
Contents

1	Instruction Cache Prefetching	3
	<i>Glenn Reinman</i> UCLA Computer Science Department	
1.1	Introduction	3
1.2	Scaling Trends	3
1.3	Instruction Cache Design	6
1.3.1	Cache Choices	7
1.4	Instruction Cache Prefetching	10
1.4.1	Next Line Prefetching	10
1.4.2	Target Prefetching	11
1.4.3	Stream Buffers	11
1.4.4	Nonblocking Instruction Caches and Out Of Order Fetch 14	
1.4.5	Fetch Directed Instruction Prefetching	15
1.4.6	Integrated Prefetching	19
1.4.7	Wrong-Path Prefetching	20
1.4.8	Compiler Strategies	22
1.5	Future Challenges	22
	References	25

Chapter 1

Instruction Cache Prefetching

Glenn Reinman

UCLA Computer Science Department

1.1 Introduction	3
1.2 Scaling Trends	3
1.3 Instruction Cache Design	6
1.4 Instruction Cache Prefetching	10
1.5 Future Challenges	22

1.1 Introduction

Instruction delivery is a critical component of modern microprocessors. There is a fundamental producer/consumer relationship that exists between the instruction delivery mechanism and the execution core of a processor. A processor can only execute instructions as fast as the instruction delivery mechanism can supply them.

Instruction delivery is complicated by the presence of control instructions and by the latency and achievable bandwidth of instruction memory. This Chapter will focus on the latter problem, exploring techniques to hide instruction memory latency with aggressive instruction cache prefetching.

1.2 Scaling Trends

Interconnect is expected to scale poorly due to the impact of resistive parasitics and parasitic capacitance. The resistance of wire is proportional to the cross sectional area of the wire (and therefore the width and thickness of the wire). Wire width must scale with the feature size. Therefore, the thickness of the wire may not be able to scale proportionally to the width due to the increased resistance that would result from the smaller cross sectional area. Electromigration, or ion transport, could also result if wires become too thin. Electromigration is the physical breakdown of the wire itself, and is therefore a major concern for processor reliability. Finally, thinner wires are

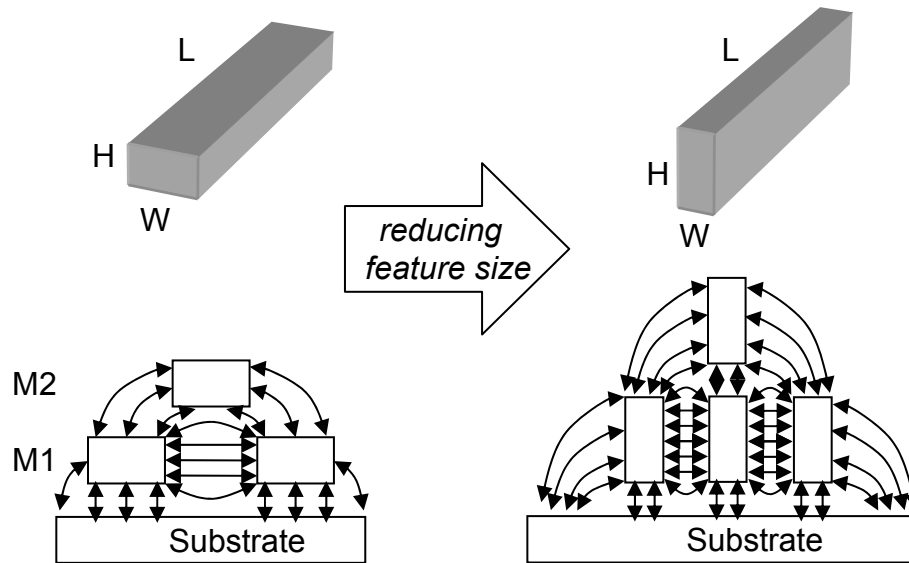


FIGURE 1.1: Interconnect Scaling: This diagram explores the scaling of a 3-D model of wire (top) to future feature sizes. As feature sizes shrink, wire height increases to reduce resistive parasitics, maintaining the same cross sectional area despite shrinking wire widths. W is wire width, T is wire thickness or height, and L is wire length. However, this impacts the interaction of wires and the substrate (shown at bottom). Capacitive parasitics increase, as demonstrated by the two metal layers $M1$ and $M2$ on a substrate. Straight lines represent parallel plate capacitance, curved lines represent fringing capacitance.

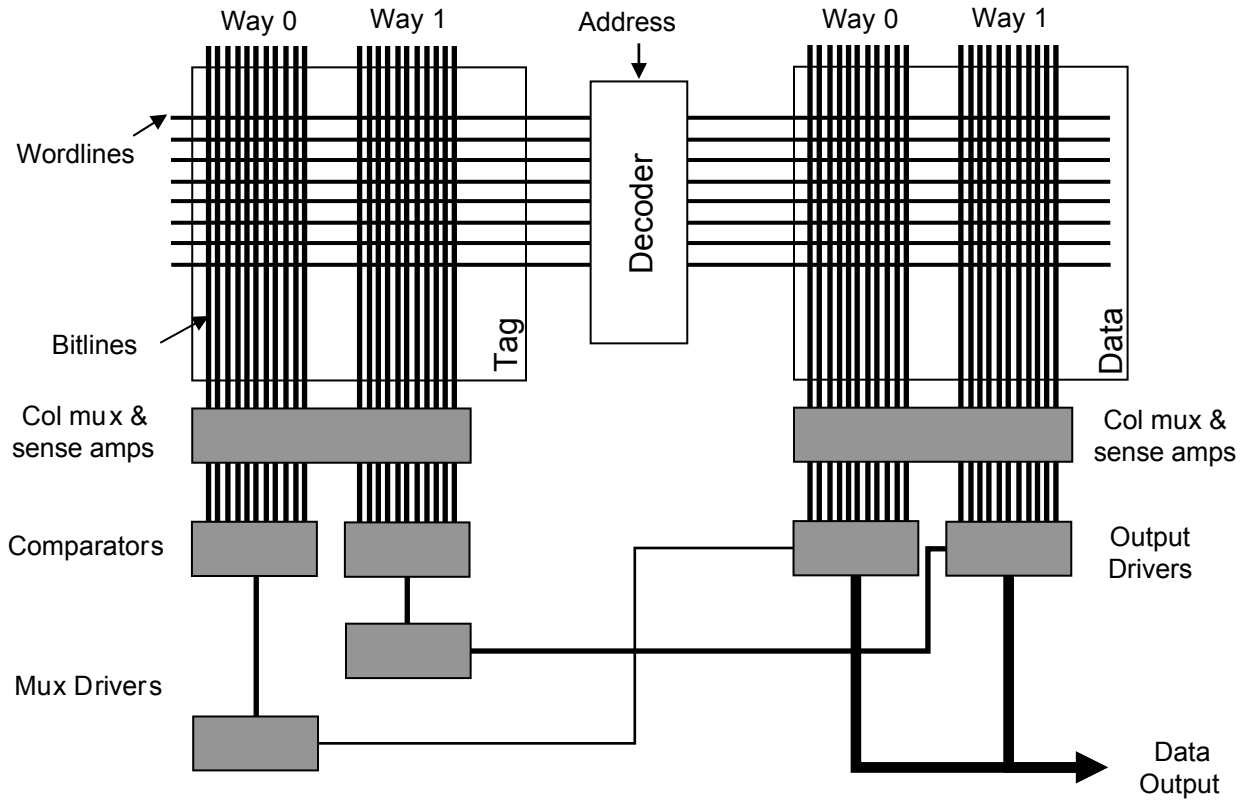


FIGURE 1.2: A Set Associative Cache Model

more difficult to manufacture.

