Aligning Single Path Loops to Reduce the Number of Capacity Cache Misses

Bekim Cilku, Roland Kammerer, and Peter Puschner
Institute of Computer Engineering
Vienna University of Technology
A1040 Wien, Austria
Email: {bekim, kammerer, peter}@vmars.tuwien.ac.at

***Abstract*—In this paper we address the problem of improving the instruction cache performance for single-path code. The properties of single-path code allow us to align single-path loops within the cache in order to reduce the number of cache misses during the loop execution. We propose an algorithm that categorizes loops in a simple way so that the loops can be aligned and NOP instructions can be inserted to support this loop alignment. Our experimental results show the predictability for cache misses in single-path loops and demonstrate the benefit of the single-path loop alignment.**

***Keywords*—time predictability; cache memories; memory hierarchy; hard real-time systems;**

I. Introduction

For hard real-time embedded systems, the time instant at which the results are produced is as important as the accuracy of the results [1]. This requirement necessitates the calculation of Worst Case Execution Time(WCET) bounds for code pieces and tasks in the design stage. Unfortunately, the software and hardware used in hard real-time systems are often highly complex which makes the WCET analysis very difficult [2]. Software is written in a way to be executed fast by having different execution paths for different input data. Different paths, in general, have different timing, and analyzing all those paths can lead to cases where the tool cannot produce results of the desired quality or the analysis gets overly expensive [3]. The inclusion of state-of-the-art hardware features (cache, branch prediction, out-of-order execution, and pipelines) makes the analysis even harder because of the state dependencies and mutual interference that have to be considered [4].

An alternative approach to deal with the complexity of WCET analysis is the use of the single-path paradigm [3]. This strategy generates single-path code by converting all input dependent alternatives of the code into pieces of sequential code and transforming all loops with input-dependent termination conditions into loops with constant execution count. Single-path code forces every execution to follow the same trace of instructions, which reduces the number of paths through every code piece or task to one and thus supports the composability of task timing from the software side – as every execution of a task follows the same trace the stream of executed instructions does not depend on the context; this in turn eliminates control-flow induced variations in execution time. However, all these properties come at the cost of a longer execution time.

In this paper we present an algorithm that aligns single-path loops with cache lines in order to reduce code execution time by decreasing the number of cache misses. Loops in single-path code have most of the time a sequential body where every instruction is executed in each iteration. Also, to eliminate time jitter, the single-path loops are bounded in their maximum number of iterations [5]. When the processor fetches loop instructions from memory, it brings them to the cache in form of chunks. If a loop is smaller than the cache capacity then cache misses will appear only in the first iteration as a result of compulsory misses. For loops that are larger than the cache size, a capacity miss will also be present. Our algorithm aligns single-path loop code in order to reduce the number of cache lines that loops need in the cache which reduce the number of cache misses in each loop iteration. The strategy itself is quite old, but so far has been implemented only for data cache [6]. The properties of generating long and sequential loops with a maximum number of iterations allows us to adapt this approach for the instruction cache.

This paper is organized as follows. The next section gives a short background on single-path code and describes the logical organization of cache memories. Section three describes the algorithm to categorize the loops that can benefit from the alignment and also describes the rules for aligning loops. In section four, the evaluation of the concept is presented while section five shows the related work. The paper is concluded in section six.

II. Background

A. Single-path Paradigm

The single-path strategy is an extension of if-conversion [7] that transforms branching code into sequential code. With this approach input dependent code alternatives are translated into sequential pieces of predicated code with the same functional behavior. After the transformation, the execution behavior of the generated single-path code becomes completely independent from the input data, which forces the execution to follow always the same trace of instructions. The essence of predicated execution is the ability to guard the processor state modification based upon the predicate's condition [8]. Predicate instructions are all executed but only those that have a predicate value *true* can change the hardware state. If the predicate evaluates to *false* the instruction will behave like a NOP instruction without changing any hardware state. In the following we describe the process of the transformation of conditional statements and loops into single-path code.

Conditional Statements are considered as *if-then-else* constructs. Depending on the result of the condition evaluation, the execution can continue on different traces. The transformation of such conditional branches is straightforward. If the outcome of the branching condition depends on the program input, then the single path conversion generates a code sequence that serializes the alternatives into sequential code with predicates [5]. Figure 1 shows an example of the *if-then-else* translation into branching code and predicated code, both with equivalent semantics. In the first case only one of the basic blocks (BB1 or BB2) will be executed depending on the branch output, while in the second case both blocks (BB1 and BB2) are executed but the predicate value will guard the changes of the hardware states.

