A Code Transformation-Based Methodology for Improving I-Cache Performance of DSP Applications[1]

N. Liveris[*], *N. D. Zervas*[+], *D. Soudris*[§] *and C. E. Goutis*

*Northwestern University, Evanston, IL., USA, nikos@ece.nwu.edu
[+]ALMA Technologies, Athens, Greece, zervas@alma-tech.com
[§]Dept. of E.C.E., Democritus Univ. of Thrace, 67100 Xanthi, Greece, dsoudris@ee.duth.gr
Dept. of E.C.E., University of Patras, Rio 26500, Greece

Abstract- This paper focuses on I-cache behaviour enhancement through the application of high-level code transformations. Specifically, a flow for the iterative application of the I-Cache performance optimizing transformations is proposed. The procedure of applying transformation is driven by a set of analytical equations, which receive parameters related to code and I-cache structure and predict the number of I-cache misses. Experimental results from a real-life demonstration application shows that order of magnitude reductions of the number of I-cache misses can be achieved by the application of the proposed methodology.

1 Introduction

I-cache memories are widely used to bridge the increasing cycle-time gap between fast processing elements and relatively slow main memories. Although the use of instruction cache memories generally decreases the programs' execution time, some programs fail to use them effectively. Especially in cases that the usage of large I-caches is impossible or unaffordable, as in the area of embedded systems, it has been shown that I-cache performance dominates on system's cycle and energy budget [1,2]. To improve I-cache performance, the use of mini-caches [3] and modified prefetch mechanism [4] has been proposed in the past.

Apart from these techniques, high-level code transformations can be used to improve I-cache performance. In this paper, two high-level code transformations are proposed for this purpose, namely the widely known loop-splitting and function call insertion. Both transformations aim the optimization of the code located in the scope of loop nests. Loop splitting improves locality of instruction memory references, in cases that the code enclosed by a loop nest is also distributed after the application of the transformation. With function insertion, rarely executed code segments (e.g. code in the scope of a condition) that are contained in loop nests, are replaced by function calls. In this way, less capacity misses [5] occur in the I-Cache, since only the size of the most frequently executed code remains in the scope of the loop nests. Furthermore, an insight analysis, which results to the formulation of analytical equation that can be used to predict the number of I-cache misses, is presented. This analysis enabled the development of a systematic methodology for improving

Symbol	Definition
N_misses	The number of I-cache misses
Block_ size	The size of one block of the instruction memory cache in number of bytes
Icache_ size	The size of the instruction cache memory in number of bytes
C_size	The code size inside the scope of a condition in number of bytes. It is the code size that is executed only if the certain condition is true and it does not include the instructions for the logical operation evaluation and the conditional branching.
Fc_o_cs	The code size of a function call in number of bytes
Fc_o_t	The overhead of a function call in time
L_size	The code size contained in the scope of a loop nest in number of bytes. It includes the instructions that are placed inside the scope of the loop and the instructions for implementing the loop (condition checking, branching, loop iterator incrementing instructions)
N_iterat	The number of iterations of one loop
p_true	The ratio of the number of times a condition is true to the number of times this condition is checked.

Table 1: Basic Parameters

[1] This work was partially supported by the project IST-2000-30093 EASY of European Commission.

I-cache performance through the application of high-level transformation. Specifically, analytical equations are used to identify the conditions under which the application of a certain transformation has the desirable results. Based on that a flow for the iterative application of the I-Cache performance optimizing transformations is proposed. Experimental results from a real-life demonstration application show that order of magnitude reductions of the number of I-cache misses can be achieved by the application of the proposed methodology.

2 Loop Nests and I-Cache Misses

This section provides an insight analysis of a direct mapped I-cache behaviour. The analysis results in a set of analytical equations for I-cache misses, which are used to drive the proposed methodology. In Table 1 the basic analysis' parameters are defined. It is mentioned here that with the presented analysis it is assumed that all function are in lined and that no conditional branches are taken.

