Code Positioning for VLIW Architectures

Andrea G.M. Cilio and Henk Corporaal

Delft University of Technology
Computer Architecture and Digital Techniques Dept.
Mekelweg 4, 2628CD Delft, The Netherlands
{A.Cilio,H.Corporaal}@et.tudelft.nl

Abstract. Several studies have considered reducing instruction cache misses and branch penalty stall cycles by means of various forms of code placement. Most proposed approaches rearrange procedures or basic blocks in order to speed up execution on sequential architectures with branch prediction. Moreover, most works focus mainly on instruction cache performance and disregard execution cycles. To the best of our knowledge, no work has specifically addressed statically scheduled ILP machines like VLIWs, with control-transfer delay slots.
We propose a new code positioning algorithm especially designed for VLIW-style architectures, which allows to trade off tighter schedule for program locality. Our measurements indicate that code positioning, as a result of tighter program schedule and removed unconditional jumps, can significantly reduce the number of execution cycles, by up to 21%, while improving program locality and instruction cache performance.

1 Introduction

Several trends of today's architectures contribute to make the instruction memory system a performance bottleneck. Reduced instruction set architectures have almost doubled [1] the instruction fetch bandwidth requirements. (Instruction-level parallel machines like VLIWs and superscalars further increase the required bandwidth.) The gap between memory and processor speed has incremented the memory cycle latency and consequently the cache miss penalty. Finally, deep pipelines and high ILP levels have increased the penalty which a processor can incur when a control transfer instruction is executed. These trends render several compile-time optimizations, like compile-directed prefetching of code and data, loop transformations for data caches, and code positioning critical for achieving optimal performance. This paper addresses *code layout*, or positioning for VLIW-style architectures. By positioning the code blocks adequately, the following improvements can be obtained:

1. *More effective use of the instruction cache.* The spatial locality of instruction cache accesses can be increased, while reducing the number of conflict misses, thereby improving cache utilization [2][3][4].
2. *Reduction of branch penalty overhead.* For architectures with branch prediction basic blocks can be arranged so as to reduce the branch penalty [5][6][7].
3. *Reduction of unconditional branches.* The basic blocks can be rearranged so as to eliminate frequently executed unconditional jumps [5][6].

In contrast to architectures with branch prediction, some recent VLIWs machines, like Trimedia TM1000 [8] and the TI C62x family [9] expose the branch penalty to the compiler by introducing several branch delay slots. One of the goals of code positioning for these architectures is then to maximize the number of useful operations that can be imported into delay slots from other basic blocks while minimizing the number of frequently executed delay slots that cannot be sufficiently filled. Application-specific architectures may not even have an instruction cache: the whole program is downloaded to a local, fast-access instruction memory before starting execution. Clearly, for such machines the locality of the program code is irrelevant. This gives a degree of freedom in the selection of the branch direction, which can be exploited for reducing the execution cycle count.

These considerations suggest that the cost model and the goals of code positioning for a statically scheduled, VLIW-style machine with control transfer delay slots may differ from those found by previous works. The purpose of this paper is to explore alternatives to the existing code positioning algorithms specifically suited for this class of machines (with different instruction cache configurations), and to investigate the effects on the program schedule.

This paper is organized into 5 sections. Section 2 reviews previous work on code positioning. Section 3 presents our code positioning algorithm, which is evaluated on a group of benchmarks in Section 4. Section 5 summarizes the results.

2 Related Work

Code positioning can be applied at various levels of granularity. Early work in this direction focused on reducing page faults of virtual memory machines by positioning memory pages. Later, as the instruction cache began to play a critical role in the overall performance, the attention shifted towards finer levels of granularity, namely single procedures, segments of procedures and single basic blocks. Most of the techniques mentioned below, as ours, base their algorithms on profiling information gathered by previous program executions using representative input data.

