Intraprocedural Code Layout Optimization

Pedro Artigas and Mengzhi Wang
{artigas,mzwang}@cs.cmu.edu

May 2, 2001

Abstract

With the advent of memory hierarchies in modern computer systems maximizing cache locality turned into a very important problem for optimizing compilers. Several compiler based solutions were proposed to maximize data cache locality and, more recently, as the memory footprint of applications become a very significant issue, solutions to the code cache locality problem were proposed. In this work we present the design and evaluation of a simple intraprocedural basic block layout algorithm similar to the one presented in [7]. The work, presented here, has no novel aspect but an implementation of such algorithm in a contemporary compiler research infrastructure and experiences obtained during the implementation justify the effort.

1 Introduction

The gap between the speeds of memory systems and processors in modern computer systems is increasing dramatically. Cache behavior has never had such a large impact on overall system performance. As a consequence, optimizations tailored towards improving memory cache behavior turned into a very hot research topic in the compiler area as well as several other areas.

Several compiler based approaches have been proposed to maximize data cache locality. For example, the data cache behavior can be improved significantly by rearranging the data in the virtual memory address space so that data accessed closely in time can reside closely in the data cache. Other techniques, such as prefetching, have also proved to be very effective in reducing or hiding data and instruction cache miss latencies.

Recently, the instruction cache behavior of programs received renovated attention as larger and larger instruction footprints were observed in important applications such as database systems. Ideas similar to the ones used for improving the data cache behavior were applied to improve the instruction cache behavior as well. The compiler based code layout technique is among them. It is somewhat similar to the data layout technique, the basic idea used to optimize the code layout is to generate frequently executed code paths close together to shrink the cache perceived instruction footprint size.

Several approaches for code layout have been proposed. They can be divided into two categories regarding the granularities they work on. Interprocedural code layout algorithms try to map correlated procedures to different parts of the instruction cache in order to reduce the conflict miss rate.

On the other hand, intraprocedural code layout algorithms work within the boundary of procedures and move basic blocks within procedures. By putting the most likely follower of the block as its layout successor, frequently executed blocks are effectively packed together. Thus, the cache perceived instruction footprint size is reduced. Side effects related to branch penalties are also minimized.

In this work, we investigated the greedy algorithm for intraprocedural basic block layout and evaluated it using a benchmark suite consisting of several SPEC95 and SPEC2000 benchmarks.

This paper is organized as follows: In section 2 a high level overview of the problem is presented and high level aspects of the proposed solution are discussed, in section 3 implementation specific aspects of the proposed solution are elaborated. Section 4 contains the evaluation of the implemented algorithm using standard benchmark programs, section 5 is a short summary of other related research papers, section 6 presents our conclusions and, finally, section 7 present options for future work in the topic.

2 Design

2.1 Intraprocedural Code Layout

The intraprocedural Code Layout problem may be defined as follows: Find the best legal permutation of the basic blocks for each procedure in the program. In general terms, the best permutation is the one that leads to the smallest program execution time. The correctness of the program is guaranteed by updating the control transfer instruction in each basic block.

There are several issues involved in designing and implementing an intraprocedural code layout algorithm. First, the quality of the information about program run time behavior, or the accuracy of a model used to predict it, determine how close to an optimal solution an algorithm can possibly achieve. Theoretically, if perfect information about the underlying hardware and program execution was available, the optimal layout could be found by exhaustively enumerating all the possible layouts. Of course, it's almost impossible for the compiler to achieve this goal due to both the lack of information and the requirement of reasonably bounded compile time. Real implementations tend to rely on either runtime program information obtained though profiling or run time estimation heuristics.

A second issue is the cost model used to compare different layouts as it may have a great impact on the quality of the final code. Clearly, there is a tradeoff between the complexity of the model and the efficiency of the compiler algorithm that implements it. That is, complex cost models are usually more accurate, but they require more complex compiler algorithms implying a longer compilation time.

A final issue, search algorithms explore the possible layouts using different strategies. As the number of possible layouts is exponential in the number of basic blocks heuristic rules to search the space are required to ensure that the produced compiler is reasonably efficient. Different heuristics may also affect the quality of the generated code as they may not, and that is usually the case, guarantee that an optimal solution is obtained.

