Improving Instruction Cache

Javier Salamero Sanz
Asier Fernández De Lecea Navarro

Context

In the context of computer architecture, logic is becoming increasingly faster in a lower area footprint, improving performance massively. This constant improvement is known as Moore's Law, which describes the observation of transistors of an integrated circuit doubling every two years, which has slowed down in the last decades due to reaching underlying physical limitations of materials. This situation has put increasing pressure on memory accesses, whose access speeds are not matched with modern processor frequencies and haven't been able to scale and improve at the same rate.

Computer engineers have put their efforts into further reducing access times by incorporating memory hierarchies into computers. The memory hierarchy consists of a set of memory structures hierarchically ordered closer/farther from the CPU in a tradeoff of capacity vs speed, placing smaller but faster memories called caches closer to the processor. In current trends, instruction and data caches are separated to reduce the memory size even more. Despite these efforts, memory accesses are still one of the main performance bottlenecks in modern applications with intensive memory workloads.

Most of the current state-of-the-art work is focused on data caches, as mentioned in the papers. Still, the penalty cycles induced by instruction access misses could cause up to 20% performance loss in certain applications. There are different types of proposals for caches, from the most simple implementation, direct-mapped caches, to more complex fully-associative caches. Associativity is capable of reducing miss rates as it can mask the problem of aliasing to a certain degree, though this increase in complexity adds to hardware costs and access times.

An orthogonal approach to keeping the instruction caches small and simple is to apply compiler optimizations to programs to make better use of this resource. By carefully managing the positioning of code and instructions in memory, the number of conflicts of competing instructions for the same sets in a direct-mapped cache can be reduced, as well as the overall number of instruction cache misses. The papers evaluated focus on these objectives with a series of techniques, which we aim to analyze and explain in the following sections.

Interestingly, the motivation section of most of the papers on this topic revolving around compiler-centered program optimizations regarding the instruction cache pose the issue of the memory hierarchy as still a matter of debate and discussion. From the claims and wording of the proposed works, we got the impression that, at the time of publication of the papers, even the separation of cache memory into instruction-cache and data-cache wasn't an established fact yet. Thus, the aim of these publications was geared towards, perhaps,

establishing the reasoning basis that kickstarted the current trends and common-knowledge assumptions made with regard to compilers and computer architecture at large, championing the utilization of compiler-based methodologies to surpass the memory wall and rendering the instruction cache configurations used in these works as essential for higher-performance aspirations.

Optimisation techniques

The goal of the studied techniques is to reduce the number of conflicts in the instruction cache while keeping it as simple as possible. Therefore, works are centered around improving a direct-mapping cache, as opposed to just resorting to bigger, more complex cache configurations of higher associativity that attempt to fix the shortcomings of direct-mapped caches to the detriment of access times and area/power costs. Additionally, direct-mapped caches are preferable to other cache configurations regarding compiler-based optimizations because their behavior is much more straightforward and predictable (and, consequently, controllable) than dealing with high degrees of associativity. As additional benefits, placing frequently-executed code close in memory also results in a reduction of the number of TLB and page misses.

In direct-mapped caches, most instruction cache misses result from interference between memory regions competing for the same entries in the cache, this is called aliasing. If the compiler finds two memory regions of a program belonging to different procedures that need to be in the cache at the same time according to the semantics of the program (exploiting spatial and temporal locality, that is), the key idea of all the following techniques covered revolves around cleverly positioning procedures in specific memory regions and/or managing several of them as unified computational units so to avoid this interference and reduce cache miss rates.

The key idea is to reduce the aliasing for instructions close in execution time by placing them in addresses that do not collide in the same set. This positioning of the code requires measuring and analyzing how the code behaves in terms of how frequently some parts of the code are executed and how to control transfers take place, which is generally assumed to be provided by profiler information.

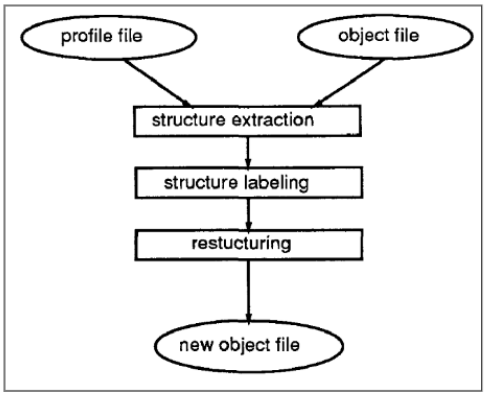
1. Repositioning of instructions to avoid cache interferences

Paper on this subject: McFarling, S. - Program Optimization for Instruction Caches - Proceedings of ASPLOS III, 1989

Most instruction cache misses in direct-mapped caches arise from interference between instructions/memory regions competing for the same entries in the cache. If instructions from two procedures need to be in the cache at the same time, ideally the instructions could be positioned so that they do not map into the same cache line. Another relevant source of misses is the lack of room for more instructions, thus interference being seemingly unavoidable. These instructions not only miss, they may replace useful cache lines of their own procedures and even other procedures, potentially doubling the miss rate.

In order to address these issues, the compiler can make sure that certain procedures and their mapped memory regions (1) either bypass the need to be included into cache and go straight to the processor (this would require hardware support in the cache to dynamically turn it off, adding extra information to the opcode/address fields of an instruction) or (2) only exclude cache lines belonging to the same procedure so that interference is caused only between within specific procedures. Obviously, the hardware-support version would yield much better results, but this also comes at the cost of added hardware complexity and implementation efforts.

It is necessary to know (1) which instructions of which procedures will interfere with each other considering the entire scope of a program and (2) how instructions from procedures should behave in relation to each other if said procedures are closely related both in terms of temporal and spatial locality regarding the instruction cache. For this, certain changes in the compilation process of a program must be implemented, mainly to do with analysis of program structure, generation of a directed acyclic graph (DAG) representing the dependencies between loops and procedures and their relative frequencies, detection of conflicting procedures within loops (which are the sources of cache interference that this work aims to tackle) and managing of the instruction placement of conflicting procedures. For this, it is only necessary to supply an initial object file and some profiler information, ultimately resulting in a new object file.