In [3] Hwu *et al.* present a set of compiler techniques that improve instruction cache performance; among these are *Function layout* (i.e., basic block positioning) and *Global layout*, which tries to arrange whole procedures in a sequential order that minimizes conflict misses. McFarling [10], who also addresses both levels of granularity, proposes a code positioning algorithm that minimizes conflict misses; he also proposes to avoid caching instructions that are not frequently executed. Pettis and Hansen [5] present a refined version of Hwu's algorithm and evaluate also the reduction of branch penalty cycles and executed instructions. Gloy and others [4] build on the work of Pettis by taking into account the *temporal* ordering of executed procedures. Their work is restricted to procedure placement (i.e., global layout). Mendlson *et al.* [2] consider reducing of cache misses due to conflicting blocks in a loop. Their approach uses the concept of *abstract caches* to place a set of blocks of the same loop in a conflict-free manner. Controlled code duplication is used when multiple uses of the same blocks in different loops would conflict with other blocks in the respective loop. Differently from other works, this approach is directed by a static cost model and does not use program profil-

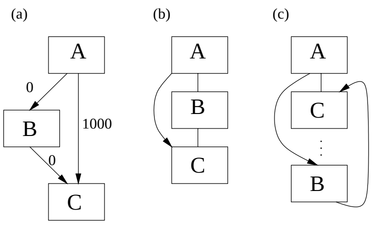


Fig. 1. Example of Code Placement. The numbers next to the edges represent execution counts obtained from profiling.

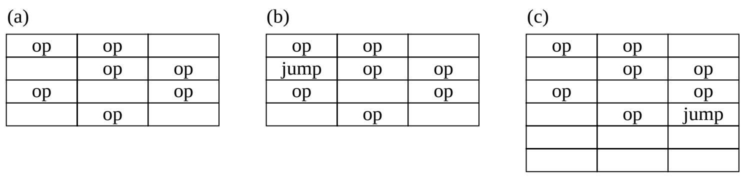
ing. Calder and Grunwald [6] exclusively address branch cost reduction. This restricted form of the code positioning problem, termed *branch alignment*, does not consider program locality. The authors propose improved models of the branch costs that expose the branch prediction schemes of the underlying machine microarchitecture. Young *et al.* [7] modeled the branch alignment optimization problem as a Directed Traveling Salesman Problem (DTS) and attained near-optimal speedup.

All these works evaluate sequential architectures, as opposed to explicitly programmed ILP machines. Furthermore, all these works have mainly been aimed at reducing the instruction cache misses or the branch penalty. We will show that for VLIW architectures with branch delay slots code positioning has a significant impact on the program schedule and the execution cycle count.

3 Code Positioning Algorithm

Traditional compilers generate code in which segments rarely executed during typical runs are interspersed with frequently used parts. Often, infrequent code is executed to handle exceptional situations. See for example block B in the control flow graph (CFG) of Fig.1(a). A traditional compiler would place this CFG as shown in Fig. 1(b), thus forcing the processor to take a branch every time block A is entered. Figure 1(c) shows a better code placement of the same CFG. In this case, the branch will never be taken, and the control flow will fall through to block C. Even in architectures (like VLIWs) for which the branch penalty is identical in both paths, the second layout is preferable, because it increases the probability that blocks A and C end up in the same cache line.

We consider the problem of code positioning in the context of explicitly programmed ILP architectures of the VLIW type in which control transfer instructions have architecturally visible delay slots. Delay slots are an architectural feature to hide the penalty of control transfer instructions. The scheduler tries to fill the delay slots with operations always useful, which may logically precede the control transfer or may belong to basic blocks that are dominated by it. In machines with jump delay slots the scope and the algorithm of the instruction scheduler interact with code positioning and affect its effec-



(a)

op	op	
	op	op
op		op
	op	

(b)

op	op	
jump	op	op
op		op
	op	

(c)

op	op	
	op	op
op		op
	op	jump

Fig. 2. Effect of adding a jump to a basic block: (a) original, partially scheduled code; (b) jump added with zero cost; (c) jump added with cost proportional to $L = 2$.

tiveness. Our scheduler uses the *region* scheduling scope [11][1] which allows to extract parallelism from several paths and move operation across join points. The scheduler first schedules the current basic block with a list-based algorithm and then tries to import operations from successor basic blocks in order to fill the unused operation slots and possibly reduce the overall path length to the successors.

