# Accurate Simulation and Evaluation of Code Reordering

John Kalamatianos & David R. Kaeli
Northeastern University
Department of Electrical and Computer Engineering
Boston, MA 02115, USA
E-mail: kalamat,kaeli@ece.neu.edu

## Abstract

*The need for bridging the ever growing gap between memory and processor performance has motivated research for exploiting the memory hierarchy effectively. An important software solution called* code reordering *produces a new program layout to better utilize the available memory hierarchy. Many algorithms have been proposed. They differ based on: 1) the code granularity assumed by reordering algorithm, and 2) the models used to guide code placement.*

*In this paper we present a framework that provides accurate simulation and evaluation of code reordering algorithms on an out-of-order, superscalar processor. Our approach allows both profile-guided and compile-time approaches to be simulated. Using a single simulation pass, different graph models are constructed and utilized during code placement. Various combinations of basic block/procedure reordering algorithms can be employed. We discuss the necessary modifications made to a detailed simulator of a processor in order to accurately simulate the optimized code layout.*

## 1 Introduction

The growing gap between memory and processor performance has generated a lot of interest in improving memory performance. The cost of cache misses can dominate the overall performance of an application. One software approach addressing this issue is to try to reduce the number of conflict misses in the cache using *code reordering*. The idea is to rearrange the order by which code modules are laid out in memory in order to make the most efficient use of the available memory and cache address space. Several algorithms have been proposed, most of which work at compile-time, and use either control-flow analysis [12, 13] and/or profile data [18, 15, 7]. Recently, several systems have been proposed that attempt to reorder code at run-time [16, 3].

Besides the issue of when to reorder, there remains the issue of what granule size to use when reordering. Procedure reordering [7, 6, 11], interprocedural basic block reordering [18, 9], and combined basic block and procedure reordering [15, 10, 4] are the most popular approaches.

In this paper we present an approach that provides a single pass simulation of multiple code reordering algorithms and their accurate cycle-based evaluation on a modern out-of-order superscalar processor. The code reordering framework first simulates the rearrangement of different types of code modules using different graph models and creates a memory map of the new text segment for each code reordering algorithm. Each generated memory map is fed into a modified CPU simulator that produces cycle-based statistics used to compare the performance of the modified executable with the original one.

The rest of the paper is organized as follows. Section 2 contains a brief description of previous simulation approaches on code reordering. Section 3 presents the step of generating the memory map while section 4 describes the necessary changes made to the CPU simulator in order to accurately emulate the reordered text segment of the new binary image. Section 5 concludes the paper and proposes future research directions.

## 2 Related work

SimpleScalar is an execution-driven CPU simulator that consists of a detailed model of a dynamically scheduled, out-of-order processor along with a multiple level memory hierarchy [2]. Although it can simulate the effects of speculative execution it does not provide any support for dynamically remapping memory addresses or simulating a modified code layout.

ATOM is a tracing tool that allows selective instrumentation of executables [17]. Although it can provide a wealth of information about a program's behavior, it does not consider speculative activity and can not optimize or modify an existing executable.

Alto is a link-time optimizer that enables a series of code transformations including profile-guided interprocedural basic block reordering [14]. It can generate an executable based on the optimized code layout. However, it requires large amounts of memory to construct its internal data structures, limiting the range of benchmarks that can be optimized. In addition, a binary optimized by Alto can not currently be traced via ATOM.

Spike is another link-time optimization framework utilizing, among others, profile-guided procedure and basic block reordering for Alpha executables running under Windows NT [4]. Spike, however, is not a public domain tool so source code is not available.

In general, the process of evaluating an application optimized by profile guided code reordering can be accomplished by running both versions of the executable on a native machine. The final execution times is the primary factor deciding the usefulness of the optimization. More information can be obtained if we can use performance counters and measure certain sensitive microarchitectural parameters before and after optimization, such as I-cache miss ratios, branch prediction ratios, etc.

Another approach is to employ execution-driven simulation which allows us to measure a significantly larger number of parameters on a range of microarchitectures. It is very difficult to generate detailed execution times. None of the above methods though, permits the exact measurement of factors such as the number of extra instructions added after code repositioning, cache misses among heavily interacting procedures, speculative traffic before and after code reordering, etc.. This is not feasible even if we use a tracing tool such as ATOM [17] because we need a way to tag the instructions/basic blocks/procedures of interest and collect data related only to those.