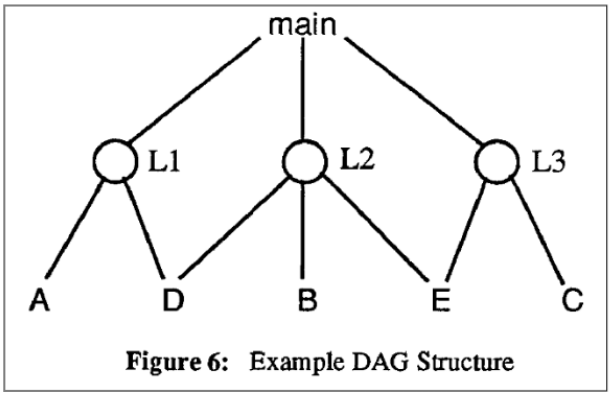


Figure 6: Example DAG Structure

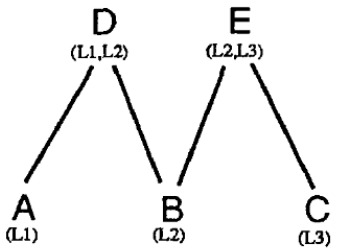
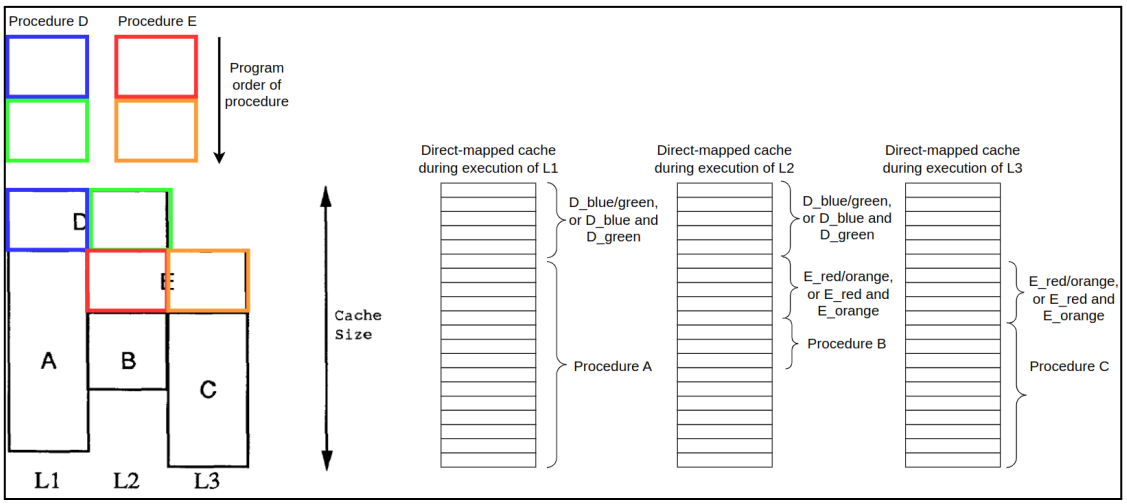


Figure 7: Example LABELDAG

Let's see an example of what each phase of the algorithm would output. After the "structure extraction" phase, Figure 6 is an example of a program's structure. L1, L2 and L3 are loops, and A, B, C, D and E are procedures called by the detected loops. Figure 7 corresponds to a dependence graph generated in the "structure labeling" phase detailing how procedures are associated to one another. A and D are related since they are called within loop L1, B-D-E are also related since they are called in loop L2 and similarly with E-C concerning L3. The "restructuring" phase of the process decides how exactly related procedures in the LABELDAG shown in Figure 7 should be mapped in the instruction cache so as to minimize the interferences amongst themselves. This "restructuring" phase employs a greedy algorithm to solve what, visually, at least, looks like a distant cousin of the knapsack problem.

The figure below shows what the greedy algorithm comes up with from the LABELDAG depicted in Figure 7. The greedy algorithm results in a specific mapping of how loop-related (i.e. conflicting) procedures should be mapped in cache if two/more procedures together do not fit in the instruction cache. Since procedures A, B and C are only called by only one loop each and they fit in the cache, they are added in their entirety to a specific region of the direct-mapped cache for the entire duration of their respective caller loops. However, procedures D and E are called by two loops each, so they must be divided up to fit in the remaining space in cache after the procedures A, B and C have already been placed.

Let's consider two scenarios, one in which the direct-mapped cache supports dynamic shut-offs, and thus certain compiler-marked instructions can be directly sent to the processor (these instructions are known as "excluded"), and a second case in which the instruction cache is a regular direct-mapped cache.



Looking at the execution of loop L1, in the first case (the extended instruction cache with dynamic shut-offs) if the blue section of procedure D is marked as "included" (the opposite of instruction/region "exclusion"), then only the blue section of procedure D will ever be in cache for the entire execution of L1 and whenever the green section is needed, its instructions will bypass the cache and go directly to the processor. Similarly, if the green section is marked for "inclusion" and the blue one for "exclusion", the opposite will be true for L1's execution. Notice the "*D_blue/green*" annotation that aims to reflect this mechanism, meaning that only one of the sections of procedure D is present in the cache during the execution of loop L1.

In the second case, the normal direct-mapped cache, the blue section of procedure D should be extended with an additional branch instruction that points to the beginning of the green zone of the procedure, such that the green section of procedure D can be stored in the same top part of the direct-mapped cache. The end of the green section of procedure D should also be extended with an additional branch instruction to connect with the previous execution flow of instructions. This way, both sections of procedure D are stored in the same section of the instruction cache during the execution of loop L1, as detailed by the annotation "*...or D_blue and D_green*". The same logic applies to the execution of loop L2 with procedures B, D and E, where B remains at all times present in cache and procedures D and E are divided into subsections to be managed by a direct-mapped cache with or without dynamic shut-off capabilities. Loop L3 is more of the same, this time only dealing with procedures C and E.