We implemented code positioning as a pre-scheduling pass because of its simplicity. A post-scheduling pass would require partial rescheduling. Though simpler, a pre-scheduling pass leaves us with the difficulty of finding an accurate cost model to guide positioning. Recent works suggest that to achieve better results in cache miss reduction, a cost model based on cache and basic block size must be used [2][4]. However, this is not possible in a VLIW instruction scheduler with global scope because the block sizes depend on the program scheduling and are very difficult to estimate. Some blocks can be enormously enlarged[2], while other blocks may disappear altogether, absorbed into predecessor blocks owing to operation importing. To make things worse, the exact schedule itself depends on the code layout, as will be shown below. For these reasons, we will use the proximity heuristic to minimize cache misses: blocks that are placed closer have less chances of competing for the same cache line.

Cost Estimates. Differently than for architectures with branch prediction [6][7], the exact cost of branches and unconditional jumps strongly depends on the schedule. For example, the cost is very low if the scheduler succeeds in placing it in an empty operation slot L cycles before the end of the basic block, where L is the number of delay cycles that follow a jump or a branch. See the example in figure 2(a–b); each tile represents an operation slot and "op" indicates that a slot is occupied.

The goal of our algorithm is to arrange the basic blocks of a procedure in an order that minimizes the number of operations executed and maximizes the locality of the instruction fetch. This objective can be reformulated as the problem of deciding for each edge $e = a \rightarrow b$ whether block b must be the sequential successor of block a.

[1] Regions, which correspond to loop bodies, are the most general scheduling scope, in the sense that traces, superblocks and trees are also regions. Our schedule supports speculation and guarding, therefore can achieve an effect equivalent to hyperblock if-conversion.

[2] This can be due, e.g., to the presence of dependencies from long-latency operations, or to the fact that a block is a duplication point for operations imported up along another path.

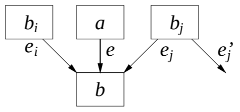


Fig. 3. Example of Control Flow Used to Determine the Cost Model.

This decision is guided by a function which estimates the cost of placing b immediately below a.

Definitions. In order to describe the algorithm, the following definitions are given:

1. $CFG(B, E, s)$ is a directed cyclic graph representing a procedure;
2. nodes $b \in B$ are the basic blocks of the procedure;
3. directed edges $b_i \to b_j \in E$ ($b_i, b_j \in B$) are the control flow edges from a source block to a destination block;
4. let $b \in B$, then $pred(b) = \{b_i : \exists b_i \to b \in E\}$, $succ(b) = \{b_i : \exists b \to b_i \in E\}$;
5. $\exists! s \in B : pred(s) = \emptyset$; i.e. there is one and only one block (the entry point of the procedure) that does not have predecessors;
6. to each edge $e \in E$ is attached a weight f_e that gives the execution count obtained from the profiling.

We will begin with a cost model that considers only the execution cycles and ignores locality. Let us consider an edge $e = a \to b$ of an unconditional jump, as shown in fig. 3. If we knew the exact effect on scheduling of making b the sequential successor of a, we would be able to compute the cost of this decision. Given the complexity of our operation-based, global list scheduler [11], it is not possible to predetermine the effect of reordering a block without backtracking. We therefore resort to a cost estimate.

Let $E_u \cup E_c \subset E$ be the set of incoming edges of b; E_u is the set unconditional edges, while E_c is the set of edges which come from a basic block terminating with a conditional branch; f_e is the frequency of execution of the control flow edge e. As a first approximation, we consider only the *local* effects of placing b in the fall-through path of a, i.e., consider only the effects on the schedule of the predecessors of b and ignore the other basic blocks. The general formula of the local cost (estimated number of additional cycles required) of taking b as sequential successor of a is then:

