2 STATE OF THE ART

In this work we are trying to increase fetch performance using both software and hardware techniques, combining them to obtain maximum performance at the minimum cost and complexity. In this chapter we introduce the state of the art regarding both compiler optimizations for instruction fetch, and the fetch architectures for which we try to optimize our applications.

The first section introduces code layout optimizations. The compiler can influence instruction fetch performance by selecting the layout of instructions in memory. This determines both the conflict rate of the instruction cache, and the behavior (taken or not taken) of conditional branches. We present an overview of the different layout algorithms proposed in the literature, and how our own algorithm relates to them.

The second section shows how instruction fetch architectures have evolved from pipelined processors to the more aggressive wide issue superscalars. This implies increasing fetch performance from a single instruction every few cycles, to one instruction per cycle, to a full basic block per cycle, and finally to multiple basic blocks per cycle. We emphasize the description of the trace cache, as it will be the reference architecture in most of our work.

2.1 Code layout optimizations

The mapping of instruction to memory is determined by the compiler. This mapping determines not only the code page where an instruction is found, but also the cache line (or which set in a set associative cache) it will map to. Furthermore, a branch will be taken or not taken depending on the placement of the successor basic blocks.

By mapping instructions in a different order, the compiler has a direct impact on the fetch engine performance. In this section we provide a brief description of the different algorithms proposed to select where each instruction should be mapped.

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 which can be obtained from them.

2.1.1 Basic block chaining

Basic block chaining organizes basic blocks into traces, mapping together those basic blocks which tend to execute in sequence. There have been several algorithms proposed to determine which basic blocks should build a trace [3, 20, 32, 59, 87].

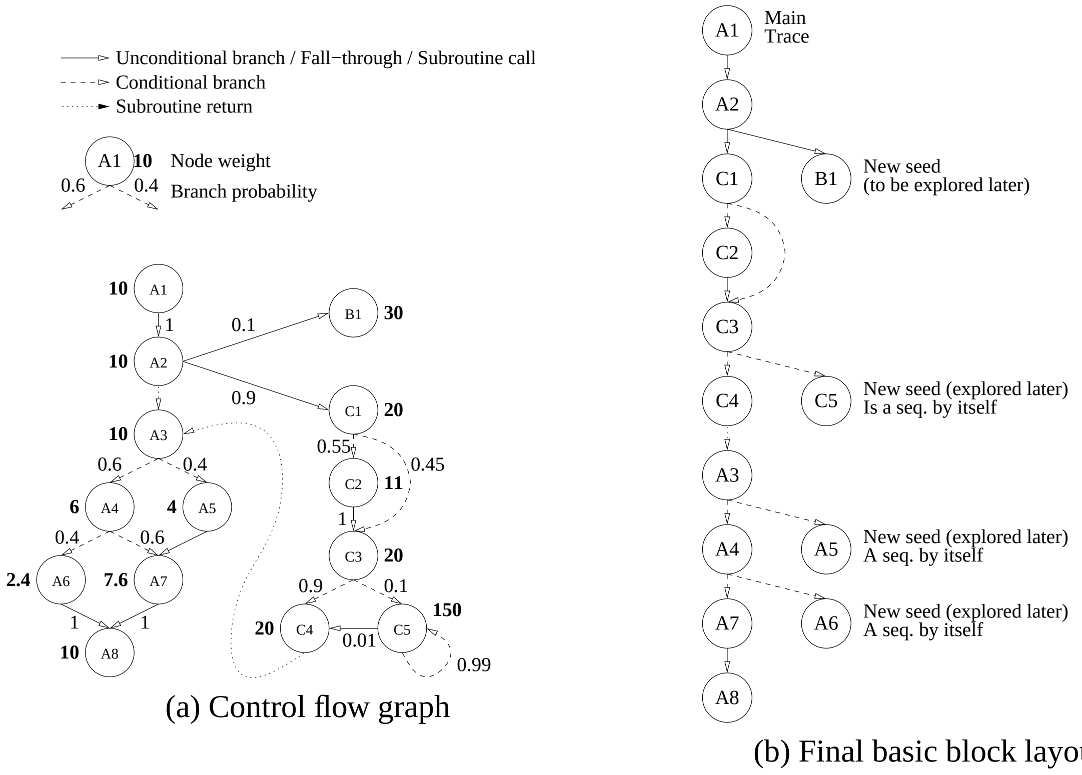

Unconditional branch / Fall−through / Subroutine call
Conditional branch
Subroutine return
Node weight
Branch probability
(a) Control flow graph
Main Trace
New seed (to be explored later)
New seed (explored later) Is a seq. by itself
New seed (explored later) A seq. by itself
New seed (explored later) A seq. by itself
(b) Final basic block layout

Figure 2.1. Example of a basic block chaining algorithm. Basic blocks are mapped so that the execution trace is consecutive in memory.

As an example, Figure 2.1 shows the chaining algorithm used in [32, 87]. It is a greedy algorithm, which given a *seed*, or starting basic block, follows the most frequent path out of it as long as that basic block has an execution frequency larger than a given *ExecThreshold*, and the transition has a probability higher than a given *BranchThreshold*. 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 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 [59], and used in [13, 49, 84]. 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.

However, 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 [3]. 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. In this case, the correspondence between the profile data and the basic block chains which should be built is immediate.

Our basic block chaining algorithm derives from the one used in [87]. As we show in Chapter 4, we improve their chaining algorithm by automating some parts of the algorithm which require human intervention in [87] like the seed selection, and removing the Exec and Branch threshold values from the algorithm.

2.1.2 Procedure splitting

After a new ordering of the basic blocks has been established for a given procedure, the frequently executed basic blocks are mapped towards the top of the procedure, while infrequently used basic blocks will move towards 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.