	OPT	optimized some cached	optimized all cached	direct mapped	set assoc.
bigfm	7.6	10.3	11.5	20.7	19.1
ccal	15.9	18.1	22.3	31.9	32.4
compare	2.9	3.7	5.3	15.9	7.8
dnf	1.3	2.1	2.5	16.6	9.5
hopt	6.9	11.2	14.0	30.4	20.4
macro	8.5	12.8	15.2	23.4	22.0
pasm	8.7	11.0	16.8	27.0	25.7
pcomp	14.7	20.2	23.7	32.8	32.1
simu	15.2	16.8	19.5	24.8	22.8
upas	13.6	18.9	21.4	27.5	28.7
avg.	9.5	12.5	15.2	25.1	22.1

Table 2: Miss Rates for Various Cache Organizations, Size = 512 Words

Table 2 reflects the miss rates for various cache configurations. OPT refers to a fully-associative cache, the second column to a direct-mapped cache with dynamic shut-off plus the proposed optimisations, the third column to a regular direct-mapped cache plus the proposed optimisations, the fourth column to a direct mapped cache with no optimisations and the final column to an 8-way set-associative cache as the one utilized in a relevant (at the time, at least) CPU architecture. The observed trends in miss rates hold true also for greater cache sizes, even maintaining the same order of miss-rate performance.

2. Procedure merging, callee-with-caller merging

Paper on this subject: McFarling, S. - Procedure Merging with Instruction Caches - PLDI 1991, 71-79

Procedure merging (also referred-to as inlining) deals in merging the calls made to external programs or procedures with the instructions of the source/caller program, thus creating an uninterrupted flow of “useful” instructions between caller and callee for any call instances selected for merging. In the context of instruction-cache-related optimizations that the compiler can perform, the decision of which call instances to merge with the caller program to improve performance is a complicated one.

On the one hand, the merging of a procedure within the caller program can result in the total number of instructions executed being reduced, since any explicit call to outside procedures implies the execution of a series of instructions pertinent to the context switches necessary to both “go-to” and “return-from” the outside procedure (call and return code, loads and stores of register values…). Thus, some performance gains can be obtained by integrating the “source code” or the actual instructions of the callee into the caller. However, these performance gains may come at the expense of total caller program size if the procedure plus the callee is big enough or if the procedure is called from multiple places. This last point effectively results in an increase of instruction cache miss rates, not only because of insufficient instruction cache size, but also due to the possible replacement of other, to-be-used-also-in-the-near-future cache lines (cache interference).

The problem is further exacerbated if the supposed merging-decision algorithm that a compiler can use must also consider the frequency at which calls to outside procedures are made (think of, for example, complicated loops with calls to outside procedures such that external procedures can or cannot be called at any given iteration of the loop because they are nested in if-else clauses, causing some procedures to be executed much more often than others), or if the compiler must also account for the calls to external procedures that were nested within a merged callee after the fact.

In order to make informed decisions about which procedures to be merged with the caller program, it would be immensely useful for the merging-decision algorithm to have a model of how exactly the miss rate of the instruction cache fluctuates if a certain procedure was to be inlined within the caller program or not. With such a model of the instruction cache, the cost arising from the merging of procedures in the caller program could be weighed against their expected benefits in performance, providing the compiler’s merging-decision algorithm with a basis of quantifiable information to decide whether to perform instruction-cache-related optimizations in the form of merging for a given program. The merging-decision algorithm would require (1) profiler information and program-structure details to construct flow and call graphs, (2) total cache size and (3) the cache model[1] defining the expected variations in instruction-cache-miss rates.

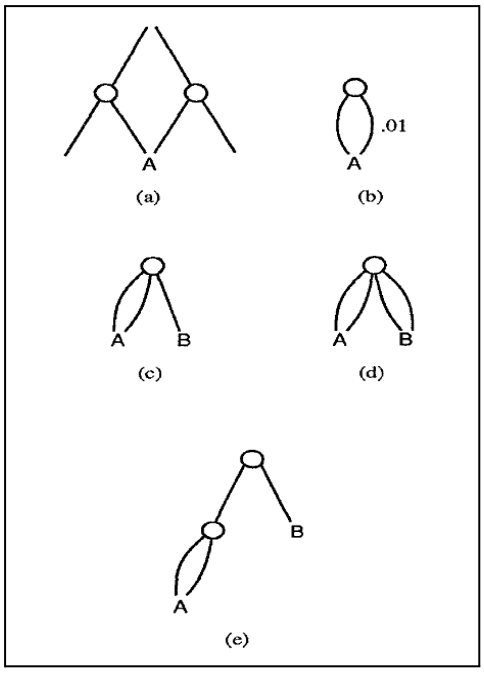
---

[1] The cache model employed corresponds to an optimal-cache-based model as described in the first paper, as (1) it is simple to model and (2) if the instruction cache miss rate is of concern, then programs are assumed as needing to be optimized for the cache in the first place, and, after such optimization, a cache behaves much like an optimal cache.

Merging a call, again, affects performance for better or worse not only via a reduction or increase in the instruction cache's miss rate, but also by reducing the number of instructions executed. The change in the number of instructions executed has several causes, like the removal of call and return code or the removal of loads and stores of parameters, among others. Accurate predictions of performance regarding these two effects is complex. However, some other adjacent works empirically show that the removal of call and return code tends to dominate the other factors when trying to predict the total number of instructions executed. Thus, a simple performance gain estimate can be used based on (1) a baseline average of how many call-and-return instructions are eliminated for each removed call coupled with (2) the change in instruction cache miss rate.