$$SeqCost(e) = \sum_{e_i \in E_u \cup E_c} \phi(e_i) - \phi(e). \tag{1}$$

where $\phi(e)$ is the cycle penalty associated with an edge e when it cannot be selected as fall-through path. The cycle penalty depends on the characteristics of the source and destination blocks of the edge e. For an unconditional edge $\phi(e) = Lf_e \cdot c_u(e)$, where L is the number of delay slots of a control flow operation and $c_u(e)$ depends on how early the scheduler can place the jump operation of e (compare figure 2(b) and 2(c)). For a conditional branch edge the value of $\phi(e)$ is more problematic, because it depends on the placement of the alternative edge:

$$\phi(e) = \begin{cases} 0, & \text{if the other branch edge } e' \text{ is sequential} \\ (L+1) \cdot \min\{f_e, f_{e'}\}, & \text{if neither branch directions is fall-through.} \end{cases} \qquad (2)$$

if either branch direction can be selected as fall-trough path then no unconditional jump needs to be inserted. Equation (2) takes the minimum frequency between the two branch edges because if a basic block with an unconditional jump must be inserted, it is always possible to assign the less frequent direction to this jump. Note that it is possible to have branches for which neither direction is fall-through. As noted in [6], in some cases such branch alignment can be preferable.

The expressions for $SeqCost(\cdot)$ and $\phi(\cdot)$ are local approximations: since the decisions taken in other blocks of the procedure affects the scheduling globally, $\phi(\cdot)$ depends on the ordering of all blocks of the scheduling scope. Given a basic block b, its best sequential predecessor is indicated by the edge $e \in pred(b)$ that minimizes $SeqCost(\cdot)$. If we assume that the sum terms in (1) are not affected by the positioning, then the minimization becomes a problem of maximizing the only term that is different in each expression. For unconditional edges the best fall-through path is then the most frequent path, as in the greedy algorithms of the literature [5][6]. This path is also the best choice from the point of view of program locality, given our proximity heuristic. For branch edges $\phi(\cdot)$ completely depends on the placement of the alternative edge. On the one hand, we would like to postpone the decision for less critical branches until we know the placement of one of the two edges and we can use (2) to estimate the local costs. On the other hand, a critical branch edge should be decided first.

This problem motivated the introduction of a *priority function* to determine the order in which edges are selected for placement. Our priority (applied to blocks) ensures that outgoing edges of more critical blocks are selected first, thereby postponing the decision for less critical branch edges until we might have a better estimate for $\phi(\cdot)$. As priority, we take the frequency for the blocks with unique outgoing edge, and the following function for the blocks with outgoing branch edges e, e':

$$prio(b) = \min\{f_e, f_{e'}\}. \qquad (3)$$

For the cost of a branch edge when the placement of the alternative edge is unknown we take the following formula, in which $c_b \in [0, 1]$ reflects the probability that the branch needs an additional unconditional jump: $\phi(e) = c_b(L + 1)\min\{f_e, f_{e'}\}$.

These cost and priority functions are not adequate if program locality is also to be taken into account. If only the program locality is relevant, a more suitable priority is simply based on the edge execution frequency, therefore a block terminating in a branch has the same priority of its most frequent outgoing edge:

$$prio(b) = \max\{f_e, f_{e'}\}. \qquad (4)$$

Similarly, the cost $\phi(\cdot)$ for a branch edge uniquely depends on its frequency, like for unconditional edges: $\phi(e) = Lf_e$.

3.1 Algorithm Description

Our algorithm can be divided into two parts; the first part sorts the basic blocks for later selection according to a priority, the second part selects the basic blocks and places them according to the local cost estimate described above.

First, a priority is assigned to every basic block of a procedure. For blocks with a unique successor, the priority is proportional to the execution count of the jump. Blocks which end with a conditional branch are assigned the following priority, which combines (3) and (4):

$$prio(b) = \alpha \cdot \min\{f_e, f'_e\} + (1-\alpha) \cdot \max\{f_e, f'_e\}. \tag{5}$$

The parameter $\alpha \in [0,1]$ allows to balance execution count reduction $(\alpha = 1)$ against enhancement of program locality $(\alpha = 0)$. The latter choice results in the same selection order of the greedy algorithms found in the literature.