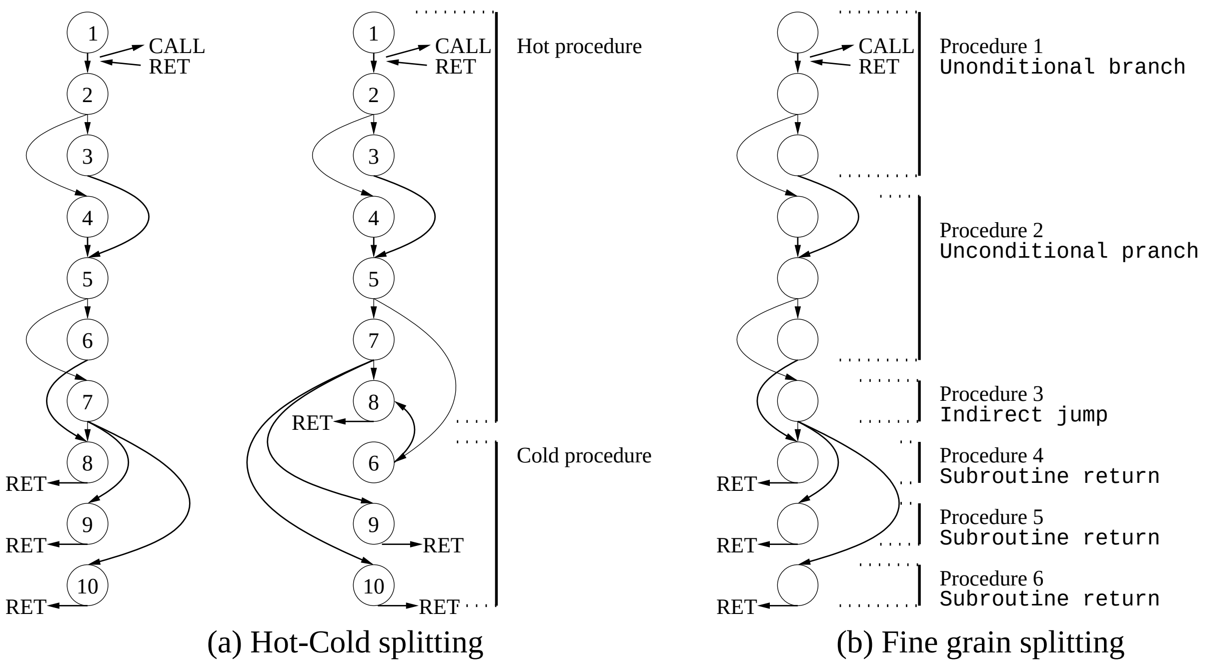


Figure 2.2. Examples of the procedure splitting algorithm. Infrequently used basic blocks are mapped to a different procedure, and moved away of the execution path.

Figure 2.2 shows two different ways of splitting a procedure body. A coarse grain splitting would split the routine in two parts [49, 13, 59, 84]: 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 [32, 87]. 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.

2.1.3 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.

Figure 2.3 shows the mapping algorithm used in [49, 84, 59, 13]. 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 [24].

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.

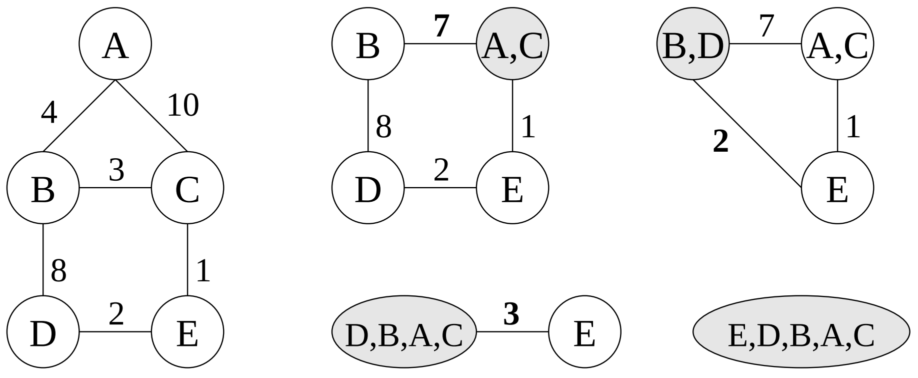


Figure 2.3. Routine mapping algorithm by Pettis & Hansen. Two routines which call each other will be mapped close in memory to avoid conflicts among them.

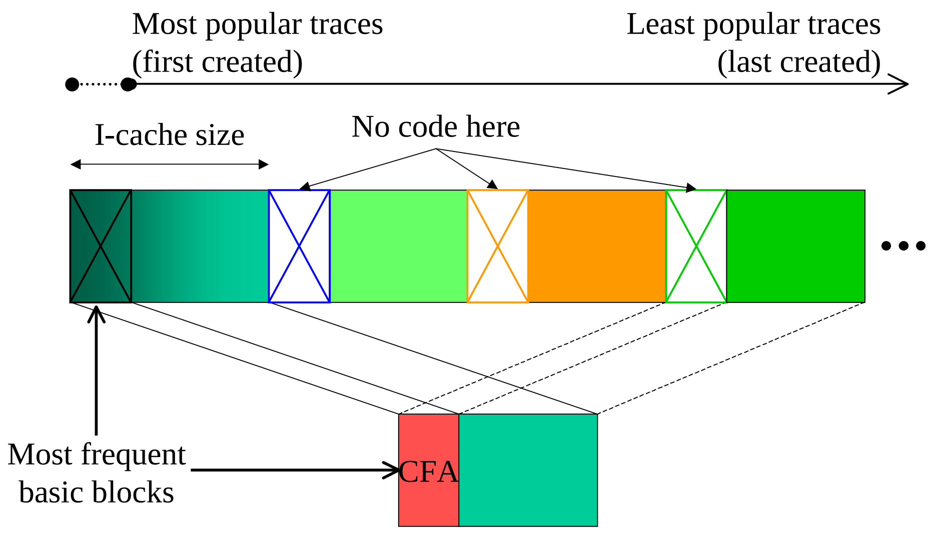


Figure 2.4. Code mapping algorithm by Torrellas *et al*. The most heavily executed basic blocks are mapped to a reserved area of the instruction cache, eliminating conflicts in important parts of the code.

Figure 2.4 shows the mapping algorithm used in [87]. It follows the basic block chaining phase described in Section 2.1.1. After all basic blocks have been mapped to chains, the chains are ordered by popularity. The most popular chains are mapped to the beginning of the address space, while the least popular ones are mapped towards 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 basic blocks by ensuring that no other code maps to that same range of cache addresses. These basic blocks are pulled out of whichever chain they mapped to, and moved into this privileged cache space. This ensures that the most frequently used basic blocks will never miss in the cache due to a conflict miss. This reserved cache space is called the *Conflict Free Area* (CFA). The size of the CFA is determined experimentally.

In Chapter 4 we improve on this mapping algorithm by keeping all the basic blocks in a chain together. That is, instead of mapping individual basic blocks into the CFA, we map the whole chain, which increases spatial locality, and avoids taken branches whenever possible. We also extend the algorithm to include a heuristic to determine the CFA size automatically.

In [28, 36], 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.

2.2 Processor architecture

2.2.1 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. This fact 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 [43].

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

Figure 2.5 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 [43].

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 *vs.* not taken) [80, 95], and predicting the branch target address [40]. 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 di-

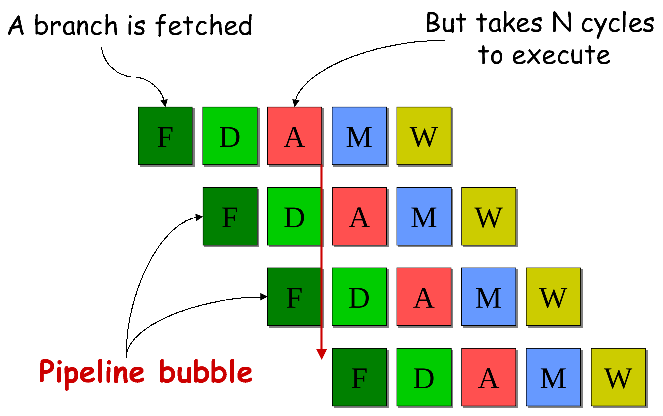


Figure 2.5. Pipelined execution of a branch instruction. The branch is not resolved until the ALU stage, which introduces two delay slots.

rection 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.

Figure 2.6 shows the execution cost of a branch instruction for several branch architectures as shown in [43][1] The execution cost of a branch shows the average number of cycles it takes to correctly execute a branch.

The results shown correspond to a 5-stage pipeline architecture (such as the one shown in Figure 2.5), 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:

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 [80], 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.