The miss rate model of the instruction cache works by estimating the average loop iteration size of a loop and, from this, estimating the number of misses an iteration would cause on the instruction cache, applying this calculation to all loops of the program. This analysis leads to a loop-based estimate from which to obtain a miss rate prediction for the whole program (this model is shown to be quite accurate for multiple cache sizes and benchmark programs, see section 4 for further details). Thus, when trying to decide on whether a procedure is to be merged or not, the same calculation process for the expected miss rate is performed with the updated value of the average loop iteration size if the procedure was to be inlined.

Usually, merging a procedure increases code size, resulting in new average sizes of the loops containing the call and, thus, in different miss-rate behaviors. If the procedure to be merged is frequently called from only one place, then the average size of the loop will be reduced by the amount of the call and return code. If the merged procedure is called from multiple sites, then some of the loop sizes may increase and the model will predict more misses. Whether a given call should be merged depends on what else is in the loops that contain it, which ultimately determines the loop's size. Additionally, merging one procedure causes the calls within the merged procedure to be copied, further complicating the problem.



The adjacent figure shows several examples of how these factors can affect whether or not a merge should be performed. In the figure, circles represent loops, A and B represent procedures, and the arcs represent calls. Diagram a) shows a procedure called only once in each loop it is in. Merging such a call will always increase performance because the number of instructions that need to be kept in the cache over any loop lifetime is decreased. Merging is always beneficial if there is only one call to a procedure or if only one call is ever executed.

Diagram b) shows a procedure called from two places in a loop. One call is executed on every iteration and the other is rarely executed. If both calls are merged, the procedure body from the frequent call will need to be in the cache every iteration, which corresponds

closely to the need to keep the entire original procedure in the cache each iteration. The code from the rare call will demand additional space in the cache, but very infrequently. This corresponds to a small increase in the average loop iteration size. Unless the called procedure is huge, the reduction in the number of call and return instructions executed will more than compensate for any additional cache misses.

Diagram c) shows a loop that contains calls to procedures A and B, with B called from one site every iteration and A called from two sites on each iteration as well. Merging B is always worthwhile since it would reduce the size of the program. Whether A should be merged depends on the sizes of the procedures relative to the size of the cache. If the loop body will fit in the cache even if procedure A is duplicated, then clearly A should be merged. If A is large and the cache is already full, then A should not be merged since each copied instruction will miss on each loop iteration.

In diagram d), procedures A and B are called from two sites within a loop. Whether or not either procedure should be merged depends on their sizes and the size of the cache. Since the savings in instructions executed from merging either procedure is approximately the same, it is usually better to merge the calls of the smaller procedure first. Subsequently, if there is still space in the cache, the other procedure should also be considered. Smaller procedures should be considered first for merging before larger procedures called the same number of times.

In diagram e), there are two calls to procedure A nested more deeply than a call to procedure B. Whether or not A should be merged depends on the sizes of the procedures and the number of iterations of the inner loop. If procedure B will not fit in the cache if A is copied, then the additional misses in B must be weighed against the removal of instructions within the inner loop. This can be calculated using the model from the average number of iterations of the inner loop, the sizes of A and B, and the size of the cache.

Deciding which combination of calls from the entirety of a program to merge is difficult because the decisions for each call interact. Merging one call affects how much space is left in the cache and how often other calls are executed. In addition, merging one procedure causes the calls within the merged procedure to be copied. Deciding whether these new calls should be merged will further complicate the problem.

```
Procedure MergeDecision
begin
    while more calls
        pick aCall with highest (call frequency/callee size)

        savings := aCall.timesCalled * CostOfCall;
        cost := additionalMisses * CostPerMiss;

        if savings < cost then
            Mark aCall to be merged
            update data structures
        end;
    end;
end MergeDecision
```

The calls to external procedures are selected through a greedy algorithm, in decreasing order of the frequency of the call divided by the average size of the called procedure. In other words, calls are considered in decreasing order of the expected path length savings (i.e. reduction in executed instructions) versus how much inlining might increase the demand for the cache. This greedy ordering criteria results in small frequently called procedures to be considered first.

Each loop that contains the call is analyzed to determine the expected change in the number of misses of the program through the loop average size model. The change in misses is then weighted by the miss penalty (the cost of each miss) and compared with the expected savings in path length to determine whether merging the call will be beneficial. If the model recommends merging the call, the representation of the program is modified to reflect the situation after inlining. If the callee has internal calls that have not been considered for merging, both the original call and the new copy are placed on the undecided list. Any calls already decided keep their current status. Finally, if the duplication of the inlined procedure body creates new loops, these loops are considered during any future merge decisions.

The results yielded by the greedy algorithm based on the performance estimations given by the average-size-of-loops model of miss rate changes are shown to outperform other, simpler merging decision algorithms on three categories for increasing miss penalties (in cycles) of the cache model: change in overall performance, total path length of final program and overall miss rate. One of the simpler algorithms decides to merge procedures or not based solely on whether procedures are above/below a certain size, such that procedures whose sizes are less than a threshold are merged. The second one uses both the size of the callee and the number of times each call site is executed, such that calls are merged when the ratio of the number of calls to the size of the procedure exceeds a threshold value.