Algorithm 1 places the basic blocks of CFG by assigning a label to every edge $e \in E$. Initially, all edges are labelled as undefined. An edge $e = b_i \rightarrow b_j$, is labelled SEQ if b_j is the sequential successor of b_i; vice versa, $label(e) = NSEQ$ if b_j does not follow b_i. The actual placement is performed after all edges have been assigned a label and merely involves moving pointers to list elements around; this pass inserts any unconditional jumps and basic blocks necessary to ensure that the code executes correctly. The function BestCandidate estimates the sequential cost with (1) and returns the destination block with smaller cost. The estimation of $\phi(e)$ for a branch edge e combines execution cycle reduction and locality enhancement objectives:

$$\phi(e) = \begin{cases} 0, \text{if only the other edge, } e' \text{ is sequential,} \\ c_b(1-\alpha)Lf_e + c_b\alpha(L+1) \cdot \min\{f_e, f_{e'}\}, \text{otherwise.} \end{cases} \tag{6}$$

Note that at this point $\phi(e)$ may be exactly determined for some branch edges because their branch has been already aligned. When both branch directions are known not to be fall-trough we take $c_b = 1$.

Like the greedy algorithm proposed by Pettis and Hansen [5], this algorithm is able to restructure the placement of loop basic blocks so that a loop header appears in the middle of the loop. In addition, this algorithm partially solves the problem indicated in [6], whereby a conditional edge is given precedence over an unconditional edge with the same frequency, without need to exhaustively search several possible orders, simply because unconditional paths can be given priority over conditional edges.

A few additional, practical considerations are in order. For blocks that contain indirect (computed) jumps the positioning of the successors is relevant only for the locality. All successor edges are marked $NSEQ$ before starting LayoutBasicBlocks. This improves the placement of other blocks with branches. We must check for cyclic chains of edges marked SEQ. If such a chain would be formed by assigning a SEQ label, the edge is assigned label $NSEQ$ (and the cycle is broken).

4 Evaluation

Our placement algorithm has been evaluated in a series of experiments. This section first describes the method used to perform our measurements and the target machines

Algorithm 1 LayoutBasicBlocks (CFG).

```
AssignLabels($CFG$)
for all $b \in B$, sorted by $priority(\cdot)$ do
  if $|succ(b)| = 1 \wedge \neg label((b, b_1)) = NSEQ$ then
    $label((b, b_1)) = SEQ, \quad label((b', b_1)) = NSEQ \quad \forall b' \in pred(b_1), b' \neq b$
  else if $|succ(b)| = 2$ then
    if $label((b, b_1)) = NSEQ \wedge label((b, b_2)) = NSEQ$ then
      // Nothing to do
    else if $label((b, b_1)) = NSEQ$ then
      $label((b, b_2)) = SEQ, \quad label((b', b_2)) = NSEQ \quad \forall b' \in pred(b_2), b' \neq b$
    else if $label((b, b_2)) = NSEQ$ then
      $label((b, b_1)) = SEQ, \quad label((b', b_1)) = NSEQ \quad \forall b' \in pred(b_1), b' \neq b$
    else
      $b_{best}$ = BestCandidate($b_1, b_2$)
      $label((b, b_{best})) = SEQ, \quad label((b, b_{worst})) = NSEQ$
      $label((b', b_{best})) = NSEQ \quad \forall b' \in pred(b_{best}), b' \neq b$
    end if
  else
    $label((b, b')) = NSEQ \quad \forall b' \in succ(b)$
  end if
end for
```

used. Then the benchmark programs are briefly presented. The rest of this section is dedicated to the results of the simulated execution and their analysis. Also, we compared our algorithm with an improved version of Pettis and Hansen's [5] which, as proposed in [6], uses a specific cost model for our architecture.

4.1 Experimental Setup

To perform our measurements we used a cycle-accurate simulator for transport-triggered architectures (TTAs)[11]. TTAs are a class of statically scheduled architectures for which data transports and bypasses of the general-purpose registers are explicitly programmed. For the purpose of this paper, we can consider our test architecture to be comparable to a VLIW. The simulator is part of a software design system that allows to schedule and simulate the execution of programs for a range of machines. The cache performance has been measured by using the *cheetah* cache simulator [12], which can model the behavior of different cache configurations. In this way we are able to evaluate the effect of code placement on the cache performance.