One solution is to use the optimizer to instrument the code we need to examine and run the application to collect more accurate statistics. That presupposes that the tool has lightweight instrumentation capabilities to allow minimal overhead during the tracing step.

Our approach simulates code reordering on a statically linked executable. We allow multiple code reordering algorithms to be simulated in a single pass, therefore we reduce the overall effort of exploring different configurations. Furthermore, we are able to label (at an instruction level of granularity) different parts of a program and collect statistics. Accurate evaluation of the modified executable is achieved through execution-driven and cycle-based simulation on a modified CPU model.

## 3 Simulating Code Reordering

The current implementation of our code reordering framework utilizes profile data to build graph models that are used to guide code layout. Profile data are used to weight the edges of the graph models, hence it is possible to use compile-time generated edge weights to accomplish the same task. We have implemented algorithms to rearrange procedures and basic blocks to achieve a lower number of cache conflicts. Profile data are gathered with the ATOM tracing tool on a Compaq Alpha-based workstation running Unix [17].

First, we create a memory map for the entire text segment of the application. There are two kinds of entries in the memory map. The first one marks the beginning of a procedure (and the end of the previous procedure) and consists of a unique procedure ID, the procedure's starting virtual memory address and the procedure size measured in instructions. Such entries will be referred to as *p-entries*. The second entry type corresponds to a single basic block and consists of the block's starting virtual memory address, a unique ID and the block size in instructions. It will be referred to as a *bb-entry*. The p-entries are used to construct a procedure graph, where every node models a procedure.

Next, we construct a procedure graph and weight its edges based on profile data. We generate a trace, with every trace entry consisting of the basic block ID, the block's size and its associated procedure ID. Currently, our framework is capable of weighting edges according to three different graph models: (a) a *Call Graph* (CG), (b) a *Temporal Relationship Graph* (TRG) and (c) a *Conflict Miss Graph*. All of these models have been successfully applied in procedure placement: CG in [15, 7], TRG in [6, 1] and CMG in [11]. Although the CG exploits interaction between procedures that directly call each other, the TRG and the CMG exploit temporal interaction between procedures that lie even further apart in the call chain, or even between procedures that lie in different call chains [5]. To capture this interaction, constructing a TRG or a CMG is more computationally intensive than a CG. After building these graphs, a pruning step is applied on the graph edges. The idea is to remove those edges containing minimal information. Every procedure that has at least one edge after pruning is called *popular*. Pruning allows the placement algorithm to focus on popular procedure pairs. It also reduces the time spent placing procedures.

We have also added the ability to simulate intraprocedural basic block reordering into our framework. The particular algorithm we implemented is similar to the one proposed in [15]. The goal is to make most branches fall through. Currently, basic block reordering is applied only to popular procedures, but could be easily extended for the remaining procedures in the program. Basic block reordering requires weighting the control flow edges of a procedure. We use profile data to perform weighting, but compile-time generated edge weights could also be used [8]. Modifying the order of basic blocks requires a fix-up step to guarantee correct program semantics. Since we are simulating the reordering step, we have detected all different cases and marked every basic block with a label. The label uniquely defines each case or denotes that no fix-up was required. All the cases handled by our algorithm are shown in Fig. 1. Notice that the fix-up step works independent of the employed basic block reordering algorithm.