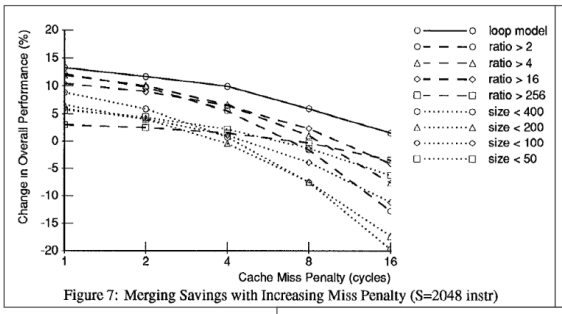


Figure 7: Merging Savings with Increasing Miss Penalty (S=2048 instr)

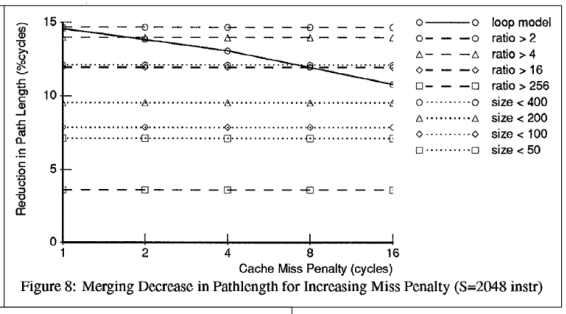


Figure 8: Merging Decrease in Pathlength for Increasing Miss Penalty (S=2048 instr)

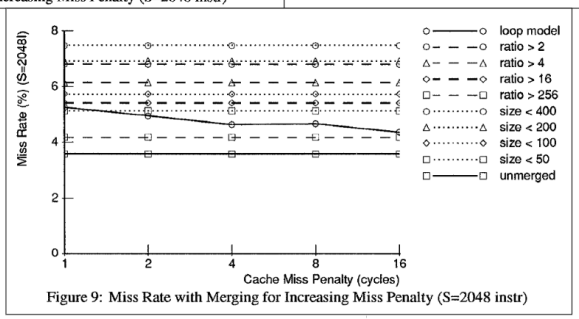


Figure 9: Miss Rate with Merging for Increasing Miss Penalty (S=2048 instr)

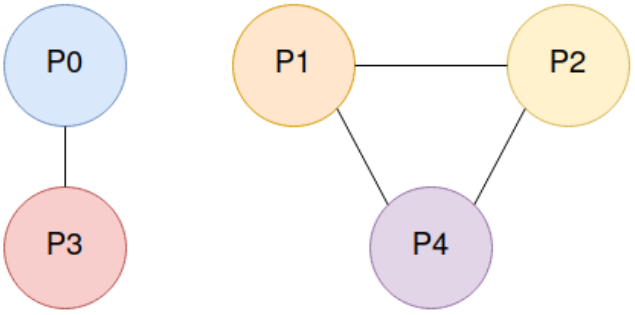
3. Procedure Positioning

Papers on this subject:

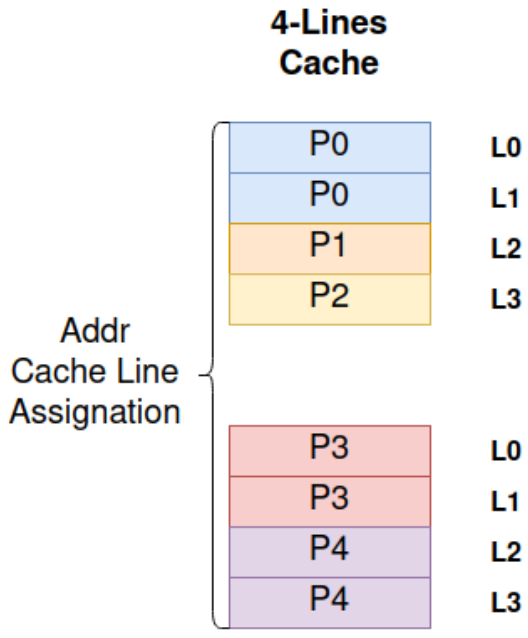
- Pettis, K; C. Hansen, Robert - Profile Guided Code Positioning - SIGPLAN Not. 25, 6 (Jun. 1990), 16-27
- Hisu, WW; PPChang, PP - Achieving High Instruction Cache Performance With An Optimizing Compiler - The 16th Annual International Symposium on Computer Architecture, June 1989.

The idea behind procedure positioning is to have procedures frequently executed together, as close as possible in memory. This approach has two main advantages. The first is the reduction of TLB and page misses. By placing frequently-executed procedures closer in memory, it is possible in some cases to have them on the same page, inevitably reducing the number of TLB and/or page miss penalties. The second and most impactful advantage is the reduction in cache misses. If frequently executed procedures are consecutively placed in memory and fit inside the cache, they will not compete for the same cache lines and will avoid aliasing, reducing the amount of replacing.

For example, some procedures have the following call graph:

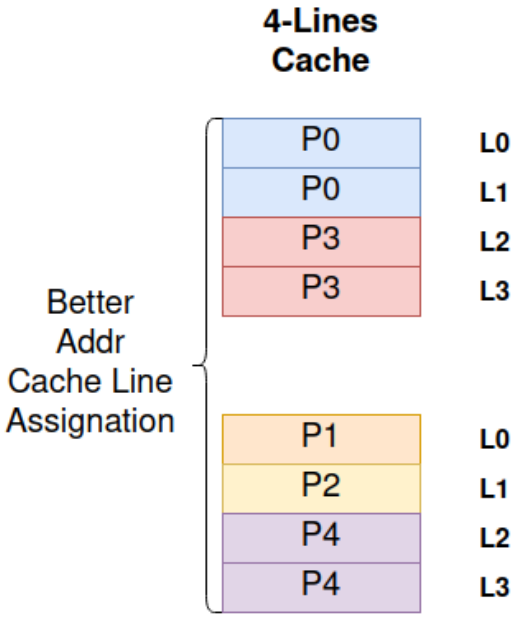


The compiler assigns the addresses in order of appearance in the source code, assigning to the direct-mapped cache lines the process as follows:



In this case, P0 and P3 will be competing for the same cache lines, despite being executed contiguously, producing a replacement every time they have a control transfer. The same applies to P1, P2 and P4.

A better address assignment would eliminate this problem, reducing the number of conflicts.



