Instruction Fetch Architectures and Code Layout Optimizations

ALEX RAMIREZ, JOSEP L. LARRIBA-PEY, AND MATEO VALERO, FELLOW, IEEE

Invited Paper

The design of higher performance processors has been following two major trends: increasing the pipeline depth to allow faster clock rates, and widening the pipeline to allow parallel execution of more instructions. Designing a higher performance processor implies balancing all the pipeline stages to ensure that overall performance is not dominated by any of them. This means that a faster execution engine also requires a faster fetch engine, to ensure that it is possible to read and decode enough instructions to keep the pipeline full and the functional units busy.

This paper explores the challenges faced by the instruction fetch stage for a variety of processor designs, from early pipelined processors, to the more aggressive wide issue superscalars. We describe the different fetch engines proposed in the literature, the performance issues involved, and some of the proposed improvements. We also show how compiler techniques that optimize the layout of the code in memory can be used to improve the fetch performance of the different engines described.

Overall, we show how instruction fetch has evolved from fetching one instruction every few cycles, to fetching one instruction per cycle, to fetching a full basic block per cycle, to several basic blocks per cycle: the evolution of the mechanism surrounding the instruction cache, and the different compiler optimizations used to better employ these mechanisms.

Keywords*—*Branch prediction, code layout, instruction fetch, trace cache.

I. INTRODUCTION

Superscalar processors represent the major trend in high-performance processors in the past several years [54]. These processors naturally evolve from pipelined architectures, and try to obtain higher performance in two ways: first, by simultaneously executing several independent instructions in parallel; second, by increasing the clock rate to speed up instruction execution.

Manuscript received January 17, 2001; revised June 15, 2001. This work was supported by the Ministry of Education and Science of Spain under Contract TIC-0511/98 and by CEPBA.

The authors are with the Universitat Politecnica de Catalunya, D6 08034 Barcelona, Spain (e-mail: aramirez@ac.upc.es; larri@ac.upc.es; mateo@ac.upc.es).

Publisher Item Identifier S 0018-9219(01)09684-0.

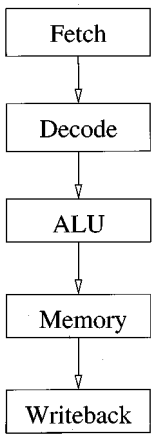


Fig. 1. Example stages of instruction execution.

When designing a high-performance processor, it is important to keep all parts of the processor balanced, avoiding bottlenecks whenever possible. For example, as shown in Fig. 1, if we design a high-performance processor capable of executing five ALU operations at once, it is also important to ensure that we can feed the ALU stage and retire those instructions without stalling the pipeline. This means fetching and decoding at least five instructions per cycle, to keep the ALU stage busy, and writing results and graduating instructions at a fast enough rate.

But the fetch stage does not behave like other pipeline stages in the sense that it can not be widened by simply replicating it, or adding more functional units. It has to follow the control path defined by branch instructions, which have not been executed yet. The fetch stage first evolved to include branch prediction, and used it to fetch instructions from speculative execution paths.

This ability to follow speculative paths independently of the execution stages leads to a decoupled view of the processor, as shown in Fig. 2. The fetch engine reads instructions from memory and places them in an instruction buffer. Then,

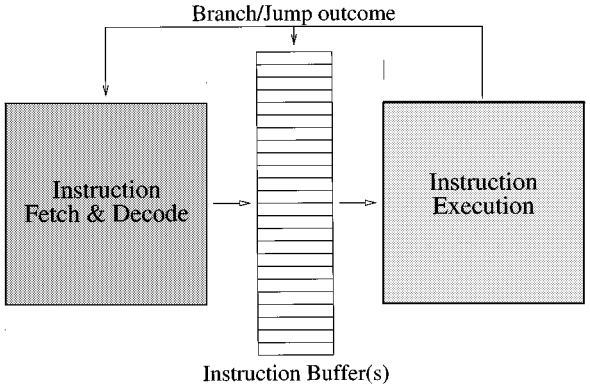


Fig. 2. Decoupled view of the processor: a fetch engine produces instructions, and an execution engine consumes them.

an execution engine reads instructions from the buffer and generates the required results, providing feedback to the fetch engine regarding the actual outcome of branch instructions.

But the fetch engine also has had to adapt to the increasing clock rates of each new processor generation. This usually implies moving the more complex parts of the fetch process out of the critical path, and adding a new hierarchy to the different memory structures used, like the branch prediction tables.

This paper shows how the fetch engine evolves with each new generation of processors, increasing fetch performance to keep up with the execution engine. We show the different fetch mechanisms proposed and the different performance issues related to those fetch engines.

This paper also shows how compiler techniques such as trace and superblock scheduling [11], [20], [30], or code layout optimizations [4], [15], [19], [25], [42]–[45], [59] affect all aspects of fetch engine performance, from instruction cache misses, to branch prediction accuracy, and the effective fetch width. Throughout the paper, we show detailed results on how the use of the different reordering techniques improves the performance of the discussed fetch engines.

As we advance through the paper, we show how the introduction of novel architecture features required the fetch engine to evolve, introducing newer challenges, and introducing newer performance metrics to measure fetch performance.

For example, while executing only one instruction at a time (with no pipeline stage overlapping), the only obstacle faced by the fetch mechanism is the memory latency. The time it takes to read an instruction from memory is computed together with the time taken to execute the instruction. If the memory latency is large, it quickly becomes the major component of the processor time. Minimizing this memory delay is usually approached using cache memories and prefetching schemes. Given the popularity of caches, the fetch performance metric used for these processors is usually the instruction cache miss rate.

Pipelined processors quickly became the major organization technique used by computer designers to reach higher single-processor performance [31], [32]. By overlapping the execution of several instructions, pipelined processors allow the processor to complete the execution of one instruction every cycle, instead of taking several cycles to execute one instruction.

In addition to the instruction cache performance, the fundamental problem of the fetch engine of pipelined architectures is that branch instructions disrupt the flow of instructions through the pipeline. The problem arises because the outcome of the branch is not known until several cycles after it has been fetched, depending on the pipeline depth. By the time the branch has been fully resolved, several instructions may have entered the pipeline and may need to be squashed. If instruction squashing can not be implemented, the processor pipeline is stalled for several cycles every time a branch is fetched.

The fetch performance metric used for these processors is the branch execution cost, which measures how many cycles it takes to effectively execute a branch, including any delays introduced in the pipeline. The way branches are executed, the number of pipeline stages, and other factors, will determine the amount of execution slots lost to each branch executed. It was the need to keep fetching instructions without waiting for a branch to resolve which quickly led to branch prediction, and speculative instruction execution.

Superscalar processors attempt to obtain higher execution performance by exploiting instruction level parallelism (ILP), which basically means replicating the different pipeline stages to execute several instructions in parallel. In order to execute multiple instructions per cycle, it becomes imperative to fetch multiple instructions per cycle. But simply duplicating the number of functional units in the fetch stage does not resolve the problem. The fetch engine evolves to increase its capabilities and read a full basic block of instructions from memory in a single cycle [4], [48], [64].

With superscalar processors, a single-cycle delay represents an undetermined number of wasted instructions, which increases the importance of an accurate branch prediction mechanism. The performance metric changes from branch execution cost to branch execution penalty, which measures the number of wasted cycles due to a branch instruction. Given that the number of instructions executed per cycle varies depending on the available ILP, the number of lost cycles can not be easily translated to the number of lost instructions.

Wide superscalar processors try to achieve even higher performance by exploiting larger amounts of ILP, using large numbers of functional units, speculative techniques for memory disambiguation, and value prediction to execute eight to 16 instructions per cycle. An execution engine capable of issuing 16 instructions per cycle requires a fetch engine capable of supplying at least the same amount of instructions, which requires even higher fetch performance.

With an average basic block size of 5–6 instructions, integer codes require multiple basic blocks per cycle to feed a 16-wide execution engine. This need to fetch instructions past several branches in a single cycle introduces two new challenges: first, it is necessary to obtain multiple branch predictions in a single cycle; second, instructions belonging to different basic blocks may not be stored in sequential memory positions, not even in the same cache line. This

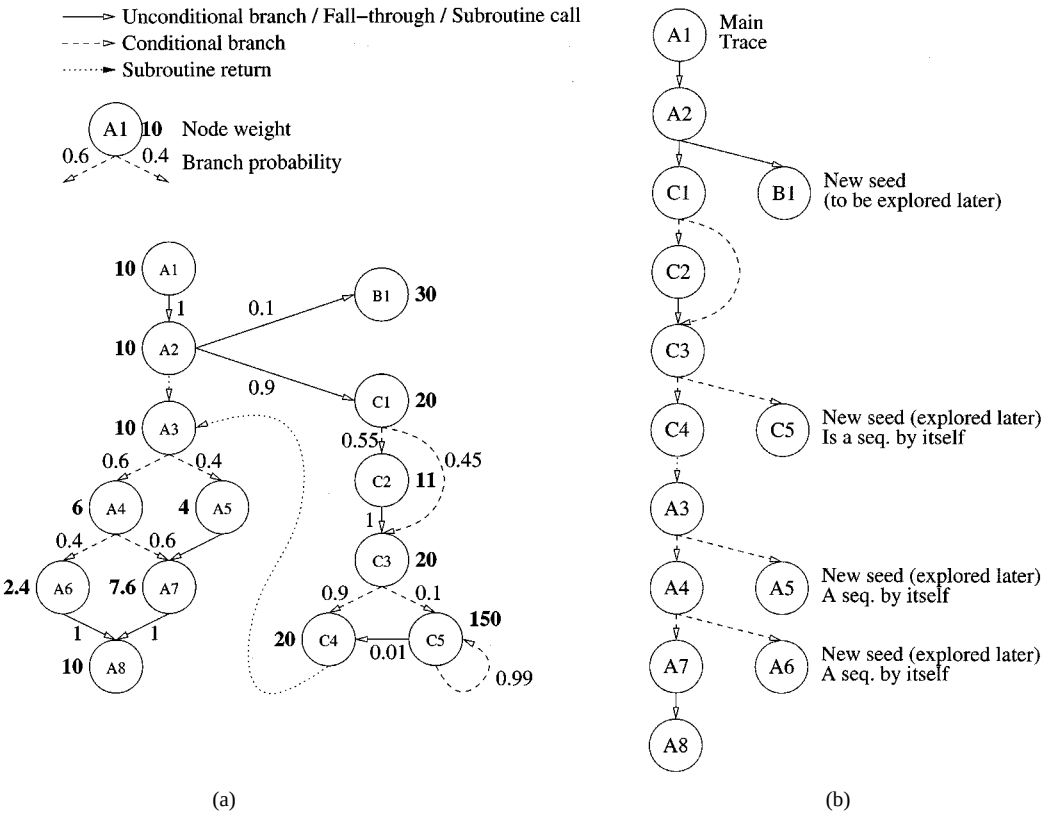


Fig. 3. Example of the software trace cache basic block chaining algorithm. Basic blocks are mapped so that the execution trace is consecutive in memory.

introduces the problem of the effective fetch width of the fetch engine [8], [13], [41], [50], [61].

To this point we have identified three elements that determine fetch performance: instruction cache misses, branch prediction accuracy, and the effective fetch width.

This paper does not discuss the fetch engine of VLIW and CISC processors. Although they present interesting and hard-to-solve problems of their own, we will concentrate instead on the fetch engine of superscalar processors. Also, we will concentrate on the description of the branch architectures used in the fetch engine. Other elements found in the front-end engine of superscalar processors, such as the instruction decode and rename logic stages, will not be treated here.

It is also worth noting that we did not reproduce all prior work to generate the results for this paper. Instead, we present results taken from previously published work. This means that the data presented in two separate graphs was generated using two different simulation environments, and that their results should not be compared directly.

The rest of this paper is organized as follows. In Section II, we describe code reordering optimization algorithms that will been used throughout the paper to improve fetch performance. Section III shows how branch prediction and speculative execution were introduced in pipelined architectures. Fetching multiple instructions per cycle is approached in Section IV with superscalar architectures, and Section V explores wide superscalar architectures from the fetch perspective, focusing on the trace cache. Finally, in Section VI, we present our concluding remarks.

II. Code Reordering Techniques

Along this paper, we show how code layout optimizations improve the different aspects of fetch performance, from instruction cache miss rate to the effective fetch width.

By introducing the different code layout optimizations first, we intend to provide the reader with a clearer idea of how they help improve the fetch performance of the different fetch architectures that will be presented later.

We can divide code layout optimizations in three parts: the layout of the basic blocks inside a routine, the splitting of a procedure into several different routines or traces, and the layout of the resulting routines or traces in the address space. In this section, we will describe some algorithms for each of these optimizations, and point the benefits that can be obtained from them.

A. Basic Block Chaining

Basic block chaining organizes basic blocks into traces, mapping together those basic blocks that tend to execute in sequence. There have been several algorithms proposed to determine which basic blocks should build a trace [1], [11], [19], [42], [44], [59].

