7 FETCHING INSTRUCTION STREAMS

To this point we have optimized the application code so that it uses more effectively the underlying fetch architecture, and proposed modifications to the fetch architecture to adjust to the newly generated code. Along this path, we have gained a vast amount of information on the special characteristics of optimized applications.

However, we still rely on the existing fetch architectures for high performance, and such high performance architectures prove costly and complex to implement. However, there is little performance advantage in increasing front-end performance beyond what the back-end can consume. For each processor design, the target is to build the best possible fetch engine for the required performance level. A fetch engine will be better if it provides better performance, but also if it takes fewer resources, requires less chip area, or consumes less power.

In this chapter we propose a novel fetch architecture based on the execution of long streams of sequential instructions, taking maximum advantage of code layout optimizations. We describe our front-end architecture in detail, and show that it requires less complexity than other high performance fetch architectures, while providing important advantages such as a latency tolerant multiple branch predictor, and a high bandwidth front-end.

Our results show that our stream front-end engine increases performance by up to 27% over the same processor using a classic BTB+gshare front-end, and is only 7% slower than the same processor using a trace cache. Fetching instruction streams effectively exploits the special characteristics of layout optimized codes to provide a high fetch performance, close to that of a trace cache, but has a much lower cost and complexity, similar to that of a BTB architecture.

7.1 Introduction

As we have shown in Chapter 4, layout optimized codes exhibit some special characteristics which could be exploited to increase fetch performance, and reduce energy consumption: First, optimized codes execute long chains of sequential instructions, often containing multiple basic blocks,

and crossing multiple not taken branches. Second, they pack only useful instructions in a cache line, moving unused portions of code to other memory locations where they do not interfere. Finally, they map a branch and its target in close memory locations, often sharing the same cache line.

In this chapter we propose a fetch architecture designed to exploit these special characteristics. We take those long instructions sequences as the basic fetch unit, and call them streams. A stream is a chain of sequential instructions starting, at the target of a taken branch, and going on until the next taken branch.

We propose a branch predictor which provides stream level sequencing, based on the next trace predictor [34]. Each stream prediction provides information about a full sequential chain, potentially containing multiple not taken branches, and a single taken branch which terminates the stream. We store those predictions in a fetch queue, and use the information stored in that queue to drive the instruction cache, like proposed in [71].

In order to exploit the high spatial locality present in optimized codes, we design our instruction cache with long lines, which often contain a whole loop body, or both a branch and its target. We use a buffer to remember the last fetched cache line for as long as it is needed, which avoids unnecessary accesses to the instruction cache.

7.2 Fetching instruction streams

The stream front-end we propose in this chapter is based on the notion of an instruction stream. An instruction stream is a sequential run of instructions, from the target of a taken branch, to the next taken branch. It is possible for an instruction stream to contain multiple basic blocks, and multiple branches, as long as all the intermediate branches are not-taken.

As such, an instruction stream is fully identified by the starting instruction address and the stream length. Unlike traces, streams do not require information about the behavior of the branches contained in the stream, because it is implicit in the definition: all intermediate branches are not taken, and the terminating branch is always taken.

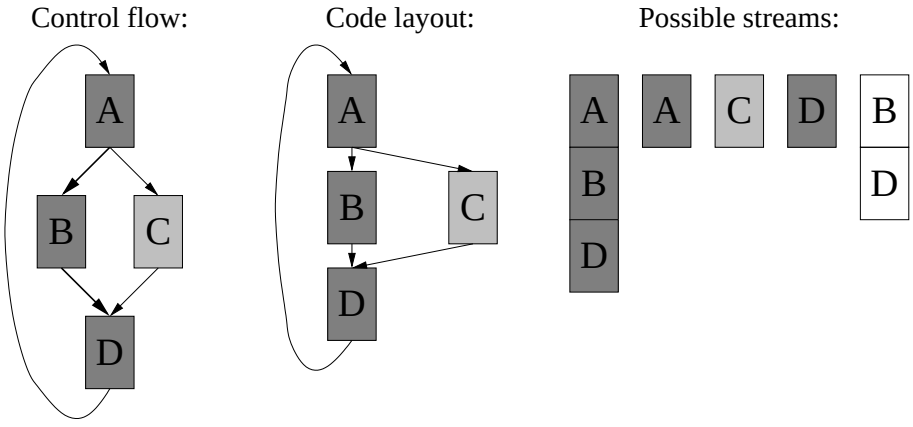


Figure 7.1. Example instruction streams. A stream is a sequential run of instructions.

Figure 7.1 shows an example control flow graph from which we will find the possible streams. The figure shows a loop containing an if-then-else structure. Our profile data shows that $A \rightarrow B \rightarrow D$ is the most frequently followed path through the loop. Using this information, we lay

out the code so that the path $A \rightarrow B$ goes through a not-taken branch, and the same for $B \rightarrow D$. Basic block C is mapped somewhere else, and can only be reached through a taken branch at the end of basic block A.

From the resulting code layout we may encounter four possible streams composed by basic blocks ABD, A, C, D. The first stream corresponds to the sequential path starting at basic block A and going through the frequent path found by our profile. Basic block A is the target of a taken branch, and the next taken branch is found at the end of basic block D. The infrequent case follows the taken branch at the end of A, goes through C, and jumps back into basic block D.