We performed our evaluations on DSP applications and on the Unix programs *compress* and *cjpeg*, the standard JPEG compressor. The DSP programs, taken from [13], are subdivided into audio applications (*arfreq, g722main, music*), and image processing applications (*edge, expand, smooth*). These benchmarks were compiled with *gcc* (ported to our architecture) into sequential code and then scheduled. We profiled and simulated complete programs, including library code.

The simulations have been performed on two different machines, one "low-end" (M1) and one "high-end" (M2). Table 1 summarizes their characteristics. Both ma-

Table 1. Simulated Target Machines. "Long immediates" is the number of long immediates that can be specified each cycle.

	quantity				**quantity**	
resource	**M1**	**M2**	**unit**	**latency**	**M1**	**M2**
transport busses	3	8	ld/st unit	2	2	3
long immediates	1	2	IALU	1	2	4
integer registers	24	64	multiply	3	1	1
FP registers	16	48	divide	10	1	1
boolean registers	2	4	FPU	3	1	1

chines have 2 cycles of delay for control transfer operations. The transport busses are used for the data traffic between functional units or registers. Two transport busses roughly deliver the same instruction bandwidth of one conventional VLIW operation slot. Unoccupied transport slots contain a no-transport code.

4.2 Experimental Results

Table 2 summarizes the effect of code positioning on the program schedule. The cycle counts do not include the stall cycles due to instruction cache miss. Data memory accesses are assumed to always hit the cache. Each row of the tables refers to a different benchmark. For each machine, the left column below label *cycles* shows the cycles spent to execute the benchmark without code positioning optimization (i.e., using the code layout generated by gcc); the right column gives the cycle count reduction obtained with code positioning. This reduction is the performance improvement to be expected on a machine in which the fetch subsystem never stalls the execution engine. The columns below *size* show the static code size (in instruction words) of each benchmark before code positioning and the size reduction achieved by code positioning. The next two columns show the percentage of control transfer operations *(CT)* executed in the original and in the optimized program. Function calls are not influenced by this optimization and are not included in the counts (their percentage is very low). Although the extent of the speedup varies largely, all benchmarks improve. The programs showing best speedup are *compress*, *smooth* and *expand*; this is partly explained by the high frequency of control flow transfers (16–24% of operations are control transfers).

Our algorithm tries to reduce conflict misses and exploit cache line locality in the instruction cache by ordering frequently executed paths sequentially. This results in a longer sequence of instructions executed between two taken control transfer operations. To evaluate the instruction cache performance, we measured the miss reduction of a number of direct-mapped caches. We simulated caches with 16 instruction-word lines, with sizes ranging from 128 to 8192 instruction words.[3] Figure 4 shows the cache miss count in the original and in the optimized code for a selection of benchmarks. All

[3] Our binary encoder produces instruction words of 12 and 32 bytes for M1 and M2 machines, respectively. Much denser encodings are easily attainable, but we did not investigate this aspect, since it is marginal to our discussion.

Table 2. Effect of Code Placement on the Program Schedule.

	machine 'M1'						machine 'M2'					
benchmark	**exec. cycles**		**size**		**CT %**		**exec. cycles**		**size**		**CT %**	
	orig.	**red.%**	**orig.**	**red.%**	**orig.**	**opt.**	**orig.**	**red.%**	**orig.**	**red.%**	**orig.**	**opt.**
compress	22.3M	16.1	6015	7.2	20.4	13.1	18.9M	21.3	5143	9.6	16.2	10.2
cjpeg	4.4M	3.7	13031	0.2	12.3	10.9	3.2M	4.0	9838	1.0	12.1	10.9
arfreq	11.5M	1.8	1541	7.7	8.3	7.9	8.3M	2.4	1352	8.4	7.5	7.1
g722main	19.1M	11.2	7353	8.5	11.9	8.1	12.1M	20.1	5804	10.3	10.1	7.1
music	37.6M	3.4	6494	-0.3	10.2	9.6	24.8M	8.7	5648	-0.6	9.1	8.0
edge	1.0M	11.8	8941	19.3	19.6	16.0	0.8M	17.2	8155	22.5	15.9	12.1
expand	0.6M	16.5	6766	4.5	24.7	19.7	0.5M	21.2	6005	5.9	23.6	18.1
smooth	0.4M	12.5	6049	11.1	20.5	16.7	0.4M	19.8	5377	13.6	17.1	12.6
average	12.1M	9.6	7023	7.3	16.0	12.8	8.6M	14.3	5915	8.8	13.9	10.8