2.2 Greedy Code Layout Algorithm

We have chosen to use the greedy basic block code layout algorithm proposed by Pettis and Hansen in this work. The authors refer to it, in [7], as the bottom-up position algorithm. This algorithm works by rearranging basic blocks intraprocedurally. It does not consider moving code past procedure boundaries. The main factor influencing our algorithms choice was it's simplicity and the fact that it only requires intraprocedural control flow information. More ambitious algorithms could have been implemented if more elaborate control flow profile information could be obtained in our current infrastructure.

We give, here, a brief description of the steps involved in the use of the greedy basic block code layout algorithm. The first step involves obtaining basic block control flow information through a profiling step, this information is, then, combined into the IR of our compiler infrastructure. In a latter step, the information is then converted into edge frequency information and used to guide the code layout algorithm. The used profiling techniques are discussed in the next section in further detail.

The cost model used to produce the final code layout, in [7], is relatively simple. It only takes the absolute number of branches into consideration. Given a layout, the score is the total number of the fall through branches at runtime. This indicates that the program will experience less taken branches since the total number of branches is fixed. Therefore, a higher score, implies a better code layout.

The search for a final code layout, in the space of possible layouts, proceeds as the algorithm's name suggests. The algorithm is initialized with the empty layout and greedily adds blocks to it by considering edges of decreasing execution frequency. As each edge is considered, the algorithm tries to lay out the predecessor and successor blocks consecutively. If that is not possible the algorithm simply skips the current edge and proceeds. The complexity of this algorithm is linear in the number of edges present in the control flow graph.

2.3 Profiling

The greedy algorithm used ultimately requires *edge profiling*, as it uses a *weighted control flow graph* to guide it's decisions. Instead of implementing edge profiling directly we opted to use *block profiling*. That is, simply profile the number of times each basic block is entered. The motivation for this choice is that the latter is simpler to implement, usually incurs a lower compile time overhead and matches the utilized compiler infrastructure more naturally. Also, if both techniques are implemented naively, the latter incurs less space overhead during the profiling run; as the space requirements are proportional to the number of blocks and not to the number of edges in the control flow graph. As a consequence the required profiling time is also smaller. One undesired side effect is that the block profiling information needs to be converted into edge profiling information before the greedy algorithm can be applied. If the control flow graph is reducible, block profiling and edge profiling have the same expression power. That is, it is possible to derive a weighted control flow graph from block profiling information precisely. Let's now present the strategy used to obtain the edge frequency using basic block profiling techniques.

After the profiling run, each basic block is annotated with the number of times it was entered with one exception, loop header blocks (blocks that are the target of a back edge that is part of a well formed loop). For those blocks, the annotation contains the number of visits to the complete well formed loop. In other words, these blocks are annotated with the frequency of the edges that enter the loop, back edges are ignored. The motivation for this design decision will become clear when we discuss the algorithm to convert the block profile information into an edge profile.

The Algorithm employed to convert the block profiling information into edge profiling proceeds, edge by edge, as follows: Name E the current edge, if E is a back edge, ignore it. (The code layout algorithm does not require the frequency of back edges) If not, let's name P, for predecessor, the basic block that the current edge originates. Let's name S, for successor, the basic block the edge points to. We now need to consider several individual cases to extract the edge execution frequency from the execution frequency of the blocks related to this edge.

1. If block S has only one predecessor, the execution frequency equals to the number of visits to block S. (Figure 1(a).) Observe that S is not a loop header of a well formed loop as a header of a well formed loop in a structured program has, necessarily, two predecessors.

2. If block S has two predecessors it may be a loop header of a well formed loop (Figure 1(b).), test if the other edge into S is a back edge. If so the frequency of E equals the frequency in S as we store only the number of loop visits in a loop header node.

3. If block S has two predecessors and block P has only one successor, the execution frequency of the edge is the number of visits to block P. (Figure 1(c).) Also observe that block P cannot be a loop header of a well formed loop. S is not a loop header and has two predecessors, therefore P does not dominate S as it has more than one predecessor and P has only one successor. Therefore if P was a loop header this loop would not be well formed and node P would not contain the number of loop visits.

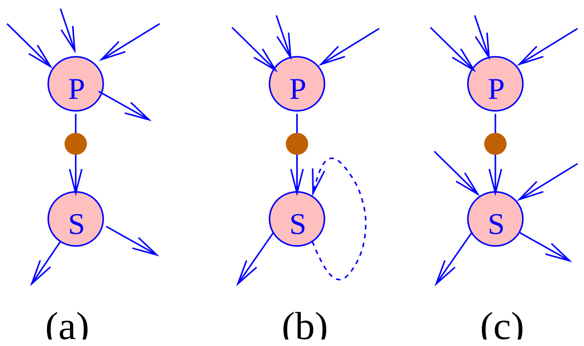

Figure 1: From block profiling information to weighted control flow graph, simple cases.