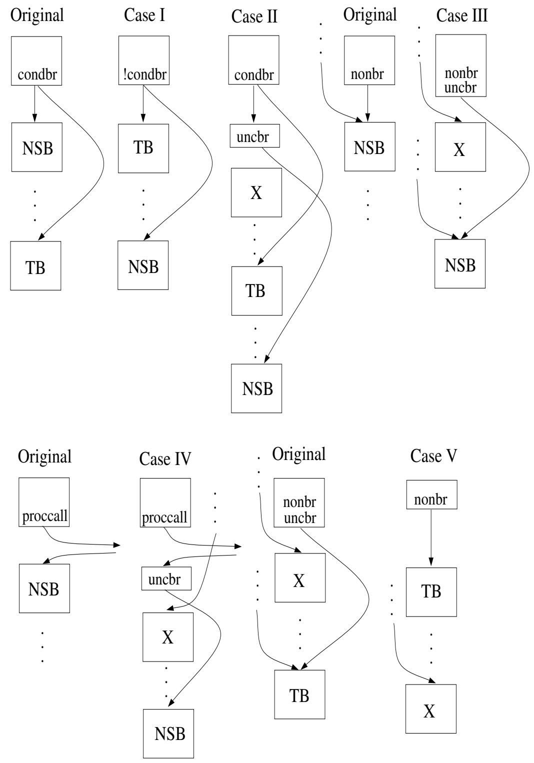


**Figure 1. Code fix-up cases for intraprocedural basic block reordering.**

Every box in Fig. 1 corresponds to a basic block. $TB$ and $NSB$ are the Target Block and Next Sequential Block, respectively. $condbr$ is a label for a conditional branch, $uncbr$ for an unconditional branch, $proccall$ for a procedure call and $nonbr$ for a non-branch instruction. Every basic block has such a label attached to it, indicating the type of the last instruction in the block. As it can be seen from Fig. 1 several cases arise during code fix-up:

- Case I: the opcode of the conditional branch must change to its dual.
- Case II and IV: a new basic block consisting of one unconditional branch must be inserted.
- Case III: the basic block must be extended with one unconditional branch.
- Case V: the unconditional branch found at the end of the block can be removed.

If the unconditional branch deleted in Case V is the only instruction of the basic block, then the basic block is not removed from the list of basic blocks in the procedure. It remains in its original position, being normally labeled with a Case V identifier. Its size has been set to 0, indicating that it is an empty block. The output of the basic block reordering step is an ordered list of basic blocks for every popular procedure. This list includes all additional basic blocks introduced by the fix-up step. The list is attached to the node of the procedure graph representing the procedure whose blocks have been rearranged.

Reordering procedures in the text segment is performed after pruning. The remaining edges are sorted in decreasing weight order, and popular procedures are considered in pairs and placed in the *cache address space* in such a way so that conflicts between them are minimized. This process is known as *cache line coloring* [7]. Recently, we have extended this approach to multiple levels of caches. Basically, the coloring step of two popular procedures has to place procedures into multiple cache address spaces with the goal of minimizing the conflicts between the two procedures on all cache levels [5]. Reordering for single or multiple cache levels is controlled by a parameter passed to the coloring routine. Placing procedures in the cache is equivalent to assigning a starting cache set to every procedure. Notice that unpopular or unactivated procedures are not mapped into the cache.

The last step is associated with positioning all procedures in the text segment address space. This is equivalent to assigning a starting virtual memory address for every procedure/basic block in the executable. We first consider popular procedures, since their placement is subject to the cache coloring constraints. We sort these procedure in a list, based on their assigned cache set. If we exercise multiple cache level coloring, we create a number of lists equal to the number of cache levels. Then we pick a procedure at a time, starting from memory address 0, and find all valid memory addresses (according to the cache coloring constraints) where the procedure can be assigned. The selection of the memory address depends on the minimum distance from the boundaries of the previously mapped procedure. If the minimum distance is larger than the page size, then we relax one of the coloring assignments (usually the one imposed by the lowest level cache) and iterate. Checking against the page size is a heuristic that helps keep the final size of the executable, and its working set size, small.

After assigning a starting memory address for every procedure, we place all remaining procedures in memory. First, we attempt to fill in any gaps in the memory address space that were created after placing popular procedures. A best-fit algorithm is used. Second, all remaining procedures are appended at the end of the image. All assigned memory addresses are essentially offsets from address 0, so that the entire memory map is relocatable. In addition, all procedures are aligned within cache line boundaries.

After mapping all procedures to memory, we proceed by mapping basic blocks. Assigning a memory address to a block when no basic block reordering is done, is simple. The first basic block in the ordered list is assigned with the procedure starting memory address and the remaining blocks simply follow. When basic block reordering is performed, special care has to be taken because fix-up code has been added. For case I in Fig. 1, no special attention has to be paid because the code layout is not affected. Cases II, III and IV add one extra instruction. Therefore, the basic block immediately following the one containing the fix-up is placed 4 bytes (instruction size for the Alpha ISA) after the new memory address of the later. In Case V, an instruction is deleted. All instructions from the modified basic block, excluding the deleted one, are mapped normally. The deleted instruction is mapped onto the starting memory address of the immediately following basic block. The later is mapped right after the new address of the last (non-deleted) instruction. Fig. 2 illustrates the solution.