This optimisation is done at the linker level in a two-phase fashion:

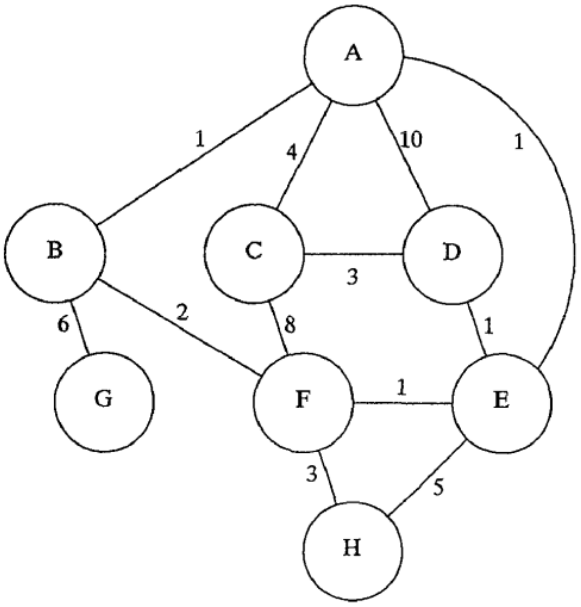
- First, profiling information has to be collected to establish and count the relationships between caller and callee. This phase could be done by recompiling the code, adding profiling instructions to the source, or modifying the linker to add these instructions. The second approach seems a better solution as it doesn't require the source code for re-compilation. With the profiling information, an indirect, weighted call graph is built, where nodes represent procedures, and the weighted arcs represent control transfers.

- Second, the procedure orders. Once the weighted call graph is obtained, the second phase is deciding how the ordering of procedures should take place. Selecting the ordering strategy is a key aspect of the optimisation.

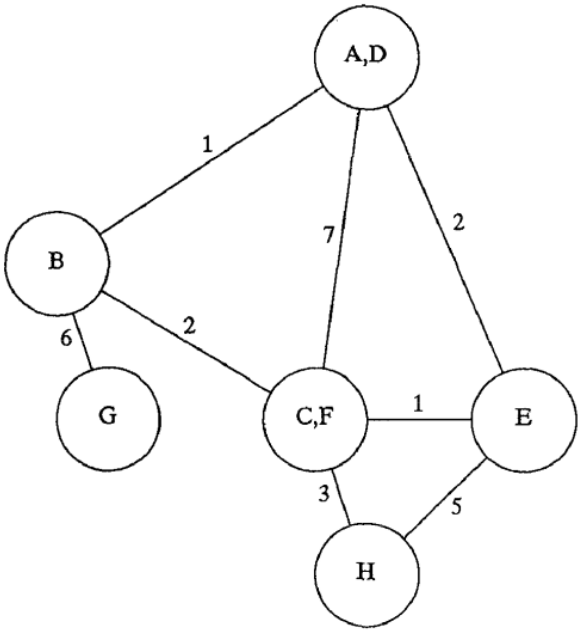
  One proposal is the strategy "closest is best". An algorithm implementing this strategy would reason as follows: the two nodes with the highest-weighted arc are merged into one, and the arcs of the other nodes are combined. After that, the next nodes with the highest arc are merged following the same procedure until no more nodes can be merged, remaining one final node or unconnected nodes. However, with an indirect graph, there can be some ambiguities in the ordering. For example, a merged A-B-C could be ordered differently, p.e. node A-B-C is the same as a C-A-B, C-B-A or B-A-C, and either of the four options could show behaviours regarding their miss rate. In this case, original weights are considered, so if A is more related to C than B, the final order should be C-A-B or B-A-C (reverse ordering).

Results with this optimisation present an improvement of up to 10% in performance for specific benchmarks and platforms.

The following graph gives a complete example:



The first combination of nodes is A-D because it has the highest-weighted arc, 10. After merging it and combining its arcs, the next highest arc is 8 between the nodes C-F. After merging them and combining the arcs, the resulting graph follows:



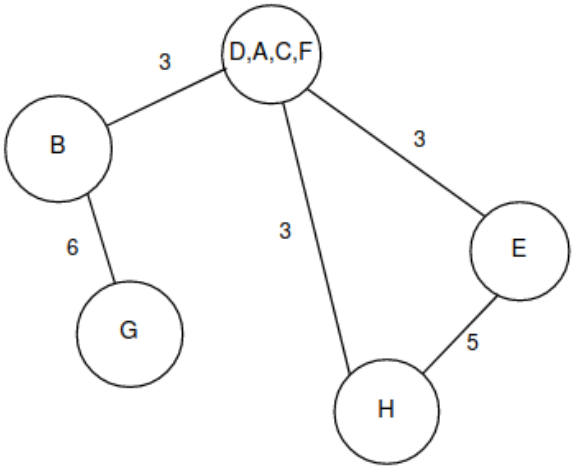
The next step will be to merge de nodes A-D and C-F, as they have the highest arc in the graph. However, some ambiguity appears in how to merge the graph, as the following combinations are possible:

A-D-C-F, F-C-D-A, A-D-F-C, C-F-D-A,
D-A-C-F, F-C-A-D, D-A-F-C, C-F-A-D

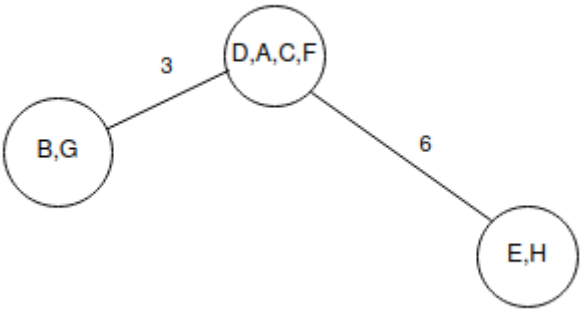
In the original graph, F is not connected to A or D, so it doesn't have to be close to them, the possibilities are reduced to:

A-D-C-F, F-C-D-A, D-A-C-F, F-C-A-D

C is more connected to A (4) than D (3) in the original graph so it should be closer to it. The final ordering shall be D-A-C-F or F-C-A-D. After the merge, arcs to other nodes are combined.