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

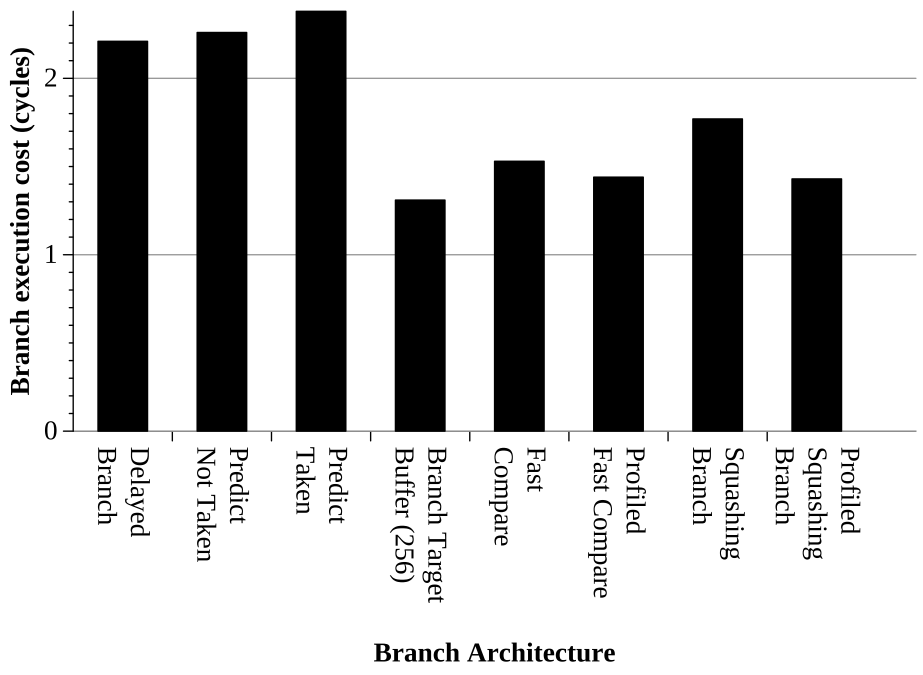


Figure 2.6. Comparison of the branch execution cost for different branch architectures using both software and hardware techniques.

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 slots can always be filled with useful instructions, adding an instruction squash mechanism which 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.

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 [9] 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.

Further performance improvements can be obtained by enhancing the branch alignment optimization with other code transformations which increase the static branch prediction accuracy. Among these optimizations we find unconditional branch removal [48], conditional branch removal [47], using branch correlation in static predictions [96, 38], and value range propagation [57].

2.2.2 Superscalar processors

Superscalar processors replicate the pipeline in the execution engine, allowing the simultaneous execution of several independent instructions, exploiting Instruction Level Parallelism (ILP) [79]. But the fetch engine can not be replicated in the same way, which introduces 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 4-issue processor with instructions to keep its functional units busy. 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 [8, 93].

Furthermore, with a 20% branch frequency, it is likely that one branch per cycle will be fetched, increasing the importance of the branch prediction mechanism.

This motivates 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 [40], and the addition of a saturating two-bit counter to predict the branch direction [80].

Two-level adaptive branch predictors [92, 95] 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).

Figures 2.7 and 2.8 show two examples of two-level adaptive branch predictors. The gshare predictor [42] correlates a branch with the outcomes of all previous branch instances, that is, the prediction for the current branch depends on the behavior of the previous branches. The first level table consists of a single register which contains the global branch history. For each dynamic

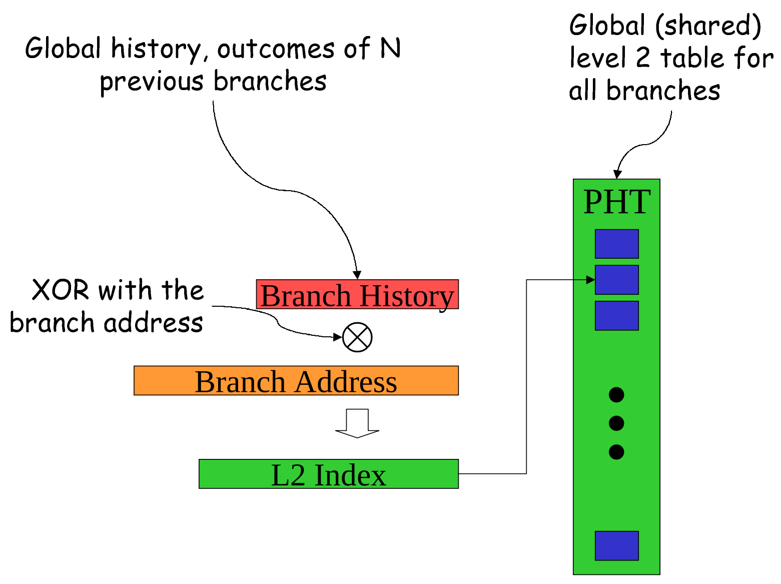

Global history, outcomes of N previous branches
Global (shared) level 2 table for all branches
PHT
XOR with the branch address
Branch History
Branch Address
L2 Index

Figure 2.7. gshare (global history)

branch instance, we shift the outcome of the branch into the register. Then we use the contents of this register and the branch address to index the second level table, which contains a set of 2-bit counters which provide the final prediction.

The PAp predictor [95] correlates a branch with the previous instances of the same static branch, that is, the prediction for the current branch depends on the past behavior of the same branch. The first level table contains a history register for each static branch. For each dynamic branch instance, we shift the outcome of the branch in the appropriate history register. Then we use the contents of the register to index into the second level table, also private for each branch, and access the 2-bit counter which provides the final prediction.

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 due to space limitations, 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 which reduce this effect. Among this predictors we find the agree predictor [82], the bi-mode predictor [39], and the gskew predictor [46].

Figure 2.9 shows the implementation of some dealiased branch predictors. The agree predictor adds a direction bias bit to the BTB, and changes the semantics of the 2-bit counters in the second level table. The bias bit predicts the behavior of the branch (taken or not taken), and the 2-bit counter predicts if the branch will agree with the bias bit or not. This way, two branches with opposite behavior may share the same 2-bit counter with no harmful effect, because both will agree with the bias bit, which will be different.

The bimode predictor reduces harmful interference because it splits branches in two groups: the mostly taken set, and the mostly not taken set. Then, it uses a separate prediction table for each set. If two branches share the same 2-bit counter, it means that they belong to the same set, so it is likely that the interference will not be negative because both branches push the counter in the same direction.