As an example, Fig. 3 shows the chaining algorithm used in [19], [44], [59]. It is a greedy algorithm, which given a *seed*, or starting basic block, follows the most frequent path out of it. This implies visiting the routine called by the basic block, or following the most frequent control flow out of the basic block. All secondary targets from that basic block are

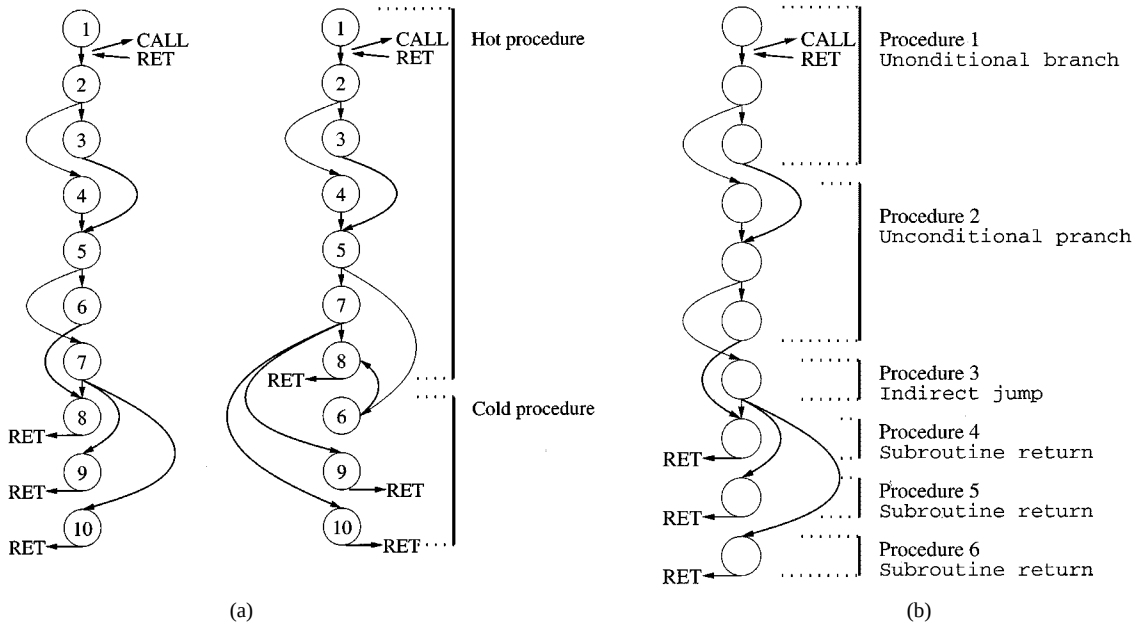


Fig. 4. Examples of the procedure splitting algorithm. Infrequently used basic blocks are mapped to a different procedure, and moved away of the execution path.

added to the list of seeds to be explored later. If the most likely path out of a basic block has already been visited, the next possible path is taken. If there are no possible paths out of a basic block, the algorithm stops, and the next seed is selected.

A second alternative is the *bottom-up* algorithm proposed in [42]. The heaviest edge in the graph (the edge with the highest execution count) is selected, and the two basic block are mapped together. The next heaviest edge is taken, and processed in the same way, building basic block chains. Edges reaching or leaving a basic block in the middle of an already existing chain are ignored. After all basic blocks have been mapped to chains, the different chains are mapped in order so that conditional branches map to forward/usually not taken branches.

But a control flow graph with weighted edges does not always lead to a basic block representing the most frequent path through a subroutine. The solution to this problem is path profiling [1]. A path profile counts how many times each path through a subroutine was followed, not simply how many times a branch was taken/not-taken. This gives an immediate correspondence between basic block chains and the profile data.

The chaining optimization improves fetch performance in several ways: it improves instruction cache performance by increasing the amount of spatial locality available, because basic blocks that execute in sequence will usually be mapped together, and because unused basic blocks will be moved toward the end of the procedure, avoiding unused space in the cache lines. It can also improve branch prediction accuracy, specially for the static prediction schemes (Section III), by aligning branches in a given direction. Finally, it can also increase the effective fetch width by reducing the number of taken branches, which allows fetching of more consecutive instructions (Section V).

B. Procedure Splitting

After a new ordering of the basic blocks has been established for a given procedure, the frequently executed basic blocks are mapped toward the top of the procedure, while infrequently used basic blocks will move toward the bottom of the procedure body. Unused basic blocks will be mapped at the very end of the routine. By splitting the different parts of the procedure we can significantly reduce its size, obtaining a denser packing of the program.

Fig. 4 shows two different ways of splitting a procedure body. A coarse grain splitting would split the routine in two parts [42]: one containing those basic blocks which were executed in the profile (the hot section), and another one containing those basic block which were never executed for the profiling input (the cold section).

A fine grain splitting would split each basic block chain as a separate procedure [19], [44], [59]. The end of a chain can be identified by the presence of an unconditional control transfer, because after reordering it is assumed that all conditional branches will be usually not-taken. Unused basic blocks would form a single chain, and be kept together in a new procedure.

The procedures resulting from splitting do not adhere to the usual calling conventions, there is no defined entry or exit point, and do not include register saving/restoring. This is done to avoid overhead associated with standard procedure control transfers.

The benefits of the procedure splitting optimization do not lay within the splitting itself. That is, there is no immediate benefit in splitting a procedure into several smaller ones. The benefit of splitting reflects on the improvements obtained with the procedure placement optimizations, because mapping smaller procedures gives these optimizations a finer grain control on the mapping of instructions without undoing what the basic block chaining optimizations obtained.

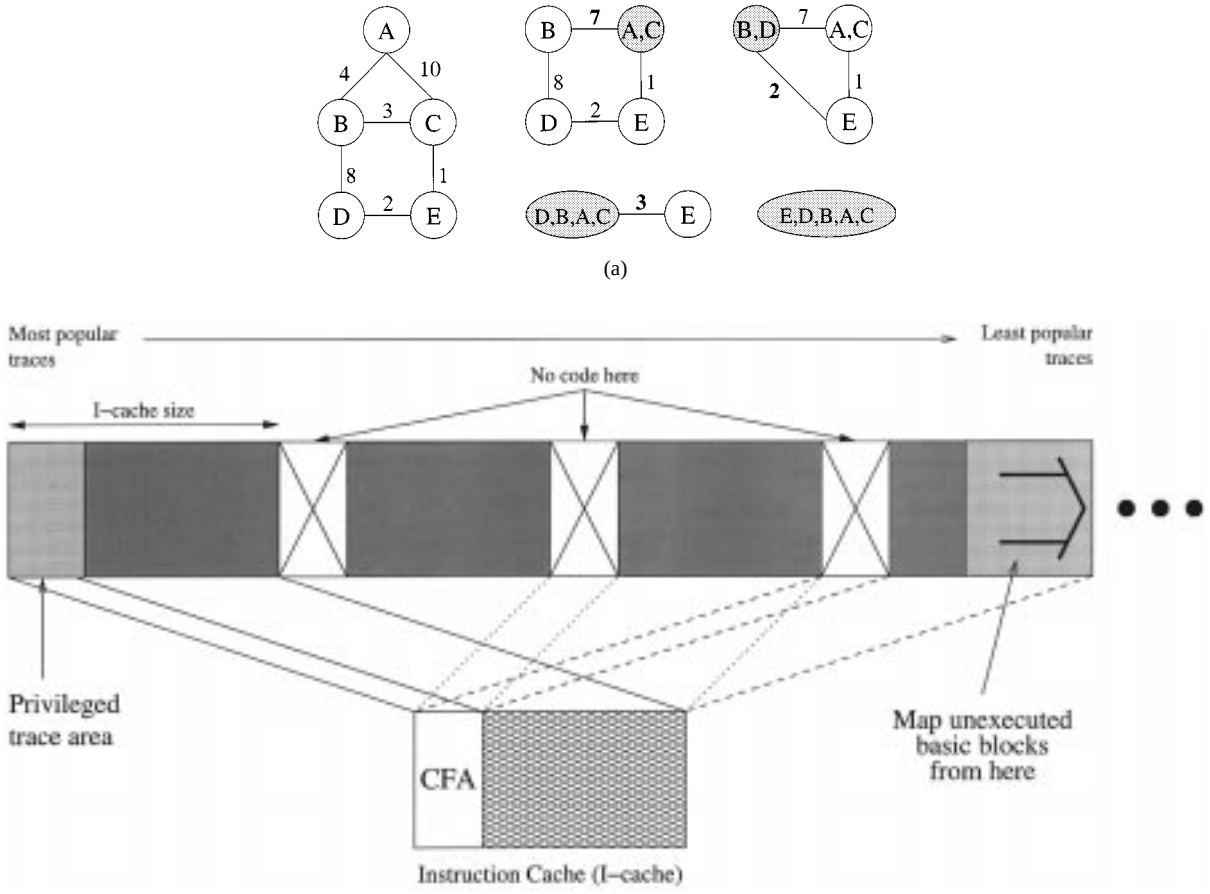


Fig. 5. Two different algorithms for procedure mapping.

C. Procedure Mapping

Independently of the basic block chaining and procedure splitting optimizations, the order in which the different routines in a program are mapped has an important effect in the number of code pages used (and, thus, on the instruction TLB miss rate), and on the overlapping between the different procedures (and, thus, on the number of conflict misses).

The simplest procedure mapping algorithm is to map routines in popularity order: the heaviest routine first, and then in decreasing execution weight order. This ensures that there will not be conflicts among two equally popular routines.

Fig. 5(a) shows the mapping algorithm used in [7], [36], [42], [57]. It is based on a call graph of the procedures with weighted edges. The edge weight is the number of times each procedure call was executed. This algorithm can be extended to consider the temporal relationship between procedures and the target cache size information, as described in [15].

Starting from the heaviest edge, the two connected nodes are mapped together, and all incoming/outgoing edges are merged together. When two nodes containing multiple procedures should merge, the original (unmerged) graph is checked to see which way they should join. For example, the third merging joins nodes (B,D) and (A,C). The original graph shows that A-B is the strongest relationship, thus, they merge as (D,B-A,C). The fourth (final) merging also checks the original graph, and determines that D-E is the strongest relationship.

Fig. 5(b) shows the software trace cache mapping algorithm [43], [44]. Derived from the algorithm used in [59], it follows the basic block chaining phase described in Section II-A. After all basic blocks have been mapped to chains, the chains are ordered by popularity. The most popular traces are mapped to the beginning of the address space, while the least popular traces are mapped toward the end. Chains containing the unused basic blocks are mapped at the very end of the program.

In addition to mapping equally popular chains next to each other, a fraction of the instruction cache will be reserved for the most popular chains by ensuring that no other code maps to that same range of cache addresses. This ensures that the most frequently used traces will never miss in the cache due to a conflict miss.

In [17] and [25], an optimized procedure layout is generated by performing a color mapping of procedures to cache lines, inspired in the register coloring technique, taking into consideration the cache size, the line size, the procedure size, and the call graph.

The mapping optimizations mainly improve the instruction cache performance, by reducing the number of conflict misses. Also, in combination with the procedure splitting optimization, it improves spatial locality by moving the unused

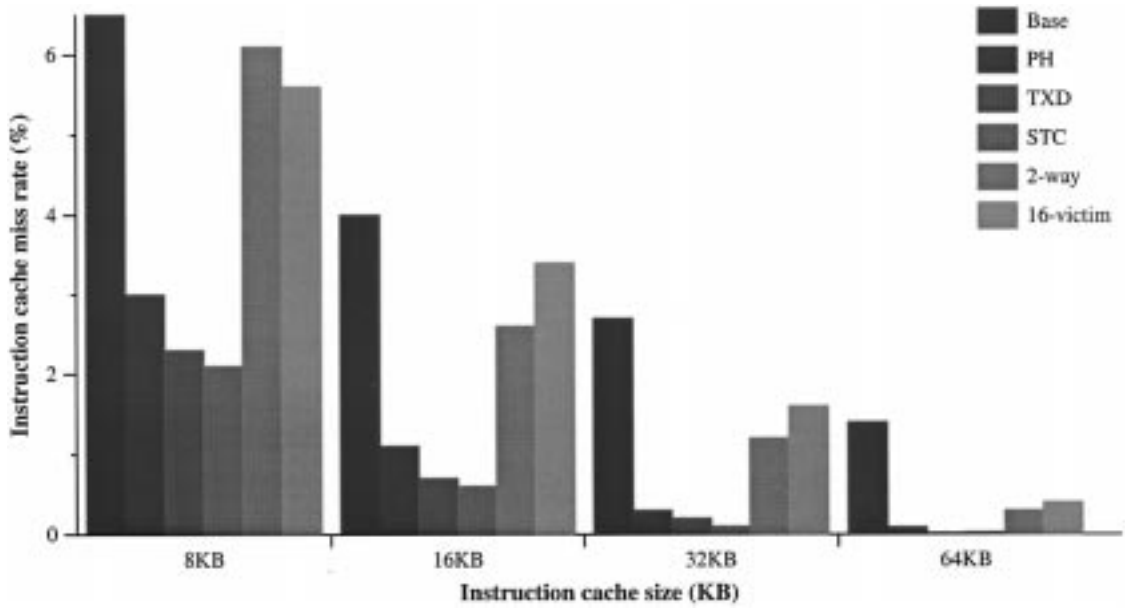


Fig. 6. Effect of different code reordering techniques on instruction cache performance.

procedures to a separate region of the code, obtaining a better packaging of instructions to cache lines. In addition to improving instruction cache performance, procedure mapping optimizations have a beneficial effect on all the instruction memory hierarchy: the instruction TLB, and the L2 cache. Plus, the reduced instruction miss rate in the shared L2 cache can have a beneficial impact in the data miss rate, due to the reduced interference between data and instructions.

D. Code Placement and Instruction Cache Performance

The layout of the instructions in memory determines their mapping onto the instruction cache. For this reason, the mapping of the program routines largely determines the number of conflict misses that will be encountered in the instruction cache. At the same time, the mapping of the basic blocks inside a routine in conjunction with the cache line size determines the amount of spatial locality exploited, and can also affect the number of instruction cache misses encountered.

Fig. 6 shows the effect of several code reordering algorithms on the instruction cache miss rate. The different setups explored are a baseline setup with a direct mapped instruction cache, the baseline setup using a code optimized using the Pettis & Hansen algorithm [42], a code optimized using the Torrellas, Xia, and Daigle algorithm for operating system code [59], a code optimized with the software trace cache algorithm [44], the baseline code layout using a two-way set associative cache, and the baseline setup using a 16-way fully associative victim buffer.

The results show that code reordering techniques can be very effective at reducing instruction cache miss rate, for both small and large sized caches. The effectivity of code layout optimizations proves much better than that of pure hardware approaches, like set associativity [18], [52], victim caches [24], and prefetching schemes like stream buffers [10], next line prefetching [28], [60], and context sensitive predictors [23], [49].

III. Pipelined Processors

Before pipelined processors, instructions were executed one at a time. The instruction is fetched from memory, and it goes through all stages of execution. Once it has been completed, the next instruction is read from memory, and the process continues.