4. If block S has two predecessors and block P has two successors, namely S and $S1$, we can derive the structure of the control flow graph from the dominance information. Several subcases need to be considered:

   (a) If $S1$ doesn't dominate P, S must postdominate $S1$. (Figure 2(a).) That is, P is a fork node and S is the join node. Otherwise, the graph is irreducible (Figure 2(b).) For reducible graphs like in (a), the block frequency of S provides the total number of visits to the whole structure; and the block frequency information of $S1$ provides the number of times the execution goes to $S1$ from P. As a consequence, the execution frequency of the edge E from P to S is the number of visits of S minus the block profiling information on $S1$. This is correct even if $S1$ or P is a loop header, as shown in (Figure 2(a).) given the special treatment of well formed loop header nodes. (Block S

can not be a loop header if the graph is reducible). As, in this cases, $S1$ and P contain the number of loop visits. Therefore, the number on $S1$ always provides the number of times execution does not fall though directly from P to S.

(b) If $S1$ dominates P, P must post-dominate $S1$. (Figure 2(c).) That is $S1$ is the loop header and P is in the loop . Otherwise, the graph is not reducible. (Figure 2(d).) For reducible graphs, the number of times we enter block S from P is exactly the number of times the loop whose header is $S1$ is visited. Therefore, the execution frequency of the edge E from P to S is exactly the block profiling information of $S1$.

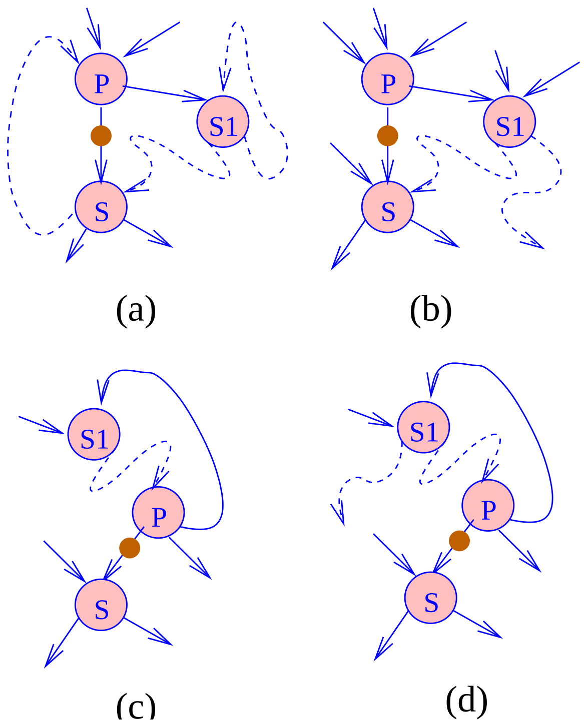

Figure 2: From block profiling information to weighted control flow graph, S has two predecessors.

5. If block S has more than two predecessors, it must be the join node of a `switch` statement. Otherwise this control flow could not be generated by a structured program. Since S has multiple predecessors, in order for the control flow to be reducible, it must be an S such as the one presented in (Figure 3(a)). That is the only case that may occur in a well behaved `switch` statement. Let's now define a well behaved `switch` statement: In a well behaved `switch` statement all `case` statements produce disjoint paths from the fork node to the join node. As a consequence, there are no two `case` statements that share code among them. An example of a `switch` statement that is not well behaved is presented in (Figure 3(b)). In this case the first step is to test if this is a well behaved `switch` statement, if this test (not described here) fails the edge E is assigned execution frequency zero. If the test succeeds we proved that S is the join node of a well behaved the `switch` statement (Figure 3(a)). The execution frequency of the edge E may be computed as the number of times the `switch` statement is entered but execution directly fall through to S. (the block profiling information on S minus the total number of times all `case` blocks are executed)

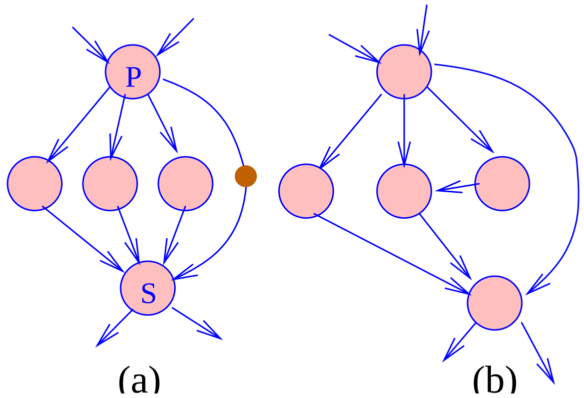

Figure 3: From block profiling information to weighted control flow graph, switch statement.

