# CODE POSITIONING TO REDUCE INSTRUCTION CACHE MISSES IN SIGNAL PROCESSING APPLICATIONS ON MULTIMEDIA RISC PROCESSORS

*Hans-Joachim Stolberg* *Masao Ikekawa*[1] *Ichiro Kuroda*[1]

Laboratory for Information Technology, University of Hannover, Germany
[1]Information Technology Research Laboratories, NEC Corporation, Kawasaki, Japan
Email: stolberg@mst.uni-hannover.de

## ABSTRACT

Real-time operation of signal processing applications on multimedia RISC processors is often limited by high instruction cache miss rates of direct-mapped caches. In this paper, a heuristic approach is presented which reduces high instruction cache miss rates in direct-mapped caches by code positioning. The proposed algorithm rearranges functions in memory based on trace data so as to minimize cache line conflicts. Moreover, a new method to extract potential cache misses from trace data is introduced which enables accurate cache behavior analysis and greatly enhances code positioning efficiency. Application of code positioning to an MPEG-1 video decoder implementation on the V830 multimedia RISC processor reduced instruction cache refill cycles by 66–98 %. The proposed code positioning algorithm does not require hardware modifications; it can easiliy be integrated in an object linker to automate the optimization process.

## 1. INTRODUCTION

Complex real-time signal processing tasks are increasingly implemented on multimedia RISC processors [1][2], enabled by rapid advances in clock speed and performance as well as by integration of DSP execution units. RISC-based implementation offers the advantages of easy programmability and availability of excellent optimizing compilers, thus reducing both design time and cost. However, because of high clock speed in RISC processors, access to instructions and data in external memory cannot be accomplished without stall cycles. Therefore, RISC processors incorporate caches to keep a subset of instructions and data on-chip where they are accessible within one clock cycle. Thus, external memory accesses and resulting stall cycles only occur if a requested instruction or datum is not found in cache. In order to achieve a tradeoff between stall cycles, storage capacity, and implementation cost, a multi-level memory hierarchy may be employed with a small, fast-accessible on-chip cache as first level, a larger external cache as second level, and the main memory as highest level. For refillment, caches are subdivided into cache lines containing multiple instructions or data in order to exploit spatial locality and to utilize fast memory access modes.

Due to their simplicity compared to other cache organizations, direct-mapped caches are the most cost-effective solution for multimedia RISCs as they offer the highest storage capacity on a given silicon area; they are furthermore characterized by short access times, which in turn account for high processor clock speed. With direct-mapped organization, each memory entry has exactly one location where it can appear in cache, and only one entry can reside at that location at a time. Thus, when an instruction or datum not present in cache is accessed, it will be loaded to its destined cache line, replacing the entire previous cache line entry. As a result, cache misses caused by conflicts between memory addresses that map to the same cache line are more likely to arise in direct-mapped caches.

Real-time signal processing applications are especially prone to instruction cache misses in direct-mapped caches as they typically involve a limited set of functions executed periodically to process the incoming data. Therefore, with an unfavourable code layout where frequently executed code parts map to the same cache lines, instruction cache misses will repetitively occur in the same functions, degrading performance seriously. The effect is even aggravated for cost-sensitive applications where a second-level cache cannot be afforded, which dramatically increases the miss penalty. In consequence, in spite of high CPU performance offered by modern multimedia RISC processors, in many cases real-time operation of signal processing tasks is severely threatened by a prohibitive high number of instruction cache refill cycles.

In order to make the implementation advantages of multimedia RISC processors accessible for real-time signal processing applications, code positioning can be employed to minimize mutual instruction replacement and resulting cache misses. In this paper, a new code positioning scheme is presented which rearranges functions of a signal processing application in memory based on trace data so that the mapping of frequently accessed code parts to same cache lines in a direct-mapped cache is avoided.

In section 2, the code positioning approach—comprising trace data processing and function mapping—is described. Section 3 presents simulation results for the application of code positioning to an MPEG-1 video decoder implementation on the V830 multimedia RISC processor, and section 4 concludes the paper.

## 2. CODE POSITIONING

The proposed code positioning approach consists of two steps: trace data processing and function mapping. At first, however, trace data has to be collected while executing the target application with typical input data. The collected

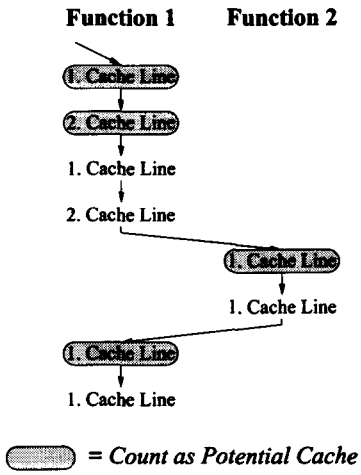


**Figure 1. Scheme for counting potential cache misses in a sequence of memory accesses.**

trace data has to contain the sequence of memory accesses during program execution—information on cache hits or misses is not required at this stage.