The only performance bottleneck for the fetch engine of these processors is the perceived memory latency. Only one instruction is needed at a time, and there is no need to speculate on the program control flow, as the previous instruction has completed its execution, and the next PC is known. But if reading the next instruction from memory takes too long, instruction execution must stop waiting for that instruction.

Pipelined processors introduce the problem of fetching one instruction per cycle without waiting for the previous instruction to finish. In an attempt to fetch and execute one instruction per cycle, the outcome of a branch instruction is unknown until one or more cycles after the instruction was fetched, making the program control flow uncertain at that point, and introducing execution bubbles in the pipeline. That introduces the branch architecture as a major fetch issue, next to the memory latency problem. Fetching one instruction per cycle makes the fetch width issue unimportant, as reading a single word does not pose any problem.

For example, in the early Berkeley and Stanford RISC machines, the pipeline required one empty execution slot after each branch to account for its execution delay. With a 20% branch frequency, those machines saw a loss of 6% to 30% compared to another machine with single-cycle branches. A two-cycle delay in the branch execution could easily account for a 40% performance loss, and a three-cycle delay for a 60% waste [31].

A first solution encountered for that limitation is making the problem visible to the programmer, continuing to fetch and execute instructions sequentially until the branch is re-

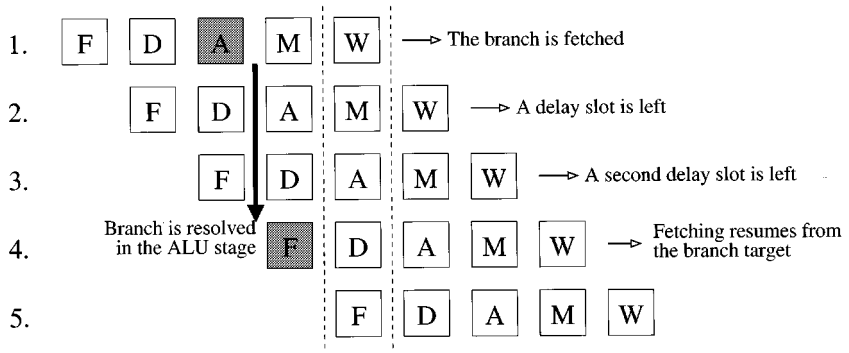


Fig. 7. Pipelined execution of a branch instruction. The branch is not resolved until the ALU stage, which introduces two delay slots.

solved. These are the architected delay slots present in the IBM 801, RISC II, and MIPS architectures.

Fig. 7 shows the pipelined execution of a branch instruction. The branch instruction is fetched in cycle 1. At the end of cycle 1, the branch has not yet resolved, and the target address is still unknown. In cycle 2, the branch is decoded, but its outcome is still unknown. In cycle 3, the branch condition is evaluated and the target address calculated in the ALU stage. Meanwhile, the next instruction has been fetched. In cycle 4, the correct instruction can be fetched by setting the PC to the computed branch target. If the architecture does not define branch delay slots and the branch was taken, the instructions fetched in cycles 2 and 3 must be squashed, and they represent wasted cycles. If the architecture defines delay slots, the instructions were meant to be executed anyway, so they do not waste any resources.

The fetch performance of such mechanism with architected delay slots heavily depends on the ability of the compiler to allocate useful instructions to the delay slots. A single branch delay slot can be successfully filled with a useful instruction around 70% of the time, a second delay slot can only be filled 25% of the time [31].

The performance degradation due to longer pipelines motivated the research in branch prediction techniques. There are two aspects in branch prediction: predicting a conditional branch direction (taken versus not taken) [53], [63], and predicting the branch target address [29]. Predicting the branch direction can reduce the branch delay by one cycle if the branch is predicted not taken, but introduces an additional penalty if the branch was mispredicted. Plus, in order to reduce the delay due to taken branches, it is also necessary to predict the branch target address. If both direction and target are predicted in the fetch stage, the next correct path instruction can be fetched the next cycle, and no delay is paid. If one or both predictions are wrong, it will be necessary to squash the wrongly fetched instructions, and maybe pay an additional penalty.

Fig. 8 shows the execution cost of a branch instruction for several branch architectures as shown in [31].[1] The execution cost of a branch shows the average number of cycles it takes to correctly execute a branch.

[1]bigfm, dnf, hopt (Fortran applications). Average of 63% taken branches (37% not taken).

The results shown correspond to a five-stage pipeline architecture (such as the one shown in Fig. 7), which executes branches in the third pipeline stage (the ALU stage). This stage both evaluates the branch condition, and calculates the branch target address.

The different branch architectures shown are as follows.

Delayed branch: The architecture defines two branch delay slots. As stated before, the first delay slot could be filled 70% of the time, and the second delay slot was filled only 25% of the time.

Predict not taken: Assume that the branch will be not taken, and keep fetching instructions sequentially. If the assumption was incorrect, the wrongly fetched instructions must be squashed.

Predict taken: Assume that all branches will be taken, and stop fetching until the target address is calculated at the decode stage.

Branch target buffer: Uses dynamic branch prediction to decide the branch direction using a 2-bit counter [53], and a cache memory to determine the target address.

Fast compare: Implement certain comparison operations in the decode stage, so that branches depending on easy compares (compares against zero) can execute early. This reduces the number of delay slots to just one.

Profiled fast compare: Same as fast compare, but uses profiling to restructure the code and to fill the delay slots.

Squashing branch: Always fills the delay slots with potentially useful instructions, assuming that the branch will be taken. If the branch is not taken, and the squashing bit is on, the instructions from the delay slots are squashed.

Profiled squashing branch: Same as squashing branch, only the branch prediction is given by profile data. If the branch was mispredicted, and the squashing bit is on, the delay slots are squashed.

The main problem of the compiler approach is filling the branch delay slots with useful instructions, which causes the poor performance of the *delayed branch* approach. Meanwhile, the different *fast compare* approaches obtain much better performance because they require only a single delay slot, which can be filled with useful instructions most of the time. The *squashing branch* approach ensures that the delay

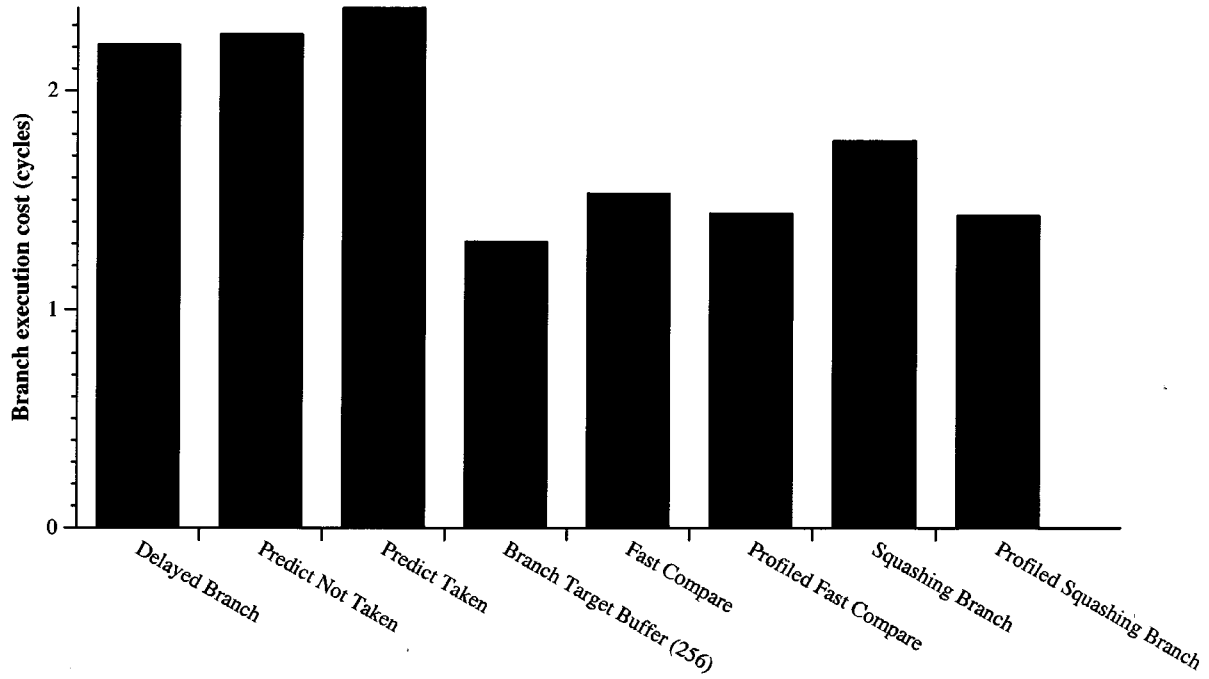


Fig. 8. Comparison of the branch execution cost for different branch architectures using both software and hardware techniques.

slots can always be filled with useful instructions, adding an instruction squash mechanism that activates on a wrong assumption.

However, the results show that dynamic branch prediction using a BTB and 2-bit saturating counters provides the best fetch performance. But the implementation cost of a 256-entry BTB was too high at that point, which made software approaches much more amenable.

A. Branch Alignment Techniques

The need to use static branch prediction techniques and other software approaches to implement efficient branch architectures motivated the research on code layout optimizations to reduce the cost of branches. In addition to improving instruction cache performance, code reordering techniques can map basic blocks so that the conditional branches contained follow the heuristic expected by the architecture.

Branch alignment optimizations [5] usually rely on profile data to obtain information about the most likely branch direction. Once the branch behavior is known, then it is possible to map the two possible successors in a way that make the branch adhere to a specific heuristic that can be computed at run-time. For example, assume a given branch has two targets *A* and *B*, and the most likely target is *A*. If the run-time heuristic used is to assume that all branches will be taken, then we would map our code (and set the branch condition) so that *B* is the fall-through target of the branch. If the run-time heuristic says that branches will be not taken, we would map *A* as the fall-through target for the branch.

Looking at the results in [45],[2] it can be observed how the use of code layout optimization can be very successful at aligning branches toward a specific heuristic. For example, aligning branches so that they are usually not taken improves the prediction accuracy of an *always not taken* predictor from 49% to 77%, and aligning them to use a *backward taken/forward not taken* heuristic improves from 60% to 81%, coming close to the 92% prediction accuracy of a perfect static predictor.

It is important to note that the use of reordering optimizations to align branches does not limit their ability to reduce the instruction cache miss rate.

Further performance improvements can be obtained by enhancing the branch alignment optimization with other code transformations that increase the static branch prediction accuracy. Among these optimizations we find unconditional branch removal [35], conditional branch removal [34], using branch correlation in static predictions [26], [65], and value range propagation [40].

IV. Superscalar Processors

Superscalar processors replicated the pipeline in the execution engine, allowing the simultaneous execution of several independent instructions, exploiting ILP [54]. But the fetch engine can not be replicated in the same way, which introduced the problem of fetching multiple instructions per cycle to be able to feed the execution engine. The effective fetch width of the processor suddenly becomes important, which means fetching several instructions per cycle, and fetching them from the correct execution path.

A full basic block of instructions is usually enough to provide a four-issue processor with instructions to keep its functional units busy. Furthermore, with a 20% branch frequency, it is likely that one branch per cycle will be fetched,

[2]SPECint95 except go, and adding PostgreSQL running TPC-D.

increasing the importance of the branch prediction mechanism. As all instructions in a basic block are stored sequentially in memory, it is enough to fetch a whole cache line, and then select the required instructions from it.

This solves the fetch width problem for narrow superscalar processors, however, the branch prediction mechanism has to provide two pieces of data: the number of instructions to fetch until the terminating branch is found, and the start address of the next executed basic block [4], [64].

This motivated the research for better branch predictors, in search of high prediction accuracy and low access time. The classical approaches to branch prediction used were the use of a branch target buffer (BTB) for target address prediction [29], and the addition of a saturating two-bit counter to predict the branch direction [53].

Two-level adaptive branch predictors [62], [63] represent a novel organization of the branch prediction tables. This new organization not only stores information about the past behavior of a branch (taken or not taken), but also relates the behavior of a branch to either its own past behavior (private history) or the behavior of the previous branches (global history).

A major factor in the loss of accuracy of two-level branch predictors is aliasing. When two branches end up sharing the same 2-bit saturating counter, interference happens. If the two branches have different behavior, this interference results in a loss of prediction accuracy. Based on this observation, a new generation of predictors (called dealiased predictors) use novel organizations that reduce this effect. Among this predictors we find the agree predictor [56], the bi-mode predictor [27], and the gskew predictor [33].

Fig. 9 shows the fetch mechanism proposed by Yeh and Patt in [64] for fetching a full basic block of instructions per cycle. All blocks are accessed in parallel using the fetch address as index: a cache line is fetched from memory, the top of the return address stack (RAS) is read, and the BTB is checked. On a BTB hit, it means that the basic block ends in a branch instruction. On a BTB miss, fetching continues sequentially.

The BTB contains all the information necessary to generate the next fetch address: the branch type (conditional, unconditional, subroutine call, or subroutine return), the branch direction prediction, and the target and fall-through addresses. If the branch is unconditional, or the conditional branch is predicted taken, the target address is used. If the branch is predicted as not taken, the fall-through address is used. If the branch is a return, the top of the stack is used as target address. If it is a subroutine call, the next instruction is pushed onto the stack. If a branch is discovered later on in the pipeline, a BTB entry is allocated for it, and its outcome is predicted using static information.

This fetch mechanism approaches all three fetch performance factors relevant at this point: the memory latency is hidden using an instruction cache; the fetch width is increased by reading a whole basic block of instructions, which reside in consecutive memory positions; and the branch prediction accuracy is increased using a BTB in conjunction with a two-level adaptive branch predictor [63].

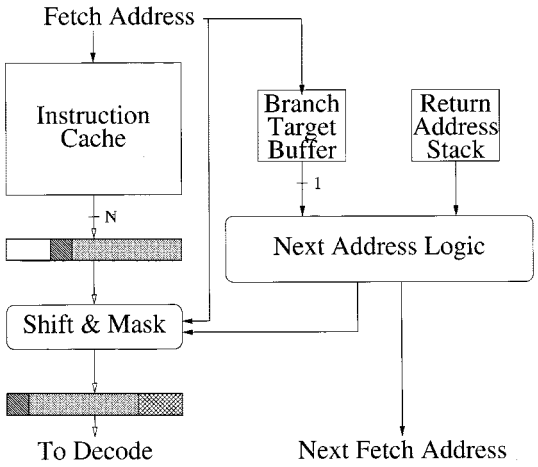


Fig. 9. A fetch mechanism capable of reading one basic block per cycle.