These are all the possible streams found as long as branch prediction is correct. If we account for branch mispredictions, there is a fifth possible stream which does not follow the definition above. Stream BD does not start at the target of a taken branch: it starts at the target of a branch misprediction. If the branch at the end of A is predicted taken, but it turns out to be not taken, execution proceeds from basic block B, but there is no full stream starting there.

To avoid this situation, we define a *partial* instruction stream as a stream which starts at the target of a branch misprediction, and goes on until the next taken branch.

The use of streams has several advantages over the use of traces. First, a stream is not defined by an arbitrary heuristic, it is a natural program entity (often called a dynamic block). However, it is still a dynamic program construct (like traces), and has to be detected by the processor before it is known by the architecture. Capturing a stream does not require a buffer to store the instructions, it simply requires a register to remember the address of the last taken branch target, and the ability to detect a taken branch. This provides the stream start an end addresses, and thus the stream length. To detect partial streams we only need an additional register to remember the last branch misprediction target.

However, the main advantage of instruction streams is that they reside in the instruction cache. They do not require additional instruction storage, nor a special purpose cache. Like traces, streams are sometimes redundant with each other. That is, two different streams may share a large fraction of their code, making them partially redundant. But this redundancy among streams does not waste any storage in the instruction cache, because the redundant parts of the streams naturally overlap.

The main drawback of fetching instruction streams instead of traces is that streams may not be long enough to provide sufficient fetch bandwidth, but this is not the case. Figure 7.2 shows the average length in instructions of a basic block, an instruction trace, and an instruction stream for the SPECint95 codes after the branch alignment optimization has been applied.

The traces collected have a limit of 16 instructions, no maximum number of branches, and may contain a single indirect branch which terminates the trace. This heuristic limits traces to a length under 16 instructions, which makes their average length of 14 instructions quite high. The trace length could be increased by relaxing the heuristics, and setting the maximum trace length to 32 or more instructions, but that would severely increase the trace cache miss rate, reducing or eliminating the benefits of longer traces. However, instruction streams have an average length of 16 instructions, and it can still be improved by applying more aggressive optimizations such as routine inlining and loop unrolling, which would not hurt severely the instruction cache miss rate. Overall, instruction streams are longer than traces, and thus, capable of providing a high fetch bandwidth to the rest of the processor.

Previous results in Section 4.2.2 show that a sequential fetch engine obtains a fetch performance similar to that of a trace cache when using layout optimized codes. The sequential engine

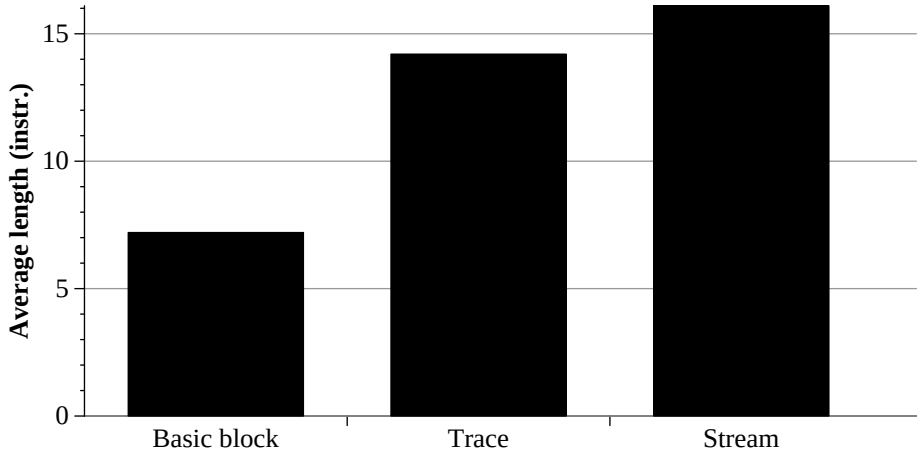


Figure 7.2. Average length of a basic block, an instruction trace, and an instruction stream. Streams are longer than traces in optimized codes.

used in that work is the SEQ.3 unit described in Chapter 2. However, such engine proves very complex to implement, and is still limited to 3 consecutive basic blocks per cycle. Next, we describe a fetch engine designed to fetch sequential code, up to a whole instruction stream per cycle, that overcomes the 3 basic block limitation without requiring more complexity than the classic BTB front-end engine.

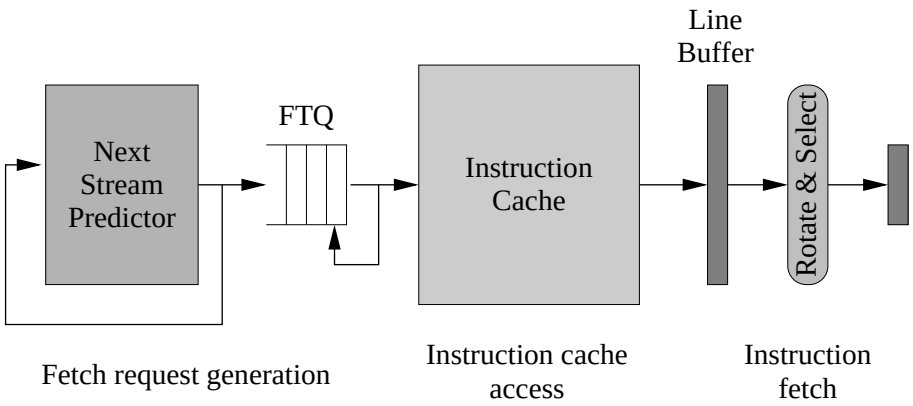


Figure 7.3. The proposed stream fetch engine. The instruction stream becomes the basic fetch entity.