The gskew predictor reduces branch aliasing by adding skew associativity to the branch pre-

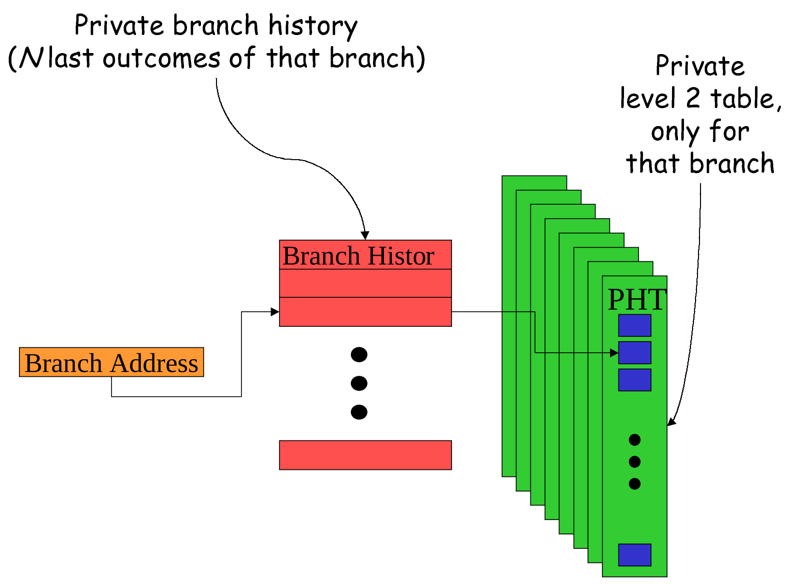


Figure 2.8. PAp (private history)

dictor. It uses three separate prediction tables, each one indexed using a different hashing function. It is very unlikely that a branch will suffer aliasing problems in all tables, so it is assumed that at least two tables will not suffer negative interference, and a majority vote is used to decide the final prediction.

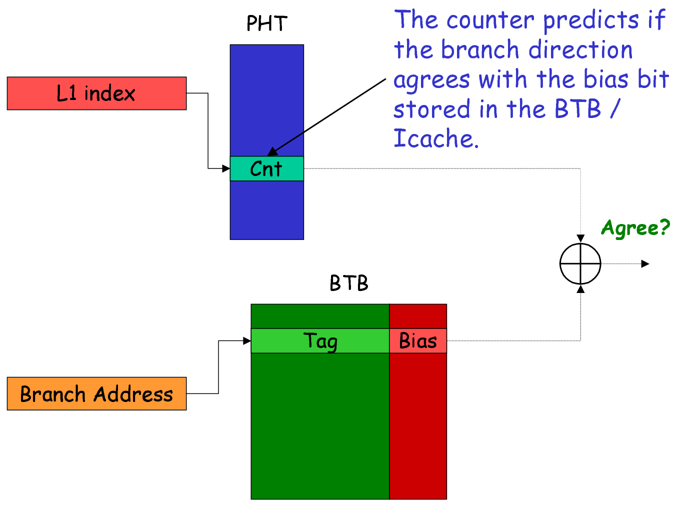
Figure 2.10 shows the fetch mechanism proposed by Yeh and Patt in [93] for fetching a full basic block of instructions per cycle. All blocks are accessed in parallel using the fetch address as index: a cache line is fetched from memory, the top of the return address stack (RAS) is read, and the branch target buffer (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.

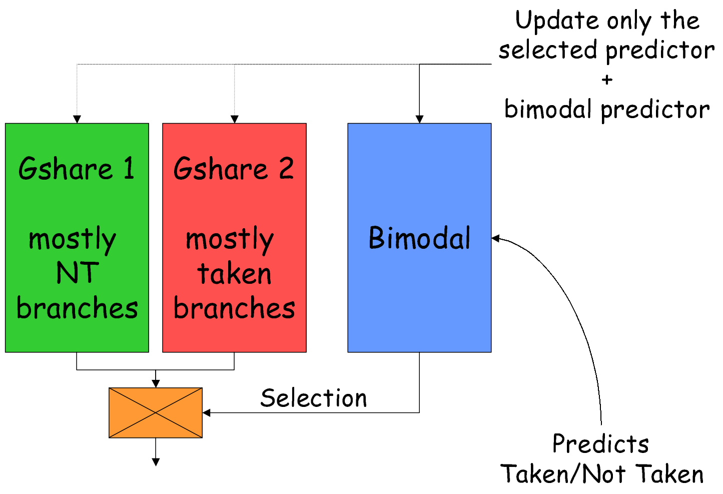
This fetch mechanism approaches all three fetch performance factors relevant at this point: the memory latency is hidden using an instruction cache; the fetch width is increased by reading a whole basic block of instructions, which reside in consecutive memory positions; and the branch prediction accuracy is increased using a BTB in conjunction with a 2-level adaptive branch predictor [95].

Figure 2.11 shows two performance metrics for the fetch engine described in [93][2] : the branch execution penalty (BEP), and the instructions per fetch cycle (IPFC). The BEP represents the

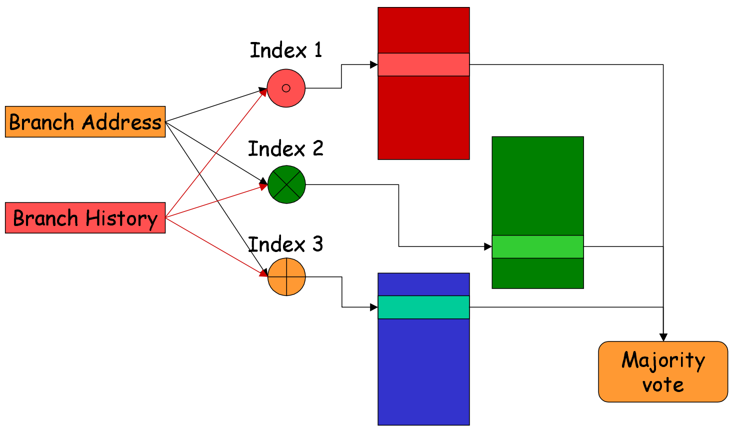
[2]eqntott, espresso, gcc, li, doduc, fpppp, matrix300, spice2g6, and tomcatv from SPEC92 on a Motorola88100 instruction level simulator


PHT
The counter predicts if the branch direction agrees with the bias bit stored in the BTB / Icache.
L1 index
Cnt
Agree?
BTB
Tag
Bias
Branch Address

(a) agree


Update only the selected predictor + bimodal predictor
Gshare 1 mostly NT branches
Gshare 2 mostly taken branches
Bimodal
Selection
Predicts Taken/Not Taken

(b) bimode



(c) gskew

Figure 2.9. Dealiased branch predictors.

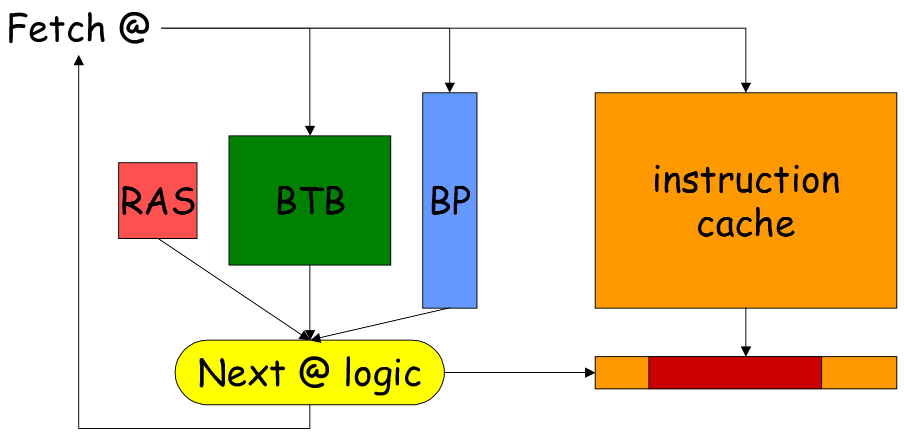


Figure 2.10. A fetch mechanism capable of reading one basic block per cycle.

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 2 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 4 integer, and 5 FP codes from the SPEC92 benchmark set (eqntott, espresso, gcc, li, doduc, fpppp, matrix300, spice2gr6, tomcatv).