### 2.1. Trace Data Processing

A new method is introduced which exploits the address sequence information inherent in instruction trace data as input for the code positioning algorithm. So far, existing code positioning methods [3][4][5][6] had to rely on purely statistical data such as call graphs or execution frequencies that do not contain information on sequential program behavior. Address sequence information, however, is essential for cache behaviour optimization since occurence of cache misses is related to the sequence of memory accesses rather than to their mere number.

The new method evaluates the address sequence of memory accesses in order to extract only those accesses which can actually lead to cache misses during code execution. As each function of the original implementation will be preserved in the code positioning without modification, repeated memory accesses to same cache lines within execution of one function are always sure to be served by cache hits. They can therefore be identified in the instruction trace and discarded as they are irrelevant for cache behavior optimization. In most of the existing methods [3][4][5][6], this way of exploiting the address sequence in trace data is not possible as they rearrange code in basic block units, which can lead to mutual replacement even during execution of one function. Here, functions denote the program subroutines delimited by their start addresses and concluding return instructions.

The new method of trace data analysis comprises the following steps:

1. In the memory layout of the original implementation, subdivide the memory range of each function into cache line units, assuming the first instruction of each function to be aligned to the start of a cache line.
2. Of the memory accesses listed in the instruction trace, count only the first access to each cache line as long as one function is executed.

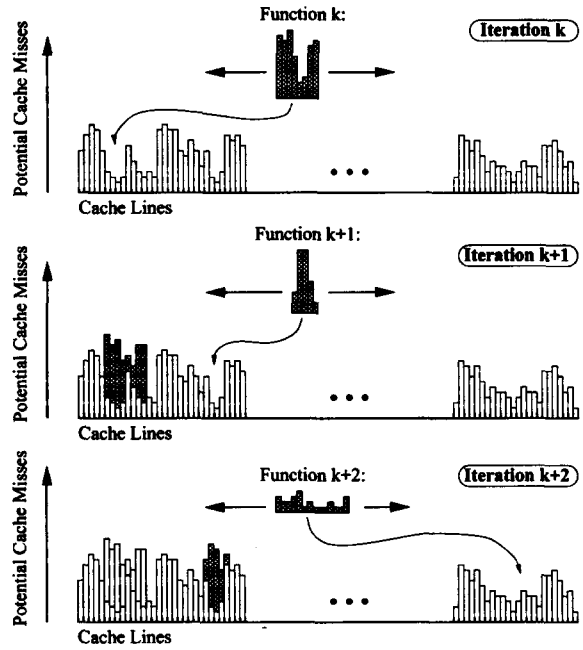


**Figure 2. Successive allocation of functions to cache space.**

3. Repeat step 2 every time a new function call has occurred.

Figure 1 demonstrates this counting scheme by example of two alternately executed functions. As the final function layout in memory is not yet defined at this stage, it always has to be assumed that a new-called function does not reside in cache as it might have been replaced by intermediate execution of other functions. Therefore, memory accesses have to be counted again for potential cache misses after every new function call, as expressed in step 3. As a result of this method, the number of potentially arising cache misses per cache line unit during execution of the target application is obtained separately for each function.

### 2.2. Function Mapping

Based on the number of potentially arising cache misses derived from trace data, the functions of the target application are then allocated to cache space so as to minimize the overlapping of code parts with high numbers of potential cache misses in the same cache line. This is achieved by the new heuristic code positioning algorithm comprising the following steps:

1. Sort functions for their average number of potential cache misses per cache line in descending order.
2. For the first function in the sorted list, find that position in cache space which leads to the most homogeneous distribution of potential cache misses over the entire cache space, considering all already allocated functions.
3. Repeat step 2 for each next function in the sorted list, until all functions are successively allocated.

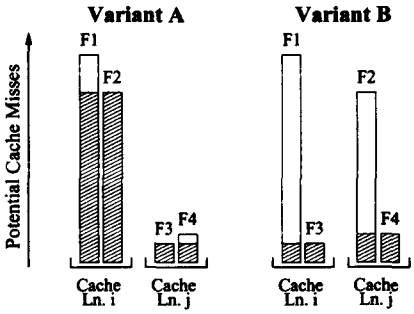


**Figure 3. Two variants of function overlapping in cache lines.**

4. Map function start addresses from cache space to memory space so that each instruction will be assigned to a unique memory address.