Figure 7.3 shows the block diagram for the proposed stream fetch engine. The next stream predictor provides the fetch engine with stream level sequencing. That is, given the current stream starting address, it provides the current stream length, and the next stream starting address. The predicted next stream address is used as the fetch address for the next cycle. The current stream address, and the current stream length are stored in the fetch target queue (FTQ), and represent a fetch request for a full instruction stream.

The instruction cache is driven by the fetch requests stored in the FTQ. The stream starting address is used to access the instruction cache, which provides one or more consecutive cache lines. If the cache lines provided contain the whole stream, the FTQ is advanced to the next

request, if not, the fetch request is updated to reflect the remaining part of the stream to be fetched.

Not all instructions provided by the instruction cache need to be fetched in a single cycle. The cache lines provided are stored in a cache line buffer. Each cycle, the contents of the cache line is aligned, and valid instructions are selected up to the maximum fetch width of the processor. Once all the valid instructions in the buffer have been passed to the processor, new instructions are fetched from the instruction cache.

As described, the stream fetch engine can be run as a single pipeline stage, or can be divided in several shorter stages using the different intermediate structures (FTQ and cache line buffer) to separate them. In the following sections we describe the different fetch sub-stages in detail.

Stream prediction

The stream fetch engine we propose is driven by a branch predictor providing stream level sequencing. Such is the purpose of the *next stream predictor*. Given a stream starting address, the predictor provides a stream identifier and the next stream starting address. The stream identifier consists of the stream starting address and the stream length.

Figure 7.4 shows a possible implementation of the next stream predictor. Each table entry contains information about one stream: start address, length (bytes or instructions), terminating branch type (for return stack management), the next stream address, and a 2-bit saturating counter. The table is indexed using a hash of the current stream address and the previous stream starting addresses. For this work we have used the same DOLC scheme used for the next trace predictor [34].

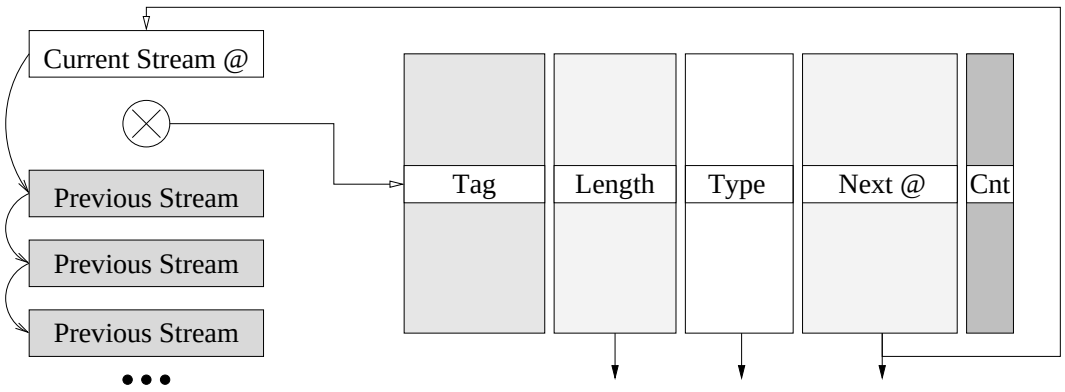


Figure 7.4. Path correlated implementation of the next stream predictor.

The predictor maintains two separate history registers: a lookup register which is updated speculatively, and an update register which is updated using correct path information only.

The confidence counter is used to implement the replacement policy. When a new stream is completed, the prediction tables are checked. If the stream is already there (the table is being updated with the same information already stored in that entry) the confidence counter is increased. If the new stream data and the data stored in the table do not match, the counter is decreased. If the counter reaches zero, the old data is replaced by the new data and the counter is set to one.

Each access to the predictor provides information about a whole instruction stream, possibly containing multiple basic blocks as long as they are connected by not taken branches. This is how the stream front end engine takes advantage of the characteristics of layout optimized codes: an average of 80% of all conditional branch instances are not taken. By stepping through the code

passing through not taken branches, we are able to fetch much longer code sequences than if we had to stop at all branches to predict them individually.

Fetch target queue

Following the proposal of Reinman, Austin and Calder [71] we have decoupled the branch prediction stage from the instruction cache access stage. The stream predictor provides information about an instruction stream with each prediction, the predictions are stored in a Fetch Target Queue (FTQ), and are used to drive the instruction cache access.

The average stream contains over 16 instructions, which means that the average fetch request stored in the FTQ is too large to be fetched in a single cycle. Instead of dividing a large fetch request into several smaller ones, we have implemented a fetch request update mechanism as shown in Figure 7.5.

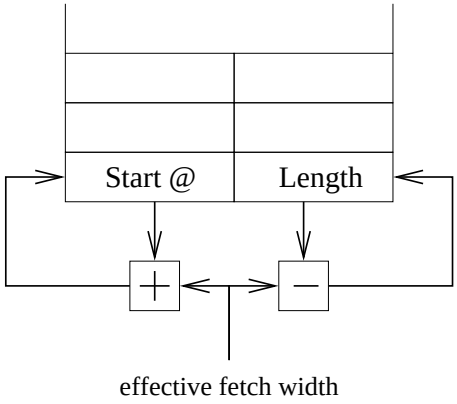


Figure 7.5. Fetch target queue update mechanism. A fetch request contains a whole stream and may take several cycles to be fetched.