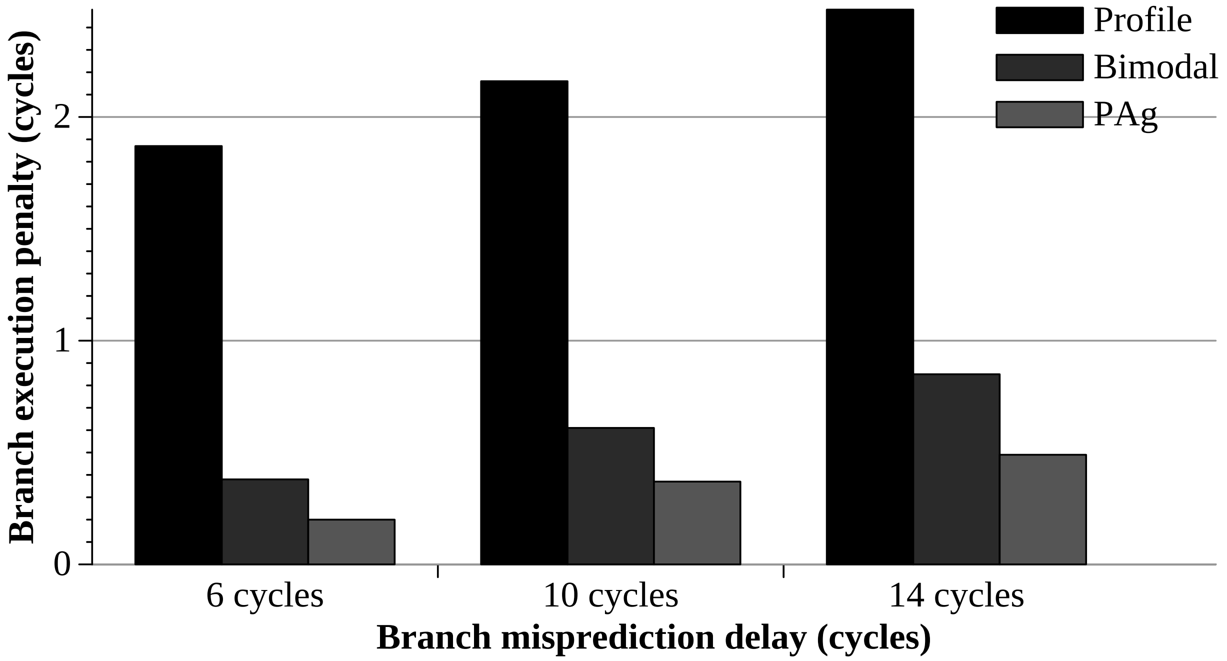
The different fetch engine configurations tested are:

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.

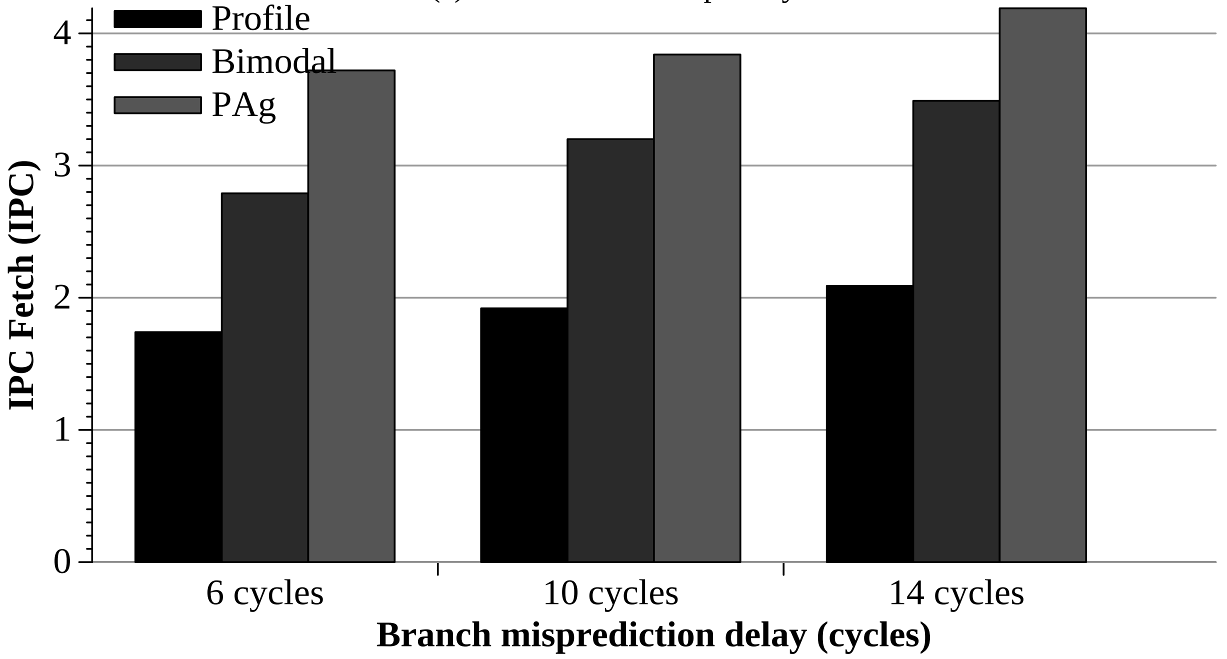
Bimodal: Uses the saturating 2-bit counter proposed in [80] to predict conditional branch directions. Both the 2-bit counter and the target address are obtained from the BTB. The BTB has 512 entries, and is 4-way associative.

2-level pred.: Uses the private history two-level branch predictor proposed in [95]. 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 4-way associative. The history registers are 12-bit long, and index into a 4K-entry PHT.

Previous studies have shown how profile based branch prediction can be as accurate as that obtained with 2-bit saturating counters [21]. The poor fetch performance of the *profile* setup is mainly due to the 2-cycle delay in the calculation of the branch target address of taken branches.



(a) Branch execution penalty



(b) Fetched instructions per cycle

Figure 2.11. Performance metrics for the fetch engine proposed by Yeh & Patt.

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 *2-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 [8]:

Decoupled BTB and branch predictor: The original design does not allow dynamic prediction for branches which 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 2-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 [10], represents an alternative mechanism for fetching instructions in a superscalar processor. 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 Figure 2.11). The PowerPC 604 [81] 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 21264 [26] 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 [52], 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 which 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.

Figure 2.12 shows the mechanism proposed by Reinman, Calder and Austin in [71] to alleviate the negative impact of this trend. The fetch 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 which reads the fetch target blocks from memory and provides instructions to the decode stage.