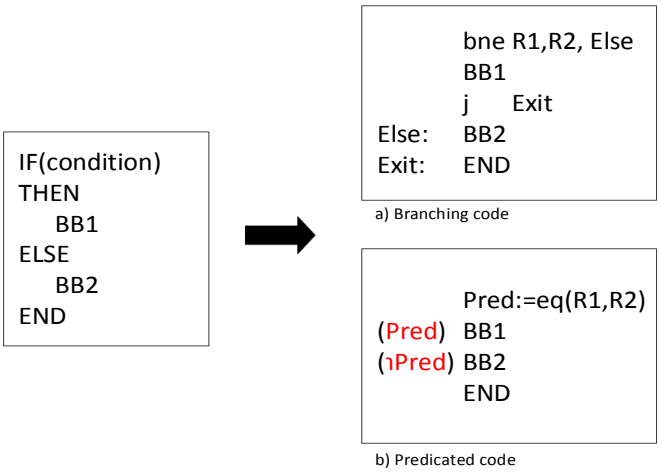

Fig. 1: Converted branching code

Loops with input-data dependency are transformed in two steps. First, the original loop is replaced by a *for-loop* and the number of iterations is assigned. The iteration count of the new loop is set to the maximum number of iterations of the original loop code. The termination of the new loop is controlled by a new counter variable in order to force the loop to iterate always a constant number. The second step generates a statement from the termination condition of the original loop to change the predicate value so to keep the semantic of the original one [3]. Figure 2 shows a simple example of the two-step *while loop* conversion. The loop is converted to a *for-loop* and the iteration count is set to the maximum. Before entering the loop, the maximum number of iterations for the loop is assigned, and on each iteration this value is decremented. At the end of the loop, the program counter jumps back to the header, until the counter will reach the value equal to zero. The termination condition of the old loop is assigned to the predicate in order to keep the semantic of the old loop. In case the predicate value changes (termination) the loop will still continue to execute, but without changing any hardware state from then on.

B. Understanding Cache Behavior

Caches are small and fast memories that are used to improve the performance between processors and main memories

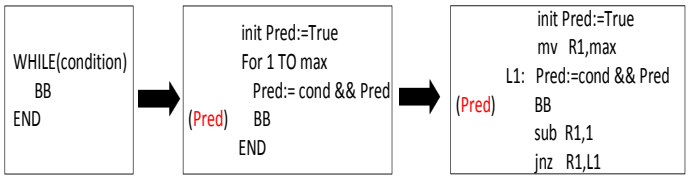

Fig. 2: Converted loop code

based on the principle of locality. The property of locality can be observed from the aspects of temporal and spatial behavior of the execution. Temporal locality means that the code that is executed at the moment is likely to be referenced again in the near future. This type of behavior is expected from the program loops in which both data and instructions are reused. Spatial locality means that the instructions and data whose addresses are close by will tend to be referenced in temporal proximity because the instructions are mostly executed sequentially and related data are usually stored together [9].

References instructions (or data) that are found in cache are called *hits*, while those that are not in the cache are called *misses*. Usually the processor stalls in case of cache misses until the instructions/data have been fetched from main memory. The time needed for transferring the instructions/data from the main memory into the cache is called *miss penalty*. To benefit from the spatial locality properties of the code, the cache always fetches one or more chunks of data, called *cache blocks*, and places them into the *cache lines*. Those misses that are generated because a reference is accessed for the first time are called *compulsory misses*. If a cache cannot store all the blocks needed during the execution of the program a *capacity miss* will occur, because blocks can be evicted and later referenced again [10]. Cache memories with more than one cache line can be organized as a fully associative (cache blocks can be stored in any cache line), set associative (cache lines are grouped into sets), or direct mapped (cache blocks can be placed only on a particular cache line).

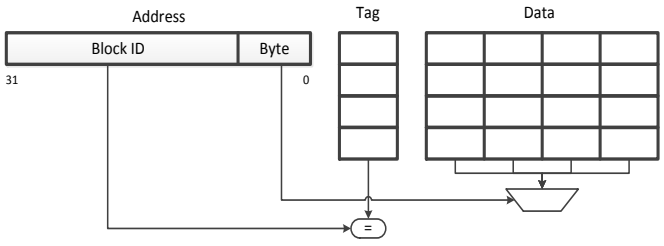

Fig. 3: Cache organization

The basic structure of fully associative caches is illustrated in Figure 3. As an application is executed over the time, it makes a reference to the memory by sending the address. At this step the cache compares the *Block ID* part of the address with tags from the cache. If a match is found, it generates a cache-hit signal and then it uses the *Byte* part of the address to select the requested data from the cache block.