Each fetch request contains the stream starting address and the stream length. The starting address is used to access the instruction cache and fetch one or more consecutive cache lines. Depending on how many cache lines were fetched, and the cache line width, a number of instructions belonging to the stream will be fetched.

The fetch request is updated using the actual number of instructions obtained from the instruction cache. The stream starting address is advanced, and the stream length is reduced appropriately. If the stream length reaches zero, then the fetch request has been satisfied, and the FTQ is advanced to the next request.

Instruction cache

Instruction streams are composed of sequential instructions. The easiest way to fetch an instruction stream is to read multiple consecutive cache lines from the instruction cache until the whole stream has been fetched. However, fetching a large number of cache lines in a single cycle is not always feasible, nor cost-effective.

The most simple fetch mechanism would be to fetch a single cache line per cycle. In this scenario we must face the problem of instruction misalignment, as shown in Figure 7.6.

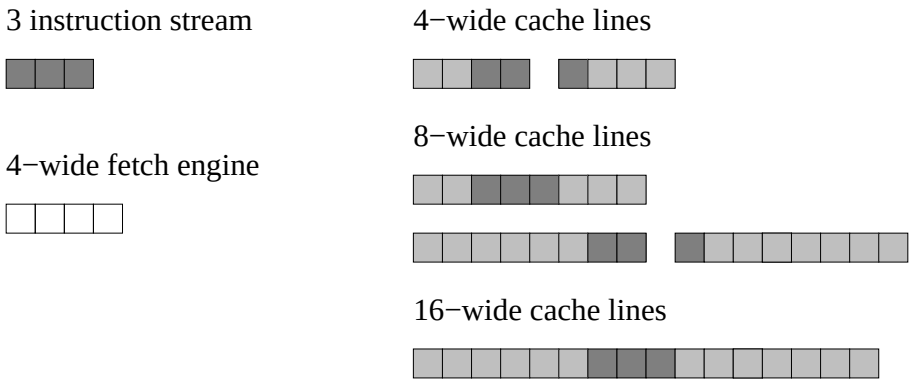


Figure 7.6. The instruction misalignment problem. A 3 instruction stream may take more than one cycle to fetch on a 4-wide engine if the instructions cross the cache line boundary.

A fetch request consisting of 3 consecutive instructions should be fetched in a single cycle by a 4-wide fetch engine, but such is not always the case. It is possible that the 3 instruction stream is split across two separate cache lines, which means that it will take two cycles to fetch if we fetch a single cache line per cycle.

The use of long instruction cache lines alleviates this problem. A longer cache line reduces the possibilities of the instruction stream crossing the cache line boundary.

Section 4.2.1 shows that layout optimized codes benefit from long cache lines more than unoptimized codes due to a denser packing of useful instructions to cache lines. In particular, these results show that a 128-byte line (32 instructions) is usually fully used before being replaced (Figure 4.8).

Considering both benefits together (the reduced stream misalignment, and the instruction cache miss rate benefits), we have adopted a simple instruction cache design which reads a single line per cycle, but we use a very long line size.

Cache line buffer

The instruction cache is the only source of instructions in our stream fetch architecture. In order to minimize the problem of stream misalignment, and to obtain maximum benefit of the code layout optimizations, we are using very long instruction cache lines (4 times wider than the actual fetch width).

However, fetching a cache line wider than the actual fetch width of the processor represents a wasted number of instructions transferred, because not all the instructions in the cache line will be actually fetched. It is frequent to fetch the same cache line in several consecutive cycles.

In order to avoid this redundant instruction cache access, we consider the use of an instruction cache line buffer which remembers the last cache line fetched. The cache line buffer mechanism is shown in Figure 7.7. We require the presence of a simple logic which reads the current fetch address from the FTQ, and calculates the next fetch address looking at the start address for the next FTQ entry if the current length is less than the fetch width. Using this information (current fetch address, and next fetch address) it is possible to calculate which cache line will be required the next cycle. If the same cache line will be used in two consecutive cycles, the buffer contents is reused and the instruction cache is not accessed.

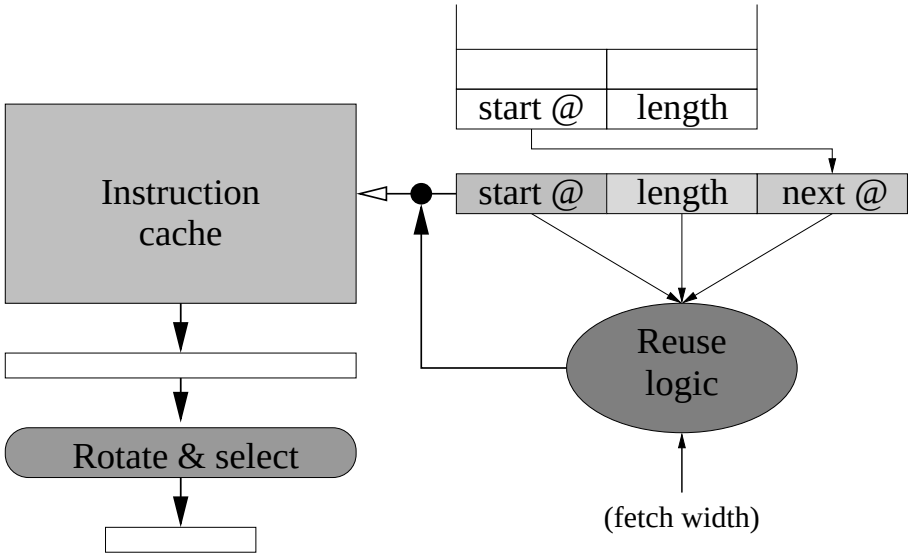


Figure 7.7. Cache line reuse buffer mechanism. The line buffer reduces instruction cache activity and exploits longer cache lines.