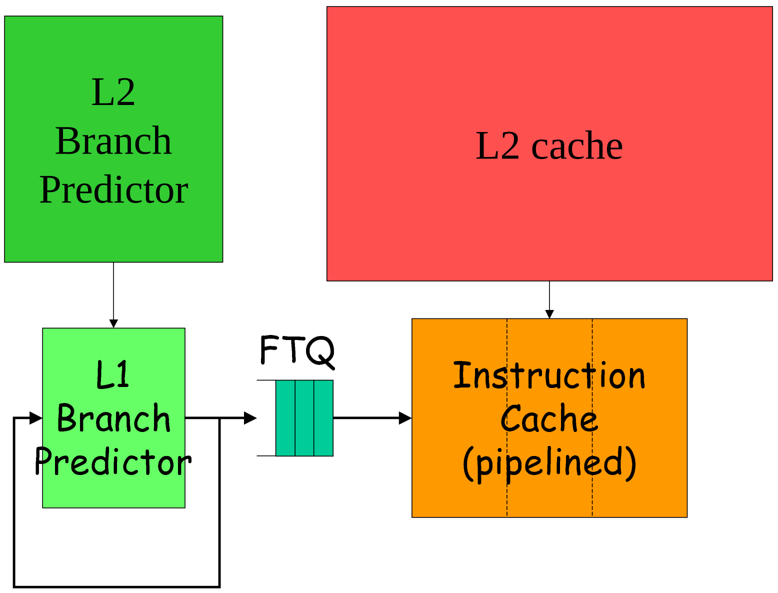


Figure 2.12. Decoupling the fetch stage: an independent branch prediction mechanism provides fetch blocks to a Fetch Target Queue, and the pipelined instruction cache reads the blocks from memory.

The branch prediction architecture used in [71] 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 [8], 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 4-issue processors. It combines all the optimizations proposed for the superscalar fetch engine on a more scalable design which 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 [85], 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, 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 [71]. 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 Fetch Target Queue (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 [72].

2.2.3 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.

Figure 2.13 shows an extension of the superscalar fetch engine to read multiple consecutive basic blocks per cycle (this is the fetch engine described in [74] and called SEQ.3). The instruction cache is interleaved to allow reading two 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 non-consecutive instructions, like the branch address cache [94], the collapsing buffer [14], and the trace cache [22, 58, 74].

The branch address cache mechanism proposed by Yeh, Marr and Patt in [94] is composed of 4 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

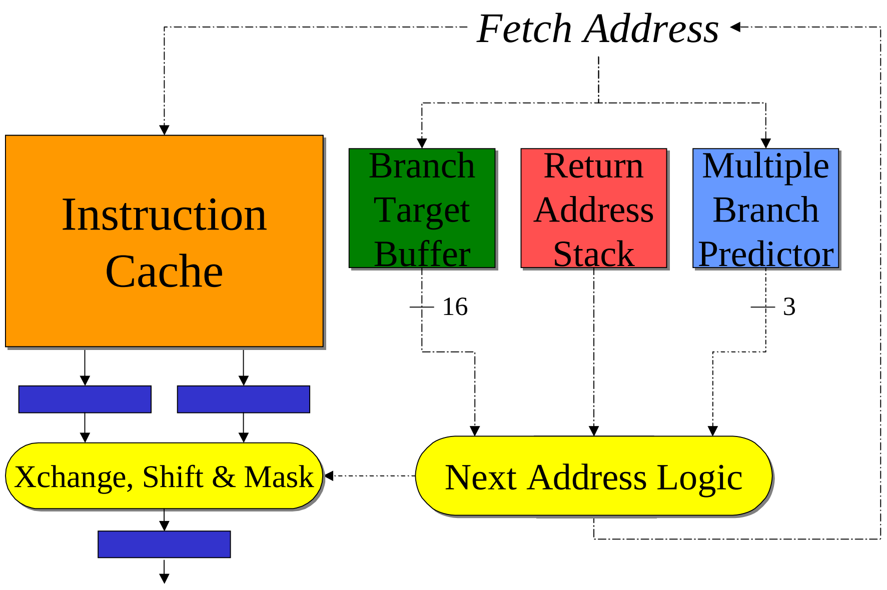


Figure 2.13. Extension of the superscalar fetch engine with a multiple branch predictor to read multiple consecutive basic blocks per cycle.

predictions obtained per cycle (2 addresses for 1 branch, 7 addresses for 2 branches, 15 addresses for 3 branches). The multiple branch predictor is used to select the tree branch which follows the current fetch address, and generate the basic block addresses which 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.

Figure 2.14 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 [94][3].

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 3 basic blocks per cycle obtains a fetch performance increase of around 100%.

The collapsing buffer proposed by Conte *et al* in [14] is composed on 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 [93], but the BTB is able to detect *intra-block* branches (branches with the target in the same cache line). The collapsing buffer uses this information to merge the

[3]eqntott, espresso, gcc, li from SPECint89, doduc, fpppp, matrix300, spice2g6, tomcatv from SPECfp89. Instruction cache is a 2-way set associative (8-way interleaved), 32KB, with 16-byte lines

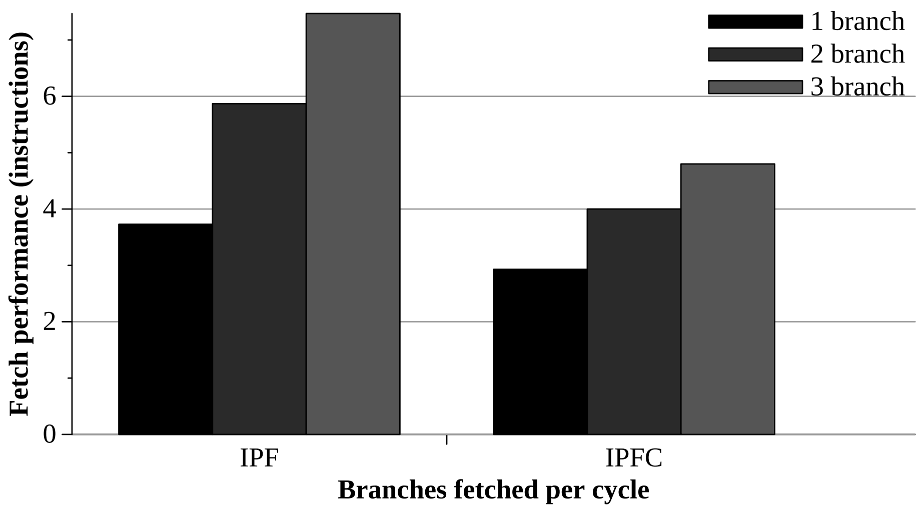


Figure 2.14. Fetch width provided by the branch address cache (IPF), and fetch engine performance measured in instructions per fetch cycle (IPFC).

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.

Figure 2.15 shows the processor performance obtained in [14] using several fetch policies on an 8-issue superscalar processor[4].

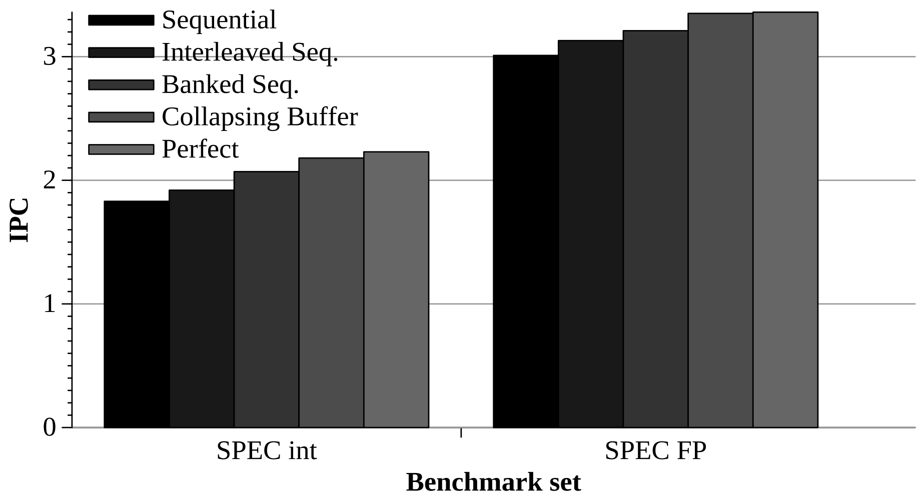


Figure 2.15. Processor performance using different sequential fetch policies and using the Collapsing Buffer.