III. Techniques to Align the Single-path Loops

In following we describe the algorithm for tuning single and nested single-path loops.

A. Single Loop

The first instruction of the loop, based on the memory address, can be located in any position within a cache line. If this position is not aligned with the cache line bound then the cache line will also contain instructions that do not belong to the loop. The same can also happen with the cache line that contains the last instruction. Loops that are bigger than the cache capacity will bring these instructions into the cache on each iteration due to the capacity misses. In figure 4 we show a simple example of a sequential loop with 15 instructions. Depending on the starting address of the loop, the loop instructions can be spread into four or five cache blocks. If we assume that the cache size is equal to one cache block, then on each iteration, the first two cases will generate four cache misses, the last two cases five cache misses. By padding NOP instructions at the end of the upper basic block (the last two cases), we align the start of the loop with the next cache block boundary. In this way, on each iteration, the number of cache misses will be reduced by one compared with the original code structure.

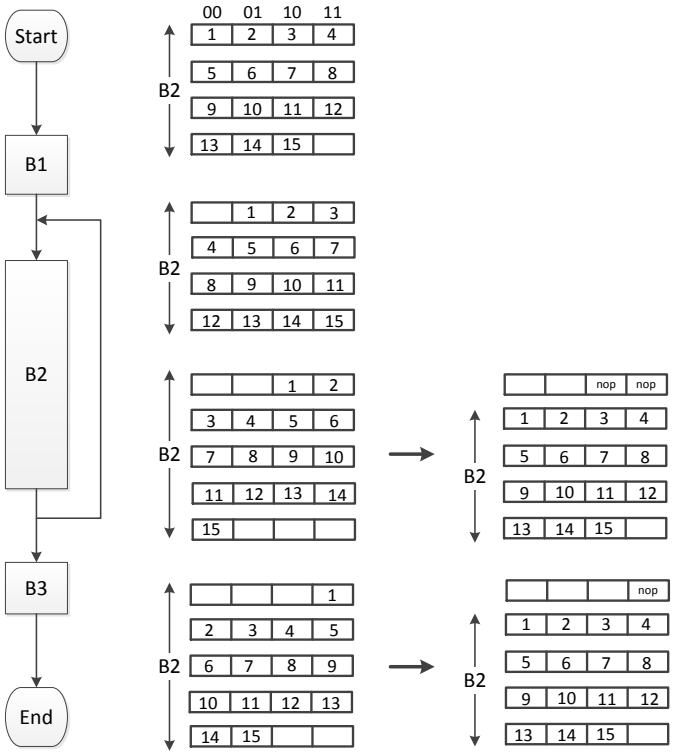

Fig. 4: Mapping the B2 loop block into cache lines

Before aligning the loops, the algorithm firstly has to check which loops do not fit completely in the cache in order to mark loops that generate capacity misses. Only loops that are bigger than the cache size are vulnerable to capacity misses. The amount of cache blocks that a loop occupies in the cache is calculated as the difference between the BlockIDs of the first and the last loop instruction, multiplied by the size of the cache block (1). The produced value is compared with cache size (2).

$$Loop_{size} = (BlockID_E - BlockID_B + 1) * CacheBlock \quad (1)$$

$$Loop_{Size} > Cache_{Size} \quad (2)$$

The next condition of the algorithm is to check if the loop can be aligned. The requirement is that the size of all data entries (instructions) that are fetched at the beginning and the end of the loop but are not executed (N_B and N_E) to be greater or equal than the size of the cache line. In Figure 5 we show the first and the last cache block of the loop. X_B is the byte offset of first loop instruction while X_E of the last one. If X_B is bigger than X_E then we know that N_B is smaller than N_E and the loop can be aligned (3). The number of NOP instructions that have to be inserted before the loop is calculated with equation (4). NOP instructions now belong to the upper basic block. They will be executed only once, before the loop is entered. In this way the loop will be spread on N-1 cache lines which reduces the number of cache misses by one on each iteration.