Fig. 10 shows two performance metrics for the fetch engine described in [64][3]: the branch execution penalty (BEP), and the instructions per fetch cycle (IPFC). The BEP represents the average number of cycles wasted for each executed branch. This assumes a standard number of wasted cycles each time a branch is mispredicted (shown in the x axis), and a two-cycle delay for each branch misfetch (taken branches not found in the BTB), misfetched branches are predicted using static heuristics. The IPFC shows the average number of correct path instructions provided by the fetch unit, and thus the performance of an ideal machine able to execute all instructions in a single cycle. The benchmarks used for this study were four integer and five FP codes from the SPEC92 benchmark set (eqntott, espresso, gcc, li, doduc, fpppp, matrix300, spice2gr6, tomcatv).

The different fetch engine configurations tested are as follows.

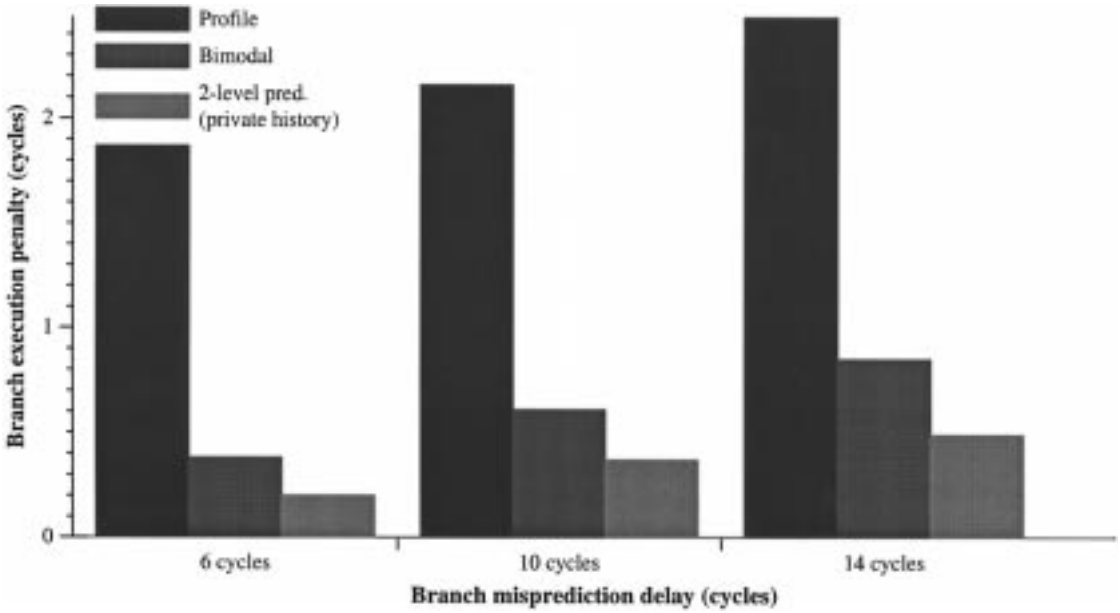
Profile: Uses profile data to predict conditional branch directions. The target address for taken and unconditional branches is not available until it is calculated from the decoded instructions.

Bimodal: Uses the saturating 2-bit counter proposed in [53] to predict conditional branch directions. Both the 2-bit counter and the target address are obtained from the BTB. The BTB has 512 entries, and is four-way associative.

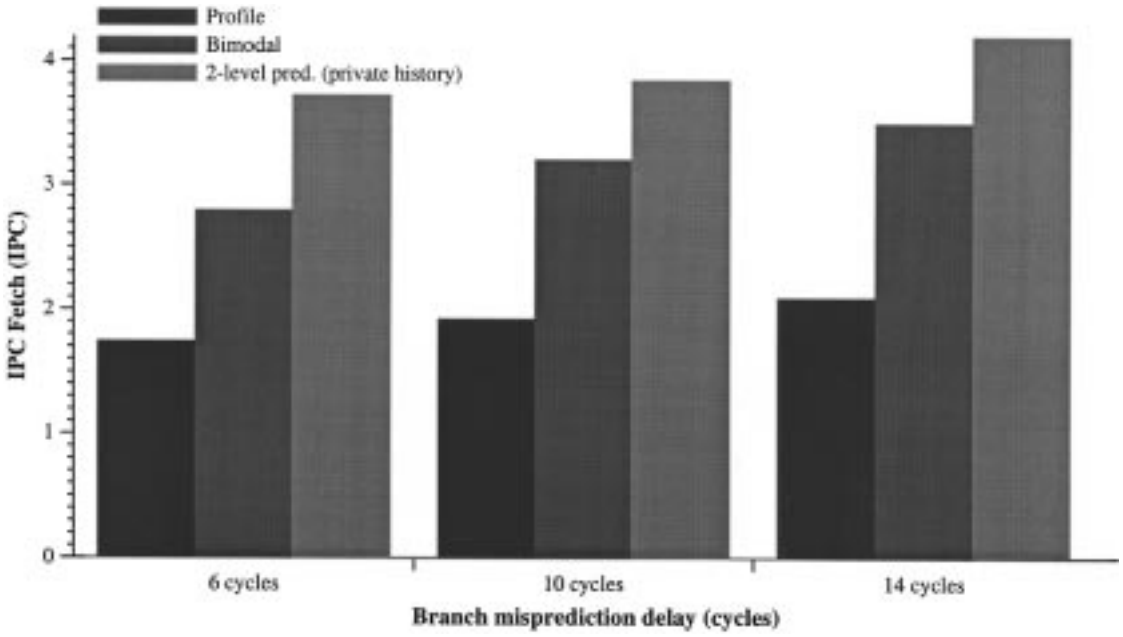
Two-level pred.: Uses the private history two-level branch predictor proposed in [63]. This stores the target address and the private branch history in the BTB, and uses the branch history as an index into a separate pattern history table (PHT) consisting of 2-bit saturating counters. The BTB has 512 entries and is four-way associative. The history registers are 12 bits long, and index into a 4-k-entry PHT.

Previous studies have shown how profile based branch prediction can be as accurate as that obtained with 2-bit saturating counters [12]. The poor fetch performance of the *pro-*

[3]eqntott, espresso, gcc, li, doduc, fpppp, matrix300, spice2g6, and tomcatv from SPEC92 on a Motorola88100 instruction level simulator



(a)



(b)

Fig. 10 Performance metrics for the fetch engine proposed by Yeh and Patt.

file setup is mainly due to the two-cycle delay in the calculation of the branch target address of taken branches. The *bimodal* setup solves this problem by using a BTB to store the target address of the most recently executed branches. However, the improvement obtained with the *two-level pred.* setup shows the importance of accurate branch prediction for superscalar fetch performance. The BTB is the same as the one used in the *bimodal* setup, but the branch direction predictor proves much more accurate, and is more effective at avoiding branch mispredictions.

Given the relevance of the branch prediction mechanism in the superscalar fetch engine, Calder and Grunwald proposed two improvements to the baseline mechanism [4].

Decoupled BTB and branch predictor: The original design does not allow dynamic prediction for branches that miss in the BTB, relying instead on static prediction methods. If the branch predictor operates independently of the BTB, it is possible to obtain dynamic prediction for branches discovered in the decode stage. The higher prediction accuracy of the dynamic predictor avoids many cycles of wasted work fetching from the wrong path.

Only taken allocate: The effectiveness of the BTB can be increased by only allocating entries for taken branches. A branch missing in the BTB but resulting as not taken, will not be allocated an entry. This avoids displacing information about taken branches, as not

taken branches do not really benefit from the BTB because they always fetch the next sequential instruction.

Their results show that two-level adaptive branch predictors can be implemented independently of the BTB, which allows a separate resource allocation for target address prediction and branch direction prediction. The increased accuracy of a larger direction predictor obtains substantial reductions in the branch misprediction rate, and thus on the branch execution penalty.

In addition, the *only taken allocate* makes a more effective use of the BTB space, storing only those branches which will benefit from its target address prediction capabilities. By selectively storing branches in the table, more branches will fit, reducing the number of branch misfetches and further increasing fetch performance.

Next line and set prediction (NLS), by Calder and Grunwald [6], represents an alternative mechanism for fetching instructions in a superscalar mechanism. Instead of predicting which is the next instruction to fetch, NLS predicts which instruction-cache line contains the next instructions to be fetched.

By providing pointers into the instruction cache, NLS allows the next instruction to be fetched, while the previous branch is decoded and its target address calculated (not predicted). NLS is based on the distinction between a branch misfetch and a branch mispredict. When a branch is detected as such, and predicted incorrectly, the fetch engine will spend several cycles fetching from a wrong execution path until the branch is executed. Meanwhile, if a branch is fetched but not detected as such, it will be identified at decode time, and its target address can be calculated at that point, potentially saving many cycles.

It is possible to find examples of this mechanisms in real processors. For example, the Intel Pentium uses a BTB for target address prediction, and complements each BTB entry with a 2-bit saturating counter for branch direction prediction (the *bimodal* setup in Fig. 10). The PowerPC 604 [55] uses the decoupled BTB and branch predictor setup, with a 64-entry fully associative BTB, and a 512-entry PHT for direction prediction. Finally, the Alpha 21 264 [16] uses the NLS mechanism.

To this point, we still have not mentioned the scalability concerns regarding the fetch engine of superscalar processors. The current trend in superscalar processors achieves higher performance by increasing the clock rate, which also requires increasing the number of pipeline stages.

With faster clocks, the amount of useful work that can be done in each stage is reduced [38], requiring deeper pipelines. And deeper pipelines require even more accurate branch prediction to avoid paying an excessive cost for each branch instruction.

A faster clock rate also reduces the amount of memory that can be accessed in a single cycle, leading to smaller sized caches and branch prediction tables. A smaller cache will have a higher miss rate, and a smaller branch predictor will be less accurate, which is the opposite of what a deeply pipelined processor needs. The deeper the pipeline, the higher the misprediction penalty, which means that deep pipelines require more accurate branch predictors.

Fig. 11 shows the mechanism proposed by Reinman *et al.* in [48] to alleviate the negative impact of this trend. The fetch

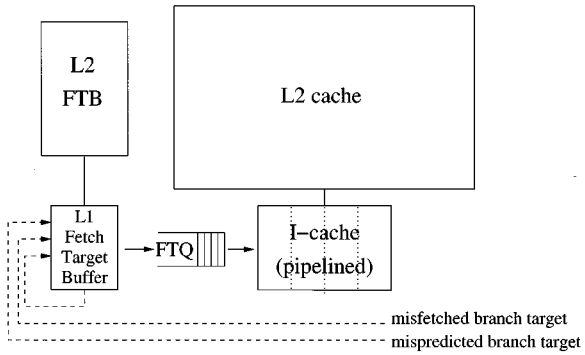


Fig. 11. Decoupling the fetch stage: an independent branch prediction mechanism provides fetch blocks to a fetch target queue, and the pipelined instruction cache reads the blocks from memory.

engine is decomposed in two separate engines. The first engine contains a fully autonomous branch predictor, which follows speculative execution paths and provides fetch blocks to a fetch target queue (FTQ). The second engine contains a pipelined instruction cache that reads the fetch target blocks from memory and provides instructions to the decode stage.

The branch prediction architecture used in [48] is the fetch target buffer (FTB). It extends the BTB architecture to include information about a complete fetch block, which potentially contains several branches. Using the *only taken allocate* optimization proposed in [4], the FTB is not aware of not taken branches, which can enlarge the fetch block by including two or more basic blocks separated by frequently not-taken branches. To mitigate the effect of using a smaller prediction table, a second-level FTB is defined. This L2 FTB is much larger, and thus should be more accurate. It is accessed on an L1 FTB miss, and its prediction is used a few cycles later when it is available.

Their results show that the FTB architecture has a prediction accuracy similar to that of a BTB architecture, and the additional benefit of reading larger fetch blocks, emphasizing the importance of fetch width even on four-issue processors. It combines all the optimizations proposed for the superscalar fetch engine on a more scalable design that also obtains slightly higher performance.

Further improvements on the superscalar fetch engine include techniques to perform instruction prefetching using idle instruction cache ports on the event of a cache miss or a pipeline stall.

One possibility would be to keep fetching instructions even after an instruction cache miss [58], placing the instructions in their rightful place in the instruction window, leaving space for those instructions coming from the upper levels of the memory hierarchy. This would be effectively fetching instructions out of order, effectively hiding some of the memory latency paid for the cache miss.

The second options would be to use the decoupling of the branch prediction stage from the instruction cache read stage proposed in [48]. This decoupling allows the branch predictor to continue generating the fetch sequence on the presence of a cache miss, and to store the future fetch path in the FTQ. Using this future information stored in the FTQ, it is possible to prefetch the instructions using idle instruction cache ports, guided by the branch predictor [49].