these programs have a rather small footprint (all fit in a 128KB cache). The miss rate reduction is of course relevant only for caches where conflict and capacity misses occur. For *compress*, for example, the reduction is above 50% for all relevant cache sizes. Other benchmarks, like *cjpeg*, show more modest improvement. The case of *arfreq* is interesting; this benchmark shows a slight miss reduction increase for certain cache sizes. This is due to the presence of two very critical loops in different procedures. Depending on the placement of the two procedures, these loops may or may not generate a large number of conflict and capacity cache misses. Such problem can be substantially alleviated by procedure positioning, as shown in [2][4].

The overall execution time is computed by combining the execution cycle count with the stall cycles due to instruction cache miss. Table 3 summarizes the results obtained by simulating the benchmarks on target machine M2 with a cache size of 128 and 1024 instruction words, (4KB and 32KB, respectively). For each cache, the column *cycles* shows the total number of cycles (including stalls). Next columns show the execution cycle speedup and the overall cycle count reduction including stalls. The columns under *stalls* show the fraction of stall cycles in the original and the optimized program. The results show that the speedup obtained is highly dependent on the cache size and so is the effect of the two optimizations: program locality and execution cycle count. For large caches, e.g., execution cycle reduction dominates the achieved speedups.

Alongside our code placement algorithm we implemented and tested the block placement algorithm described by Pettis and Hansen. Our algorithm always yields lower execution cycle counts, although the difference is very small. On average, we measured 14.3–13.5% execution cycle reduction (depending on the value of α) versus 13.2%. The cache performance is also very similar. Pettis and Hansen's algorithm performs slightly better when $\alpha = 1$ is chosen for our algorithm and the cache is small. For low values of α our algorithm always performs better.

We also evaluated the impact of the heuristic parameter α (described in section 3.1). For lack of space, the discussion of the results are omitted. In summary, while most programs, as we anticipated, showed a trade-off between execution cycle count and program locality, the entity of such differences were minimal.

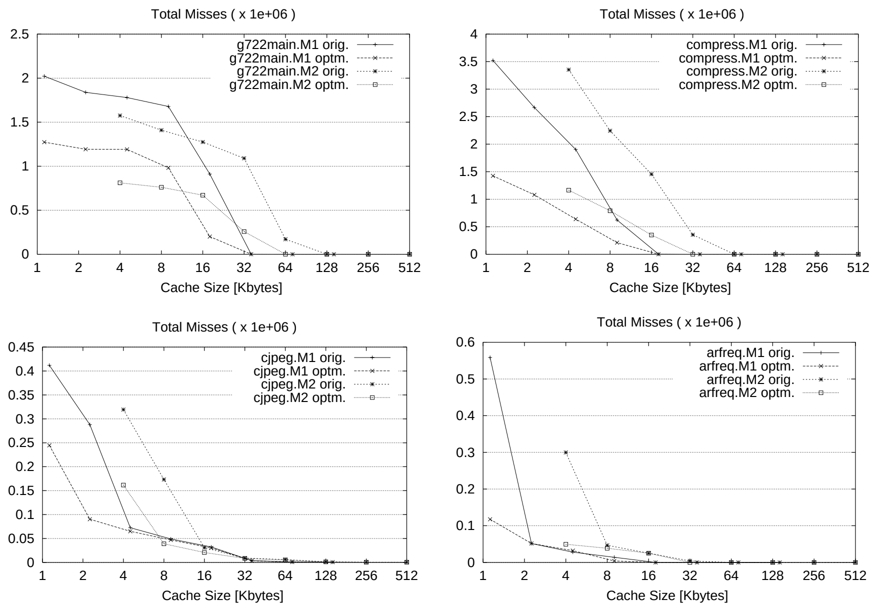


Fig. 4. Miss Reduction with Various Instruction Caches for *g722main*, *compress*, *arfreq* and *cjpeg*.