This optimization not only reduces instruction cache activity when a cache line will be used several times to read a long instruction stream. If a whole loop body is contained in a single cache line, it is possible to fetch all the instructions for all the loop iterations from the line buffer, without having to access the instruction cache.

The actual instruction fetch proceeds from the contents of the line buffer. The instructions are aligned using the starting address, and valid instructions are selected up to the maximum fetch width (or the remaining instructions in the buffer). The actual number of instructions fetched is then used to update the FTQ entry, or to advance the queue if the request was completely fetched.

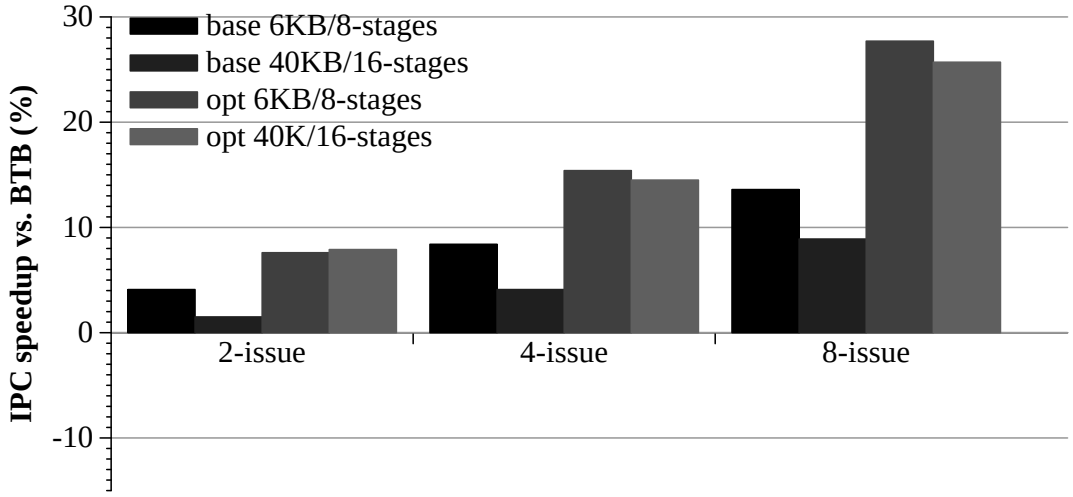
7.3 Performance evaluation

In Section 7.2 we have described the stream fetch architecture: a low cost, low complexity fetch engine designed to exploit the characteristics of optimized code layouts. As we already mentioned in the introduction, a fetch engine is better than others if it has a lower cost, requires fewer resources, or has a lower energy consumption. However, raw performance is still a very important metric when designing a high performance processor.

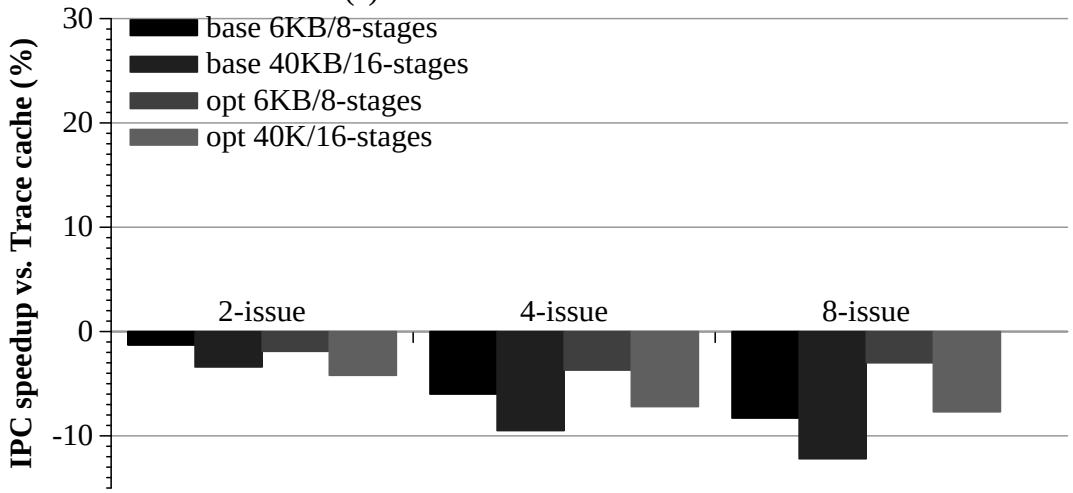
Figure 7.8 compares the processor performance obtained using (a) a BTB architecture and (b) a trace cache architecture with the stream fetch architecture. The graph shows the IPC speedup or slowdown (percent) of the stream fetch architecture compared to the BTB and the trace cache.

The results show that fetching instruction streams provides significant performance improvements over the classic BTB fetch engine, specially for high issue widths where fetch performance becomes more important. On an 8 issue processor, the BTB proves unable to provide enough instructions per cycle to keep all functional units busy, given that a single basic block does not contain enough instructions. The stream front-end is able to provide instructions from multiple basic blocks per cycle, and overcomes this limitation.