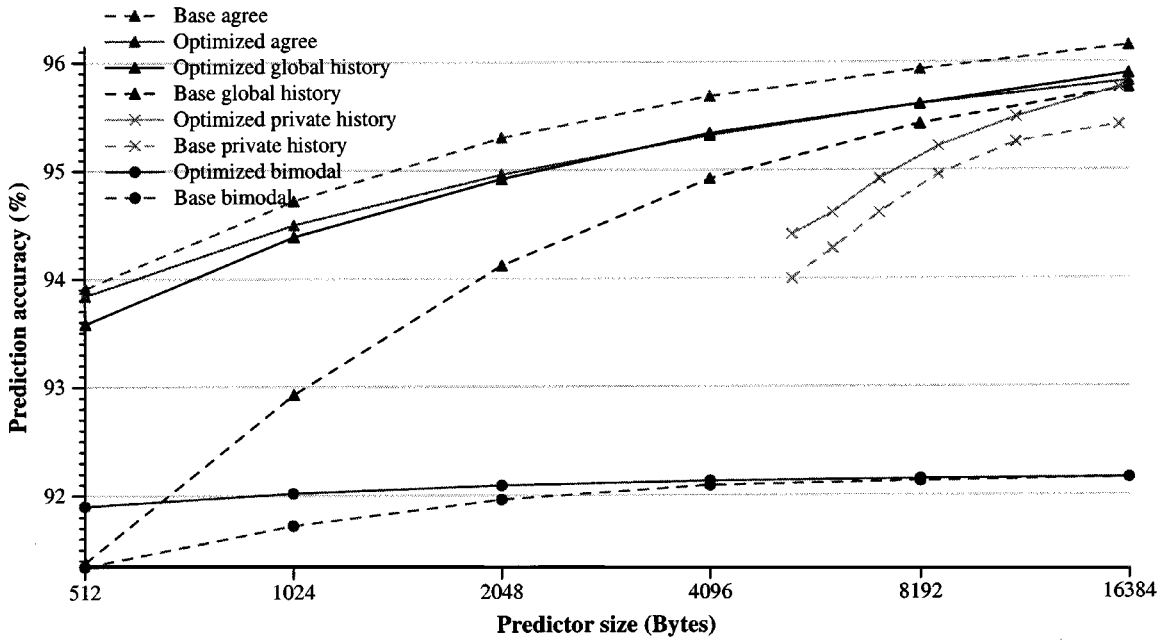


Fig. 12. Interaction between code reordering optimizations and the branch prediction mechanism. Prediction accuracy of optimized an unoptimized binaries.

A. Code Reordering and the Superscalar Engine

Given the relevance of the branch prediction mechanism, and the instruction cache performance improvements obtained with code reordering techniques, it makes sense to analyze their interaction. We already examined the relationship between them when the branch alignment optimization was used to improve fetch performance on pipelined processors [5], but the interaction between these optimizations and the more complex dynamic branch predictors could be more subtle.

Fig. 12 shows the branch prediction accuracy obtained with two-level adaptive branch predictors and dealiased branch predictors (the agree predictor [56]), as explored by Ramirez *et al.* [45].[4]

The results for two-level adaptive predictors show that optimized binaries obtain higher prediction accuracy than the unoptimized ones. This shows a beneficial effect of layout optimized codes: most branches are biased toward not taken after reordering, which means that when two branches end up sharing the same 2-bit counter, they will both push the counter in the same direction, reducing negative interference.

Meanwhile, the results for the agree predictor exhibit the opposite behavior: optimized binaries obtain lower prediction accuracy than unoptimized ones. Dealiased predictors already eliminate the negative interference in the prediction tables, and do not benefit from the effect described above. Meanwhile, the accumulation of not taken branches causes a worse data distribution in the prediction tables, leading to a slight decrease in prediction accuracy.

This loss in prediction accuracy could represent a performance loss, even in the presence of other benefits like improved instruction cache performance. Fig. 13 shows overall processor performance measured in IPC using two code layouts (baseline and optimized), a 64-kB instruction cache, and two branch predictors: a gshare predictor, which increases its accuracy with the optimized layout, and an agree predictor, which loses accuracy with the optimized layout.[5]

As pointed by Navarro *et al.* [37], it is more important to have a good instruction cache performance than a good branch prediction accuracy. However, the paper also shows that this statement is less true as pipeline depth increases and misprediction penalties become higher.

For a short pipeline processor, such as the one simulated by the Simplescalar tool set [3], Fig. 13 shows that the optimized code layout always obtains better performance than the baseline layout, although the agree predictor proves more accurate for the baseline layout. It is important to note that the gshare predictor obtains slightly better performance than the agree predictor when using an optimized code layout.

V. Wide Superscalar Processors

In an effort to exploit larger amounts of ILP, researchers have explored the possibility of wider issue processors: 8-wide, or even 16-wide. Fetching one basic block per cycle is not enough to keep a 16-wide execution engine busy: it is necessary to fetch instructions from multiple basic blocks per cycle. Without reducing the importance of the instruction cache performance, or the relevance of the branch prediction mechanism, the effective fetch width becomes a major performance issue for this kind of processors.

Fig. 14 shows an extension of the superscalar fetch engine to read multiple consecutive basic blocks per cycle (this is the fetch engine described in [50] and called SEQ.3). The instruction cache is interleaved to allow reading two

[4]m88ksim, gcc, li, ijpeg, vortex from SPECint95 plus PostgreSQL running TPC-D.

[5]four-issue processor similar to the Alpha 21 264.

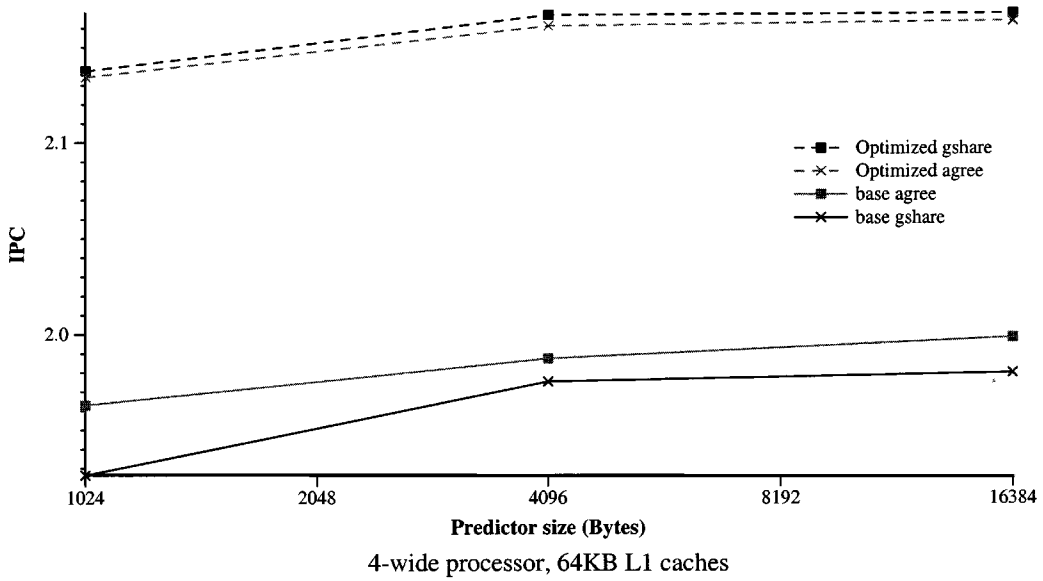


Fig. 13. Overall processor performance for two code layouts (base and optimized) and two branch predictors (gshare and agree).

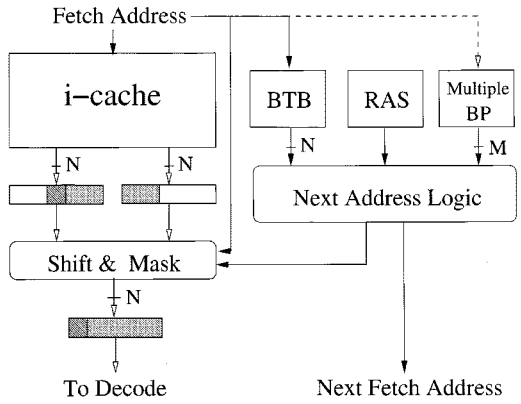


Fig. 14. Extension of the superscalar fetch engine with a multiple branch predictor to read multiple consecutive basic blocks per cycle.

consecutive cache lines. This allows fetching code sequences crossing the cache line boundary, guaranteeing a full width of instructions. The BTB is also interleaved, and all banks are accessed in parallel. The BTB must be N-way interleaved, where N is the number of instructions to provide per cycle. This allows all instructions in a cache line to be checked for branches in parallel. The branch predictor must also provide several branch predictions at once. The next address logic combines the N fields provided by the BTB, the M branch predictions, and the top of the return address stack to provide both the next fetch address, and the instruction mask.

But, unlike the instructions in a single basic block, instructions belonging to different basic blocks may not be stored in sequential memory positions, not even in the same cache line. There have been several solutions to the problem of fetching nonconsecutive instructions, like the branch address cache [61], the collapsing buffer [8], and the trace cache [41], [50], [13].

The branch address cache mechanism proposed by Yeh *et al.* in [61] is composed of four components: a branch address cache (BAC), a multiple branch predictor, an interleaved instruction cache, and an interchange and alignment network. The BAC extends the BTB by keeping a tree of target addresses following a basic block. The width of the tree depends on the number of branch predictions obtained per cycle (two addresses for one branch, seven addresses for two branches, 15 addresses for three branches). The multiple branch predictor is used to select the tree branch that follows the current fetch address, and generate the basic block addresses that will be obtained from the interleaved instruction cache, and the next fetch address. Finally, the interchange and alignment network will arrange the data obtained from the instruction cache to build the dynamic instruction sequence predicted, and present it to the decode stage. Such a mechanism can be delayed by branch mispredictions, target address mispredictions, and instruction cache bank conflicts.

Fig. 15 shows fetch performance measured in instruction per fetch (IPF), which measures the raw width of instructions provided (max 16); and in instructions per fetch cycle (IPFC), which accounts for the delay introduced by instruction cache misses and branch mispredictions. The results are shown in [61].[6]

These results show the importance of fetching instructions from multiple basic blocks in the context of wide issue superscalar processors. Fetching instructions from a single basic blocks effectively limits the performance to 3–4 instructions per cycle for integer codes, while fetching three basic blocks per cycle obtains a fetch performance increase of around 100%.

The collapsing buffer proposed by Conte *et al.* in [8] is composed of an interleaved instruction cache, an interleaved BTB, a multiple branch predictor, and an interchange and alignment network featuring a *collapsing buffer*. The mechanism works in a similar way to the sequential extension of the fetch engine proposed in [64], but the BTB is able to detect *intrablock*

[6]eqntott, espresso, gcc, li from SPECint89, doduc, fpppp, matrix300, spice2g6, tomcatv from SPECfp89. Instruction cache is a two-way set associative (eight-way interleaved), 32 kB, with 16-byte lines.

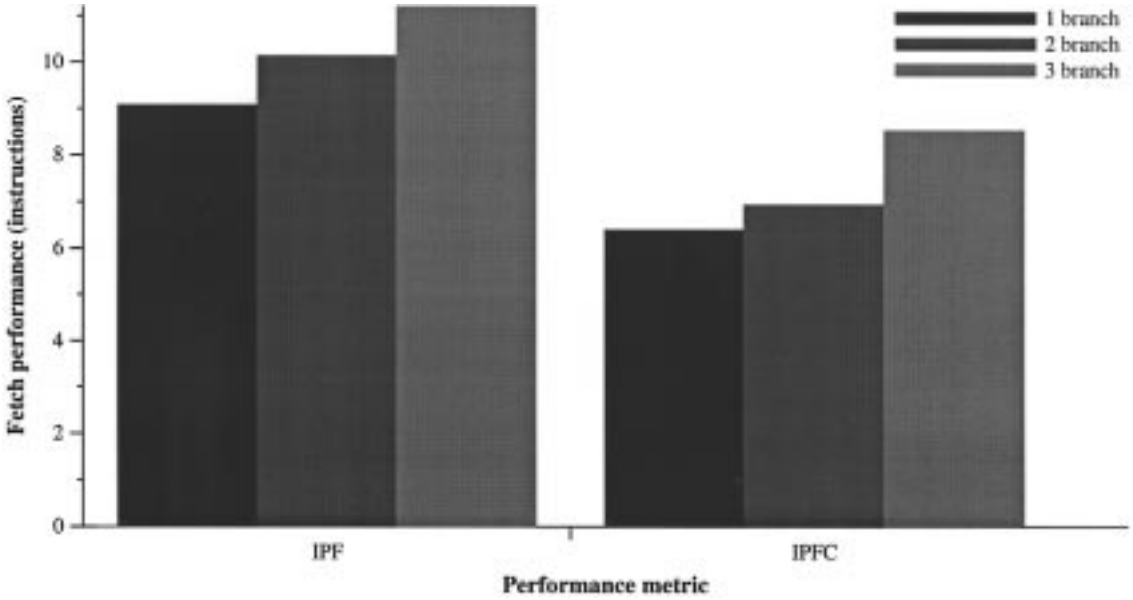


Fig. 15. Fetch width provided by the branch address cache (IPF), and fetch engine performance measured in instructions per fetch cycle (IPFC).

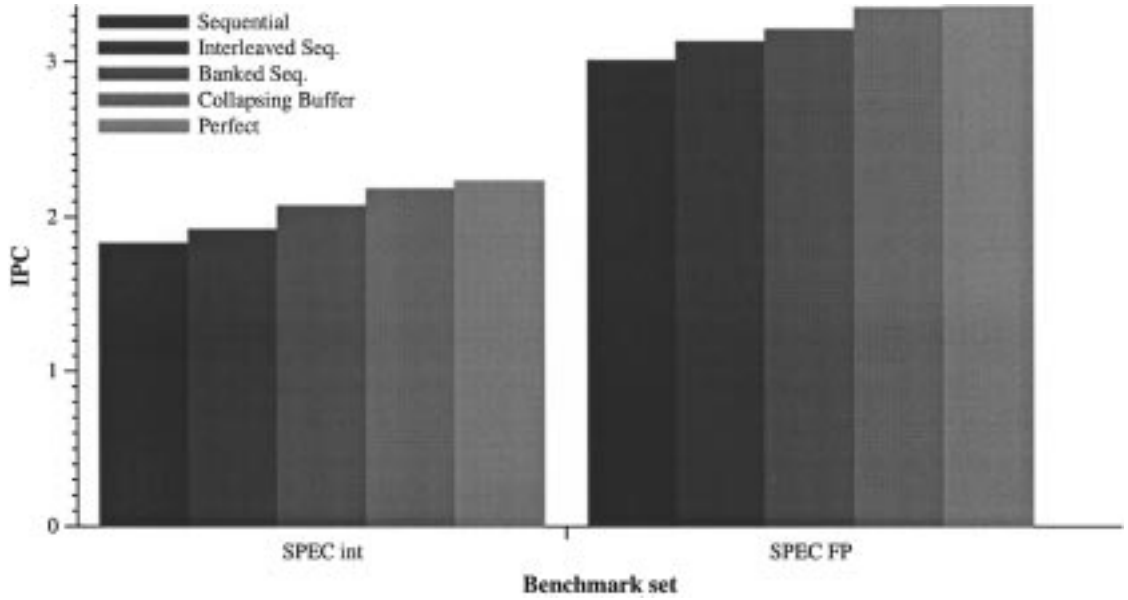


Fig. 16. Processor performance using different sequential fetch policies and using the Collapsing Buffer.

branches (branches with the target in the same cache line). The collapsing buffer uses this information to merge the discontinuous instructions from the cache lines fetched. In addition, a single fetch cycle goes through two BTB accesses, which allows fetching instruction blocks from two separate cache lines, as long as they belong to different cache blocks.