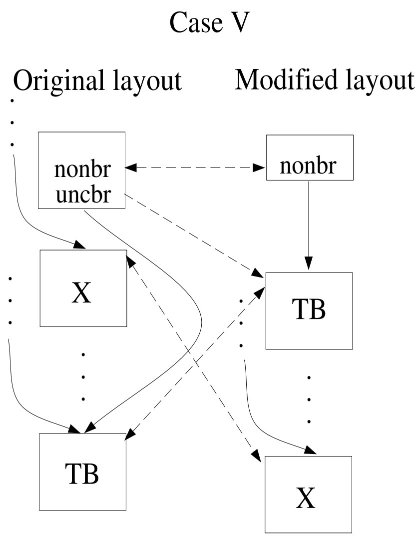


**Figure 2. Memory Mapping for Case V.**

Dashed lines represent the mapping between the original and the modified addresses. This mapping scheme not only models the effect of deleting instructions in the final executable, but also facilitates its simulation in the CPU model (see section 4). The final memory map for the text segment has the following format:

`old_address new_address block_size status`

The number of entries in the memory map file is equal to the number of basic blocks in the original executable. $Old\_address$ and $new\_address$ are the starting memory address of the block in the original and modified layout, respectively. $Block\_size$ is the size of the block in instructions in the unoptimized binary and $status$ is the identifier for the code fix-up cases.

Our code reordering framework can simultaneously build any of the three supported graph models [1] from the same profile data. It will generate different edge weights for each graph and attach multiple starting colors to every node (each corresponding to a placement according to a specific graph). The basic block reordering step is independent of the particular graph used, and is performed only once. The coloring algorithm is performed separately for every graph. It can run twice for each graph, applying either single or multiple level cache coloring. All configurations use the same cache hierarchy which is specified in a header file.

The memory placement routine is called by default four times for every graph, based on the four combinations of single/multiple cache coloring and basic block reordering (or no reordering). The user is able to limit the number of configurations by turning off either the basic block reordering or the multiple cache level option. Similarly, the memory map generator is called four times and creates a different map for each configuration. A memory map is distinguished by the graph model, the basic block reordering decision and the number of cache levels used in procedure coloring.

## 4 Evaluating Code Reordering on a Superscalar Processor

We modified SimpleScalar v3.0 running the Alpha ISA [2] to model the effects of code reordering. The most critical pipeline stages that require modification are the fetch and dispatch stage of the pipeline. Fig. 3 shows a simplified diagram of the simulator's structure.

The solid lines show part of the flow of the simulator pipeline. During the fetch stage the fetch engine decides upon the next address and the predicted PC is used to access the I-cache and the I-TLB. The fetch engine utilizes components that predict the outcome and target of every branch in the instruction stream. Since SimpleScalar is an execution-driven simulator, it needs to access the actual application image and load the instructions. This type of flow is displayed with the dashed lines in Fig. 3. During the fetch stage, instructions are accessed from the application image and fetched (using the instruction fetch queue of the simulated processor) to the dispatch stage. At dispatch, the instructions are natively executed.

Since we simulate repositioning of blocks and procedures in the text segment, we can remap addresses on the fly. Remapping is done by employing two different hash

[1]Although memory requirements may prevent the framework from building all the requested graphs.