The trace cache also overcomes this limitation, and is able to provide instructions from basic



(a) Stream fetch vs. BTB fetch



(b) Stream fetch vs. Trace cache

Figure 7.8. IPC speedup of the stream fetch engine compared to a BTB fetch engine and IPC slowdown compared to the trace cache.

blocks separated by a taken branch, which is beyond the abilities of the stream front-end. This provides the trace cache with an extra degree of flexibility which translates into a slightly improved performance.

These results show that when using optimized codes, the stream fetch architecture, which has a cost and complexity similar to the BTB architecture obtains performance close to that of the trace cache, a much more complex and costly solution.

When using unoptimized codes, the stream fetch architecture experiences a significant performance loss due to the presence of many taken branches, which limit the effective fetch width. This loss makes the stream fetch architecture 12% slower than the trace cache architecture. However, even in the absence of optimized codes, the stream fetch architecture is still 14% faster than the BTB architecture. Furthermore, dynamic code optimization infrastructures such as the one proposed by Merten *et al.* [45] or Dynamo [2] can ensure that the code layout has been optimized, and that the stream fetch architecture can obtain the maximum fetch performance.

Table 7.1 shows other performance metrics which contribute to the fetch engine performance: the actual fetch width on the 8-wide processor, and the branch prediction accuracy measured as the number of instructions fetched between two consecutive mispredictions (IPM). The instruction cache miss rate is very low in all three architectures, and hardly contributes to the IPC performance difference.

Fetch engine	fetch width	IPM 6KB	IPM 40KB
BTB	4.5	104	137
Stream	6.0	124	163
Trace	6.6	115	145

Table 7.1. Fetch performance metrics: actual fetch width and branch prediction accuracy.

The results show that the main advantage of the stream and trace cache architectures over the BTB is the fetch width increase, obtained fetching instructions from multiple basic blocks per cycle. This fetch width increase means that the stream front-end provides 33% more instructions per fetch than a BTB architecture, and 10% fewer instructions than a trace cache architecture.

However, fetching more instructions per cycle does not mean higher IPC, unless the instructions are fetched from the correct path. The number of instructions per misprediction (IPM) shows how often we mispredict a branch (any kind of branch). Our results show that changing the branch prediction scope from individual branches (individual basic blocks) to instruction traces or instruction streams can increase branch prediction accuracy, providing 5–20% more instructions between mispredictions than an BTB+gshare pair, with the stream predictor begin slightly better than the trace predictor.

The combination of increased fetch width and improved branch prediction accuracy explain the IPC improvements obtained fetching instruction streams. The fact that the trace cache provides only 10% more instructions per fetch than the stream architecture, and that the trace predictor is slightly less accurate than the stream predictor show why the stream architecture comes so close to the trace cache performance.

Branch predictor latency

The core of the stream fetch architecture is the next stream predictor, which replaces the classic BTB + two-level predictor pair and provides stream level sequencing.

Given that each stream potentially contains multiple basic blocks, each stream prediction is effectively predicting multiple branches: several not taken branches, and one taken branch which terminates the stream. This increased information density makes the stream predictor more latency tolerant: each stream prediction contains a whole stream and generates a fetch request for the whole stream, but it is likely that fetching that stream will take several cycles. That means that it is not necessary to provide a new stream prediction each cycle. A single stream prediction will keep the processor busy for several cycles. However, the same is not true for the BTB+gshare pair: each prediction contains information about a single basic block, which can usually be fetched in a single cycle.

This section does not show results for the next trace predictor nor the trace cache. Given that the next trace predictor is also a multiple branch predictor, all the conclusions drawn for the next stream predictor are also applicable to the next trace predictor, except that the trace predictor still needs a BTB as a support predictor.

Table 7.2 shows the performance degradation (percent) experienced by the BTB and stream fetch architectures when the branch predictor latency is increased from 1 to 2 cycles. The branch predictor is not pipelined. That is, a 2-cycle predictor provides one prediction every two cycles, not one prediction per cycle with a longer startup time.

Arch.	Pred. size	2-issue	4-issue	8-issue
BTB	6KB	4.0	9.2	21.4
BTB	40KB	1.9	7.5	21.5
Stream	6KB	3.5	4.3	8.2
Stream	40KB	2.2	3.3	7.3

Table 7.2. IPC degradation (%) of the BTB and stream fetch architectures when the latency of the branch predictor increases to 2 cycles.

Our results show that as long as a single basic block takes several cycles to fetch, there is little performance degradation in both architectures. But, as the fetch width increases, more basic blocks can be fetched in a single cycle, the BTB fetch engine runs out of branch predictor information, and stalls losing up to 20% performance. However, streams are much longer than basic blocks, and take longer to fetch. This allows the stream fetch architecture to experience a much lower performance degradation, which is mostly due to the increased pipeline depth than to lack of branch predictor information.

This latency tolerance of the stream predictor allows the use of a larger and more accurate, but slower predictor. The slower predictor leads to a small slowdown, while the increased prediction accuracy may increase performance beyond the slowdown introduced resulting in an overall performance increase.