It should be clear now that the algorithm presented above is correct even if loops exist in the graph due to the special treatment of well formed loop header nodes. We observe that if loops that are not well formed or irreducible graphs exist the algorithm provides no guarantee in terms of the precision of the computed edge frequency, it works using a best-effort approach. If the graph is irreducible, it will eventually encounter situation in which accurate edge frequency information cannot be computed from block profiling information. One such example is presented in (Figure 2(b) and (d)). In the first case (b) the path thought block $S1$ not always lead to S, in the second case (d) the node P does not post-dominate $S1$. In this scenario the algorithm cannot compute the execution frequency of the edge E. In the current implementation, such edges are assigned a zero execution frequency. Any edge that makes the control flow graph irreducible is treated with the same technique. Fortunately, we observed that irreducible graphs are very rare in our benchmark suite.

3 Implementation

The organization of the compiler system used is shown in Figure 4. It consists of an augmented version of SUIF1 [10] using the MachineSUIF [9] back-end. All the analysis passes were implemented in the SUIF1 compiler infrastructure. The code layout algorithm, described in this paper, was implemented as the final code layout pass.

3.1 Overview

The compilation process consists of three phases. First, profile information is collected during profiling passes (shown in Box 1 in Figure 4.) This involves two new compiler passes, applied after the standard SUIF passes. The first pass, the *Control Flow Instrumenting* pass, instruments the intermediate representation of the program, adding instructions to collect the basic block profile information in a profiling run. The block profile information is, then, obtained. The second pass, the *Control Flow Combiner* pass, plugs the block profile information back into the compiler intermediate representation so that later passes can access the information.

Second, a conversion pass (Box 2 in Figure 4) is applied to the code to guarantee that the block profile information will be preserved for future MachineSUIF passes. The block profile information, produced by the the combiner pass, is associated with high level constructs, e.g `loop` structures. Those high level constructs in the SUIF IR are discarded when the program is converted into the MachineSUIF representation since the MachineSUIF representation is a low level one. The conversion pass is responsible to move the block profile information into no-op instructions that are inserted into the beginning of each basic block. Such instructions have the same representation in both IRs and, therefore, can be used to forward information to the final, MachineSUIF based, code layout pass. We observe that the added instructions have no impact in the run time of a compiled program as they are removed prior to the final code generation step.

Third, the last pass of interest, the final code layout pass, is applied. This pass was implemented to replace the original MachineSUIF code lay-out pass. We observe here that original code layout algorithm in MachineSUIF is somewhat naive. It consists of defining, as the layout successor of each basic block, the first block that succeeds the one under consideration that has not been laid out yet. If no such block is found an arbitrary block is used as the successor.

In the following sections, we describe the implementation of the profiling passes and the greedy code layout algorithm in detail.

3.2 Profiling

Two passes are involved in obtaining the basic block profile information. Both work on the SUIF1 IR representation of the program and are applied to the program after the standard SUIF front-end.

The Control Flow Instrumenting pass is used, as it's name suggests, to instrument the code in order to obtain control flow information. During this pass every program structure that affects the control flow, (i.e. `if` statements, `loop` structures), receives an unique identifier, also, calls to construct specific profiling library routines are inserted in order to capture the control flow behavior of each structure. The profiling library is responsible to maintain counters for all the program structures and collects the number of visits for each one them. It is linked to the program to be profiled, the profiling run, then, occurs. Finally, the execution frequencies for each identifier are dumped to a disk file at the end of the instrumented program execution.

The information obtained during a profiling run is then fed back into the SUIF IR by the Control Flow combiner pass. The combiner pass reads the source code and stores the basic block profile information as annotations in the respective structures in the program IR that generated them.