Fig. 16 shows the processor performance obtained in [8] using several fetch policies on an 8-issue superscalar processor.[7]

The fetch policies shown are as follows.

Sequential: A single cache line is fetched, and a full block of sequential instructions is obtained from it.

Interleaved sequential: Two consecutive cache lines are fetched. This allows the sequential instruction block to cross the cache line boundary.

Banked sequential: Allows fetching instruction from two nonsequential basic blocks as long as the two basic blocks reside in different banks.

Collapsing buffer: Full implementation of the collapsing buffer in the alignment network. Useless instructions between an intrablock branch and its target are removed.

Perfect: Ideal fetch engine, able to fetch instructions from nonsequential basic blocks.

[7]compress, eqntott, espresso, gcc, li, sc from SPECint92, doduc, mdljdp2, nasa7, ora, tomcatv, wave5 from SPECfp92, plus bison, flex, and mpeg_play (shown as SPECint). 12-issue processor, 128 kB direct mapped instruction cache with 64-byte lines, 1024-entry BTB with 2-bit counters for branch prediction.

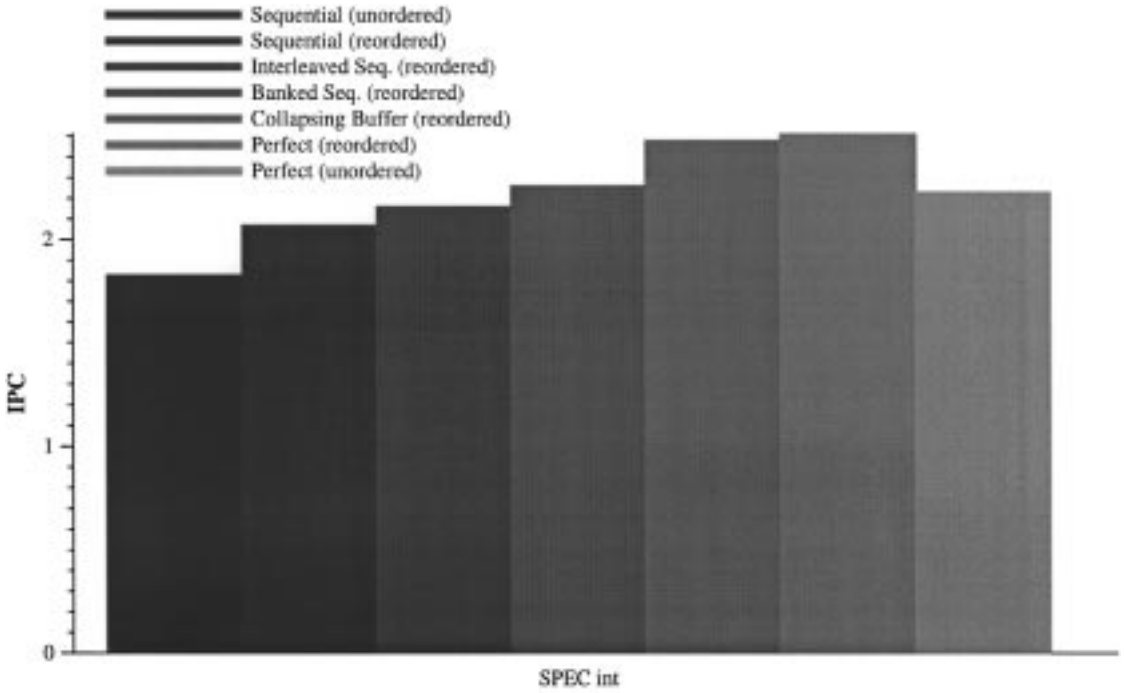


Fig. 17. Fetch performance using the collapsing buffer and code layout optimized binaries.

Again, the results show the importance of fetching multiple basic blocks per cycle on a wide issue superscalar processor. The performance of the single-line all-sequential fetch engine can be easily improved by allowing a limited degree of freedom to allow wider instruction fetch. First allowing the instruction sequence to cross the cache line boundary, then fetching two basic blocks from two different cache lines, and finally using the collapsing buffer to allow fetching past intraline branches.

A. Code Reordering and the Collapsing Buffer

The collapsing buffer and related techniques are limited by their ability to fetch across taken branches, while fetching across not taken branches or intraline branches poses no problem. A reduction of the number of taken branches such as the one provided by code reordering optimizations effectively mitigates this limitation.

Fig. 17 shows the overall processor performance of the different sequential fetch engines and the collapsing buffer in combination with code reordering techniques, as shown by Conte *et al.* in [8].

The use of code layout optimizations greatly improves the performance of the collapsing buffer and related engines. The reduction in the number of taken branches makes the sequential fetch engine reach a similar performance to the perfect engine using the unoptimized code layout. The banked sequential achieves higher performance than the perfect (unordered) engine, and the collapsing buffer largely surpasses that performance, coming close to that of the perfect (reordered) engine.

Clearly, the use of code reordering techniques proves more beneficial for wide issue superscalars, both due to its effect on the instruction cache performance, and to the effective fetch width increase obtained.

However, both the BAC and the collapsing buffer use a complex interchange and alignment network to organize the data fetched from the instruction cache banks into a sequential block of instructions before it is presented to the decode stage. The implementation complexity of this network potentially makes the fetch stage the critical stage for determining the clock rate, and could require an extra pipeline stage to work.

Next, we present the trace cache, the most widely adopted mechanism for fetching instructions from multiple basic blocks per cycle, which resolves this complexity issue by moving the alignment network out of the critical path.

B. The Trace Cache

The trace cache is a fetch mechanism patented by Peleg and Weiser [41] that captures the dynamic stream of instructions, divides it in *traces*, and stores these traces in a special purpose cache (the *trace cache*), expecting that the same dynamic sequence of instructions will be executed again in the future.

The dynamic instruction stream is captured from the retirement pipeline stage, so this process is out of the critical execution path. A fill buffer reads the graduated instructions and organizes them into *traces*, storing them in a special purpose cache. The trace cache does not require any additional work to align the instructions before passing them to the decode stage. The complexity of aligning the instructions has moved from the fetch stage (branch address cache and collapsing buffer) to a separate pipeline after the retirement stage.

A trace is composed of at most N instructions and M branches, where N is the width of the data path, and M

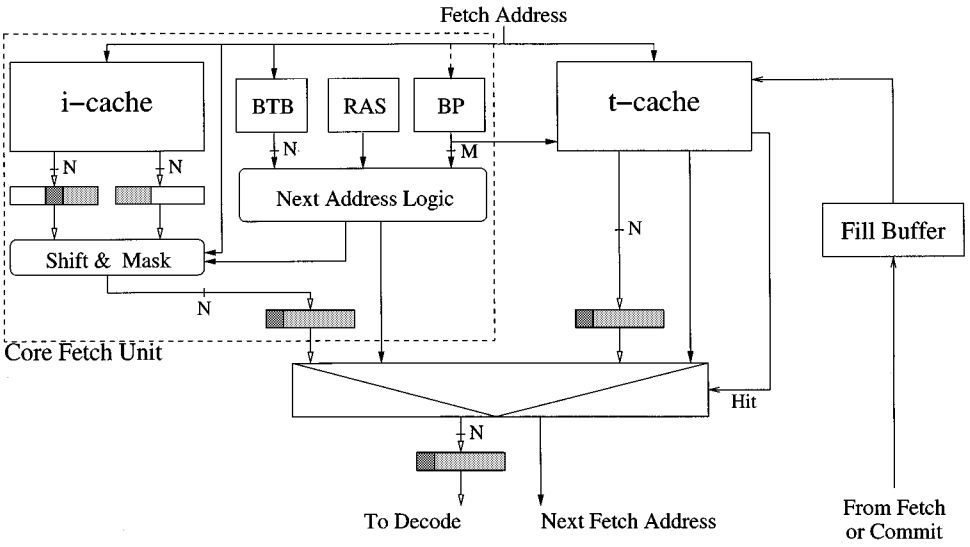


Fig. 18. Extension of the wide superscalar fetch engine with a trace cache to allow fetching of nonconsecutive basic blocks in a single cycle.

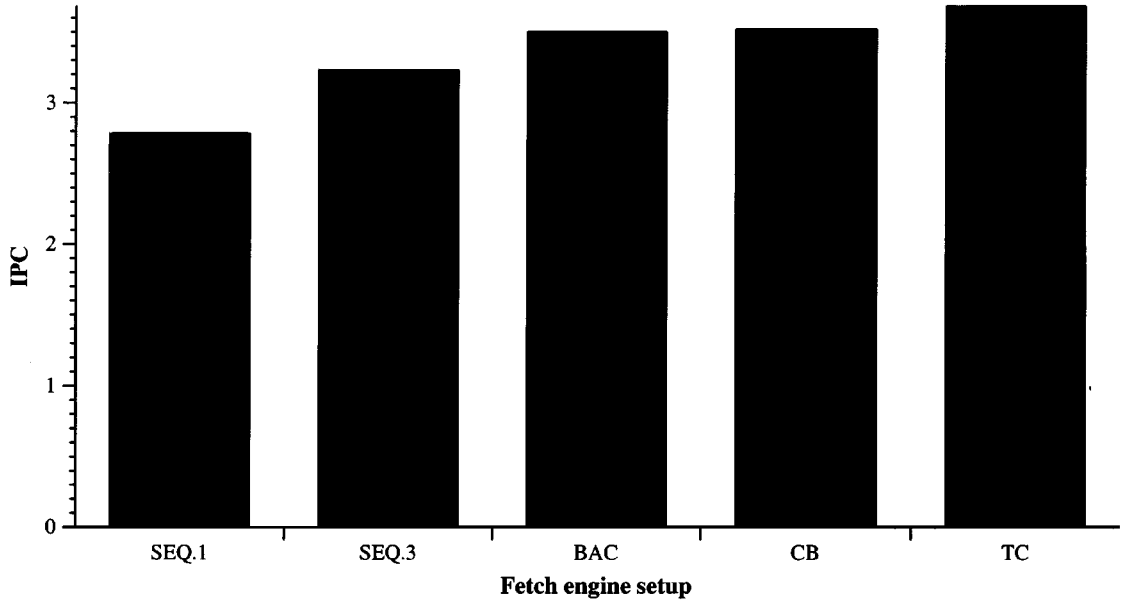


Fig. 19. Comparison of the processor performance using different fetch engines, including the trace cache.

is the multiple branch predictor throughput. The trace is identified by the starting address, and the outcome of up to M-1 branches to describe the path followed.

Fig. 18 shows the wide superscalar fetch engine (which we will call *core fetch unit*) extended with a fill buffer, which captures the dynamic instruction stream and breaks it into traces, and a trace cache where the traces are stored. The core fetch unit and the trace cache are accessed in parallel. On a trace cache hit, the instruction trace stored in the trace cache is passed to the decoder. On a trace cache miss, fetching proceeds from the core fetch unit.

In addition to storing the instructions in a trace, the trace cache also stores the fall-through and taken target addresses that could be followed after the trace. This allows the trace cache to provide the next fetch address on a trace cache hit.

Fig. 19 shows the average processor performance obtained with different fetch engines for the SPECint95 and IBS benchmark sets, as shown by Rotenberg *et al.* [50].[8] The figure shows IPC results for a single-block fetch engine (SEQ.1), a three-block sequential fetch engine (SEQ.3, shown in Fig. 14), the branch address cache (BAC), the collapsing buffer (CB), and the trace cache (TC).

Again, these results show the relevance of fetching multiple basic blocks per cycle on a wide-issue superscalar processor. All three methods described to fetch multiple basic blocks per cycle (BAC, CB, and TC) obtain significant performance improvements over the three-block sequential

[8]16-wide fetch/dispatch rate, three branches per cycle, 2048-entry instruction buffer, 4096-entry global history predictor, 1-k-entry BTB, 128-kB instruction cache, 64-entry trace cache, 1-k-entry BAC.

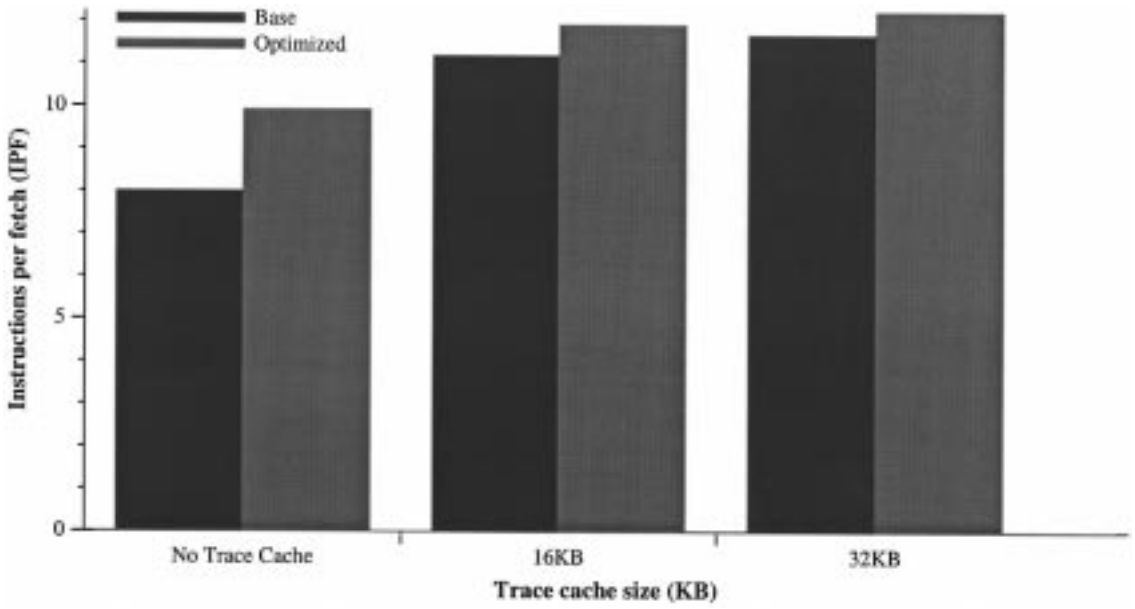


Fig. 20. Effect of code layout optimizations on the trace cache performance.

fetch engine (SEQ.3). The trace cache obtains a slightly higher performance, and adds less complexity to the fetch stage, reducing its impact on cycle time or the number of pipeline stages.

