Contents

I This is a Part **7**

1 Trace Caches **11**

Eric Rotenberg North Carolina State University

1.1 Introduction 12
1.2 Instruction Fetch Unit with Trace Cache 15
1.2.1 Traces 15
1.2.2 Core fetch unit based on instruction cache 16
1.2.3 Trace cache operation 18
1.3 Trace Cache Design Space 22
1.3.1 Path associativity 22
1.3.2 Indexing strategy 22
1.3.3 Partial matching 22
1.3.4 Coupling branch prediction with the trace cache . . . 23
1.3.5 Trace selection policy 24
1.3.6 Multi-phase trace construction 25
1.3.7 Managing overlap between instruction cache and trace cache 25
1.3.8 Speculative *vs.* non-speculative trace cache updates . 26
1.3.9 Powerful *vs.* weak core fetch unit 26
1.3.10 Parallel *vs.* serial instruction cache access 27
1.3.11 L1 *vs.* L2 instruction cache 27
1.4 Summary 28

References **29**

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

List of Tables

1.1 Growth in instruction-level parallelism (ILP). 12

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

List of Figures

1.1 Conventional instruction cache. 13
1.2 High-level view of trace cache operation. 14
1.3 Instruction fetch unit with trace cache. 16
1.4 Two-way interleaved instruction cache. 17
1.5 Contents of trace cache line. 21

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

Part I

This is a Part

Symbol Description

Symbol	Description
α	To solve the generator maintenance scheduling, in the past, several mathematical techniques have been applied.
σ^2	These include integer programming, integer linear programming, dynamic programming, branch and bound etc.
$\sum$	Several heuristic search algorithms have also been developed. In recent years expert systems,
abc	fuzzy approaches, simulated annealing and genetic algorithms have also been tested.
$\theta\sqrt{abc}$	This paper presents a survey of the literature
ζ	over the past fifteen years in the generator
∂	maintenance scheduling. The objective is to
sdf	present a clear picture of the available recent literature
ewq	of the problem, the constraints and the other aspects of
bvcn	the generator maintenance schedule.

Chapter 1

Trace Caches

Eric Rotenberg
North Carolina State University

1.1 Introduction 12
1.2 Instruction Fetch Unit with Trace Cache 15
1.3 Trace Cache Design Space 22
1.4 Summary 28

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

1.1 Introduction

To put processor performance growth in perspective: The number of in-flight instructions in various stages of execution at the same time, inside a high-performance processor, has grown from tens to hundreds of instructions in the past decade. The growth in instruction-level parallelism (ILP) is due partly to significantly deeper pipelining, and partly to increasing the superscalar issue width, as shown in Table 1.1.

TABLE 1.1: Growth in instruction-level parallelism (ILP), measured by in-flight instruction capacity.

Processor Generation	Pipeline Depth (fetch to execute)	Issue Width	In-flight Instructions
Pentium	5	1 instr.	∼5
Pentium-III	10	3 μ-ops	∼40
Pentium-IV	20	3 μ-ops	126
IBM Power4	12	5 instr.	200

Both factors – increasing width and depth – place increasing pressure on the instruction fetch unit. Increasing the superscalar issue width requires a corresponding increase in the instruction fetch bandwidth, simply because instructions cannot issue faster than they are supplied.

The relationship between increasing pipeline depth and fetch bandwidth is more subtle. Due to data dependences among instructions, finding enough independent instructions that can execute in parallel requires examining a much larger "window" of speculative instructions. Branch mispredictions cause the window to fill with useless instructions that are eventually flushed, leaving the window empty. A deep pipeline will take a long time to replenish the window after a flush. As a result, it takes a long time to ramp up execution to its peak parallel efficiency, due to an inadequate pool of instructions from which to expose independent instructions. In this case, a fetch unit that can deliver high instruction fetch bandwidth can rapidly replenish the window in a sort of "burst" mode, ramping up overall execution faster.

Unfortunately, increasing instruction fetch bandwidth is impeded by several factors. Instruction cache misses and branch mispredictions constrain instruction fetch bandwidth by disrupting the supply of useful instructions for extended periods of time. However, even absent these discrete performance-degrading events, sustained instruction fetch bandwidth is limited on a continuous basis by frequent *taken branches* in the dynamic instruction stream. Conventional fetch units cannot fetch the instructions following a taken branch in the same cycle as the branch, since the blocks containing the branch and

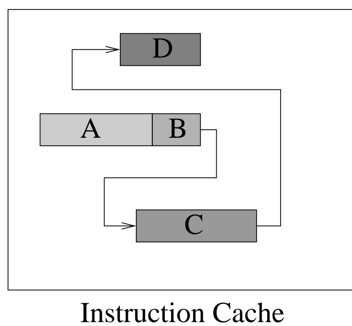


FIGURE 1.1: A conventional instruction cache stores instructions in their static program order.