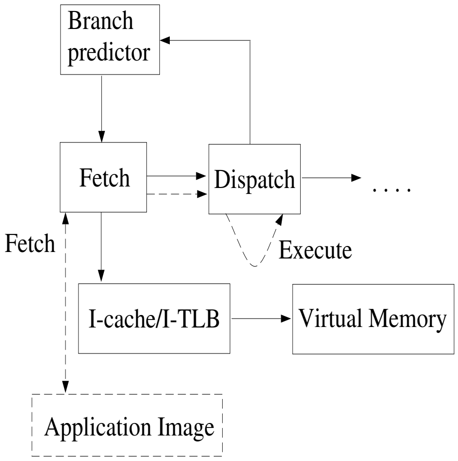


**Figure 3. CPU Simulator structure.**

tables. The *direct* hash table returns a modified address, given an original address as the key, and the *inverse* hash table performs the inverse remapping. Building these tables is done by traversing the memory map and creating an entry for every instruction found in the text segment. The direct table contains an entry for every instruction in the unmodified executable. In order for the mapping to be correct, an address in the original memory space can not be mapped onto two different addresses in the modified layout. Two different original addresses are allowed to be mapped onto the same address in the modified layout (see Fig. 2). In order to enforce the same principle in *inverse*, only non-deleted instructions are allowed to have an entry in that table. As shown in Fig. 2, an address in the new layout can be mapped only to one address in the original layout. By not creating an inverse table entry for deleted instructions, we maintain a valid inverse mapping.

An additional hash table, called *status*, is created for the last instruction of every basic block that was labeled with a Case identifier other than unmodified after code fix-up. That includes basic blocks whose last instruction was deleted. *Status* simply returns the Case identifier when presented with an instruction address as a key. For example, the last instruction (conditional branch) of the first block illustrated in Case II of Fig. 1 has an entry in the *status* table. The key is the address of the branch in the original text segment and the output is the Case II identifier which triggers a Case II scenario. That allows the CPU model to simulate the effects of an unconditional branch on the fly. This is done only if the branch is predicted non-taken.

Since we perform code fix-up only for certain basic blocks that belong to popular procedures, the memory requirements of the *status* hash table are not prohibitive. In addition, when requesting the status of an instruction whose basic block was not modified, the status table will not find an entry and simply returns the unmodified Case identifier. Hence, no special action is taken by the CPU model.

Although the original addresses still flow through the pipeline we make calls to the *inverse* and *status* hash tables every time we need to remap addresses. The first place where remapping is performed is the memory hierarchy. All code requests are made via the instruction fetch stage to the L1 instruction cache, hence, we modified the call to the I-cache and I-TLB management routines so that the incoming addresses are from the reordered text segment. The modified addresses flow through the entire cache hierarchy to main memory. All accesses to the page table are performed with the modified addresses, thus all TLB and page table statistics reflect all changes in the text segment.

We do not make any I-cache/I-TLB requests when a deleted instruction is detected in the dynamic instruction stream. Any extra unconditional branches are inserted in an queue (in program order) and are tagged with a time stamp. The time stamp specifies the clock cycle on which they should be fetched from the I-cache/I-TLB. This is always set to one cycle after the clock cycle upon which the instruction immediately preceding them in the modified executable is fetched from the memory hierarchy. All such branches are issued to the memory system when their scheduled time stamp expires. Using this mechanism, we are also able to compute I-cache misses, I-TLB misses, page faults and clock cycles that are due solely to those extra instructions.

Further modifications at the instruction fetch stage of the CPU simulator include, properly adjusting instruction fetch bandwidth consumption when extra or deleted instructions are encountered in the dynamic instruction stream. The current implementation of the instruction fetch stage discontinues fetching when a break in the control flow is encountered. This is honored whenever an extra unconditional branch is encountered. The instruction fetch queue allocates entries for both extra and deleted branches. The later is an inaccuracy since deleted branches should not be visible to the microarchitecture. The main reason for doing this is that all instructions in the fetch queue are later dispatched and executed by the ISA simulator. Since we have to execute all instructions from the original image to preserve correct execution-driven simulation, we add the deleted instructions in the fetch queue.

An alternative solution would require significant modifications to SimpleScalar. More specifically, we would need to decouple the resources used to feed the ISA simulator from those used in the CPU model. Adding a new queue, that would be used exclusively to transfer the instructions to the ISA simulator, seems to be the proper solution. The fetch queue would contain a subset of those instructions, namely only those marked as non-deleted as specified by the *status* table. In order to minimize the negative impact of the deleted branches on the final statistics, we eliminate any deleted branches after they are used by the ISA simulator. As a result, they do not occupy any instruction window entries, execution slots or waste retirement bandwidth.