Friendly *et al.* proposed several improvements on the base trace cache mechanism [13]:

> **Partial matching:** It is possible that while a trace is not found in the trace cache, a partially matching trace is found. If a trace matches only the first basic blocks from the requested trace, it is possible to obtain those from the trace cache instead of resorting to instruction cache fetching.
>
> **Inactive issue:** In addition to fetching instruction from a partially matching trace, it is possible to issue the remaining instructions in the trace *inactively*. In case of a branch mispredictions, those inactive instructions will have already been fetched, decoded, and possibly executed reducing the misprediction cost.

The research around the trace cache mechanism has continued, adding techniques to increase the length of the traces provided [39], and increasing its importance to the point of making the trace a new unit of execution: based on the execution of traces rather than the execution of instructions, we find a new branch prediction mechanism, the path-based next trace predictor [21], and a new processor design, the trace processor [51].

Plus, the fill buffer adds a new communication channel between the execution engine and the fetch engine (see Fig. 2), in addition to the feedback about branch mispredictions, now the fetch engine has information about which groups of instructions tend to execute together, which opens the door to further code optimizations at the fetch stage [14], [22].

C. Code Reordering and the Trace Cache

The trace cache stores in consecutive storage those basic blocks which are executed sequentially, and it does this task dynamically. Code reordering techniques can use profile data to do the same mapping effort at compile time as shown by Ramirez *et al.* in [44] and [43]. Layout optimizations effectively store basic blocks in execution order, doing the job of the trace cache, and increasing the effective fetch width of the core fetch unit. This performance increase of the core fetch unit has a significant impact on the overall trace cache performance.

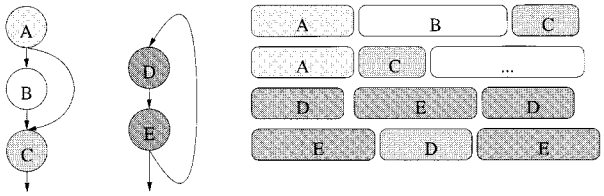


Fig. 21. The trace cache is storing redundant information, because a basic block may be present in more than one trace cache line.

Fig. 20 shows the fetch width obtained with a three-block sequential fetch unit (no trace cache), and the same core fetch unit augmented with trace caches of 16 and 32 kB. Results from [44] are shown for the baseline and optimized versions of the SPECint95 binaries.[9]

The results show that the basic block chaining optimization was successful at increasing the sequentiality of the code, mapping basic blocks in execution order. This reduction in the number of taken branches increases the effective fetch width that the core fetch unit can obtain, almost reaching the effective width of a 16-kB trace cache for unoptimized code.

The same improved core fetch unit performance increases the trace cache performance by providing a better fail safe mechanism: On a trace cache miss, the core fetch unit is capable of providing a larger instruction sequence with the optimized code. This ensures that a larger amount of instructions

[9]For a limit of 16 instructions and three conditional branches.

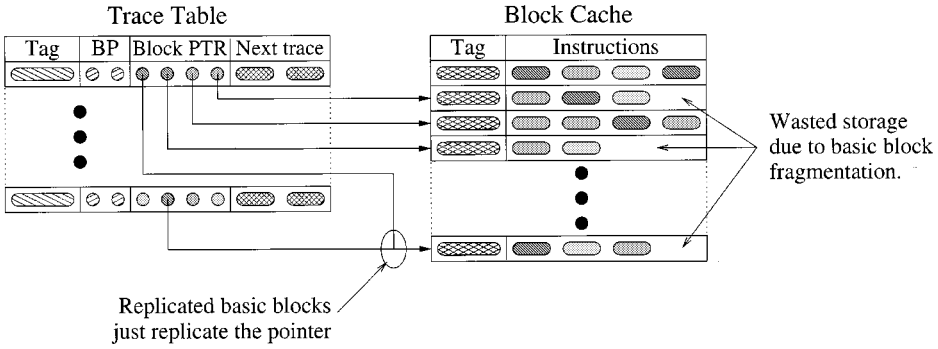


Fig. 22. The block based trace cache stores basic blocks in a special purpose block cache, and stores block pointers in the trace table, eliminating the basic block redundancy.

will be provided each cycle, regardless of where the instructions are being fetched from.

The increased core fetch unit performance obtained using code layout optimizations has motivated the research on purely sequential fetch engines based on fetching long sequential runs of instructions, such as the Stream processor [46], which relies on the compiler to create the sequential paths, and the rePLay microarchitecture [9], which creates the sequential regions dynamically.

D. Trace Cache Redundancy

As described in Section V, the trace cache is a redundant storage mechanism. In this paper, we examine two levels of trace cache redundancy: redundancy within the trace cache itself at the basic block level, and redundancy between the trace cache and the instruction cache at the trace level.

Because it captures the dynamic instruction stream, it is possible for a basic block to be present at many different points in that instruction stream. Fig. 21 shows two examples of this basic block redundancy: an IF–THEN–ELSE construct, and a loop. The figure shows some examples of instruction traces that contain several copies of the same basic blocks. As a trace is identified by the starting address and the branch outcomes contained, these traces will be regarded as different from each other, although they contain mostly the same instructions.

The block-based trace cache (BBTC) [2] is an alternative organization for the trace cache mechanism that avoids this redundant storage of basic blocks.

Fig. 22 shows the BBTC organization proposed by Black *et al.* in [2]. Basic blocks are stored in a special purpose block cache. The trace table stores the trace identifier (the start address, and the required branch outcomes), but instead of storing the instructions in the trace, it stores pointers to the block cache. This way, if a basic block is present in several traces, only the block pointer is replicated.

The results in [2] show that using an improved storage organization, an address translation to use block indexes instead of full addresses, and an improved next trace predictor that takes advantage of this address translation, the block based trace cache obtains significant performance improvements over the baseline trace cache mechanism.

In addition to basic block-level redundancy, the trace cache also presents trace-level redundancy. Some traces stored in the trace cache are redundant with what is stored in the instruction cache.

As shown in Fig. 23, those traces which do not contain taken branches can be fetched in a single cycle from the core fetch unit. But the trace cache is storing these traces as well, which means that traces containing only consecutive instructions are present in both the trace cache and the instruction cache. Furthermore, the use of code layout optimizations increases the trace level redundancy, as the increased code sequentiality also increases the number traces that do not contain any taken branch.

Selective Trace Storage, proposed by Ramirez *et al.* in [47], divides traces among those containing taken branches or some other form of control break (red traces), and traces containing only sequential instructions (blue traces). This trace division is implemented in the trace cache fill unit. When a trace is classified as red, it is stored in the trace cache. If a trace is classified as blue, it is discarded, and the next trace is initiated.

This selective storage of traces makes a more efficient use of the trace cache storage space, although it is reducing the trace cache hit rate. Blue traces will always miss in the trace cache, but they can still be fetched from the instruction cache. Meanwhile, the red trace hit rate increases, allowing fetch of a taken branch and its target in the same cycle.

Fig. 24 shows the fetch width (IPF) and fetch performance (IPFC) of the baseline trace cache, and a trace cache using selective trace storage to eliminate the trace level redundancy.[10]

The results show that the number of blue (redundant) traces increases sig-10Max 16 instructions or three conditional branches, 64 kB direct mapped instruction cache (six cycles miss penalty), perfect branch prediction. The results show how eliminating the trace level redundancy results in a much better trace cache utilization, which leads to a higher fetch performance through a wider fetch. Traces that do not contain taken branches are provided from the core fetch unit, while the trace cache extends its capabilities to fetch past taken branches.

[10]Max 16 instructions or three conditional branches, 64 kB direct mapped instruction cache (six cycles miss penalty), perfect branch prediction.

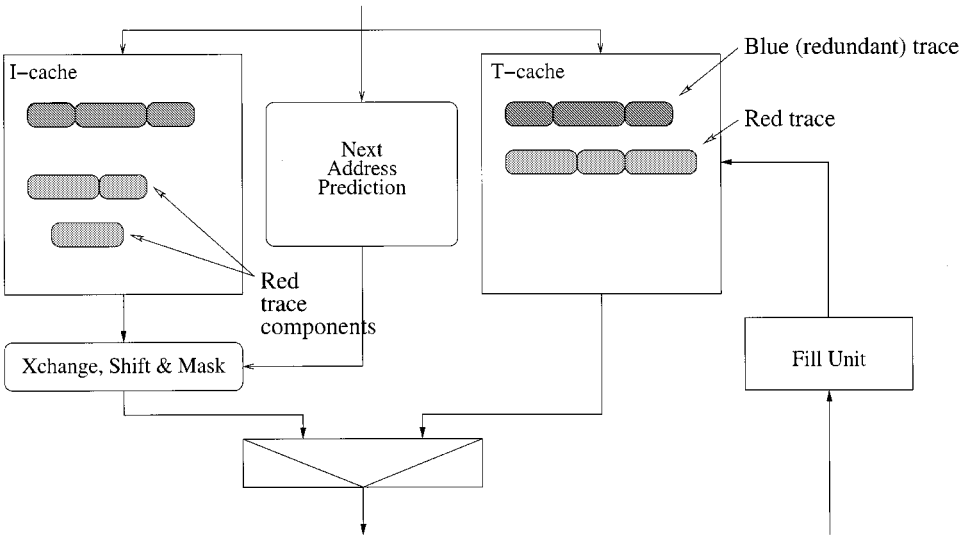


Fig. 23. The trace cache is storing redundant traces, because consecutive basic blocks can be obtained from the instruction cache.

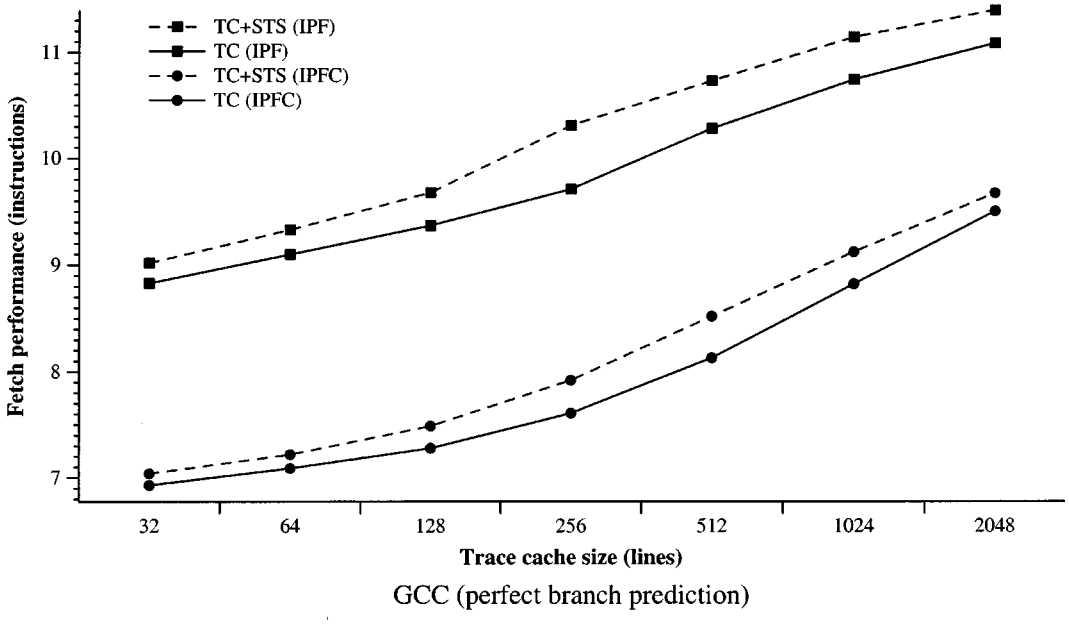


Fig. 24. Fetch performance obtained with the baseline trace cache, and a trace cache using selective trace storage.

E. Code Reordering and Trace-Level Redundancy

The use of code reordering techniques proved very effective at reducing the number of taken branches in a program by mapping basic blocks in execution order. By doing the work of the trace cache at compile time, the number of redundant traces may have increased significantly.

Fig. 25 shows a classification of the traces executed by the SPECint95 programs by the number of sequence breaks they contain, as shown in [47]. Traces containing zero breaks are considered blue (redundant) traces, because they can be fetched by the core fetch unit in a single cycle, without need of the trace cache. Traces containing one or more breaks are considered red traces and can not be provided by the core fetch unit in a single cycle, but they can be provided by the trace cache.

The results show that the number of blue (redundant) traces increases significantly (from 39% to 62%) when we use an optimized code layout. This adds another advantage to the use of code reordering techniques: aside from reducing the instruction cache miss rate, and increasing the effective fetch width, they also increase the effectivity of the selective trace storage. As more traces can be obtained from the core fetch unit, there are fewer traces remaining to be stored in the trace cache, and the same program will then fit all its red traces in a smaller trace cache.