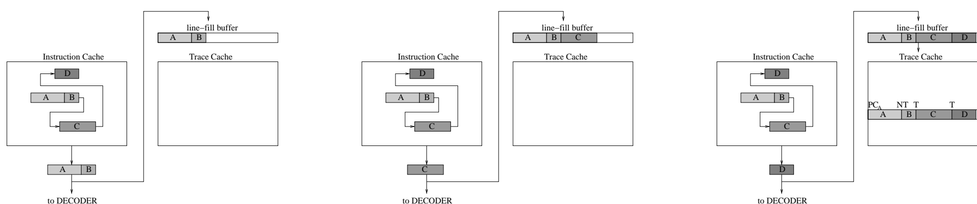
the target instructions are noncontiguous in the instruction cache. For example, it takes a minimum of three cycles to fetch the dynamic instruction sequence composed of basic blocks A, B, C, and D, shown in Figure 1.1, due to taken branches at the end of basic blocks B and C.

The taken-branch bottleneck did not really surface until microarchitects began pondering very aggressive superscalar processors, with >200-instruction windows and >10 peak issue bandwidth (16-issue processors became a popular substrate for a number of years [10, 15, 23]). Since taken branches typically occur once every 5 to 8 instructions [19] in integer benchmarks, a new approach to fetch unit design was needed.

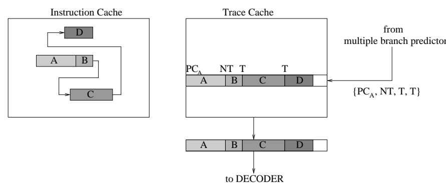
The problem lies with the organization of the instruction cache, which stores instructions in their *static program order* even though instructions must be presented to the decoder in *dynamic program order*. Reconstructing a long sequence of dynamic instructions (say 16 or more) typically takes multiple cycles, since the constituent static blocks are in noncontiguous locations throughout the instruction cache.

The trace cache was proposed to reconcile the way instructions are stored with the way they are presented to the decoder [12, 11, 7, 16, 20, 19, 14, 2]. A trace cache stores instructions in their dynamic program order. A trace is a long sequence of dynamic instructions, possibly containing multiple branches, some or all of which may be taken branches. Upon predicting and fetching a trace for the first time, the conventional instruction cache constructs the trace sequentially over several cycles, and supplies it to the decoder and execution engine. This first-time trace is also presented to and stored in the trace cache, for later use. Then, when the trace is needed again, as predicted by the branch prediction mechanism, it is found in the trace cache and the entire trace is supplied in a single cycle to the decoder, thereby exceeding the taken-branch bandwidth limit. Figure 1.2 depicts the process of filling the trace cache with a new trace (trace cache miss) and then later reusing the same trace from the trace cache (trace cache hit).

Alternative high-bandwidth fetch mechanisms have been proposed that as-



(a) Trace cache miss: Assembling first-time trace (3 cycles).



(b) Trace cache hit: Reuse trace.

FIGURE 1.2: High-level view of trace cache operation.

semble multiple noncontiguous fetch blocks from a conventional instruction cache [24, 1, 22]. The underlying idea is to generate multiple fetch addresses that point to all of the noncontiguous fetch blocks that must be assembled. The instruction cache is highly multiported, to enable fetching multiple disjoint cache blocks in parallel. This is achieved using true multiple ports (area-intensive), or, more cheaply, using multiple banks, although bank conflicts limit fetch bandwidth and steering addresses to their corresponding banks introduces complexity. Finally, the noncontiguous fetch blocks are assembled into the desired trace via a complex interchange/mask/shift network, which reorders the cache blocks and collapses away their unwanted instructions.

In essence, these alternative high-bandwidth fetch mechanisms repeatedly assemble traces on-the-fly from the conventional instruction cache. The trace cache moves this complexity off the critical fetch path, to the fill side of the trace cache, thus assembling traces only once and then reusing them many times afterwards.

In the following sections, we describe an instruction fetch unit with a trace cache, and then discuss in-depth issues regarding the trace cache design space.

1.2 Instruction Fetch Unit with Trace Cache

1.2.1 Traces

A *trace* is a sequence of dynamic instructions containing multiple, possibly noncontiguous basic blocks. A trace is uniquely identified by the PC of the first instruction in the trace, the start PC, and the sequence of taken/not-taken conditional branch outcomes embedded within the trace. The start PC plus sequence of conditional branch outcomes is sometimes referred to as the *trace identifier*, or trace id. Trace ids are fundamental in that they provide the means for looking up traces in the trace cache, regardless of specific policies for managing the trace cache (e.g., indexing policies), of which there are multiple alternatives.