The analysis focuses on the most critical part of an application code, namely the code contained in loop nests. Specifically, three different cases of I-Cache behaviour are identified according to the relation among the code size contained in a loop nest and the I-cache size:

2.1 $L\_size \le ICache\_Size$

In this case there are no capacity misses [5] since the whole code of the loop can be placed in the cache. Therefore, the only misses that occur are the compulsory misses during [5] the first iteration of the loop (Fig.1). So, in this case the number of instruction cache misses is:

$$N\_misses = \frac{L\_size}{ICache\_size} \times \frac{ICache\_size}{Block\_size} = \frac{L\_size}{Block\_size} \qquad (1)$$

Loops that belong to this category will from now-on referenced as type-1 loops.

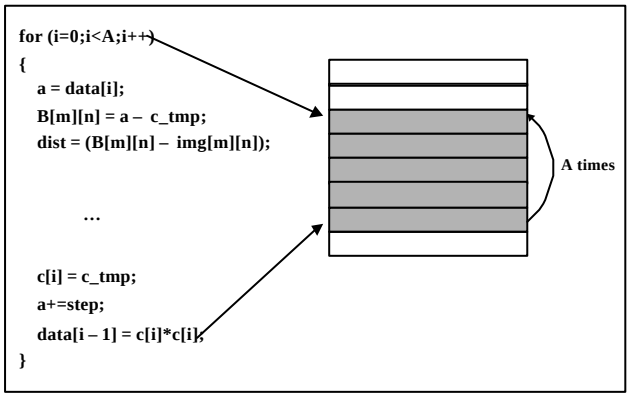

Figure 1: I-Cache Misses for $L\_size \le ICache\_Size$.

2.2 $ICache\_size < L\_size < 2 \times ICache\_size$

In this case not only compulsory misses will take place but also capacity misses will occur (Fig. 2). The number of compulsory misses from the first iteration will be the same as in the previous case, namely: $L\_size / block\_size$. In addition to this, for each of the next iterations a number of capacity misses will take place: $2 \times (L\_size \% iICache\_size) / Block\_size$. Thus, given that the number of iterations of the loop is N_iterat, the total number of instruction cache misses will be:

$$N\_misses = \frac{L\_size}{Block\_size} + (N\_iterat - 1) \times 2 \times \frac{L\_size \% ICache\_size}{Block\_size} \qquad (2)$$

The loop nests that belong to this category from now-on referenced as type-2 loops.

2.3 $2 \times ICache\_size \le L\_size$

In this case the number of compulsory misses of the first iteration equals the number of capacity misses of each of the next iterations (Fig. 3). So, the total number of instruction cache misses is:

$$N\_misses = N\_iterat \times \frac{L\_size}{ICache\_size} \times \frac{ICache\_size}{Block\_size} = N\_iterat \times \frac{L\_size}{Block\_size} \qquad (3)$$

The loops of this category will from now-on referenced as type-3 loops.

2.4 Taking into Account Conditional Branches

The number of misses in the presence of conditional branches is different only in the cases 2.2 and 2.3. Firstly, let us consider the case that $2 \times ICache\_size \le L\_size$. Assume that there is only one conditional branch with code size C_size, and that the condition has a true value in p_true percentage of the total times that it is checked. In such a case $L\_size - C\_size$ is the amount of the code that is executed in every iteration of the loop and C_size will be executed $p\_true \times N\_iterat$ times. Thus, the number of instruction cache misses during the execution of the loop will be:

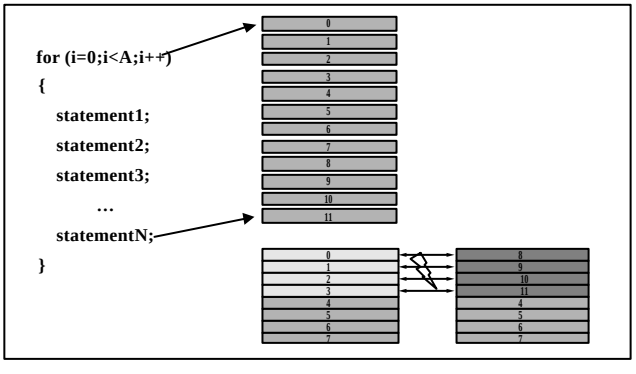

Figure 2: I-Cache Misses for $ICache\_size < L\_size < 2 \times ICache\_size$

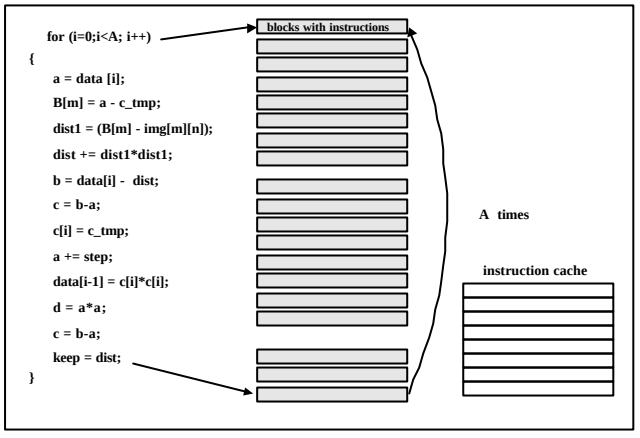

Figure 3: I-Cache misses for $2 \times ICache\_size \le L\_size$.

$$N\_misses = N\_itera \times \times \frac{(L\_siz - C\_size) + p\_true \times C\_size}{ICache\_size} \times \frac{ICache\_size}{Block\_Size} =$$

$$= N\_iterat \times \frac{(L\_size - C\_size) + p\_true \times C\_size}{Block\_size} \quad (4)$$

Now, if there are M conditional branches inside the loop, each one with a different p_true and a different C_size, then the number of misses will be:

$$N\_misses = N\_iterat \times \times \frac{\left(\left(L\_size - \sum_{i=1}^{M} C\_size_i\right) + \sum_{i=1}^{M} p\_true_i \times C\_size_i\right)}{Block\_size} \quad (5)$$

where C_size_i and p_true_i are the code size and the being true percentage of the i[th] condition.

Generally speaking, the second term of Eq. 5 contains the size of the code for each separate code part multiplied with the ratio of the number of times this part is executed to N_iterat.