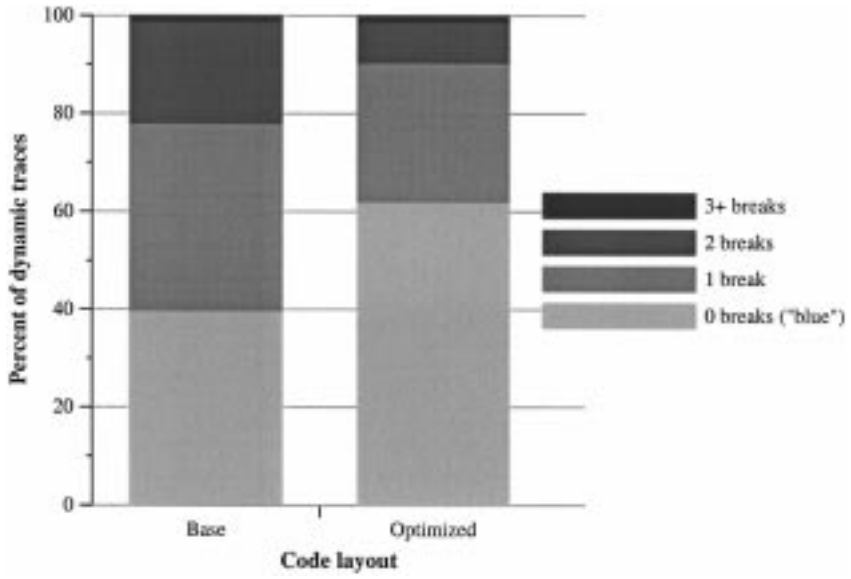


Fig. 25. Classification of traces by the number of sequence breaks contained. Traces with zero are considered blue traces.

VI. Concluding Remarks

We have shown how instruction fetch has evolved with each new processor design to provide the required number of instructions to the execution engine of the processor.

Following the ILP trend and the widening of the processor pipeline, we have seen how instruction fetch started reading one instruction every few cycles, then one instruction per cycle, then one basic block per cycle, and then several basic blocks per cycle. However, the fetch engine can not be replicated as other pipeline stages by adding more functional units: new engines had to be designed, solving the problem of fetching instructions past unresolved branches, and fetching nonsequential instructions past several branches in a single cycle.

But pipeline widening is not the only way to higher processor performance; longer pipelines and faster clock rates are also being exploited. We have also shown how deeper pipelines and shorter cycle times impose new challenges to the instruction fetch stage: smaller prediction tables and smaller caches will be required. Also, the complexity implied by some high performance fetch engines will also require changes to fit in a single pipeline stage.

Overall, we have shown how fetch performance mainly depends on three factors: the perceived memory latency, the quality of the instructions provided, and the width of instructions provided.

The memory latency issue will require better memory prefetching schemes and new memory hierarchies, specially in the context of high clock rate processors, which will be unable to access large memories in a single cycle.

The use of deeper pipelines and faster clock rates will also increase the importance of the instruction quality. The longer it takes to realize that a branch has been mispredicted, the more harmful it is to performance and power consumption. As pipelines become deeper, the need for better branch predictors will increase.

Finally, the increasing pipeline widths will require fetch engines capable of producing a high enough width of instructions without requiring large amounts of complex hardware, which could make the fetch stage the critical stage for determining the clock rate.

And for all the described mechanisms, we have shown how the use of compiler optimizations such as trace scheduling and code layout optimizations can be used to boost the performance of the chosen fetch architecture.

Acknowledgment

The authors also want to thank the reviewers for their insightful comments.

References

[1] T. Ball and J. R. Larus, "Efficient path profiling," in *Proc. 29th Annu. ACM/IEEE Intl. Symp. Microarchitecture*, Dec. 1996.

[2] B. Black, B. Rychlik, and J.P. Shen, "The block-based trace cache," in *Proc. 26th Annu. Int. Symp. Computer Architecture*, May 1999.

[3] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: The Simplescalar tool set," Univ. Wisconsin, Tech. Rep. TR-1308, July 1996.

[4] B. Calder and D. Grunwald, "Fast & accurate instruction fetch and branch prediction," in *Proc. 21st Annu. Int. Symp. Computer Architecture*, 1994, pp. 2–11.

[5] ——, "Reducing branch costs via branch alignment," in *Proc. 6th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 242–251.

[6] ——, "Next cache line and set prediction," in *Proc. 22th Annu. Int. Symp. Computer Architecture*, June 1995.

[7] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An optimizer for Alpha/NT executables," *USENIX*, pp. 17–23, Aug. 1997.

[8] T. Conte, K. Menezes, P. Mills, and B. Patell, "Optimization of instruction fetch mechanism for high issue rates," in *Proc. 22th Annu. Int. Symp. Computer Architecture*, June 1995, pp. 333–344.

[9] S. Patel *et al.*, *Proc. 33rd Annu. ACM/IEEE Int. Symp. Microarchitecture*, 2000.

[10] K. I. Farkas, N. P. Jouppi, and P. Chow, "How useful are nonblocking loads, stream buffers and speculative execution in multiple issue processors?," in *Proc. 1st Int. Conf. High Performance Computer Architecture*, Jan. 1995, pp. 78–89.

[11] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.*, vol. 30, no. 7, pp. 478–490, July 1981.
[12] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proc. 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 85–95.
[13] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative fetch and issue techniques from the trace cache mechanism," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1997.
[14] ——, "Putting the fill unit to work: Dynamic optimization for trace cache microprocessors," in *Proc. 31st Annu. ACM/IEEE Int. Symp. Microarchitecture*, Nov. 1998, pp. 173–181.
[15] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder, "Procedure placement using temporal ordering information," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1997, pp. 303–313.
[16] L. Gwennap, "Digital 21 264 sets new standard," *Microprocessor Rep.*, vol. 10, no. 14, 1996.
[17] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 1997, pp. 171–182.
[18] M. D. Hill and A. J. Smith, "Experimental evaluation of on-chip microprocessor cache memories," in *Proc. 11th Annu. Int. Symp. Computer Architecture*, June 1984, pp. 158–166.
[19] W.-M. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Annu. Int. Symp. Computer Architecture*, June 1989, pp. 242–251.
[20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Water, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. K. Haab, J. G. Hold, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *J. Supercomput.*, no. 7, pp. 9–50, 1993.
[21] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1997.
[22] Q. Jacobson and J. E. Smith, "Instruction pre-processing in trace processors," in *Proc. 5th Int. Conf. High Performance Computer Architecture*, Jan. 1999, pp. 125–129.
[23] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, June 1997, pp. 252–263.
[24] N. J. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Computer Architecture*, May 1990, pp. 364–373.
[25] J. Kalamatianos and D. R. Kaeli, "Temporal-based procedure reordering for improved instruction cache performance," in *Proc. 4th Int. Conf. High Performance Computer Architecture*, Feb. 1998.
[26] A. Krall, "Improving semi-static branch prediction by code replication," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1994, pp. 97–106.
[27] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bi-mode branch predictor," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1997, pp. 4–13.
[28] D. Lee, J.-L. Baer, B. Calder, and D. Grunwald, "Instruction cache fetch policies for speculative execution," in *Proc. 22nd Annu. Int. Symp. Computer Architecture*, June 1995, pp. 357–367.
[29] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Comput.*, vol. 17, pp. 6–22, Jan. 1984.
[30] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. 25th Int. Symp. Microarchitecture*, Dec. 1992.
[31] S. McFarling and J. Hennessy, "Reducing the cost of branches," in *Proc. 13th Annu. Int. Symp. Computer Architecture*, 1986, pp. 396–403.
[32] W.-M. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc. 16th Annu. Int. Symp. Computer Architecture*, June 1989, pp. 224–233.
[33] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, 1997, pp. 292–303.
[34] F. Mueller and D. A. Whalley, "Avoiding conditional branches by code replication," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1995, pp. 56–66.
[35] F. Mueller and D. B. Whalley, "Avoiding unconditional jumps by code replication," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1992, pp. 322–330.
[36] R. Muth, "Alto: A platform for object code modification," Ph.D. dissertation, Univ. Arizona, 1999.
[37] C. Navarro, A. Ramirez, J. L. Larriba-Pey, and M. Valero, "On the performance of fetch engines running DSS workloads," in *Proc. Int. Euro-Par Conf.*, Aug. 2000.
[38] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, June 1997.
[39] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *Proc. 25th Annu. Int. Symp. Computer Architecture*, June 1998, pp. 262–271.
[40] J. R. C. Patterson, "Accurate static branch prediction by value range propagation," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 1995, pp. 67–78.
[41] A. Peleg and U. Weiser, "Dynamic flow instruction cache memory organized around trace segments independent of virtual address line," U.S. Patent 5.381.533, Jan. 1995.
[42] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 1990, pp. 16–27.
[43] A. Ramirez, J. L. Larriba-Pey, C. Navarro, X. Serrano, J. Torrellas, and M. Valero, "Optimization of instruction fetch for decision support workloads," in *Proc. Int. Conf. Parallel Processing*, Sept. 1999, pp. 238–245.
[44] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero, "Software trace cache," in *Proc. 13th Int. Conf. Supercomputing*, June 1999.
[45] A. Ramirez, J. L. Larriba-Pey, and M. Valero, "The effect of code reordering on branch prediction," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, Oct. 2000, pp. 189–198.
[46] ——, "A stream processor front-end," *IEEE TCCA Newsletter*, pp. 10–13, 2000.
[47] ——, "Trace cache redundancy: Red & blue traces," in *Proc. 6th Int. Conf. High Performance Computer Architecture*, Jan. 2000.
[48] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *Proc. 26th Annu. Int. Symp. Computer Architecture*, May 1999, pp. 234–245.
[49] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *Proc. 32nd Annu. ACM/IEEE Int. Symp. Microarchitecture*, 1999, pp. 16–24.
[50] E. Rotenberg, S. Benett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proc. 29th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1996, pp. 24–34.
[51] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1997, pp. 138–148.
[52] A. Seznec, "A case for two-way skewed-associative caches," in *Proc. 20th Annu. Int. Symp. Computer Architecture*, May 1993, pp. 169–178.
[53] J. E. Smith, "A study of branch prediction strategies," in *Proc. 8th Annu. Int. Symp. Computer Architecture*, 1981, pp. 135–148.
[54] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proc. IEEE*, vol. 83, Dec. 1995.
[55] S. P. Song, M. Denman, and J. Chang, "The PowerPC 604 RISC microprocessor," *IEEE Micro*, vol. 14, pp. 8–17, Oct. 1994.
[56] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, 1997, pp. 284–291.
[57] A. Srivastava and D. W. Wall, "A practical system for intermodule code optimization at link-time," *J. Program. Lang.*, vol. 1, pp. 1–18, Dec. 1992.
[58] J. Stark, P. B. Racunas, and Y. N. Patt, "Reducing the impact of icache misses by writing instructions into the reservation stations out of order," in *Proc. 30th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1997, pp. 34–43.
[59] J. Torrellas, C. Xia, and R. Daigle, "Optimizing instruction cache performance for operating system intensive workloads," in *Proc. 1st Int. Conf. High Performance Computer Architecture*, Jan. 1995, pp. 360–369.
[60] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer, "Instruction fetching: Coping with code bloat," in *Proc. 22th Annu. Int. Symp. Computer Architecture*, June 1995, pp. 345–356.

[61] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the instruction fetch rate via multiple branch prediction and a branch address cache," in *Proc. 7th Int. Conf. Supercomputing*, July 1993, pp. 67–76.
[62] T. Y. Yeh and Y. N. Patt, "Two-level adaptive branch prediction," in *Proc. 24th Annu. ACM/IEEE Int. Symp. Microarchitecture*, 1991, pp. 51–61.
[63] ——, "Alternative implementations of two-level adaptive branch prediction," in *Proc. 19th Annu. Int. Symp. Computer Architecture*, 1992, pp. 124–134.
[64] ——, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *Proc. 25th Annu. ACM/IEEE Int. Symp. Microarchitecture*, Dec. 1992, pp. 129–139.
[65] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proc. 6th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Oct. 1994, pp. 232–241.

Alex Ramirez received the computer science degree from the Polytechnic University of Catalonia (UPC), Spain, in 1997.

In 1998, he joined the Computer Architecture Department at UPC where he is a Ph.D. student. His research areas of special interest are profile-guided compiler optimizations, code layout optimizations, performance studies of user and system code like database applications, and the design and implementation of the fetch stage of superscalar processors. He has been a student intern at Compaq's Western Research Lab., Palo Alto, CA, and Intel's Microprocessor Research Lab., Santa Clara, CA. Since 2000, he has been lecturing on operating systems as an Assistant Professor.

Josep L. Larriba-Pey received the computer science degree in 1989 and the Ph.D. degree in 1996, both from the Polytechnic University of Catalonia (UPC), Spain.

He is an Associate Professor in the Computer Architecture Department at UPC. His current research interests are in the relation between the architecture of the computer, the compiler and the high level applications with special interest in databases; the tuning of the basic sequential and parallel DBMS operations and, the design, analysis and tuning of sequential and parallel nonnumeric algorithms. At present, he is also involved in research and development projects with IBM and Intel.

Mateo Valero (Fellow, IEEE) received the telecommunication engineering degree from the Polytechnic University of Madrid, Spain, in 1974 and the Ph.D. degree from the Polytechnic University of Catalonia (UPC), Spain, in 1980.

He is a Professor in the Computer Architecture Department at UPC. His current research interests are in the field of high performance architectures, with special interest in the following topics: processor organization, memory hierarchy, interconnection networks, compilation techniques and computer benchmarking. He has published approximately 200 papers on these topics. He served as the general chair for several conferences, including ISCA-98 and ICS-95. He is a member of the subcommittee for the Ecker-Mauchly Award and director of the C4 (Catalan Center for Computation and Communications).

Dr. Valero has been honored with several awards, including the Narcis Monturiol, presented by the Catalan Government, the Salvà i Campillo presented by the Telecommunications Engineer Association and ACM, and the King Jaime I by the Generalitat Valenciana. He has been an Associate Editor for IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS for three years. Since 1994, he has been a Member of the Spanish Engineering Academy.