Wire capacitance has two components that compose the overall parasitic: parallel plate capacitance and fringing capacitance. Parallel plate capacitance is the capacitance between the wire and the substrate and is proportional to $\frac{W}{H}$, where W is the wire width and H is the distance between the wire and the substrate (see Figure 1.1). Fringing capacitance is the capacitance between the side-walls of the wires and the substrate and is proportional to $\frac{T}{H}$, where T is the wire thickness and H is the distance between the wire and the substrate. Parasitic capacitance exists between wires and the substrate and between wires themselves. This latter component, known as coupling capacitance, also has both a parallel plate component and a fringing component, and is proportional to the distance between wires on the same layer or in different layers. As this distance shrinks, the coupling capacitance increases. As W decreases in size, the fringing capacitance component begins to dominate the overall capacitance.

The latency of a memory device (illustrated in Figure 1.2), to a first order,

is the latency to exercise the logic in the decoder, assert the wordline wire, read the memory cell logic, assert the bitline wire, and finally exercise the logic in the bitline multiplexor to select the accessed data. As the process feature size is scaled, the latency of the transistors is scaled proportional to their size, thus the latency of the logic scales linearly with feature size reductions.

The latency of the wordlines and bitlines, on the other hand, does not scale as well due to parasitic capacitance effects that occur between the closely packed wires that form these buses. As the technology is scaled to smaller feature sizes, the thickness of the wires does not scale. As a result, the parasitic capacitance formed between wires remains fixed in the new process technology (assuming wire length and spacing are scaled similarly).

Since wire delay is proportional to its capacitance, signal propagation delay over the scaled wire remains fixed even as its length and width are scaled. This effect is what creates the interconnect scaling bottleneck. Since on-chip memory tends to be very wire congested (wordlines and bitlines are run to each memory cell), the wires in the array are narrowly spaced to minimize the size of the array. As a result, these wires are subject to significant parasitic capacitance effects. Agarwal et al. [1] conclude that architectures which require larger components will scale more poorly than those with smaller components. They further conclude that larger caches may need to pay substantial delay penalties.

1.3 Instruction Cache Design

Figure 1.3 illustrates a typical instruction delivery mechanism. A coupled branch predictor and instruction cache are accessed in parallel with the address contained in the program counter (PC). The PC stores a pointer to the next instruction in memory that is to be executed. The program itself is loaded into a portion of memory, and the PC represents the memory address where a given instruction is stored.

The instruction cache stores a subset of the instructions in memory, dynamically swapping lines of instructions in and out of the cache to match the access patterns of the application. The size of the lines that are swapped in and out of the cache depend on the *line size* of the cache. Larger line sizes can exploit more spatial locality in instruction memory references, but consume more memory bandwidth on cache misses. The line size, along with the associativity and number of sets of the cache influences the size, latency, and energy consumption of the cache on each access.

The more instruction addresses that hit in the instruction cache, the more memory latency that can be hidden. However, architects must balance this against the latency and energy dissipation of the cache. The latency of the

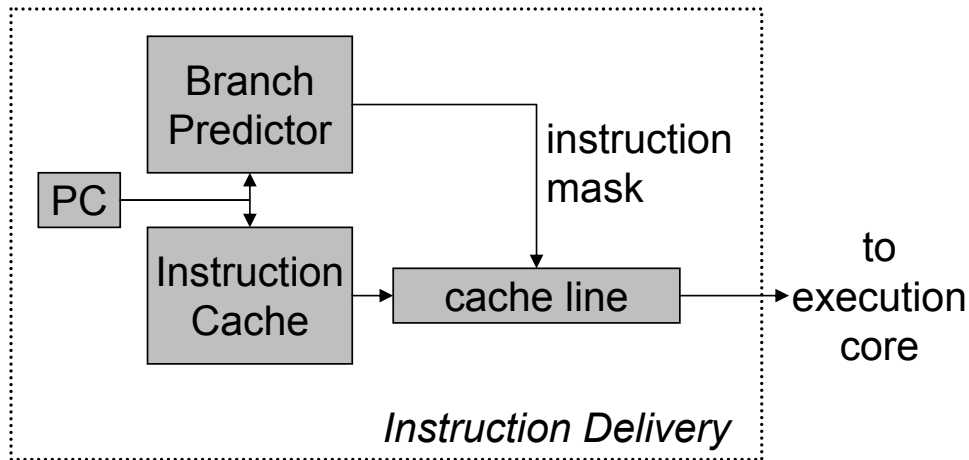


FIGURE 1.3: Instruction Delivery Mechanism: High level view of the structures of the front-end.

instruction cache access impacts the branch misprediction pipeline loop of the processor. Therefore, the processor must be designed to make the most efficient use of the available cache space to maximize latency reduction.

Unlike data cache misses, where aggressive instruction scheduling and large instruction windows can help hide memory access latency, instruction cache misses are more difficult to tolerate. Instructions are typically renamed in program order, which means that a single instruction cache miss can stall the pipeline even when subsequent accesses hit in the cache.

Typically the line size is selected based on the target bandwidth required to feed the execution core. An entire cache line is driven out, and a contiguous sequence of instructions are selected from this line based on branch prediction information and the starting PC.

1.3.1 Cache Choices

Figure 1.3 demonstrates some alternatives for instruction cache design. These alternatives differ in the latency and energy efficient of the cache, and in the complexity of the implementation.

1.3.1.1 Direct Mapped Cache

In a direct mapped cache, any given address maps to only one location in the cache. This cache can be impacted more severely by conflict misses, but is fast to access and is energy efficient. Each address can only map to one location in the cache. On an access, the data component drives out the cache line in that one location and the tag component determines whether or not there is a match.