5 Conclusions

Code placement is an important optimization technique, which has a sizeable effect on horizontally scheduled ILP machines with jump delay slots. In this paper we showed that a placement algorithm can be tuned for VLIW-type architectures by placing higher priority to reducing the unconditional jumps executed. Reducing the number of critical jump delay slots has a positive effect on the program schedule. The measurements confirm that the simple reduction in execution cycle count (up to 21%) is a significant factor of the overall speedup and becomes dominant when the cache size is large. The algorithm by Pettis and Hansen achieves comparable results, indicating that the heuristic parameters have weak influence on the speedup.

References

1. J. W. Davidson and R. A. Vaughan. The effect of instruction set complexity on program size and memory performance. In *ASPLOS-II*, pages 60–64, Palo Alto, CA, 1987.
2. Abraham Mendlson, Shlomit S. Pinter, and Ruth Shtokhamer. Compile time instruction cache optimizations. In *Compiler Construction*, pages 404–418, April 1994.
3. W. W. Hwu and P. P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *ISCA-16*, pages 242–251, Jerusalem, Israel, May 1989.

Table 3. Overall Speedup Obtained with Two Different Cache Sizes, Target Machine ’M2’.

benchmark	**128-word cache**					**1024-word cache**				
	cycles	**cycle red. %**		**stalls %**		**cycles**	**cycle red. %**		**stalls %**	
	orig.	**exec.**	**total**	**orig.**	**optm.**	**orig.**	**exec.**	**total**	**orig.**	**optm.**
compress	86.0M	21.3	55.6	78.03	61.07	18.9M	21.3	21.3	0.16	0.15
cjpeg	9.6M	4.0	34.3	66.69	51.36	3.3M	4.0	4.4	3.63	3.25
arfreq	14.3M	2.4	36.4	41.89	10.90	8.3M	2.4	2.4	0.01	0.01
g722main	43.6M	20.1	40.6	72.26	62.67	15.5M	20.1	37.6	21.99	0.14
music	108.5M	8.7	52.1	77.14	56.48	25.1M	8.7	9.0	1.16	0.86
edge	3.2M	17.2	37.9	74.50	66.00	1.3M	17.2	43.6	34.33	3.66
expand	1.6M	21.2	47.4	65.63	48.53	0.7M	21.2	37.1	21.77	2.02
smooth	1.5M	19.8	41.5	74.97	65.68	0.5M	19.8	43.4	30.10	0.86
average	33.5M	14.3	43.2	68.9	52.8	9.2M	14.3	24.9	14.1	1.4

4. Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM TOPLAS*, 21(5):977–1027, September 1999.
5. Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *PLDI*, pages 16–27, White Plains, New York, June 1990.
6. Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *ASPLOS-VI*, pages 242–251, October 1994.
7. Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. Near-optimal intraprocedural branch alignment. In *PLDI*, pages 183–193, June 1997.
8. Jan Hoogerbrugge. Instruction scheduling for trimedia. *JILP*, 1(1–2), 1999.
9. Texas Instrument Inc. *TMS320C6000 Programmer’s Guide*, 2000.
10. S. McFarling. Program optimization for instruction caches. In *ASPLOS-III*, pages 183–193, May 1989.
11. Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Technical University of Delft, February 1996.
12. Rabin Sugumar. *Multi-Configuration Simulation Algorithms for the Evaluation of Computer Architecute Designs*. PhD thesis, University of Michigan, August 1993.
13. Paul M. Embree. *C Algorithms for Real-Time DSP*. Prentice Hall, 1995.