The branch predictor is always indexed with the modified instruction addresses. All target mismatches (branch misfetches) are detected using modified addresses. No prediction decision is requested from the branch predictor when a deleted instruction is detected. Instead, we assume that the next sequential instruction is fetched. If an extra unconditional branch is found, we do not query/update the branch predictor. That greatly simplifies the mapping scheme and the simulation methodology for extra unconditional branches (Cases II, III and IV). It does generate optimistic results because we assume perfect target prediction for those branches (no BTB prediction is required) and aliasing in the BTB is reduced (fewer targets are stored). However, we do always discontinue fetching in the presence of extra unconditional branches. That is pessimistic since the BTB should be able to provide the correct target most of the time. In practice, we found that the low dynamic frequency of the cases and the high prediction accuracy of unconditional branches had a minimal impact on the generated statistics.

When the branch predictor predicts that control flow should continue from the next sequential PC ($pred\_PC$), the following adjustments are necessary:

- Case I: $pred\_PC = remap(remap(PC, direct) + 4, inverse)$
- Case V: $pred\_PC = remap(remap(PC, direct), inverse)$
- Cases II,III,IV: $pred\_PC = PC + 4$

$Remap$ takes as input two arguments, an address used as a key and a hash table. It returns a new address based on the mapping of the selected hash table. Notice that in Case I, the next sequential basic block in the reordered layout is the target basic block (TB) of the original layout (see Fig. 4). By using the formula shown above, we guarantee that the predicted PC will point to the TB in the original layout if the branch is predicted non-taken. Notice that we still have to use original addresses in order to access the text segment of the binary and continue with the program execution. Therefore, although the branch predictor is indexed with modified addresses, it keeps original addresses in its BTB. Original addresses are also kept in the Return Address Stack (RAS).

When an instruction has been deleted (Case V), the branch predictor will determine that the control flow continues with no breaks, in order to simulate the effect of statically deleting the unconditional branch. Therefore, the next sequential instruction for Case V is the first instruction of TB in the original layout. The particular memory mapping of Case V allows the above formula to successfully access the starting address of TB, as shown in Fig. 4. In the original layout, the predictor would simply predict the branch as taken and no next sequential PC would be computed.

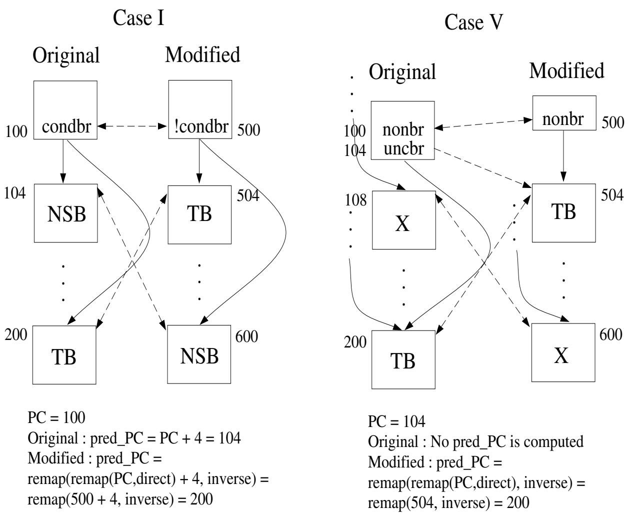


**Figure 4. On the fly remapping of addresses to accurately compute the next sequential PC.**

The dashed lines in Fig. 4 model the mapping of addresses via the hash tables. A two-headed line is created after merging two one-way arcs that exist for the instructions in those addresses. Notice that deleted instructions do not have an entry in the inverse hash table. This is denoted by the single-headed line.

During the dispatch stage of the pipeline, we also remap instruction addresses when the branch predictor is updated (our model is configured to speculatively update the predictor at the dispatch/decode stage of the pipeline). Contrary to the memory hierarchy case, we index the branch predictor with modified addresses, but store the original addresses in the BTB and RAS as mentioned above. We check branch prediction decisions based on the modified addresses and pass/return the original addresses to the predictor. The conditions that usually need to be checked regarding branch prediction are:

- $C_1$: Is the branch taken?
- $C_2$: Is the branch predicted taken?
- $C_3$: Is the prediction correct?

Condition $C_1$ is checked by comparing the actual next program counter, $next\_PC$, with $PC + 4$. $C_2$ is checked by comparing the predicted program counter (as it is returned by the branch predictor), $pred\_PC$, with $PC + 4$. $C_3$ is examined by comparing $pred\_PC$ with $next\_PC$.