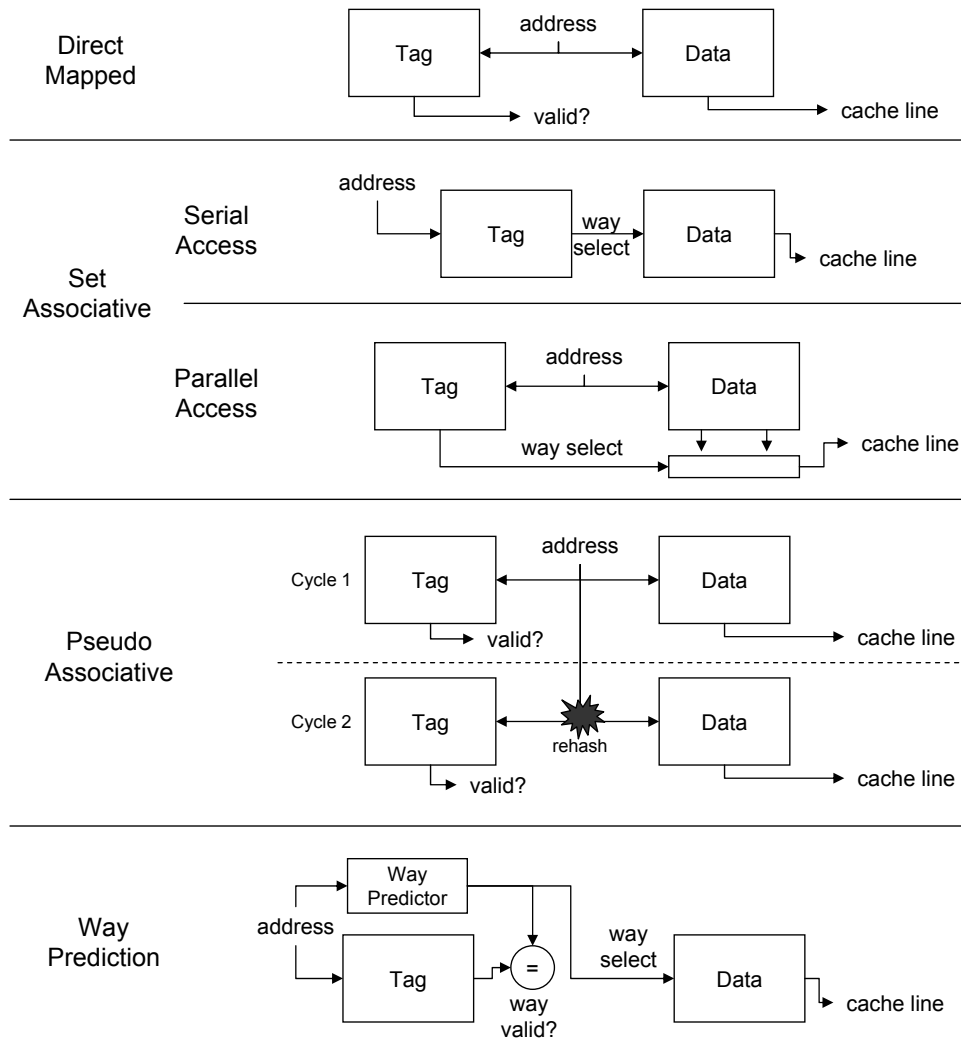


FIGURE 1.4: Different instruction cache designs, trading hit coverage, timing, area, energy, and complexity.

1.3.1.2 Set Associative Cache

In a set associative cache, any given address can map to more than one location in the cache. This cache has better tolerance of conflict misses, but can take longer to access than the direct mapped cache. There are two main types of set associative caches.

1.3.1.2.1 Serial Access A serial access cache does not overlap the tag and data component accesses as in the direct mapped cache and therefore will have longer latency. But the energy efficiency is close to that of a direct mapped cache since at most one line will be driven out of the data component per access. In fact, it may even provide an energy benefit over the direct mapped cache since there must be a cache hit for a cache line to be driven out. This benefit would depend on how often the cache misses and how much of an energy impact the associative tag comparison has.

1.3.1.2.2 Parallel Access A parallel access cache approaches the latency benefits of the direct mapped cache, as the latency of this cache is the maximum of the tag and data component latency, plus some delay for the output driver. However this cache can use dramatically more energy than the direct mapped cache. On a given access to an n -way set associative cache, n different cache lines will be driven to the data output part of the cache until the tag component can select one.

1.3.1.3 Pseudo Associative Cache

One approach to providing set associative performance with the energy and timing benefits of a direct mapped cache is the pseudo associative cache. This direct mapped cache features multiple hash functions. For example, consider a cache with two hash functions. On a cache access, if the first hash function sees a cache miss, the second hash function is used to initiate a second access in the next cycle. When the first hash function hits, the access only takes a single cycle. But when the second hash function is used, the access takes two cycle and subsequent accesses must be stalled for one cycle. The challenge of pseudo associative caches is reducing the number of times that there is a miss on the first access. However, such a cache can often compare favorably to a multicycle set associative cache. The energy impact of multiple cache accesses can often be tempered by the reduction in per access energy.

1.3.1.4 Way Prediction Cache

This cache approaches the problem of finding a middle ground between direct mapped timing and energy benefits with set associative performance in a different way. This approach starts with a set associative cache, but uses a small predictor called a way predictor to guess in what way the desired cache line will be found. The tag component is accessed in parallel with the way

predictor to verify the guess. In order for the design to make sense, the way predictor time must be much less than the tag component, to parallelize the tag and data component accesses as much as possible. If the way prediction is correct, only a single cache line will be driven out of the data component - providing the energy benefits of the serial access set associative cache with the timing benefits of the parallel access set associative cache. However, if the way prediction is wrong, the correct cache line will be driven in the following cycle based on the tag access. In this case, the latency of the access is no worse than the serial access set associative cache, but the energy dissipation is slightly worse since two cache lines are driven on a way misprediction. The way predictor must be designed to be fast enough to parallelize the access time to the cache components, small enough to avoid contributing to energy dissipation, but large enough to provide reasonable prediction accuracy.

1.4 Instruction Cache Prefetching

Without any form of prefetching, cache lines are only brought in on demand. These types of misses are known as *demand misses*. The challenge of prefetching is to hide the latency of a future cache miss as much as possible before the miss occurs, but without impacting demand misses. The goal is to bring the *right* line into the cache *before* it is needed. Prefetch schemes can be evaluated based on their accuracy and their timeliness. Prefetch accuracy is critical: it is a waste of memory bandwidth and cache space to bring in lines that will not be used before their eviction from the cache. Timeliness is also key: if a cache line is prefetched only a few cycles before that line is to be used, there is little latency hidden by the prefetch. If a prefetch is started too far in advance, the line may be evicted before it is even used. The following techniques have been proposed to guide instruction cache prefetch or tolerate instruction cache latency.

To better evaluate the following approaches, Figure 1.5 demonstrates a sample instruction address stream. The stream starts at address 992 and continues beyond address 800. For the stream of addresses, assume that 992 and 576 are the only addresses that begin in the instruction cache – the remaining addresses in the stream will be instruction cache misses.