In the second case, where the inequality

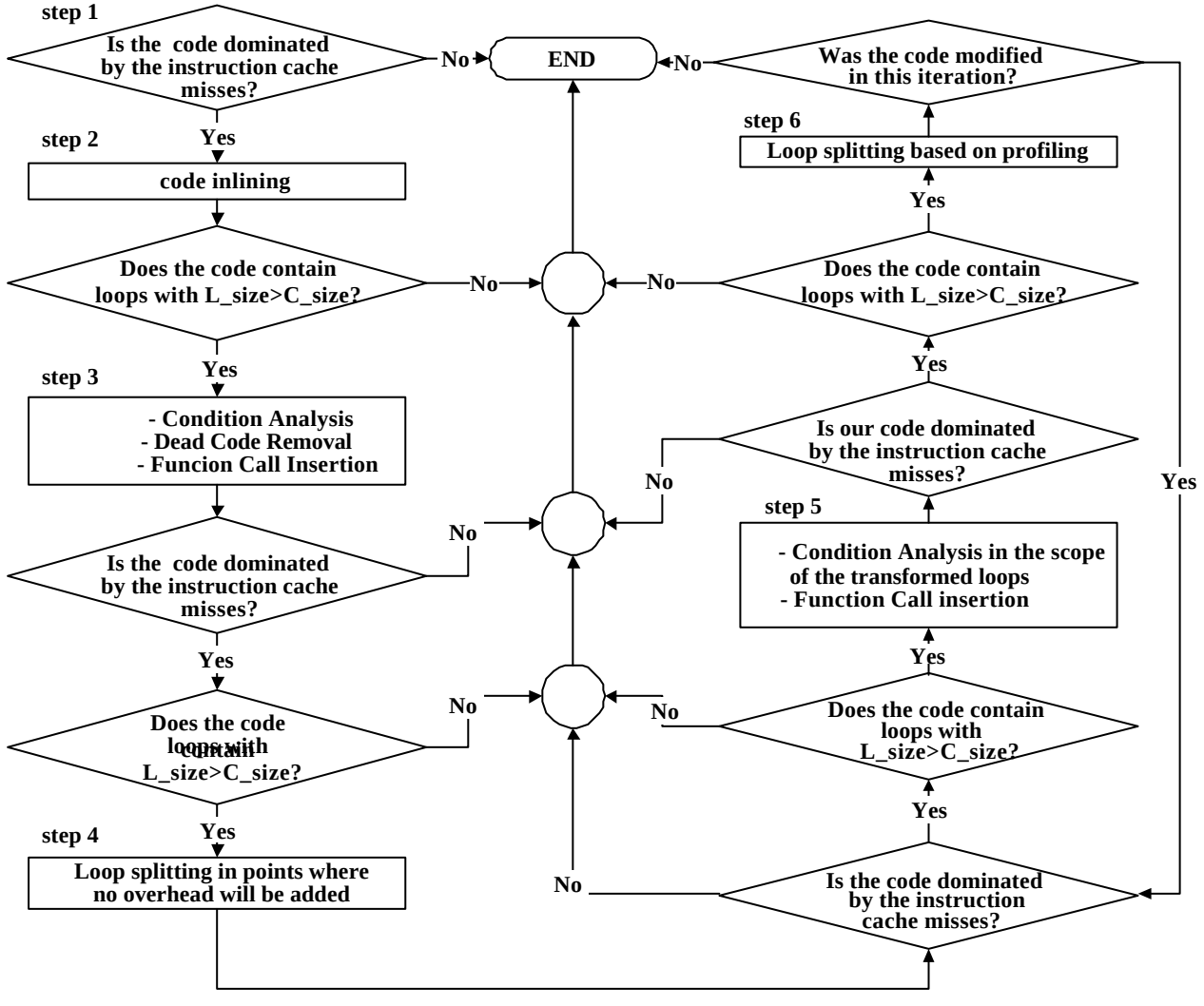

Figure 4: The proposed methodology flow.

$ICache\_size < L\_size < 2 \times ICache\_size$ is valid, things are much more complicated, since the number of misses also depends on the position of the conditional branch inside the loop. If the condition and its scope are placed in the blocks of the cache, for which only compulsory misses occur, then the number of instruction cache misses is the same as in the case where no conditions are contained in the loop. In order for the condition to be placed in one of these blocks, the following inequalities have to be valid: 1. *(code size from the beginning of the condition to the end of the loop)* $\leq$ *ICache_size, 2. (code size from the beginning of the loop to the end of the condition)* $\leq ICache\_size$.

On the other hand, the number of instruction cache misses is reduced when the condition is placed in a block, for which capacity misses occur. In this case the number of instruction cache misses is:

$$N\_misses = \frac{2\,N\_iterat\,p\_true\,(L\_size\%ICache\_size)}{Block\_size} + \frac{2N\_iterat(1-p\_true)((L\_size\%ICache\_size)-C\_size)}{Block\_size} \quad (6)$$

3 The Proposed Methodology

In this section the proposed methodology (Fig. 4) is described. The first step of the proposed methodology aims to determine whether the instruction cache misses play an important role in the total CPU time of the code. It is known that [6]:

$$CPU\_exec\_time = (CPU\_clock\_cycles + Memory\_stall\_cycles) \times Clock\_cycle \quad (7)$$

The number of memory stall cycles depends on both the number of misses and the cost per miss, which is called the miss penalty:

$$Memory\_stall\_cycles = (N\_ICache\_misses + N\_DCache\_misses) \times Miss\_penalty \quad (8)$$

The total execution time that is caused from the memory accesses (hits and misses) can also be calculated:

Total execution time due to memory accesses =
N_ICache_hits ´ cost of a hit (f1) + N_ICache_misses ´ cost of a miss (f2) + N_DCache_ hits ´ cost of a hit (f3) + N_DCache_misses ´ cost of a miss (f4)

From these 4 factors, that are added to form the total execution time due to memory accesses, only the second may be dramatically decreased by the application of the proposed methodology. The others will remain the same or will be slightly changed. Therefore, it is very important at the first step to examine whether the number of instruction cache misses is large enough to justify the application of the methodology. At this first step an approximation of the upper bound of the gain, which may be obtained by applying the methodology, can be even calculated using Amdahl's law:

$$Execution\_time_{new} = Execution\_time_{old} \times \left((1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right) \quad (9)$$

where: $Fraction_{enhanced}$ *= (f2) /(total number of execution time due to memory accesses)* and the expected $Speedup_{enhanced} \leq 100$.

So, the steps of the proposed methodology, which evaluate whether the code execution time is dominated by the performance of the instruction cache, are actually evaluations of equation (7) for which the number of instructions cache misses are taken into account.

As a second step the code of the functions, which are called inside the scope of the most critical loop nests, are in-lined. This can be considered as a pre-processing step that is necessary in order to reveal the structure of the code. The identification of the critical loop nests imposes the detection of the loops that have the largest number of iterations and the measurement of the code size enclosed by these loops nest. This process can be done statically for the loop nests that have manifest conditions and their number of iterations is known at compile time. However, for the loop nests with data-dependent conditions the mean number of iterations can be derived with profiling. If the code size of some loop nests is bigger than the size of the instruction cache, then there is a great chance that our code can be heavily optimized by this methodology. In this case the third step should be applied.

At the third step of the proposed methodology condition analysis takes place. Condition analysis aims to determine the values of p_true and C_size for all code segments that are conditionally executed. In this way, "dead code" is detected and opportunities for function call insertion are revealed. Specifically, after the functions are in lined, many conditions will be brought in different contexts, where a part of them may never be true. Therefore, the conditions, for which it can be derived at compile time that they never become true, are removed. By removing these conditions with the code inside their scope, a big decline in the code size of the scope of loops may be achieved without the addition of any kind of overhead. Another benefit from "dead code" removal is that the number of statements will be decreased and so the complexity of applying the next steps will also be reduced. Furthermore, given the values of p_true and C_size for the code segments that are conditionally executed, it can be derived the replacement of which of them with a function call will lead to reduction of the number of I-cache misses. Specifically, if a condition is found with a small p_true, then its scope is replaced with a function call, as in Fig. 5. In such a case, the code size enclosed by the outer loop is:

$$L\_size_{new} = L\_size_{old} - (C\_size - fc\_o\_cs) \quad (10)$$

where fc_o_cs is the code size overhead due to the added function call.

Now, let us assume that the inequalities: A1) $ICache\_size < L\_size < 2 \times ICache\_size$, A2)*(code size from the beginning of the condition to the end of the loop) £ ICache_size,* A3)*(code size from the beginning of the loop to the end of the condition) £ ICache_size* are valid (Fig. 5).

Under these assumptions, the number of capacity misses before function call insertion is applied is:

$$N\_misses_{(capacity)-old} = 2(N\_iterat-1) \times \times L\_size_{old} \% ICache\_size / Block\_size \quad (11)$$

After the transformation, L_size is reduced and the number of capacity misses, when the condition is not true, is:

$$N\_misses_{(capacity)-new} = = 2(N\_iterat-1)\frac{L\_size_{new} \% ICache\_size}{Block\_size} = = 2(N\_iterat-1) \times \times \frac{(L\_size_{old} - (C\_size - fc\_o\_cs)) \% ICache\_size}{Block\_size} \quad (12)$$

Therefore, the number of instruction cache misses, when the condition is not true, is reduced by:

$$\Delta(N\_misses) = \frac{2(N\_iterat-1)((C\_size - fc\_o\_cs) \% ICache\_size)}{Block\_size} \quad (13)$$