When code reordering is simulated, both the $next\_PC$ and $pred\_PC$ need to be remapped via the direct hash table since we keep original addresses in the BTB and RAS. We also need to remap $PC + 4$ from the original text segment to the modified one as follows:

- Case I, IV : $remap(PC, direct) + 4$
- Case III, II: $remap(PC + 4, direct)$
- Case V: $remap(PC, direct)$

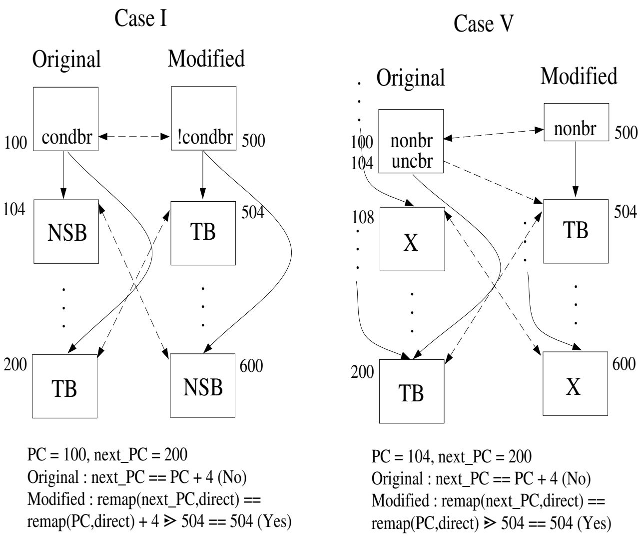


**Figure 5. On the fly remapping of addresses to accurately test condition $C_1$.**

Fig. 5 shows an example of how these formulas properly guide the branch predictor update decisions. Two examples are shown, one for Case I and another for Case V. For Case I, in the original image the branch is found to be taken since $next\_PC$ points to the target block, TB. We check condition $C_1$ by comparing $PC + 4$ with $next\_PC$, which evaluates true if the branch is taken (indicated by the $No$ label in Fig. 5). If we assume that the layout was modified according to Case I, we can not use the same formula because the branch is not-taken. Instead, we must compare $remap(next\_PC, direct)$ with $remap(PC, direct) + 4$, which will correctly indicate that the branch is actually non-taken in the new layout. A similar scenario occurs with Case V, where the direct mapping provides with the correct answer.

The above formulas for Cases III and II will always determine that the extra unconditional branch is non-taken. The reason for this is that we do not query or update the branch predictor for these extra branches. If we use the formula of Case I for Cases II and III (which would be the proper address if we wanted to fully simulate the effects of unconditional branches), then we may falsely activate a fetch squash if a misfetch is detected. By using the current formulas, the branch is always correctly predicted.

Deleted unconditional branches neither activate any prediction/update operation for the branch predictor, nor do they produce any squash operations in the pipeline. However, we still need to check for $C_1$ and $C_2$ when the simulator decides about the speculative nature of the instructions being dispatched. Since the instruction is deleted, the above formulas guarantee that the control-flow will not be disrupted. Case IV is also special because it can not be simulated when an instruction labeled with Case IV is found in the dynamic instruction stream. This is because the control flow jumps to the callee after the instruction triggering Case IV, which is always a procedure call. We do simulate the unconditional branch when the first instruction of the return basic block is found. The condition for triggering a Case IV scenario is: $status(PC - 4) = CaseIV$.

All statistics related to dynamic instruction counts take into account extra/deleted instructions. All decision points in the simulator regarding the speculative instruction nature or the transition to mispeculation are tested with the modified addresses. No checking is done when a deleted instruction is encountered.

Special attention has to be paid to indirect jumps, which are always considered to be taken in the original executable. After basic block reordering, the most frequently executed basic block is usually placed right after the block ending with a jump instruction. Checking $C_1$ would indicate that the branch is not taken if control flow was transferred to that block. This would prevent the BTB from allocating an entry for that indirect jump, leading to an unusually high number of misfetches for this class of branches. Correcting this behavior required adding a new case for indirect jumps that would force the branch predictor to consider the jump as taken.