The basic block profiling code, described above,

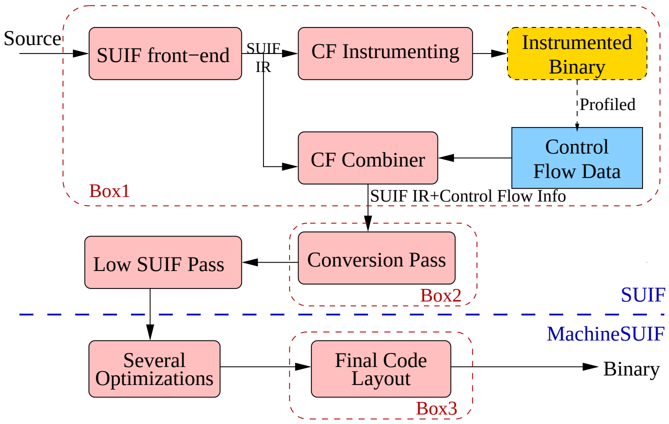

Figure 4: Organization of the compiler system used in this paper.

was adapted the from a similar pass used in the Stampede[1] project.

3.3 Code Layout

The final code layout is produced in the last MachineSUIF pass under consideration in this project. This pass is applied to the code after several standard MachineSUIF low-level optimization passes. Our implementation of the greedy basic block layout algorithm replaces the original MachineSUIF code layout pass.

The greedy basic block layout algorithm works on one procedure at a time. For each procedure, the algorithm first obtains the edge frequency from the basic block profile information, as described earlier, and stores all forward edges in a sorted list.

The second step consists of obtaining the final code lay-out, this lay-out is computed incrementally, as described latter, and stored in a data structure. This data structure consists of a two dimensional linked list as shown in Figure 5(a).

Each node in the data structure represents a basic block in the procedure, nodes that are linked together vertically represent a fragment of the final code layout; that is, a sequence of blocks, as stored in the data structure, will be part of the final code layout.

The horizontal links are only used to connect all the chunks of code obtained so far.

The linked list is empty at the beginning of the algorithm. The sorted list of the edges is then scanned in descending order of execution frequency.

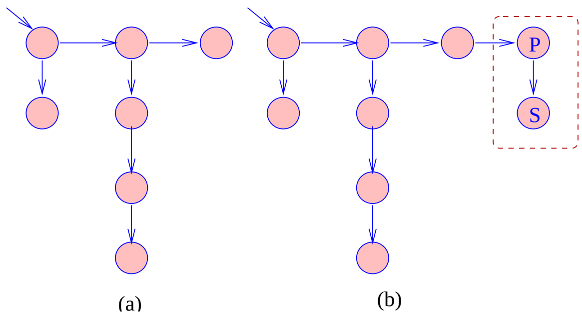

Figure 5: Two-dimensional linked list used in code layout.

Each edge is processed once, as each edge is processed basic block nodes are inserted in the two dimensional linked list data structure, after all edges are processed the final data structure contains the desired code layout for that procedure.

Each edge is processed as follows: An edge has two end points, the predecessor block P and the successor block S. If both P and S have not been already inserted in the layout list the algorithm builds two new nodes, connect them according to this edge and stores this newly created nodes as a new code layout fragment, as indicated by Figure 5(b).

If either block is already stored in the code layout data structure the algorithm must inspect the code layout computed so far. If the predecessor block already has a layout successor, or the successor block already has a layout predecessor this edge will be ignored. Observe that the code layout obtained so far is a function of edges whose execution frequency is higher than the edge under consideration, therefore it is acceptable to simply ignore the current edge in this case.

If the blocks that this edge connects to may be laid out in sequence the data structure is updated to reflect this. Nodes for the P or the S blocks are, them, created. Observe that at most one node is created in this situation. The nodes representing P and S are now linked in order to express the current code layout. Also note that lay out loops may ever be created by this algorithm, as only forward edges are ever considered.

After all edges have been processed the code is laid out as follows: The first fragment is the one that starts at the entry node of the procedure. All other fragments are appended to this one according to the order they were stored in the linked list. As edges are processed according to their frequency, and fragments closer to the head of the list were created earlier, it implies that they are 'hot' fragments, that is, are paths with higher execution frequency. As paths of higher execution frequency are laid out closer to each other, in a compact way, they will have better locality and a higher chance of not conflicting with each other. This is an important factor if the I-cache of the target machine has low associativity.