However, by inserting this function call a certain overhead is added to the code. This overhead is added every time the condition becomes true and the function is called. We will note the overhead in execution time due to a function call as *fc_o_t*. So, the criterion for applying this transformation is:

$$(1 - p\_true) \times \Delta(N\_misses) \times Miss\_penalty > p\_true \times \times N\_iterat \times fc\_o\_t \quad (14)$$

On the left side of this inequality is what the obtained gain in time by this transformation and on the right side is the paid penalty. In other words, in the proposed methodology the third step is completed with the application of this transformation in every part of the code, where the three assumptions (A1- A3) and the inequality (12) are valid. The application of this transformation is proposed during the third step, because it leads to a reduction of the code size and hence to a reduction of the complexity for the subsequent steps.

The fourth step of the methodology is loop splitting in the parts of the code where no overhead will be added. The aim of this step is to split a type 2 or type 3 loop to two different loops, from which at least one will be a type 1 loop. And in this fourth step the loop splitting should be done in parts of the code where no overhead will be introduced. That means that there should not be many data-dependencies between the two new loops and therefore it won't be necessary to introduce a new array, which will carry the intermediate results from the first loop to the second. Then the only overhead that will be added to the code is the additional loop instructions. Therefore, we can write the following equation:

$$L\_size_{old} + loop\_instructions\_overhead = = L\_size_{new}1 + L\_size_{new}2 \quad (15)$$

But now at least one of the two new loops is a type 1 loop. For example, if the loop is split into one type 3 and one type1 loop, no capacity misses will occur during the execution of the second loop. So the number of instruction cache misses that take place in the transformed code will be:

$$N\_misses_{new} = \frac{N\_iterat \times L\_size1}{Block\_size} + \frac{L\_size2}{Block\_size} \quad (16)$$

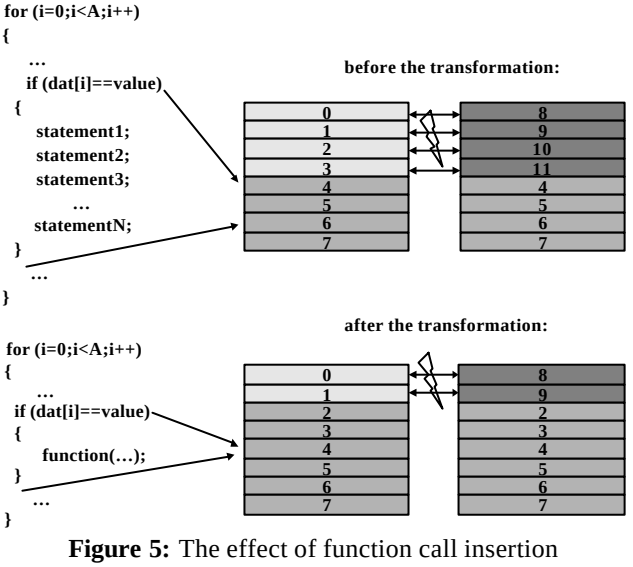

Figure 5: The effect of function call insertion

The second term is the number of the compulsory misses due to the execution of L2. For simplicity we assume again that no conditional branches are included inside the scope of the loops. With the initial form of the code the number of instruction cache misses was:

$$N\_misses_{old} = \frac{N\_iterat \times L\_size}{block\_size} \quad (17)$$

From Eq. 14 and 15, it can be derived that, since *L_size > L_size1* and *N_iterat >> 1,* the number of instruction cache misses is significantly reduced.

As a next stage of the methodology the process of measuring the code size inside the loops is repeated. If some loops are transformed in step 4, the code size inside the scope of the transformed loops has to be measured. The aim of this step is actually to determine whether there are still loops with code size bigger than the size of the

cache (type 2 and type 3 loops). If such loops exist, it makes sense to continue our process and try to optimize them as well. Otherwise, the application of the methodology has to be terminated.