1.4.1 Next Line Prefetching

The most natural instruction cache prefetching strategy is to simply fetch the next consecutive cache line from memory. For example, if the instruction cache line size is 32 bytes, and if the processor is currently fetching cache line address x from the instruction cache, then line $x+32$ would be prefetched.

992, 512, 544, 576, 352, 384, 416, 768, 800, ...

TIME \longrightarrow

FIGURE 1.5: Sample Instruction Address Stream: This sample stream of addresses will be used in subsequent Figures. The stream starts with instruction memory address 992

To limit the number of prefetches, Smith [12] suggested that each instruction cache line be augmented with a single bit to indicate whether or not the next consecutive line should be prefetched. On an instruction cache miss, the bit for the line that missed would be set. On a cache access, if the bit is set for the line that is read from the cache, the next sequential cache line would be prefetched from memory, and the bit for the current line would be cleared.

For the sample instruction stream shown in Figure 1.5, next line prefetching would capture the misses at 544, 384, 416, and 800.

1.4.2 Target Prefetching

Instruction memory references are not always contiguous, and control instructions can reduce the accuracy of next line prefetching. Hsu and Smith [4] enhanced next line prefetching with target prefetching. They used a line target prediction table to guide what cache line typically follows the current cache line. If the table predicts a next line, that line can then be prefetched while the current cache line is fetched. Target and next line prefetching can work together, either by selecting one technique for a given cache line or by using both approaches together on the same cache line.

As observed by [4], next line prefetching exploits the same spatial locality that makes larger line sizes attractive for instruction caches. Together with target prefetching, this relatively simple approach can provide high accuracy. However, recent demands for fetch bandwidth and the growing latency to memory can impact the timeliness of these approaches, as the next line may be required in the next cycle of fetch.

With correct branch prediction, Target and Next line prefetching would be able to start a prefetch for all of the misses in the sample instruction stream of Figure 1.5, but would likely start these prefetches too late to hide the majority of the memory latency.

1.4.3 Stream Buffers

Jouppi [5] originally proposed the use of stream buffers to improve the performance of direct mapped caches. On a cache miss, a stream of sequential cache lines, starting with the line that missed, are prefetched into a small, FIFO queue (the stream buffer). The prefetch requests are overlapped, so the

latency of the first line that misses can hide the latency of any subsequent misses in the stream. Subsequent work has considered fully associative stream buffers [3], non-unit stride prefetches of sequential cache lines [8], and filtering unnecessary prefetches that already exist in the instruction cache [8]. This latter enhancement is critical in reducing the number of prefetches generated by stream buffers. Idle cache ports or replicated tag arrays can be used to perform this filtering. A redundant prefetch (one that already exists in the instruction cache) not only represents a wasted opportunity to prefetch something useful, but also represents wasted memory bandwidth that could have been used to satisfy a demand miss from the instruction or data cache, or even a data cache prefetch. Multiple stream buffers can be coordinated together and probed parallel with the instruction cache for cache line hits.

With a conventional stream buffer approach, contiguous addresses will be prefetched until the buffer fills. The buffer must then stall until one of the cache lines in the buffer is brought into the cache (on a stream buffer hit) or until the buffer itself is reallocated to another miss. Confidence counters associated with each buffer can be used to guide when a buffer should be reallocated. One policy might be to use one two-bit counter with each stream buffer. On a stream buffer hit, the counter for that buffer is incremented. When all stream buffers miss, the counters for all buffers is decremented. On a cache miss (and when all stream buffers miss), a stream buffer is selected for replacement if that counter is currently 00. The counters would not overflow or underflow. This policy would allow stream buffers that are successfully capturing the cache miss pattern of the incoming instruction address to continue prefetching cache lines, and would deallocate buffers that are no longer prefetching productive streams. Similar policies can also be used to guide what stream buffer should be allowed access to a shared memory or L2 port for the next prefetch request.

Figure 1.6 demonstrates the stream buffer in action on our instruction stream example. The PC, instruction cache, result fetch queue, and four stream buffer entries are shown (nothing is drawn to scale). The *V* field in the stream buffer indicates that the entry is valid and is tracking an in-flight cache line. The *R* field in the stream buffer indicates that the entry is ready – that it has arrived from the other levels of the memory hierarchy.

Cycle *k* sees a successful cache access for address 992. In cycle *k*+1, the instruction cache misses on address 512, stalling fetch. A stream buffer is allocated for the miss. Each cycle, the next contiguous cache line in memory is brought into the cache. Assuming that the stream buffer uses the filtering approach of [8], address 576 will not be brought into the stream buffer since it already exists in the instruction cache. Eventually, the stream buffer prefetches address 608, which is not referenced in the example in Figure 1.5. In this example, the stream buffer would fill with addresses that are not referenced in the near future, and assuming only a single stream buffer, this buffer would likely be reallocated to a new miss stream. However, there may be some benefit to these prefetches even if the buffer is reallocated, as they may