Figure 2 demonstrates this scheme of function allocation for three iterations. As can be seen, functions are successively allocated to cache space aiming at a homogeneous distribution of potential cache misses over cache lines. Such a distribution is achieved by calculating for all possible cache positions of the current function the squared sums of potential misses per cache line including the already allocated functions, and by selecting that allocation which leads to the minimum overall sum. The initial sorting of functions maintains the highest allocation flexibility for the most cache-intensive code parts and rules out overlapping of those parts in the same cache lines.

Figure 3 explains the underlying strategy of the algorithm: if, as in variant A, two functions with high numbers of potential cache misses map to the same cache line, amounting to a highly unhomogeneous distribution of potential misses over cache lines, the overall cache miss probability will be high since these functions will suffer frequent mutual replacement. If however the sums of potential cache misses for all functions are equally distributed over cache space as in variant B, functions with high numbers of potential misses will come to share the same cache lines only with functions of much lower numbers of potential misses, thus leading to a low overall cache miss probability as mutual replacement can occur only infrequently. Therefore, by forcing a homogeneous distribution of potential misses over cache space, the heuristic algorithm achieves a minimum instruction cache miss rate for direct-mapped caches.

As a result of the heuristic algorithm, a new start address is obtained for each function of the signal processing application where it has to be aligned to in memory. Since the functions themselves remain unchanged, they only have to get linked together with regard to their new start addresses. As the code positioning algorithm may introduce allocation holes between functions in the final mapping, the overall memory occupied by the program code is likely to increase during the link process. However, while external memory size is not a critical issue, a significant reduction of cache miss rate and better utilization of on-chip memory is thus provided. It should be noted that memory bus load or execution cycle count are not affected by inserted allocation holes as their memory locations will never be addressed.

| *flower-garden* | *Without Code Positioning* | *With Code Positioning* | *Reduced by* |
|---|---|---|---|
| *B-Frame* | 211,729 | 71,356 | 66.3 % |
| *P-Frame* | 277,613 | 5,336 | 98.1 % |
| *I-Frame* | 27,056 | 4,386 | 83.8 % |

**Table 1. Instruction cache refill cycles for MPEG-1 sequence *flowergarden* before and after code positioning.**

Aside from the initial simulation of the target application to generate trace data, the described heuristic code positioning algorithm and the preceding trace data analysis both can easily be integrated in an object linker in order to automate the optimization process and to make it transparent for users; modifications of the compiler are not necessary. In contrast, most existing code positioning methods [3][4][5][6] require massive object code modifications as they operate on basic block level, which makes them difficult to implement.

## 3. SIMULATION RESULTS

The described code positioning algorithm has been applied to an MPEG-1 software video decoder implemented on the multimedia RISC processor V830 [7] in order to enable real-time operation by a low cache miss rate for V830's 4-kbyte direct-mapped instruction cache.

The V830 is a low-cost, low-power 32-bit RISC microprocessor with DSP capabilities, delivering 118 MIPS at 100 MHz. DSP execution units such as multiply-adder and special DSP instructions enable efficient implementation of complex multimedia algorithms. In addition, V830 offers 16 kbyte of on-chip memory, comprising instruction and data RAMs as well as direct-mapped instruction and data caches of 4 kbyte each. While on-chip memory access can always be accomplished within one clock cycle, external memory access to refill a cache line of 16 byte accounts for at least 21 clock cycles in the V830 MPEG-1 video decoder system environment when assuming the best case of SRAM as external memory.

Besides compiler and assembler, a V830 simulator is available which is able to collect trace data during simulated program execution in order to support software development. Additional tools enable several information to be extracted from generated trace data. The trace data required for code positioning as well as the simulation results presented in this paper have been obtained using the V830 simulator.

Table 1 presents simulation results for the numbers of instruction cache refill cycles in MPEG-1 video decoding before and after code positioning, and their reduction. The simulation has been performed for decoding MPEG-1 sequence *flowergarden* which comprises I-, P- and B-pictures as defined in the MPEG-1 standard. Cache refill cycles are reduced by 66–98 % for this sequence; simulation for other MPEG-1 sequences showed very similar results, including those sequences which have not been used for initial trace data collection. In general, for most signal processing applications including MPEG-1 decoding, the sequence of

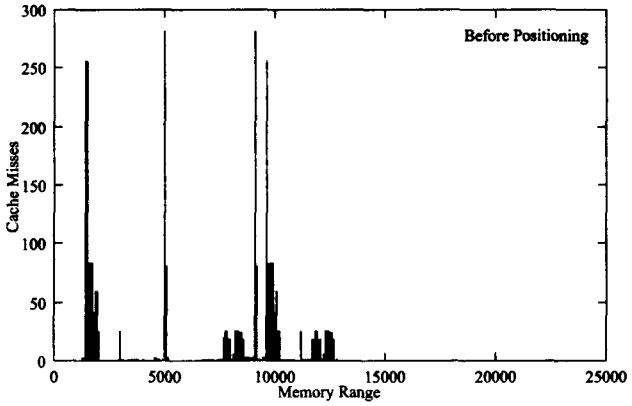


**Figure 4. Instruction cache misses over memory range in decoding one B-frame of *flowergarden* before code positioning.**