In step 5 the transformations that were introduced in the third step, has to be applied again, but this time on the transformed loops. As mentioned above, after loop splitting the code size inside some loops may be changed. That means, that there is a possibility that one assumption of the A1 – A3 has become valid due to the transformations in a part of the code where it was not valid before. This will enable the application of the replacing transformation on the scope of the conditional branch. That is the reason, that this process is proposed as the sixth step after the loop transformations of step 4.

In step 6 loop splitting will be done with the same purpose as in step 4 (type 2 or type 3 loops to type 2 or type 3 with reduced code plus type 1 loop). At this point there are not going to exist other code parts where loop splitting can be done without the addition of a serious overhead to the data memory. That is why every decision taken at this step has to be based on results from profiling. With the help of profiling or even better with the help of a simulator the achieved gain and the paid penalty will be determined for every transformation. This is of course a much more complex procedure than the one followed at step 4, since it involves profiling. Moreover, it is probably the least promising step because it introduces additional overhead in data memory and that is the reason that it should be applied after all the other six steps are completed.

If the code is modified at the seventh step, going back to step 5 is necessary. The application of these last three steps has to be done iteratively, until no more transformations should be applied. Then the application of the proposed methodology is terminated.

It is very important to state that after every transformation done on the code at every step, the number of instruction cache misses has to be measured again. If this number falls off at a great percentage, there is a chance that the total cost coming from the instruction cache misses may become less than one of the other factors of this equation. At this case the proposed methodology was successfully applied, since the biggest factor in cost was reduced so much that it became unimportant compared with the other factors. The methodology has to be terminated and other transformations should be applied that will reduce the cost from the other factors.

4 Applying the Methodology in an Real-Life Application

In this section the proposed methodology will be used to transform the code of a real life application, namely the row-column 1D Discrete Wavelet Transform assuming an input length equal 4096 and 5 levels of decomposition. The original specification of this algorithm was 255 lines of C code. A DLX [6] processor with an instruction cache of 512 Bytes and a data cache of 512 Bytes, each of which with block size equal to16 Bytes, has been used as execution platform. For translating the code the DLX compiler dlxcc has been used. In addition to this tool, the DLX simulator dlx sim and the cache simulator dinero [7] were used for taking the measurements that will be presented in this section.

As a first step the execution of the initial form of the code is simulated. The results can be seen in table 2. The number of instruction cache misses is almost 15 times larger than the number of data cache misses and approximately 25% of all the instruction memory fetches. Therefore, it makes sense to try to optimize this code by using the proposed methodology. At the second step we inline the code of the used functions inside the loops. After in-lining it can be derived that there are loops with much bigger code size than the size of the instruction cache and which have a large number of iterations. So, by applying the third step, many conditions has are detected, for which it can be derived that they will never be true during the execution of the program. Consequently, these if-constructs together with the code, which is placed inside the scope of these conditions, are removed. As a next stage, condition analysis follows. In this code there are no conditions left with small *p_true* and large code size, for which the transformation of replacing their scope with a function is justified. From table 2 it can be derived that the number of fetches to instruction memory has been seriously reduced by the application of this step. The reason for that is that the conditions, which were removed, were placed in the scope of the most critical loops. Therefore, they were executed many times and their removal has lead to large savings in instruction memory accesses. The number of instruction cache misses has also fallen off. This fact is expected, since from this transformation the code size in the loops is reduced and fewer capacity misses occur. Moreover, the number of accesses to data memory fell off, because every condition checking involved accesses to data memory for two operands at least. The number of data cache misses is not seriously affected. Since loops with bigger code size than the cache size still exist in our program, it makes sense to proceed to the next stage of the proposed methodology.

In step 4 loop splitting without the introduction of an overhead will be applied on the code. The process begins with the loop with the largest code size and the biggest number of iterations. This loop is transformed from one type 2 into three type 1 loops. As it can be seen in table 2, the number of instruction misses is now dramatically decreased. With this transformation almost 93% of the instruction cache misses do not occur anymore. This result should be expected, since capacity misses, which were the large majority of the misses, do not occur anymore in this part of the code. Only compulsory misses

take place while the three new loops are executed. Due to the inserted instructions for the separate loop execution the number of fetches from instruction memory and the number of accesses to the data memory are increased, but less than 10%. The number of data cache misses is also increased by 5%. To sum up, the total number of accesses to instruction and data memory was increased by 161,400 and the total number of misses was reduced by 105,395.