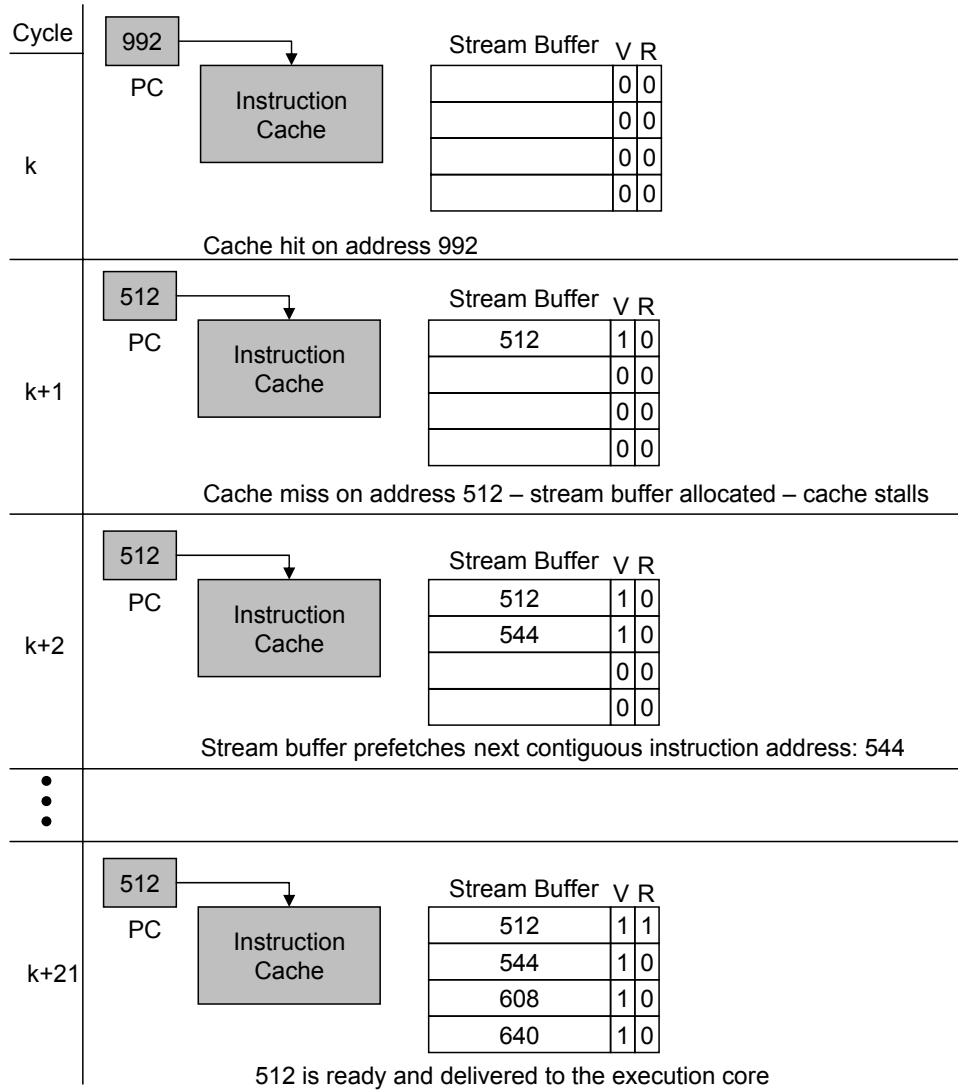


FIGURE 1.6: The stream buffer in action on the first part of the example from Figure 1.5.

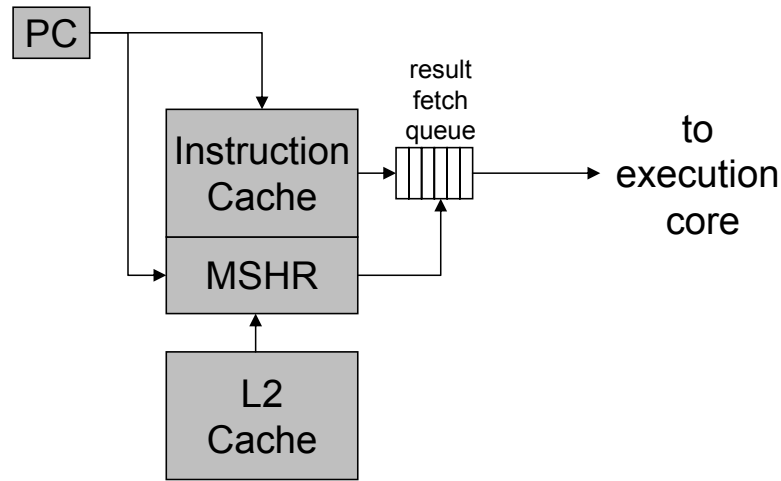


FIGURE 1.7: Out of Order Fetch Architecture with a Lockup-Free Instruction Cache

serve to warm up the L2 cache.

1.4.4 Nonblocking Instruction Caches and Out Of Order Fetch

In [6], Kroft proposed a cache architecture that could continue to operate in the presence of a cache miss. This *lockup-free* instruction cache used a number of cache line buffers - known as miss information status history registers or *MSHRs* - to track in-flight cache misses (i.e. those that have not yet returned from the levels of the memory hierarchy). On a cache miss, an available MSHR is allocated to the address that missed, and would eventually store the incoming cache block from the other levels of the memory hierarchy until it could be installed into the instruction cache. This approach left the instruction cache free to service other requests. The number of in-flight misses was limited by the number of available MSHRs.

Stark et. al [13] extended the idea of lockup-free caches as an alternative to instruction cache prefetching called *out of order instruction fetch*. This architecture is illustrated in Figure 1.7. Instructions can be fetched out of order from the instruction cache into a result fetch queue. When the instruction cache misses, an MSHR is allocated for the missed address. Their fetch architecture continues to supply instructions after the cache miss to the result queue. A placeholder in the queue tracks where instructions in in-flight cache lines will be placed once they have been fetched from other levels of the memory hierarchy. However, this architecture still maintains in-order semantics as instructions leave this queue to be renamed. If the next instruction in the result queue to be renamed/allocated is still in-flight, renaming/allocation

will stall until the instructions return from the other levels of the memory hierarchy.

Out of order fetch requires more sophistication in the result fetch queue implementation to manage and update the placeholders from the MSHRs, and to stall when a placeholder is at the head of the result queue (i.e. the cache line for that placeholder is still in-flight).

Figure 1.8 illustrates our example address stream on the out of order fetch architecture. The PC, instruction cache, result fetch queue, and four MSHRs are shown (nothing is drawn to scale). The *V* field in the MSHR indicates that the entry is valid and is tracking an in-flight cache line. The *R* field in the MSHR indicates that the entry is ready – that it has arrived from the other levels of the memory hierarchy.

Cycle *k* sees a successful cache access for address 992, and that cache line is placed in the result fetch queue. In cycle *k*+1, address 512 misses in the instruction cache and is allocated to an MSHR. A placeholder for this line is installed in the result fetch queue. If there are no other lines in the result fetch queue, stages fed by this queue must stall until the line is ready. However, instruction fetch does not stall. In the next cycle, address 544 also misses in the cache and is also allocated an MSHR. A placeholder for this address is also placed in the queue. In cycle *k*+3, address 576 hits in the cache, and the corresponding cache line is placed in the queue. The stages that consume entries from this queue (i.e. rename) must still stall since they maintain in-order semantics and must wait for the arrival of address 512. Once 512 arrives, it will replace the placeholder in the queue, and later stages like rename can continue to feed from this queue. The cache line for address 512 will be placed in the instruction cache and the MSHR for that address will be deallocated. In this simple example, the latency of the request for address 512 helped to hide the latency of 544, 320, and 352. More address latencies could be hidden with more MSHR entries and assuming sufficient result fetch queue entries. This approach is also heavily reliant on the branch prediction architecture to provide an accurate stream of instruction addresses.

The similarity between MSHRs and stream buffers should be noted, as both structures hold in-flight cache lines. The key difference is that MSHRs track demand misses and stream buffers hold speculated prefetches.