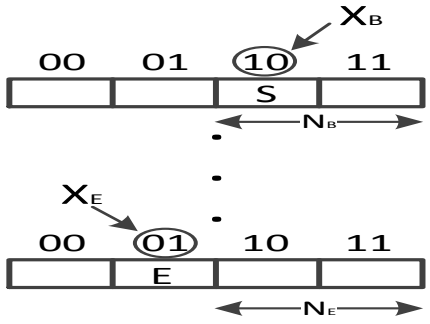

Fig. 5: Position of the first and the last loop instructions in the cache lines

$$X_B > X_E \quad (3)$$

$$N_B = CacheBlock - X_B \quad (4)$$

B. Nested Loops

For nested loops the reduction of cache misses is more complex. The alignment of one loop can shift the other loops which will generate more cache misses for the others. In order to prevent this consequence, we build a table (Table I) for all nested loops. Firstly we qualify which loop is larger

than the cache size (2) and of those we mark the ones that can be aligned (3). The inner ones that satisfy the first two requirements will be the first to be aligned. This choice is made because in single-path nested loops the inner ones iterate most. Then we continue with the other loops, until one of N_E of any loop become zero. N_E is the boundary to ensure that the shifting of the loop will not increase the number of cache lines for the other loops and with that the number of cache misses.

TABLE I: Example of nested loop table

Loop$_I$ D	Loop$_S$	Iteration	X_B	X_E	Alignment	N_B	N_E
L1	N	50	10	10	N	2	1
L2	N	15	11	00	Y	1	3
L3	F	20	01	01	N	3	2

In Figure 6 we show a nested loop with depth three. The values for these loops are given in Table 1. Based on the requirements, only *L2* can be aligned. The additional condition that has to be checked for nested loops is that the number of padding NOP instructions should not be bigger than the smallest N_E of all loops.

IV. Evaluation

In this section we present the results we obtained from our experiment. In the absence of a compiler with single-path transformation we have not been able to evaluate the performance of the algorithm for a larger number of benchmarks. However, we used an FFT program code to observe the number of cache misses that are generated for shifting the loop through the cache line in order to see the behavior of the cache misses for different numbers of NOP instructions padded before the loop. The FFT program is a single-path code that has a loop with a fixed number of iterations. The code was compiled with the LLVM compiler [11] and run on *pasim* (simulator for Patmos processor) [12]. The first instruction of the innermost loop was aligned with the cache line. During the experiments we shifted the loop with NOP instructions and executed the code for different cache-line sizes in order to monitor the number of cache misses. From the obtained results we saw that the number of cache misses was changed only when the whole cache line was filled with NOPs. In Table II we show the number of cache misses measured from the simulation. For a cache with cache line 16B the number of cache misses was increased on every fourth NOP instruction (instruction size 4B), and the cache miss difference was always a constant number. The same behavior had the other caches with 32B, 64B and 128B by increasing the number of cache misses after every eight, 16, and 32 NOPs. In all cases, when the cache line was filled with NOP instructions, the number of cache misses was increased by 63.

V. Related work

Most of the techniques for program transformation to improve cache performance are considering data cache behavior. They reduce cache misses by merging arrays, padding and aligning structures, packing structures and arrays, and interchanging loops [13]. The first three techniques change the allocation of data structures, whereas loop interchange modifies the order in which data structures are referenced. Capacity misses are eliminated by transforming the program to reuse the data before they are evicted from cache, such as loop fusion, blocking structure and array packing.

A technique that deals with capacity cache misses in the

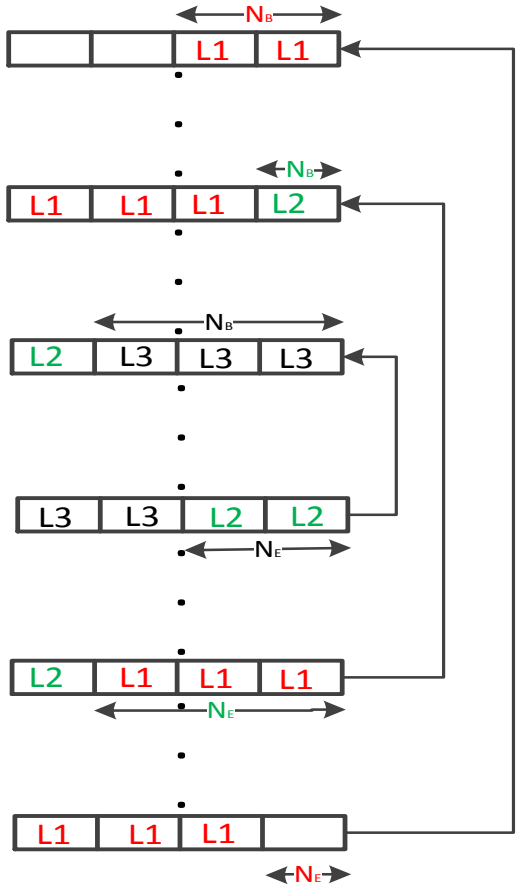

Fig. 6: Nested loops mapped into cache lines

TABLE II: The number of cache misses for different cache line sizes

Cache Line Size (B)	16	32	64	128
Number of NOPs inserted	4	8	16	32
Number of cache misses 1	4754	2480	1189	600
Number of cache misses 2	4817	2543	1252	663
Number of cache misses 3	4880	2606	1319	726