The fetch policies shown are:

[4]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, 128KB direct mapped instruction cache with 64-byte lines, 1024-entry BTB with 2-bit counters for branch prediction.

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 non-sequential 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 intra-block branch and its target are removed.

Perfect: Ideal fetch engine, able to fetch instructions from non-sequential basic blocks.

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 intra-line branches.

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.

The trace cache

The trace cache is a fetch mechanism patented by Peleg and Weiser [58] which 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 form 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 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.

Figure 2.16 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.

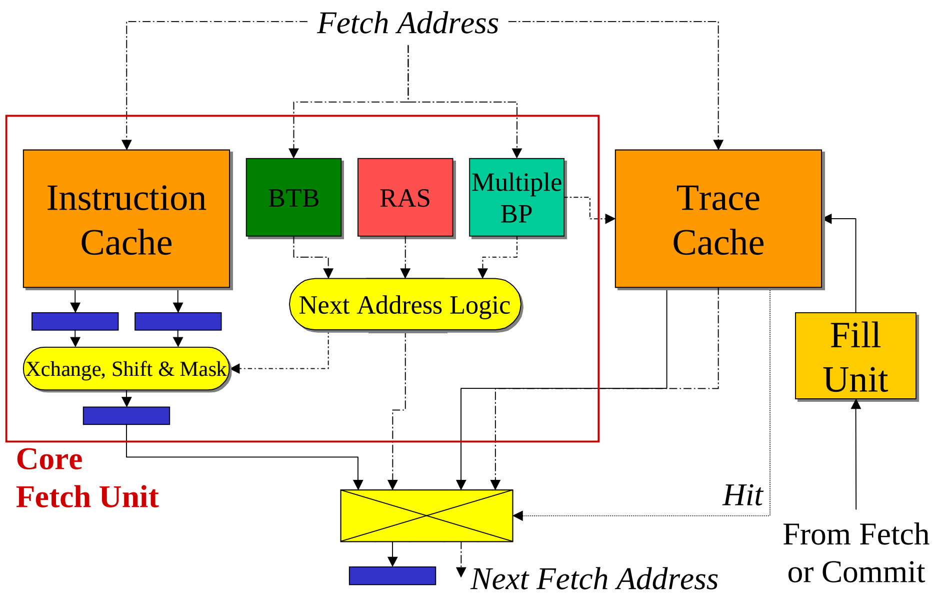


Figure 2.16. Extension of the wide superscalar fetch engine with a trace cache to allow fetching of non-consecutive basic blocks in a single cycle.

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.

Figure 2.17 shows the average processor performance obtained with different fetch engines for the SPECint95 and IBS benchmark sets, as shown by Rotenberg, Bennett and Smith [74][5]. The figure shows IPC results for a single-block fetch engine (SEQ.1), a three-block sequential fetch engine (SEQ.3, shown in Figure 2.13), 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 3-block sequential 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, Patel and Patt proposed several improvements on the base trace cache mechanism [22]:

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

[5] 16-wide fetch/dispatch rate, 3 branches per cycle, 2048-entry instruction buffer, 4096-entry global history predictor, 1K-entry BTB, 128KB instruction cache, 64-entry trace cache, 1K-entry BAC)

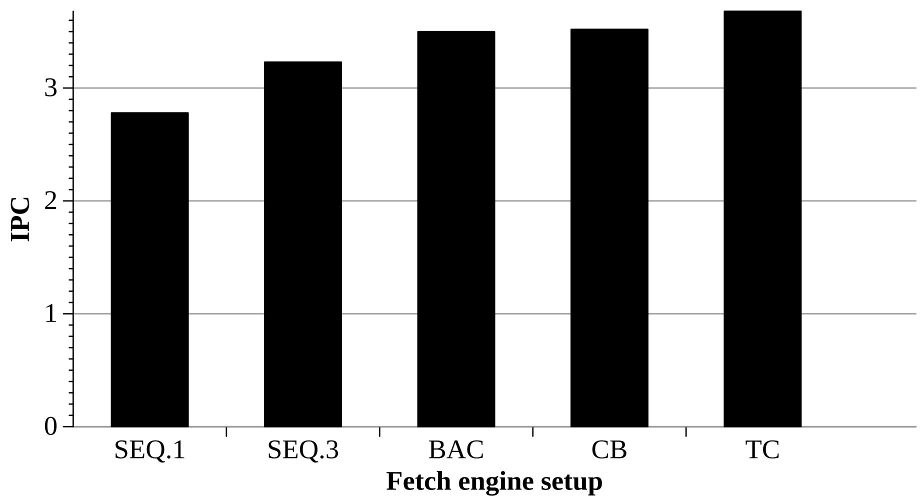


Figure 2.17. Comparison of the processor performance using different fetch engines, including the Trace Cache.

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 [54], 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 [34], and a new processor design, the trace processor [75].

Plus, the fill buffer adds a new communication channel between the execution engine and the fetch engine (see Figure 1.2), 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 [23, 33].

As described above, the trace cache is a redundant storage mechanism. Not only it stores the same instructions that could be found in the instruction cache, but it is also storing multiple copies of the same instruction.

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. Figure 2.18 shows two examples of this basic block redundancy: an if-then-else construct, and a loop. The figure shows some examples of instruction traces which 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) [6] is an alternative organization for the trace cache mechanism which avoids this redundant storage of basic blocks.

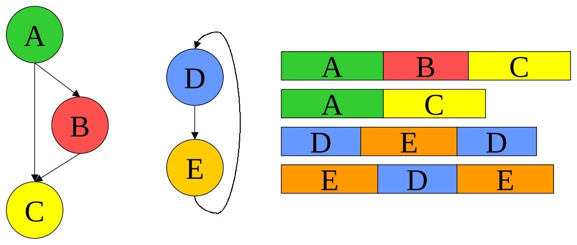


Figure 2.18. The trace cache is storing redundant information, because a basic block may be present in more than one trace cache line.

Figure 2.19 shows the BBTC organization proposed by Black, Rychlik and Shen in [6]. 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.

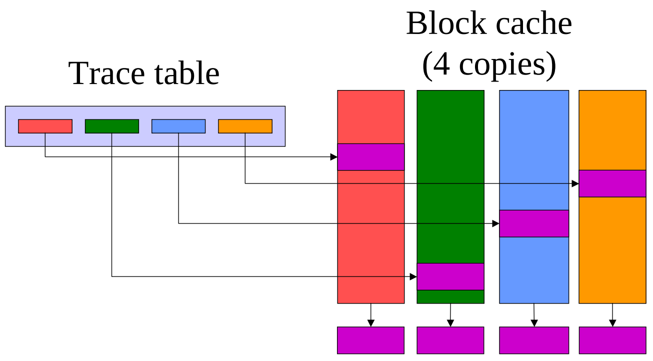


Figure 2.19. The block based trace cache stores basic blocks in a special purpose block cache, and stores block pointers in the trace table, eliminating the basic block redundancy.