1.4.5 Fetch Directed Instruction Prefetching

Another approach is to guide stream buffer allocation with the instruction fetch stream. Chen et. al [2] used a lookahead branch predictor to guide instruction cache prefetch. Reinman et. al [10] further explored this approach by decoupling the main branch predictor from the instruction cache rather than using a smaller, separate predictor. They used the stream of fetch addresses from this predictor to prefetch the instruction cache. The benefit of this approach is that the same predictor that guides instruction fetch is used to guide instruction prefetch. Aggressive branch prediction designs have im-

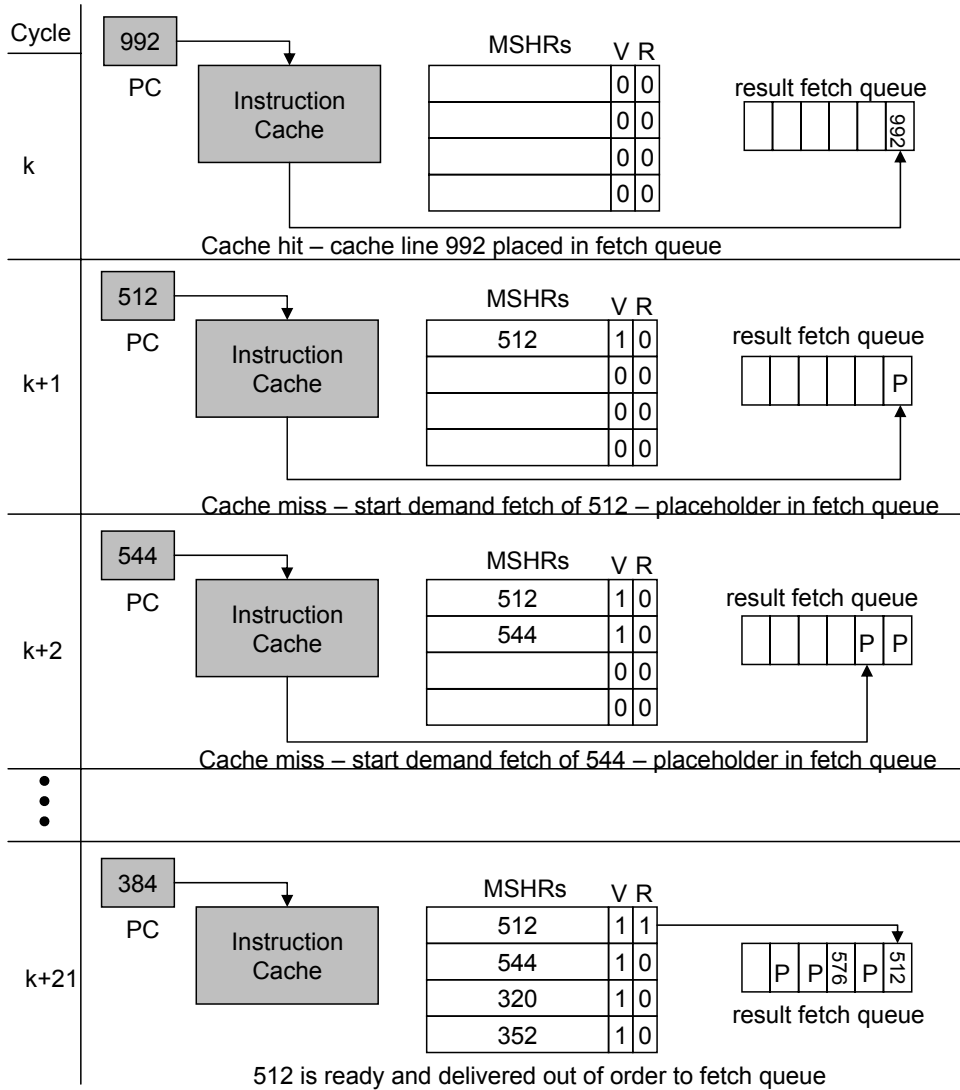


FIGURE 1.8: Out of order fetch in action on the first part of the example from Figure 1.5.

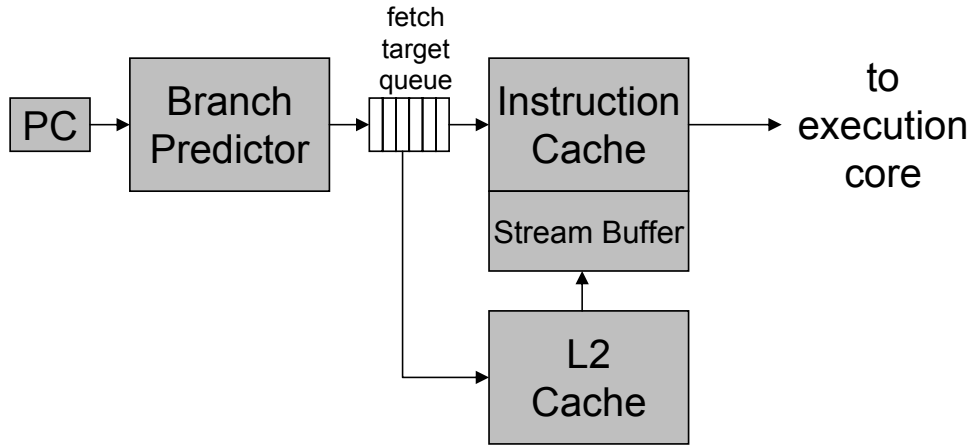


FIGURE 1.9: The Fetch Directed Prefetching Architecture

proved predictor accuracy, and the benefit obtainable through prefetching is heavily reliant on the accuracy of the prefetches made. As in stream buffer prefetching, idle or replicated instruction cache tag ports are used to filter out redundant prefetches in such schemes.

Figure 1.9 illustrates the architecture of [10]. The branch predictor and instruction cache are decoupled via a queue of fetch target addresses, the fetch target queue. The entries in this queue are used to guide instruction cache prefetching into a FIFO or fully associative stream buffer. The buffer is probed in parallel with the instruction cache.

Figure 1.10 demonstrates the performance of the fetch directed prefetching architecture on our example address stream. The fetch target queue supplies instruction addresses from the branch predictor to the instruction cache. The arrow under the queue indicates the cache line currently under consideration for prefetch. In cycle k , cache line 992 is fetched from the instruction cache while cache line 512 is under considering for prefetch. Address 512 is not found in the instruction cache using cache probe filtering and is prefetched. In the next cycle, the cache misses on address 512 and stalls until this miss is satisfied. In the same cycle, a prefetch is initiated for address 544. Prefetching continues guided by the fetch target queue up until the stream buffer is full. Once 512 is fetched from the next level of memory, fetch can continue.