After all fragments were concatenated, as described above, the code lay out for this procedure is complete. Note that the code layout of each procedure is considered in isolation, that is, after the code layout step each procedure will constitute a monolithic chunk of code, that may be a performance limiting factor if some code fragments are never executed.

4 Evaluation

In order to evaluate the effectiveness of the proposed techniques the current implementation was used to compile a subset of the SPEC95 and SPEC2000 benchmarks, the target architecture is an Alpha 21164A workstation with a 500Mhz processor. All the benchmarks were profiled and run according to the SPEC guidelines, the train input set was used to provide the profile information and the reference input set used to measure the performance of the generated binaries. All experiments were done in a lightly loaded machine and the execution time measured using the standard time command. Only the time spend in user mode was considered to compute the final results.

Speed up results were computed by comparing the execution time of a baseline binary. The baseline binary was produced using only the SUIF1 front end and the MachSUIF backend, no optimization pass described in this work was involved. The optimized version of the binary was produced using the current profile based code layout implementation.

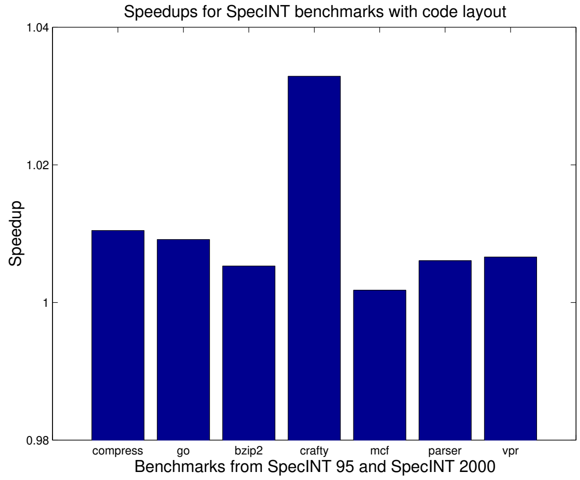

Figure 6: Speedups obtained using a subset of the SPEC95 and SPEC2000 benchmark suites.

Most benchmarks do not benefit significantly from the techniques utilized to improve code layout, on the other hand no benchmark slowed down, this indicates that the current technique, if not particularly efficient, is adequate. Most of the benefit obtained for all benchmarks but *crafty* are probably due to a slightly improved branch alignment, it is clear that no relevant number of instruction cache misses was eliminated by the technique.

The only benchmark that generated a relevant speedup, *crafty*, deserves further investigation.

4.1 *crafty* Performance

crafty consists of an elaborate Artificial Intelligence engine to obtain nearly optimal moves for a given chess board configuration. It employs a complex search routine that dominates the execution time.

The search routine consists of a very large procedure that has, itself, a very complex, but reducible, control flow graph. It is clear that for this particular procedure, the ratio of the procedure size to the instruction cache size is non-trivial. Also, as the procedure has a complex control flow and is called repetitively a very large number of times, it is clear that even a better intraprocedural code layout, may affect overall program performance as the procedure tends to conflict less frequently with other, unrelated, blocks of code; another aspect contributing to this behavior is the fact that the instruction cache of the machine used for evaluation is direct mapped, therefore conflict misses are much more likely to occur if the frequent paths of a major procedure are not laid out close together.

The result obtained for *crafty* indicates that the better branch alignment and intraprocedural code layout have a minor impact even in an ideal case scenario, on the other hand it points to the fact that more significant performance improvements could be obtained if more global, interprocedural techniques were employed.

5 Related work

The problem of maximizing instruction fetch bandwidth and minimize pipeline penalties due to instruction fetching is studied in detail in the literature. Proposed solutions to the above problem range from hardware only solutions, to several compiler based techniques, including profile guided compilation and elaborate heuristics that model program behavior at static compile time, avoiding any profiling costs.

Some well known hardware techniques like branch prediction and the trace cache are extensively used in modern processors. Almost all microprocessors use the former while the latter technique has recently been incorporated into the latest IA-32 processor. Most, if not all, compilation systems also perform optimizations to improve instruction cache locality and reduce branch related pipeline stalls.

A trace cache[8] stores contiguous dynamic instruction traces in order to improve the fetch bandwidth, it decouples the compiler defined code layout from the execution engine requiring an extra cost in terms of cache size requirements due to instruction duplication, on the other hand the trace cache stores large frequently executed instruction traces and reduce branch costs dramatically, as a consequence, it is a very successful technique to improve fetch bandwidth.