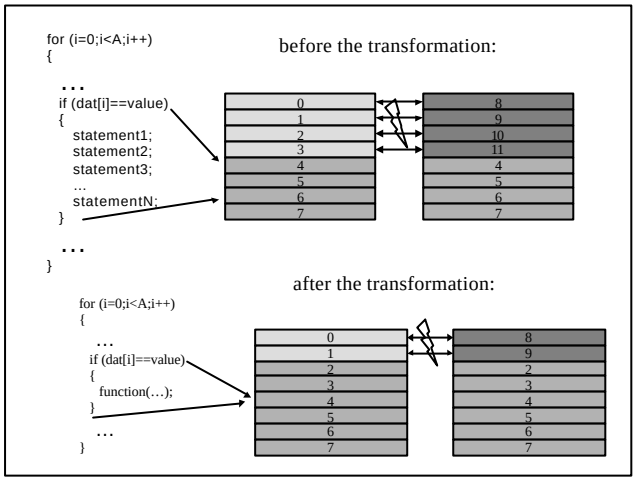

Figure 5: The effect of function call insertion.

At this step, the number of instruction cache misses is much smaller than the number of data cache misses. For that reason, the proposed methodology has to be terminated at this point.

5 Conclusions – Future Work

In this paper, a flow for the iterative application of the I-Cache performance optimizing transformations has been proposed. The procedure of applying transformation is driven by a set of analytical equations, which receive parameters related to code and I-cache structure and predict the number of I-cache misses. Experimental results from a real-life demonstration application showed that order of magnitude reductions of the number of I-cache misses can be achieved by the application of the proposed methodology. Since with the proposed methodology, decisions are made using analytical equations which parameters that can be acquired at compile time, future work will be targeted towards the automated application of the proposed methodology.

References

[1] N. D. Zervas, K. Masselos, C.E. Goutis, "*Date Reuse Exploration for Low Power Realization of Multimedia Applications on Embedded Cores*", in the Proc. of 1999 IEEE International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS'99), pp. 71-80, Oct, 1999.

[2] M. Dasygenis, et al. "*Data and instruction memory exploration of embedded systems for multimedia applications*", accepted for publication in 2001 IEEE Computer Society Annual Workshop on VLSI (WVLSI), Orlando Florida, USA, April 2001.

[3] N. E. Bellas, I. N. Hajj and C. D. Polychronopoulos, "*Using dynamic cache management techniques to reduce energy in general purpose processors*", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 8, No. 6, pp. 693 - 708 December 2000.

[4] H. C. *Young, E. J. Shekita, "An intelligent I-cache prefetch mechanism"*, in 1993 IEEE International Conference on Computer Design (ICCD'93), pp. 44 – 49, Oct. 1993.

[5] S. A. Przybylski, CACHE AND MEMORY HIERARCHY DESIGN – A Performance Directed Approach, Morgan Kaufman Publishers, 1990.

[6] J. L. Hennessy and D. A. Patterson, Computer Architecture – A Quantitative Approach, Morgan Kaufman Publishers, second edition, 1996.

[7] Alvin R. Lebeck, David A. Wood, *"Cache Profiling and the SPEC Benchmarks: A Case Study"*, IEEE Computer, vol. 27, No. 10, pp. 15-26, 1994.

	#I-Mem. Accesses	#D-Mem. Accesses	Total Number of Accesses	#I-Cache Misses	#D-Cache Misses	Total Number of Misses
Original code	2,229,916	631,918	2,931,834	568,320	38,330	606,650
After step 3	1,424,722	489,916	1,914,638	115,598	40,746	156,344
After step 4	1,547,698	528,340	2,076,038	8,275	42,674	50,949

Table 2: Experimental Results.