Fetch directed prefetching and out of order fetch are very similar, but there are some key differences. Out of order fetch is limited by the number of available MSHRs, just as fetch directed prefetching is limited by the number of available stream buffer entries. But out of order fetch is also limited by the number of entries in the result fetch queue, a structure which stores instructions. Fetch directed prefetching is limited by the number of entries in the fetch target queue, a structure which stores fetch addresses. Occupancy

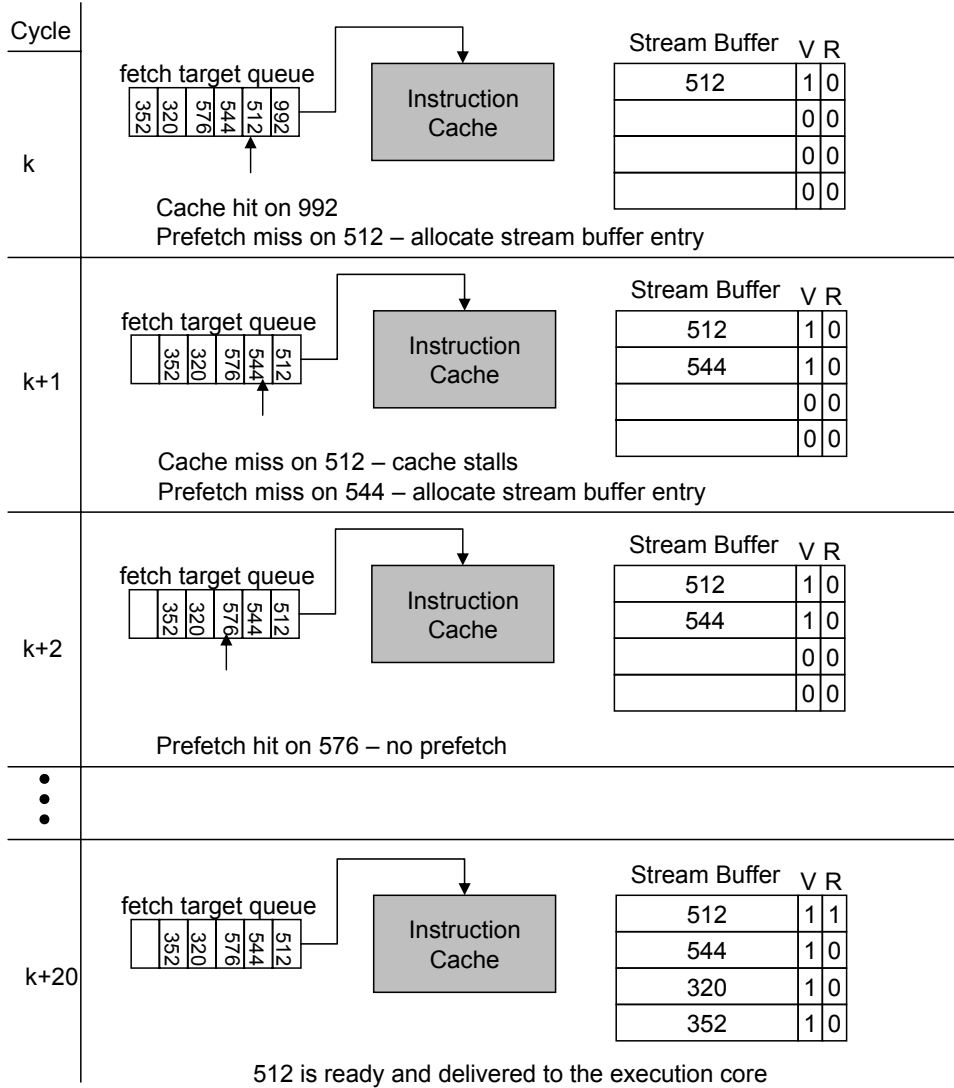


FIGURE 1.10: The fetch directed prefetching architecture in action on the first part of the example from Figure 1.5.

in either of these queues allows these mechanisms to look further ahead at the fetch stream of the processor and tolerate more latency, but the fetch target queue uses less space for the same number of entries. Fetch directed prefetching is only limited by the bandwidth of the branch predictor. To scale the amount of prefetching, the branch predictor need only supply larger fetch blocks. To scale prefetching with accurate filtering, the tag component of the cache must have more ports or must be replicated. In order to scale the number of cache lines that can be allocated to MSHRs by out of order fetch in a single cycle, the branch predictor bandwidth must be increased and the number of ports on the instruction cache must increase.

One other difference is that prefetches can often start slightly before out of order fetches. Since prefetching is a lookahead mechanism, a prefetch can be initiated at the same time that a cache line is fetched from the instruction cache. This is illustrated in the simple example of Figure 1.10 where address 512 is prefetched one cycle earlier than in out of order fetch.

1.4.6 Integrated Prefetching

One drawback to both fetch directed prefetching and out of order fetch is the complexity of the queue structures. Fetch directed prefetching uses a queue that can initiate a prefetch from any entry of the queue. Moreover, the tag component of the instruction cache is accessed twice for each cache line address - once to determine whether or not to prefetch a given cache line, and once during the actual instruction cache access.

One alternative to this policy is to more closely integrate instruction prefetching with the design of the instruction cache, as in [11]. The approach of [11] is to decouple the tag and data component of the instruction cache from one another via a queue of cache line requests (the cache block queue). For every cache access, the tag component determines what way (if any) of the cache a given cache line resides. If it is in the cache, it places the requested address and way in the cache block queue. The data component can then consume this information from the queue and provide the requested line.

Since the tag is only checked once, additional hardware is necessary to ensure that the verified cache lines stored in the cache block queue are not evicted from the cache before they are read from the data component of the cache. This mechanism is the cache consistency table (CCT). The CCT has as many sets and as much associativity as the instruction cache, but only stores 3 bits per line in the CCT. A 64KB 4-way set associative instruction cache would only need a 768B CCT. Each time a cache line is verified by the tag component, the CCT line corresponding to that cache line is incremented. Each time a cache line is provided by the data component, the CCT line corresponding to that cache line is decremented. On a cache replacement, a line will not be kicked out if its CCT entry is not zero. If the CCT entry saturates, the tag component will not verify any more requests for that cache line until the CCT counter is decremented for that entry. In addition to providing a consistency

mechanism, the CCT also provides an intelligent replacement policy. The tag component does not stall on an instruction cache miss, and therefore it can run ahead of the data component. The CCT then reflects the near future use pattern of the instruction cache, and can help guide cache replacement. The larger the cache block queue, the further ahead the CCT can look at the incoming fetch stream. This of course requires accurate branch prediction information.

On a cache miss, the integrated prefetching mechanism allocates an entry in the stream buffer (used much like an MSHR). The stream buffer can bring in the requested cache line while the tag component of the cache continues to check the incoming fetch stream. Once the missed cache line is ready, it can either be installed into the instruction cache (if the CCT can find a replacement line) or can be kept in the stream buffer. This allows the stream buffer to be used as a flexible repository of cache lines, providing extra associativity for cache sets with heavy thrashing behavior.

Figure 1.11 demonstrates the integrated prefetching architecture of [11]. The stream buffer is also decoupled, and requires a CCT of its own to guide replacement. The tag components of both the stream buffer and instruction cache are accessed in parallel. There is a single shared cache block queue to maintain in-order fetch. Each entry in the cache block queue is consumed by the data component of both the instruction cache and stream buffer. However, only one of the two data components will be read depending on the result of the tag comparison.