instruction cache is cache locking [14], which loads cache contents with instructions and locks it to ensure that the contents will remain unchanged afterwards. The cache content can be locked entirely or partially for the whole system lifetime (static cache locking) or it can be changed at runtime at predefined points in execution (dynamic cache locking) [15]. Usually the code is profiled and the most referenced fragments of the code are brought into the cache in advance and locked.

In [16] the benefit of using scratchpads is demonstrated. The main advantage of this type of memories is that the contents can be controlled explicitly by the programmer or compiler. The use of a comprehensive algorithm for scratchpad managing can improve the number of conflict misses by utilizing efficiently the loops into the scratchpad. Also, this type of memories are smaller and consume less energy than conventional caches of the same size.

VI. Conclusion and Future Work

In this paper we address the problem of improving the instruction-cache performance by aligning single-path loops with cache lines. The technique itself is old but has been used only for data caches. The properties of the single-path conversion of building single-path code allow us to adapt and implement the same technique also for instruction caches. The proposed algorithm quantifies which loops can be aligned and then estimates the minimal number of padded NOPs to improve the execution performance, thus reducing the number of cache misses. In lack of a compiler that supports the single-path conversion, we have been able to explore only the behavior of caches considering the number of the cache misses when single-path loops are executed. From the obtained result we saw that the number of cache misses for single-path loops is changed in a constant manner when the cache line is filled with NOPs. In the next step we plan to explore the cache misses for single-path loops that contain input-data independent branches.

Acknowledgment

This work has been supported in part by the European Community's Seventh Framework Programme [FP7] under grant agreement 287702 (MultiPARTES).

References

[1] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer, 2011.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problemoverview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1347375.1347389

[3] P. Puschner and A. Burns, "Writing temporally predictable code," in *Object-Oriented Real-Time Dependable Systems, 2002. (WORDS 2002). Proceedings of the Seventh International Workshop on*, 2002, pp. 85–91.

[4] M. Schoeberl, "Time-predictable computer architecture," *EURASIP J. Embedded Syst.*, vol. 2009, pp. 2:1–2:17, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1155/2009/758480

[5] P. Puschner, R. Kirner, B. Huber, and D. Prokesch, "Compiling for time predictability," in *Computer Safety, Reliability, and Security*. Springer, 2012, pp. 382–391.

[6] P. Ranjan Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "A data alignment technique for improving cache performance," in *Computer Design: VLSI in Computers and Processors, 1997. ICCD'97. Proceedings., 1997 IEEE International Conference on*. IEEE, 1997, pp. 587–592.

[7] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 177–189. [Online]. Available: http://doi.acm.org/10.1145/567067.567085

[8] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE, 1995, pp. 138–149.

[9] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, 1982.

[10] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[11] C. Lattner and V. Adve, "The llvm compiler framework and infrastructure tutorial," in *Languages and Compilers for High Performance Computing*. Springer, 2005, pp. 15–16.

[12] M. Schoeberl, P. Schleuniger, W. Puffitsch, F. Brandner, C. W. Probst, S. Karlsson, T. Thorn *et al.*, "Towards a time-predictable dual-issue microprocessor: The patmos approach," in *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, vol. 18, 2011, pp. 11–21.

[13] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[14] A. Asaduzzaman, N. Limbachiya, I. Mahgoub, and F. Sibai, "Evaluation of i-cache locking technique for real-time embedded systems," in *Innovations in Information Technology, 2007. IIT'07. 4th International Conference on*. IEEE, 2007, pp. 342–346.

[15] A. M. Campoy, A. Perles, F. Rodriguez, and J. Busquets-Mataix, "Static use of locking caches vs. dynamic use of locking caches for real-time systems," in *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*, vol. 2. IEEE, 2003, pp. 1283–1286.

[16] L. Wehmeyer and P. Marwedel, "Influence of memory hierarchies on predictability for time constrained embedded software," in *Design, Automation and Test in Europe, 2005. Proceedings*. IEEE, 2005, pp. 600–605.