The latency tolerance of the stream predictor can also lead to a single ported implementation of the branch predictor. Previous simulation results assume the existence of multiple ports to the branch prediction tables: a lookup port, used in the fetch engine; and several update ports, used in the commit stage. A single port implementation implies that the we can only obtain a

branch predictor, or update the prediction tables once, but not both at the same time, nor update the predictor multiple times per cycle.

Table 7.3 shows the performance degradations (percent) experienced by the BTB and stream fetch architectures when the number of ports to the branch predictor is limited to just one. When both an update and a lookup happen in the same cycle, the update has preference and the lookup has to wait until the next cycle.

Arch.	Pred. size	2-issue	4-issue	8-issue
BTB	6KB	2.5	13.1	17.6
BTB	40KB	1.8	9.0	13.1
Stream	6KB	1.1	2.6	4.8
Stream	40KB	0.8	1.7	3.6

Table 7.3. IPC degradation (%) of the BTB and stream fetch architectures as the number of ports to the branch predictor is limited to just one.

Our results show that for narrow width processors, a single ported branch predictor is not a problem. However, as width increases it is frequent for a branch to be committed every cycle, which keeps the branch predictor port busy, preventing the fetch engine from working at full potential. This problem is visible for the BTB+gshare case, where each committed branch has to update the two-level predictor (only taken branches update the BTB).

Meanwhile, the stream predictor presents a very small performance loss, because there is no need to obtain a new branch prediction every cycle, plus not all branches update the predictor. Only taken branches terminate a stream, which means that the stream predictor only is updated once for each taken branch. Requiring a new prediction the very same cycle that a taken branch is graduating is very infrequent.

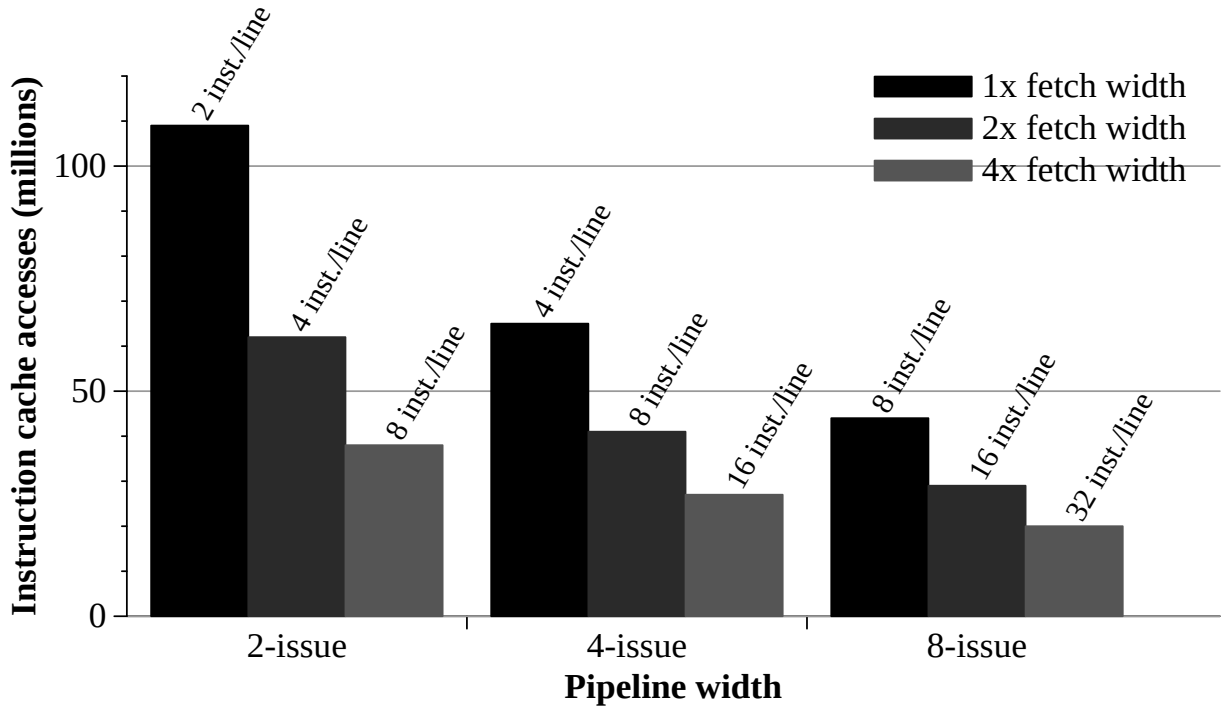
The results presented to this point show that the next stream predictor is more accurate than a BTB+gshare pair with a similar storage capacity. The results presented in this section show that the next stream predictor tolerates latencies much better than a BTB+gshare pair, which allows the use of larger predictors; and can be implemented with a single lookup/update port, making it smaller and faster.