1.4.7 Wrong-Path Prefetching

Many of the prefetching techniques described in this Chapter rely on accurate branch prediction to capture the instruction fetch address stream. However, no branch predictor is perfect, and mispredictions must be handled. While the cache block queue, fetch target queue, or result fetch queue would all be flushed on a misprediction, flushing the MSHRs and stream buffers might throw away potentially beneficial prefetches. Pierce and Mudge [9] found that prefetching on incorrectly speculated branch paths can actually be beneficial in many cases. If a branch is predicted to be taken when it is actually not taken, it may be taken at a later point in time. If addresses on the taken path are prefetched and waiting in an MSHR or stream buffer, these may avoid cache misses when the branch is seen to be taken later. For example, consider a frequently executed while loop. If control is mispredicted to exit the while loop, and the instructions following the loop are prefetched, then that will hide the latency seen when the while loop actually is exited. Another case where wrong path prefetches can help is when short forward branches are mispredicted. The caveat here is that the wrong path prefetch may be evicted by prefetches on the correct path, and that the wrong path prefetch may itself evict useful blocks.

One way to preserve entries in the MSHRs and stream buffers, but still

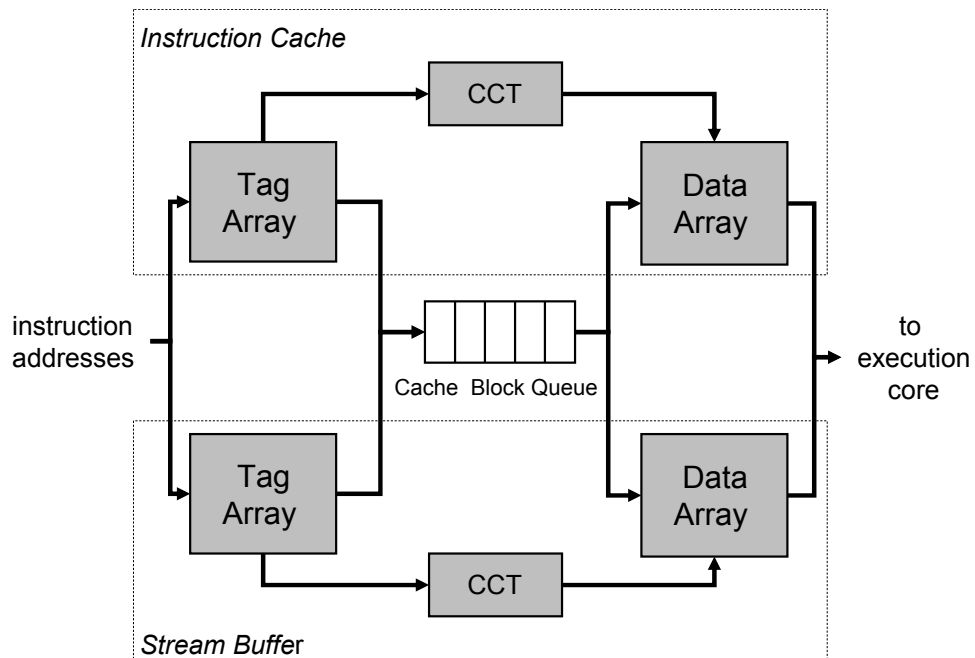


FIGURE 1.11: Integrated Prefetching Architecture: The cache and stream buffer tag components are decoupled from their respective data components. Cache and stream buffer replacement are both guided by the cache consistency tables (CCT) for the respective structures. If both the cache and stream buffer miss, the desired line is brought into the stream buffer from the L2 or main memory.

allow new prefetch requests, is to add another bit to each MSHR or stream buffer entry that indicates whether or not that entry is replaceable or not. If the bit is set, that means that while the entry may be valid, it is from a incorrectly speculated path. Therefore, it may be overwritten if there is demand for more prefetches or misses. But assuming the entry is valid and ready, it should still be probed on stream buffer or MSHR accesses to see if there is a wrong-path prefetch hit.

1.4.8 Compiler Strategies

Compiler directed approaches have also been proposed to help tolerate instruction cache misses. One common approach is code reordering, where instruction memory is laid out to keep the most frequently taken control path as a contiguous stream of instruction addresses. By keeping the most frequently taken control path as one sequential instruction address stream, simple techniques like stream buffers or next line prefetching become more effective. This of course requires some compiler heuristic or prefetching to determine the most frequently taken control path.

Another approach is to use explicit prefetch instructions in the ISA. Such instructions can be placed at strategic points in the code where prefetches should be initiated for good timeliness. The limitation is how accurately static techniques can predict the control flow of an application at run time.

The Itanium processor [7] makes use of two types of software prefetch instructions: hints and streams. A hint prefetches between 12-48 instructions depending on the type (brp.few or brp.many) and streams continue to prefetch contiguous instruction cache blocks until redirected.

1.5 Future Challenges

Instruction cache prefetching has seen improvements in both accuracy and timeliness of prefetches, but there is still room for improvement. Memory usage of applications has been growing recently, spurred on by the increasing memory capacity projected in future processors. However, as clock rates become more aggressive, the latency impact of instruction caches may prevent these caches from growing large enough to meet increased application demand. This will require more intelligent and flexible instruction delivery mechanisms to take advantage of application locality. Multiple application threads may also contribute to this problem, requiring architects to devise novel approaches to warm up threads on context switches. On an SMT machine, what is the best way to use available memory bandwidth and accelerate the particular thread mix in the processor? Researchers are also exploring ap-

plication specific customization of instruction caches and prefetching policies to individual program phases or compiler directed hints.

References

- [1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th Annual International Symposium on Computer Architecture*, 2000.
- [2] I.K. Chen, C.C. Lee, and T.N. Mudge. Instruction prefetching using branch prediction information. In *International Conference on Computer Design*, pages 593–601, October 1997.
- [3] K. Farkas and N. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.
- [4] W. Hsu and J. Smith. A performance study of instruction cache prefetching methods. In *IEEE Transactions on Computers, Vol. 47, NO. 5*, May 1998.
- [5] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.
- [6] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium of Computer Architecture*, pages 81–87, May 1981.
- [7] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. In *IEEE Micro*, March/April 2003.
- [8] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.
- [9] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *29th International Symposium on Microarchitecture*, pages 165–175, December 1996.
- [10] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *32nd International Symposium on Microarchitecture*, November 1999.
- [11] G. Reinman, B. Calder, and T. Austin. High performance and energy efficient serial prefetch architecture. In *In the proceedings of the 4th International Symposium on High Performance Computing*, May 2002.

- [12] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.
- [13] J. Stark, P. Racunas, and Y. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 34–45, December 1997.