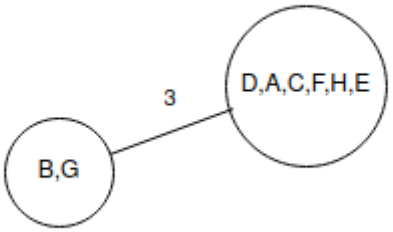


The next highest arc is B-G (6), and the following H-E (5). After the merge, the result is the following:



The next nodes to merge are D-A-C-F with E-H. Again, we have to resolve some ambiguity. The possibilities are:
D-A-C-F-E-H, H-E-F-C-A-D, D-A-C-F-H-E, E-H-F-C-A-D,
F-C-A-D-E-H, H-E-D-A-C-F, F-C-A-D-E-H, H-E-D-A-C-F

In this case, looking at the original graph, F is more connected than D(the edges of the node) to E and H, so the combinations are reduced to D-A-C-F-E-H, H-E-F-C-A-D, D-A-C-F-H-E, E-H-F-C-A-D. As F is more connected to H (3) than E(1) in the original path, the preferred order is D-A-C-F-H-E or E-H-F-C-A-D (its reverse).



In the final merge, any of the edges are connected to the others, so any of the combinations seem reasonable. However, in the original node, G is only connected to B, so the final chain should connect the nodes using B, as it is closer to the other nodes. The final solution could be G-B-D-A-C-F-H-E, E-H-F-C-A-D-B-G, G-B-E-H-F-C-A-D or D-A-C-F-H-E-B-G.

4. Basic Block Positioning

Papers on this subject:

- Pettis, K; C. Hansen, Robert - Profile Guided Code Positioning - SIGPLAN Not. 25, 6 (Jun. 1990), 16-27
- Hisu, WW; PPChang, PP - Achieving High Instruction Cache Performance With An Optimizing Compiler - The 16th Annual International Symposium on Computer Architecture, June 1989.

This second technique consists of reordering the basic block positioning in a procedure's scope before the optimiser stage of the compiler. In most applications, procedures have control paths where some basic blocks are rarely executed. A common example is the treatment of errors. The code tends to have this general structure:

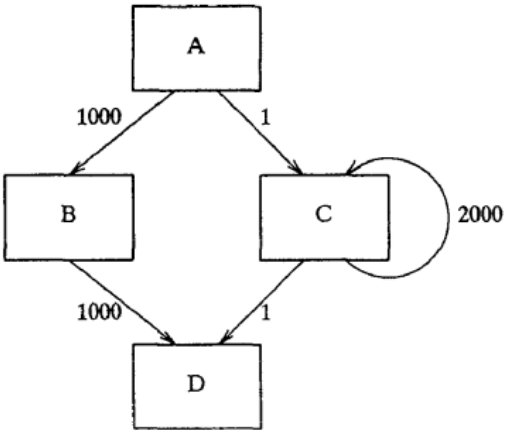
```
if (error condition) then
        error handling
rest of the code
```

This translates into a conditional jump that frequently jumps to the rest of the code lines. Having the error handling code assigned lines contiguous to the conditional evaluation lines instead of the frequently executed code is an underutilisation of the instruction cache, as those lines will rarely be executed. The objective of this technique is to move this infrequently-executed code away in the address space and to have frequently executed basic blocks close to each other.

The proposal has various advantages. It improves the instruction cache usage by reducing the number of misses and improving the number of instructions executed per cache line. It changes a frequently-taken forward branch by an infrequently-taken one, improving the predictor hit rate as they tend to predict not taken on a first appearance of a forward branch. With this optimisation, results reflect a performance improvement of up to 15% for specific benchmarks and platforms.

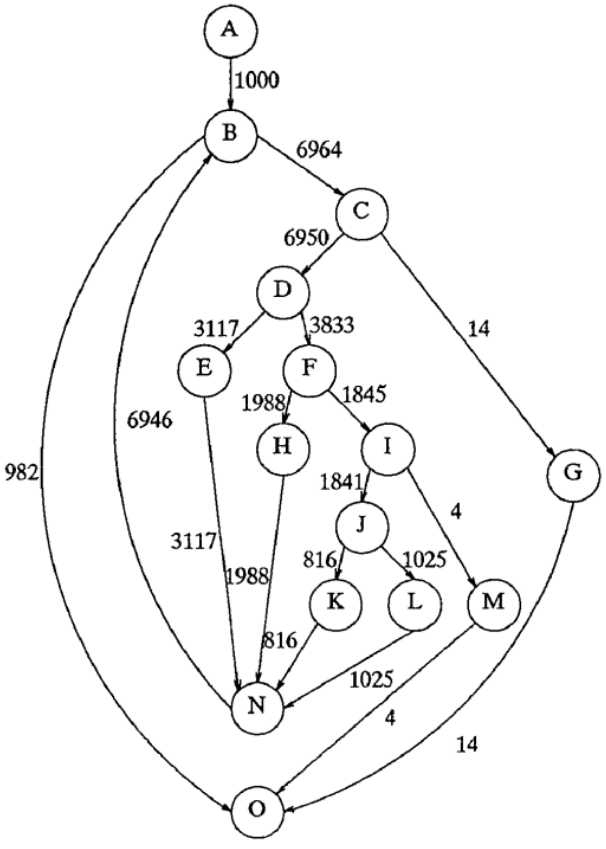
Similarly, as with procedure positioning, profiling data of execution behavioural is needed to determine the ordering of blocks.

- The measurement method selection is not straightforward, as it will have a high impact on the positioning decision. A simple solution would be to count the number of times each basic block is executed. However, this measurement doesn't give all the information required, as it hides the control transfers between basic blocks. For example, a basic block that is reached only once but is executed in a loop many times does not have to be close to any other basic block despite having a high number of executions. A better method is measuring the number of times an arc between basic blocks is traversed. In other words, each time a control transfer between procedures occurs, as shown in the following structure.