In order to measure the overhead of code reordering in the modified CPU simulator, we measured the elapsed time for completing the simulation over a predefined interval measured in the number of committed instructions. We compared the elapsed time to the elapsed time we needed to complete the simulation over the same interval with no code reordering. We used a subset of SPECINT'95 benchmarks (perl, gcc) along with a set of C++ applications. Overall, the average simulation time is 3.4 times higher when both basic block and procedure reordering is enabled and 1.8 times higher when only procedure-based reordering is performed. The reason is that basic block reordering requires code fix-up, which is time consuming to simulate. Procedure reordering does not delete or add any instructions, therefore it does not activate any code that considers the cases of Fig. 1.

## 5 Conclusions

We have presented a framework that enables the accurate simulation and evaluation of code reordering techniques on statically linked executables. Our approach allows multiple code reordering schemes to be simulated with a single pass. We can simultaneously build different graph models guiding a cache line coloring algorithm for single/multiple level cache levels. We also allow intraprocedural basic block reordering to be optionally simulated. Moreover, we showed how to modify a detailed CPU simulator to properly model the negative/positive effects of rearranging the text segment of an application and generate cycle-based results.

Future work includes decoupling the execution of the application's code from the internal data structures of the CPU simulator in order to increase the accuracy in modeling the effects of code fix-up. Modeling the effects of extra unconditional branches could also be added to further improve the accuracy of our simulator. Fine-tuning the simulator code for performance will also reduce the simulation time for modeling code reordering in the CPU simulator.

## References


[1] I. Bahar, B. Calder, and D. Grunwald. A Comparison of Software Code Reordering and Victim Buffers. In *Proceedings of the International Workshop on the Interaction between Compilers and Computer Architecture*, October 1998.

[2] D. Burger and T. Austin. The SimpleScalar Tool Set, v2.0. Technical Report UW-CS-97-1342, University of Winsconsin, Madison, June 1997.

[3] J. Chen and B. Leupen. Improving Instruction Locality with Just-In-Time Code Layout. In *Proceedings of the UNENIX Windows NT Workshop*, August 1997.

[4] R. Cohn, D. Goodwin, and P. Lowney. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.

[5] N. Gloy. *Code Placement using Temporal Profile Information*. PhD thesis, Harvard University, 1998.

[6] N. Gloy, T. Blackwell, M. Smith, and B. Calder. Procedure Placement using Temporal Ordering Information. In *Proceedings of the International Symposium on Microarchitecture*, pages 303–313, December 1997.

[7] A. Hashemi, D. Kaeli, and B. Calder. Efficient Procedure Mapping using Cache Line Coloring. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[8] A. Hashemi, D. Kaeli, and B. Calder. Procedure Mapping using Static Call Graph Estimation. In *Proceedings of the Workshop on Interaction between Compiler and Computer Architecture*, February 1997.

[9] R. Heisch. Trace-Directed Program Restructuring for AIX Executables. *IBM Journal of Research and Development*, 38(5):595–603, September 1994.

[10] W. Hwu and P. Chang. Achieving High Instruction Cache Performance with an Optimizing Compiler. In *Proceedings of the International Symposium on Computer Architecture*, pages 242–251, May 1989.

[11] J. Kalamatianos and D. Kaeli. Temporal-Based Procedure Reordering for Improved Instruction Cache Performance. In *Proceedings of the International Conference on High Performance Computer Architecture*, pages 244–253, February 1998.

[12] K.Gosmann and C. et.al. Code Reorganization for Instruction Caches. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 214–223, January 1993.

[13] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[14] R. Muth, S. Debray, and et.al. Alto: A Link-Time Optimizer for the DEC Alpha. Technical Report TR-98-14, University of Arizona, December 1998.

[15] K. Pettis and R. Hansen. Profile-Guided Code Positioning. In *Proceedings of the International Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[16] D. Scales. Efficient Dynamic Procedure Placement. Technical Report WRL-98/5, Compaq WRL Research Lab, May 1998.

[17] A. Srivastava and A. Eustace. ATOM : A System for Building Customized Program Analysis tools. In *Proceedings of the International Conference on Programming Language, Design and Implementation*, pages 196–205, June 1994.

[18] J. Torrellas, C. Xia, and R. Daigle. Optimizing the Instruction Cache Performance of an Operating System. *IEEE Transactions on Computers*, 47(12):1363–1381, December 1998.