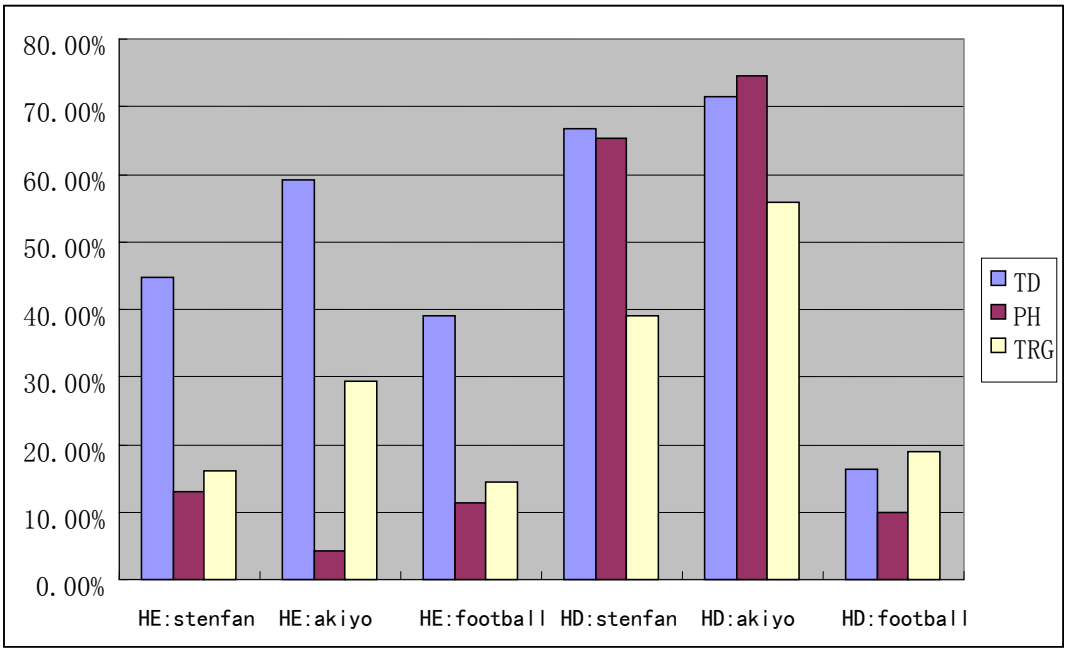

Figure 4. Reduction of I-cache misses by TD, PH and TRG with various inputs.

8. Conclusion

In our code layout work, we developed a new algorithm to position code using temporal distribution characteristics of code blocks and map them to different logical regions of the instruction cache. In this method, both capacity and conflict cache misses are effectively reduced, and the algorithm showed good adaptability to the various multimedia programs to be used in our product, especially outperforming other traditional algorithms for applications suffering from a large number of capacity misses.

Compared with other algorithms that partition code into SRAM and I-cache, our software partition approach can produce the same effect and is more flexible and adaptable for each application. Also, for systems that will run multiple applications, a static memory configuration will place a huge burden on system design, to swapping the binary between external memory and SRAM at program start. In contrast, for a pure cache approach, this is handled automatically.

9. Future work

We have put forth an attempt to better deal with capacity I-cache misses for embedded processors with small and single level cache. We feel that this approach should be effective for general purpose processors and for applications in general, not just multimedia applications. We have yet to prove this is indeed the case.

There have been precise models to evaluate capacity D-cache misses to guide data layout, but a more precise modeling of various types of I-cache misses has not been done. Such a model may be useful to guide more complex code layout algorithms and produce even better results, suggesting another avenue for future work.

Acknowledgments

The authors would like to thank Sun Chan for his encouragement and support for this work. We also thank Robert Hundt for his helpful comments in improving the quality of this paper.

References

[1] A. H. Hashemi, D. R. Kaeli, et al, “Efficient Procedure Mapping Using Cache Line Coloring,” In ACM Conference on Programming Languages Design and Implementation, pages 171–182, 1997.

[2] S. McFarling, “Program Optimization for Instruction Caches,” In ACM Conference on Architectural Support for Programming Languages and Operating Systems, pages 183-191, 1989.

[3] K. Pettis and R. C. Hansen, “Profile-guided code positioning,” In ACM Conference on Programming Languages Design and Implementation, pages 16-27, 1990.

[4] A. Ramirez, J.-L. Larriba-Pey, et al, “Software Trace Cache,” In International Conference on Supercomputing, pages 119-126, 1999.

[5] Nikolas Clemens Gloy, “Code Placement using Temporal Profile Information,” PHD thesis. 1998.

[6] Kaushal Sanghai and David Kaeli, “A Code Layout Framework for Embedded Processors with Configurable Memory Hierarchy,” 5th Workshop on Optimizations for DSP and Embedded Systemsm 2007.

[7] Alex Ramirez, Luiz Andre Barroso, et al, “Code Layout Optimization for Transaction Processing Workloads,” In International Symposium on Computer Architecture Proceedings of the 28th annual international symposium on Computer architecture. 2001.

[8] O. Temam, C. Fricker, et al, “Cache Interference Phenomena,” In Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 261-271, 1994.

[9] Chi-Keung Luk, Robert Muth, et al, “Ispike: A post-link Optimizer for the Intel Itanium Architecture,” In International Symposium in Code Generation and Optimization. 2004.

[10] Chun Xia, Josep Torrellas, “Instruction Prefetching of Systems Codes With Layout Optimized for Reduced Code Misses,” In International Symposium on Computer Architecture.1996.

[11] Chiou, D. Jain, et al, “Application-Specific Memory Management for Embedded Systems,” Design Automation Conference, 2000.

[12] Peter J. Denning, “The Working Set Model for Program Behavior,” ACM Symposium on Operating Systems Principles, 1967.

[13] Trishul M. Chilimbi, “Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality,” ACM Conference on Programming Language Design and Implementation, 2001.

[14] Open64, http://open64.sourceforge.net/