Amongst the compiler based techniques two trends exist. Profiling based [4] and heuristics based techniques, examples of the former technique [7] use edge profiling or path profiling to identify frequently executed code sequences and use this information to layout the static code in order to prioritize common executed sequences, improving instruction fetch bandwidth and reducing branch costs as the most frequent paths are usually laid out in sequence. This is advantageous as most processors incur the least amount of penalty when execution falls though branches. Interprocedural techniques also examine the problem of the optimal ordering among procedures, the abstraction commonly used to determine adequate solutions is the weighted call graph, more elaborate solutions also make use of dynamic invocation sequences of procedures, such as the work in [6].

Techniques that avoid profiling costs usually compute edge execution frequencies directly using heuristics. Common heuristics assume, for example, that loops usually execute several iterations and that if statement conditions are not satisfied if testing for equality or comparison with negative numbers, assuming that all other comparisons succeed. These assumptions are reasonable considering the coding style used in most high level languages. Even though information computed directly may not be as precise as profiled information it incurs much less compilation costs and produce adequate results in most cases. Another advantage is that final binaries are not tuned to any particular case. An example of such technique is presented in [2].

A similar but less general problem is branch alignment, several compiler based techniques addressing this problem are presented in [11, 3], another proposed solution to this problem [5] adapts VLIW techniques to more common superscalar processors, it is not surprising that the technique originally targeted VLIW machines as branching costs are much more pronounced on such architectures.

6 Conclusion

This work presents a simple Intraprocedural basic block layout algorithm. Due to the simplicity and limited scope of the proposed algorithm and the fact that most benchmarks in the utilized benchmark suite have procedures whose size is negligible when compared to the size of the instruction cache in a modern machine, such as the one used to evaluate this work, it is not surprising that the obtained gains in performance were almost imperceptible.

On the other hand further analysis of the obtained performance gains for one benchmark, *crafty*, pointed out interesting options for future work, presented in section 7. Also, this work present some interesting aspects on the engineering of a full, profile-based, compiler optimization pass implemented in a modern research compiler infrastructure; and interactions among an optimizing compiler and the platform used to execute the generated binaries, including aspects such as the underlining computer architecture and implementation.

7 Future Work

The code layout algorithm and it's implementation, presented in this paper, could be improved in several ways. Procedure placement is a natural extension to the simple basic block placement presented. Also the current implementation ignores several important aspects of instruction cache design, such as associativity and potential conflicts, the algorithm could be extended to handle these cases by incorporating, for example, a scheme similar to page coloring. Techniques, such as, distinct allocation policies to hot and cold paths also have the potential to improve the effectiveness of the current implementation.

On the other hand it is clear that, in the utilized compiler infrastructure, intraprocedural code layout is not the aspect that needs the most work, by comparing the performance of the binaries generated in the current infrastructure to the production compilers using a standard set of benchmarks it is clear that several other aspects require significant effort. On the other hand, interprocedural code layout techniques and even techniques that operate intraprocedurally but lay out only hot paths of the whole program in a compact way should generate more significant performance gains, as those problems are orthogonal to others, such as instruction scheduling, those issues could be tackled individually.

References

[1] http://www.cs.cmu.edu/ stampede.

[2] Thomas Ball and James R. Larus. Branch prediction for free. *ACM SIGPLAN Notices*, 28(6):300–313, 1993.

[3] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. *ACM SIGPLAN Notices*, 29(11):242–251, 1994.

[4] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[5] Kemal Ebcioglu, Randy D. Groves, Ki-Chang Kim, Gabriel M. Silberman, and Isaac Ziv. Vliw compilation techniques in a superscalar environment. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 36–48, 1994. Orlando, Florida, 20-24 June. SIGPLAN Notices 29(6), June 1994.

[6] Nikolas Gloy and Michael D. Smith. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, 1999.

[7] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, 1990.

[8] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, pages 24–35, 1996.

[9] M. Smith. Extending SUIF for machine-dependent optimizations, 1996.

[10] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, Shih-Wei Liao, Chau-Wen, Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF:an infrastructure for research on parallelizing and optimizing compilers. *ASM SIGPLAN Notics*, (12):31–37, 1994.

[11] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. Near-optimal intraprocedural branch alignment. *ACM SIGPLAN Notices*, 32(5):183–193, 1997.