Front-end activity

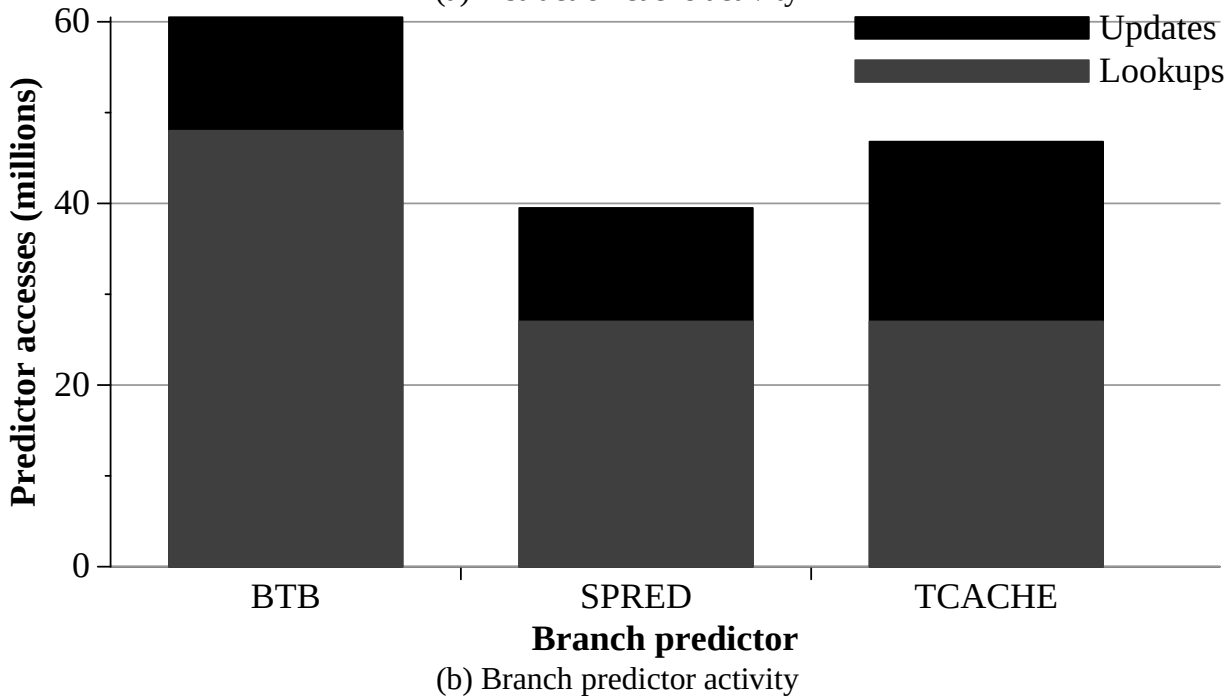
The stream fetch architecture that we propose uses long instruction cache lines for two separate reasons: first, because long cache lines reduce the stream misalignment problem (see Section 7.2); second, because previous work shows that optimized code layouts make efficient use of long cache lines, reading all instructions in the line before it is being replaced (see Section 4.2.1).

To avoid reading the same cache line in several consecutive cycles we introduce a buffer which holds the last cache line read. If the same cache line is effectively being used in several consecutive cycles, this buffer reduces the instruction cache activity. Figure 7.9.a shows the average number of instruction cache accesses required for different line sizes and pipeline widths. These results show the effectiveness of the cache line buffer, and are valid for both the stream and BTB architectures.

Our results show that doubling the line size for a given pipeline width almost halves the number of instruction cache accesses required. For example, a 4-wide fetch engine requires 40% fewer



(a) Instruction cache activity



(b) Branch predictor activity

Figure 7.9. Activity of the instruction cache and branch predictor for the BTB, stream, and trace cache architectures.

cache line reads when using 8 instructions per line (32 byte cache lines) than when using 4 instructions per line (16 byte cache lines). This result shows that we are effectively exploiting the high locality of optimized code layouts.

A second interesting observation is that the use of the cache line buffer makes the instruction cache activity depend on the cache line width, not on the fetch width. The number of accesses required by a 32 byte cache line is the same for all three fetch widths. The same can be observed for other cache line widths.

In Chapter 4 we show that optimized code layouts execute long sequences of sequential instructions, that is, they execute long instruction streams. The stream fetch architecture proposed uses the next stream predictor to provide stream level sequencing. If each instruction stream contains multiple basic blocks, each stream prediction is providing multiple branch predictions, which reduces the required branch predictor activity. Figure 7.9.b shows the number of branch predictor accesses (both lookups and updates) required by each fetch architecture.

The BTB+gshare pair require a lookup access for each executed basic block (ignoring branches which have never been taken yet), a BTB update for each taken branch, and a gshare update for each committed branch; the trace predictor requires one lookup and one update for each executed trace, plus several lookups to the support BTB in the event of a branch misprediction and an update for each executed branch (this BTB also contains a 2-bit saturating counter for direction prediction); meanwhile, the stream predictor requires a lookup for each stream executed, and an update for each stream committed (that is, one update for each taken branch).

Our results show that the stream predictor requires fewer accesses than any of the other two examined architectures: 40% fewer accesses than the BTB+gshare pair, and 24% fewer accesses than the trace cache and its support BTB. This is another example of how the stream fetch architecture exploits the characteristics of optimized codes.

7.4 Conclusions

We have presented the stream fetch architecture. A fetch architecture designed to exploit the characteristics of optimized code layouts: the execution of long streams of sequential instructions, and the dense packaging of useful instructions to cache lines.

The stream fetch architecture has a complexity similar to that of a BTB architecture, requiring a single source for instructions and a single branch prediction mechanism. However, our results show that the stream fetch architecture provides performance close to that of a trace cache, a much more complex and costly architecture.

A fetch engine will be better if it provides higher performance, but also if it takes fewer resources, requires less chip area, or consumes less power. The stream fetch performance provides higher performance than the BTB architecture, requires fewer resources and chip area than the trace cache architecture, and consumes less energy than any of the two other examined architectures.

The stream fetch architecture is a high performance, low complexity architecture which directly benefits from the characteristics of optimized codes.