- Next, the following step defines the ordering of the basic blocks in memory. There are different strategies, one of which is the formation of chains, also known as bottom-up positioning. Each node is initially a chain with its head and tail, and two chains can be merged if the arc connects the tail of one chain with the head of another. In the previous basic-blocks graph, the chains would be formed as follows: first, A-B or B-D can be merged arbitrarily as they have the arc with the highest weight (the C loop on itself can be ignored), to the chain A-B or B-D, D or A is added, respectively, forming the chain A-B-D. C cannot be merged as its tail (C) is not connected to the head of the chain A-B-D, (A), nor the tail of the chain A-B-D (D) it is connected to the tail (C). The result is two chains, A-B-D and C. Once the chains are formed, the order between them has to be established. The objective in this final step is to have the rarely-taken chain as a forward branch, as these branches are usually predicted as not taken initially, so the final positioning in memory is A-B-D, C.

Sometimes, the basic-block graph is more complex, and the final ordering of chains is not trivial.



This example contains six conditional branches. Without entering details on how the chains are formed (which could be generated by applying the algorithm presented), it has six final chains:

- A
- E-N-B-C-D-F-H
- I-J-L
- G-O
- K
- M

The ordering of them is more complex but the objective is to have rarely taken chains with a forward branch. For example, chain E-N-B-C-D-F-H has to be placed before G-O and I-J-L,

because the usual paths of the branches in the chain (C and F) are contained on E-N-B-C-D-F-H. This ordering also applies to the rest of the branches, chain I-J-L is before M and K. Sometimes, this objective cannot be fulfilled for all the conditional branches, D for example, appears after E despite of being the rarely taken path, but since E and D are part of the same chain, we cannot reorder it. In the case of having freedom ordering the chains (G-O and I-J-L of the example), the chain with the highest connecting arc is selected (I-J-L). Therefore the final order is: A, E-N-B-C-D-F-H, I-J-L, G-O, K, M.

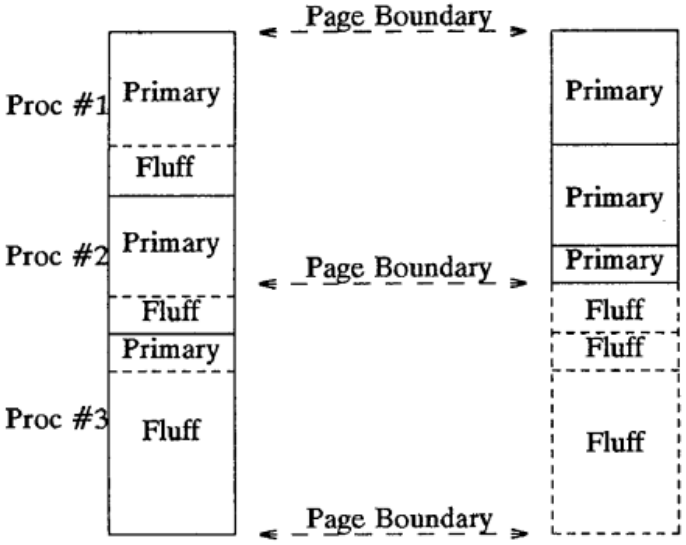
5. Procedure Splitting

Papers on this subject:

- Pettis, K; C. Hansen, Robert - Profile Guided Code Positioning - SIGPLAN Not. 25, 6 (Jun. 1990), 16-27

This optimisation combines the procedure and basic block positioning techniques. After applying them, procedures often executed together are close in memory. Within procedures, basic blocks often executed jointly are also close in the address space, leaving rarely executed blocks in the final addresses of the procedure connected with rarely taken forward jumps.

With input data used for profiling, it is common that some basic blocks related to error treatment are never executed, falling in the final addresses. These blocks are called fluff and are grouped at procedure granularity; the rest of the procedure instructions are called primary. Procedure-splitting groups fluffs globally with the goal of condensing primary blocks of procedures to reduce cache, TLB and page misses. In the following example, the primary sections of three procedures have contiguous addresses and, as a result, fit on the same page.



Conclusion

The explored techniques in the previous sections are a prime example of how, via coordinated synergy between hardware design and compiler optimizations, substantial performance gains can be obtained with relatively few conceptual idiosyncrasies and maintaining hardware complexity on the lower side of things. While this pairing process between the compiler world and the hardware world may not be without its shortcomings, we found that the proposed techniques make enough intuitive sense and show promising results so as to merit, at least, an honest chance at their implementation into both compilers and the necessary hardware support that they require. All in all, it wouldn't come as a shock to know how, indeed, these techniques served as trailblazers in a long-standing tradition of fruitful cooperation still maintained in the present day.

This is not to say, however, that the publications are free of reproach or that they aren't lacking in some aspects regarding either the experimental methodology or the feasibility of introducing the compilation-process modifications into relevant compilers of the time. For example, it was especially noteworthy how, no matter how complicated a technique might seem, no mention was ever made as to the increase in time/resource utilization of the resulting compiler after integrating the proposed technique.

Additionally, in the case of McFarling's papers, even though a series of benchmark programs are used as examples of how the proposed techniques improve the instruction cache miss rates, it is not mentioned what compiler framework or what architecture the benchmarks are performed on, whereas on the last two papers the experimental methodology is much more nuanced in its presentation and detailing.

Another source of criticism is the surface-level explanation of how the profiling of the code works: in some papers it is not mentioned how the profiling is conducted, if it is provided by the compiler or if some modifications are required. In other papers, authors mention the requirement of modifying the compiler to add this profiling code without entering more into detail concerning where and how or whether this could be easily applied to any compiler at all.

Lastly, regarding the technique of basic-block positioning, one of the aspects that surprised us is the insistence on positioning infrequently-executed code after the usual path in order to correctly predict conditional branches under the premise that backward jumps are predicted as taken and forward jumps are predicted as not taken. This could be due to the time of publication of the paper, where the usage of static branch prediction was more extended but we are not totally sure.