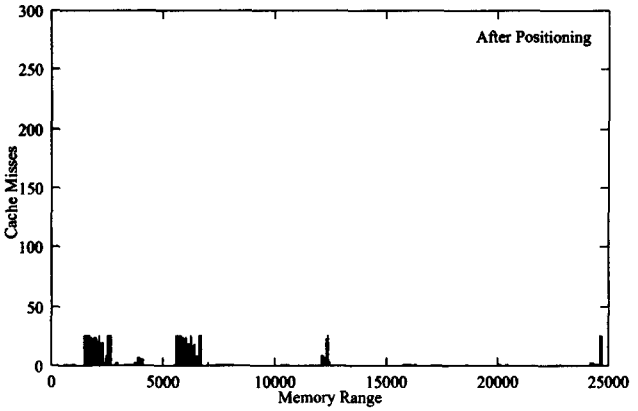


**Figure 5. Instruction cache misses over memory range in decoding one B-frame of *flowergarden* after code positioning.**

function calls during program execution can be assumed as sufficiently independent from input data so that code positioning as described here will always lead to satisfactory results.

In case of the MPEG-1 video decoder, instruction cache refill cycles have been decreased from 27.4 % of overall execution clock cycles down to only 1.6 % by a combination of using V830's internal 4-kbyte instruction RAM and applying code positioning. Although the larger part of this reduction is due to the use of V830's internal instruction RAM (down to 6.5 %), real-time operation has only been achieved after additional code positioning. On processors without the benefit of internal instruction RAM, a much higher gain can be expected.

Figures 4 and 5, which show the number of cache misses per cache line in decoding a B-frame of *flowergarden* before and after code positioning, demonstrate the effectiveness of the new algorithm in limiting the number of cache misses to low values for all cache lines as it has been expected from theoretical consideration. This effect also helps to guarantee rigid upper limits for execution times of program segments, which is an important issue in many real-time applications.

In order to reduce cache misses in direct-mapped data caches as well, it seems conceivable to apply a similar scheme for static memory layout of data structures.

## 4. CONCLUSION

In this paper, a new code positioning scheme has been introduced which allows to exploit the advantages of modern multimedia RISC processors for the implementation of real-time signal processing applications without having to suffer performance loss due to frequent instruction cache misses. The heuristic code positioning algorithm rearranges the function layout in external memory in order to minimize mapping of frequently executed code parts to same cache lines in direct-mapped caches. Efficiency of code positioning is guaranteed by the novel approach of extracting information from trace data relevant for cache behavior optimization. The algorithm can easily be integrated in an object linker in order to automate the optimization process. Application of the code positioning algorithm to an MPEG-1 software video decoder implementation on the V830 multimedia RISC processor reduced the number of instruction cache refill cycles by 66–98 %, thus proving the effectiveness of the presented method. For many signal processing applications, previously unattained real-time operation on multimedia RISC processors can become possible with the presented code positioning approach.

## 5. ACKNOWLEDGEMENT

The authors would like to thank Dr. Takao Nishitani for his continuing encouragement, support, and for providing excellent working conditions.

## REFERENCES

[1] V. Bhaskaran, K. Konstantinides, R. B. Lee, and J. P. Beck. "Algorithmic and Architectural Enhancements for Real-Time MPEG-1 Decoding on a General Purpose RISC Workstation". *IEEE Trans. Circuits and Systems for Video Technology*, Vol. 5(No. 5):380–386, 1995.

[2] K. Nadehara, I. Kuroda, M. Daito, and T. Nakayama. "Low-Power Multimedia RISC". *IEEE Micro*, Vol. 15(No. 6):20–29, December 1995.

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

[4] W.-m. Hwu and P. Chang. "Achieving High Instruction Cache Performance with an Optimizing Compiler". In *Proc. 16th Ann. Int'l. Symp. On Computer Architecture*, pages 183–191, June 1989.

[5] K. Pettis and R. Hansen. "Profile Guided Code Positioning". In *Proc. Conf. On Programming Language Design and Implementation*, pages 16–26, June 1990.

[6] A. D. Samples. *Profile-Driven Compilation*. PhD thesis, University of California, Berkeley, 1991.

[7] K. Nadehara, H.-J. Stolberg, M. Ikekawa, E. Murata, and I. Kuroda. "Microprocessor Architecture Design for Low-Cost, Low-Power Video Decoding". In *VLSI Signal Processing IX*, pages 438–447. IEEE, October 1996.