The results in [6] show that using an improved storage organization, an address translation to use block indexes instead of full addresses, and an improved next trace predictor which takes advantage of this address translation, the block based trace cache obtains significant performance improvements over the baseline trace cache mechanism.

2.3 Conclusions

There has been much work done on code layout optimizations. This includes multiple algorithm for chaining the basic blocks inside a subroutine body, splitting a single subroutine into multiple/smaller subroutines, and mapping these routines in memory. However, these optimizations usually have a narrow focus. That is, they target a very specific element in the processor architecture: either the instruction cache, the TLB, or the static branch prediction mechanism.

Regarding the microarchitecture of the fetch engine, we have shown how it evolves from fetching a single instruction every few cycles in non-pipelined processors, to fetching multiple discontinuous basic blocks in a single cycle, speculating on the behavior of multiple branches. However, there is a direct relationship between fetch performance and the cost and complexity of the fetch engine. The simple architectures are unable to feed an aggressive wide superscalar processor, and those fetch architectures capable of doing so require expensive additional structures, complex control logic, and may often introduce additional delays and pipeline stages.

In this thesis, we first approach fetch performance using compiler optimizations. However, our code layout optimizations do not target a specific element in the fetch engine, but try to improve performance in all relevant aspects: latency, width, and quality of the instructions provided. Once we have obtained an optimized application, we analyze in detail the reasons for the performance improvements obtained, and design a new fetch architecture which exploits them at the minimum cost and complexity. The end target is a combination of software and hardware techniques which result in a fetch performance high enough to feed even the most aggressive superscalar processors, while maintaining a low complexity which makes theme easier to implement.

2.4 Historical context

Of course, this thesis does not represent the end of the line in research regarding code layout optimizations, nor fetch architectures. In the time it has taken to develop this work, other research groups have published advancements in related topics, which may or may not have been influenced by our work.

Figure 2.20 shows an approximate timeline of the three main topics related to this thesis: code layout optimizations, processor and fetch architectures, and dynamic branch prediction mechanism.

Regarding compiler transformations and code layout optimizations, we start with trace scheduling [20], which leads to code layout optimizations for improved instruction memory performance (TLB, instruction cache) [24, 32, 28, 29, 36, 59, 87] and for branch alignment and static branch prediction [9, 44, 38, 48, 47, 96].

After these works appears the Software Trace Cache and our performance analysis of layout optimized applications [69, 68, 61, 60]. After the first publication of the STC, appear some dynamic optimization frameworks, which perform code layout optimizations on-the-fly [45, 2], and other works analyzing the effect of code optimizations on the trace cache performance [30].

Derived from our analysis of the impact of code layout optimizations on the branch prediction mechanism [61], and related to other dealiased branch predictors, we propose a dynamic predictor organization which makes extensive use of profile data, and exploits the characteristics of optimized applications. We call our branch predictor *agbias* [64].

Regarding processor and fetch stage architecture, this chapter describes the evolution from

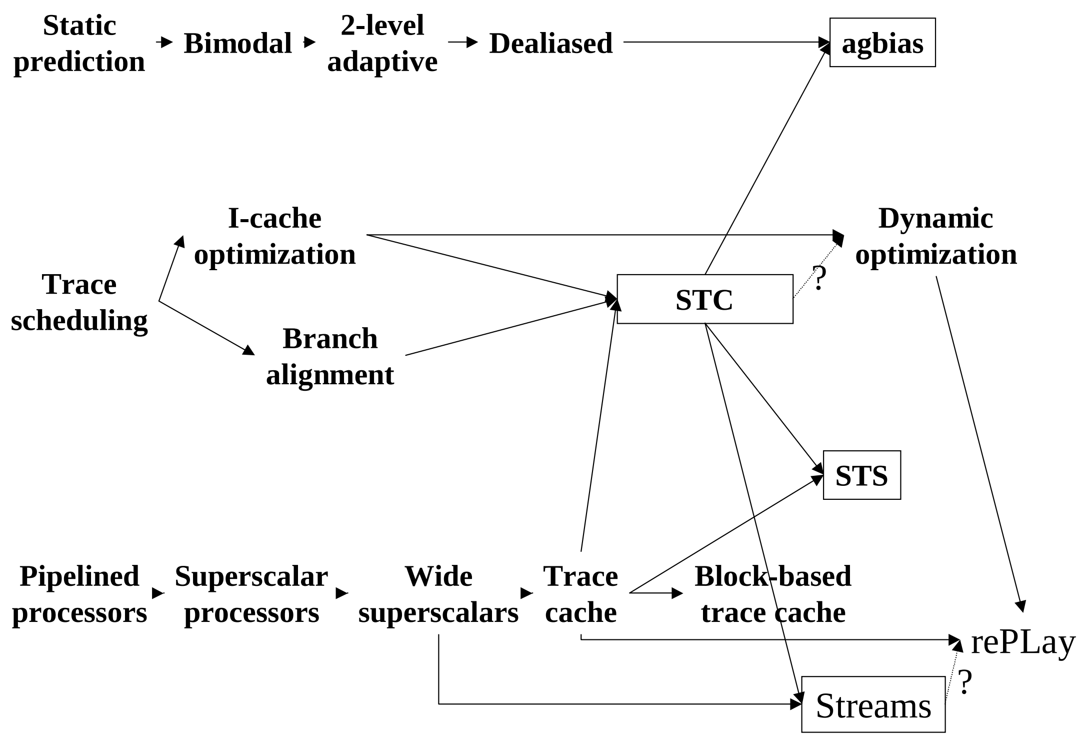


Figure 2.20. The contributions of this thesis in their historical context.

pipelined processors, to the more aggressive wide superscalars and the trace cache [22, 23, 33, 55, 54, 58, 74, 75, 76]. After the trace cache, some new architectures for trace cache-like structures appear, like the block-based trace cache [6], or the eXtended block cache [35].

Influenced by the trace cache, we evaluate the joint performance of the STC and the trace cache, and find significant redundancy in it. We propose Selective Trace Storage [70] as a trace cache modification to eliminate this redundancy.

Based on the results of our analysis of layout optimized applications, and trying to avoid the complexity and cost associated with the trace cache, we introduce the notion of an instruction stream as a long sequence of sequential instructions, and build a new fetch architecture around it [63].

Later on, directly derived from the trace cache, and the recently appeared dynamic optimization frameworks, appears the rePLay microarchitecture [53], providing a hardware infrastructure for dynamic optimizations.

This section briefly summarized the development context of this thesis. We found a strong basis of work on both code layout optimizations and fetch architectures for superscalar processors, however, there was no work done on relating compiler optimizations and wide issue superscalar processors from either the compiler or the architecture point of view.

Our work fills this gap from both sides: we develop compiler optimizations conscious of the architecture, and propose a novel fetch architecture conscious of the compiler optimizations performed.

While it is difficult to evaluate the impact such a recent work, the field of code optimizations using runtime data, and dynamic optimizations has received a lot of attention lately. We have seen appear several proposals for optimizations frameworks which do the same work as the software trace cache, but dynamically.