The hardware limits the maximum number of instructions and branches in a trace. The maximum number of instructions in a trace is governed by the line size of the trace cache. The maximum number of branches in a trace is governed by the prediction mechanism. In order to predict the next trace to be fetched, a *multiple-branch predictor* [24, 20, 14] is needed to generate multiple branch predictions in a single cycle, corresponding to a particular path through the program rooted at the current fetch PC. Thus, the maximum number of branches in a trace is determined by the throughput of the multiple-branch predictor, i.e., the number of branch predictions it can produce per cycle.

There are typically other constraints that may terminate traces early, before the maximum number of instructions/branches is fulfilled. Traces are typically terminated at subroutine return instructions, indirect branches, and system calls. The targets of returns and indirects are variable, thus their outcomes cannot be represented with the usual taken/not-taken binary indicator. Embedding returns/indirects would require expanding the trace id representation to include one or more full target addresses (depending on the maximum number of allowable returns/indirects). Although it is feasible to design a trace cache that permits embedded returns/indirects, terminating traces at these highly-variable control transfers reduces the number of unique traces that are created, reducing trace cache pressure (conflicts) and thereby improving instruction fetch performance despite reducing the average length of traces.

The criteria for forming (delineating) traces are collectively referred to as the *trace selection* policy. Overall instruction fetch bandwidth is affected by a number of interrelated trace cache performance factors, e.g., average trace length, trace cache hit rate, trace prediction accuracy, etc. The trace selection policy typically affects all of these performance factors, providing significant leverage for balancing tradeoffs among these factors.

1.2.2 Core fetch unit based on instruction cache

An instruction fetch unit incorporating a trace cache is shown in Figure 1.3. At its core is a high-performance conventional fetch unit, comprised of: (1) a conventional instruction cache, (2) a branch target buffer (BTB) [9], (3) a conditional branch predictor, and (4) a return address stack (RAS) [8].

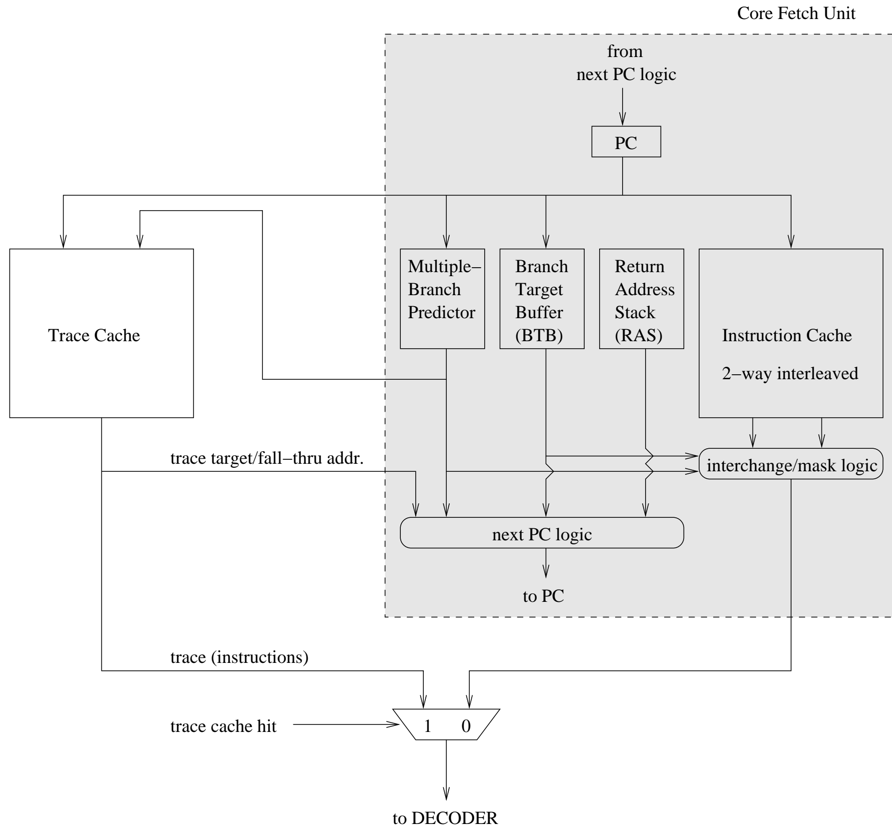


FIGURE 1.3: Instruction fetch unit with trace cache.

The instruction cache is two-way interleaved, in order to provide a full cache line's worth of sequential instructions, even if the access is not aligned on a cache line boundary [3]. Without interleaving, an unaligned access supplies fewer instructions than the number of instructions in the cache line, because the fetch PC begins in the middle of the cache line. With two-way interleaving, two sequential cache lines are fetched in parallel from two banks containing only even-address and odd-address lines, respectively. Thus, although an unaligned access begins in the middle of one of the lines (either

even or odd, depending on the fetch PC), additionally fetching instructions from the next sequential line enables gathering a full line of instructions, as shown in Figure 1.4.

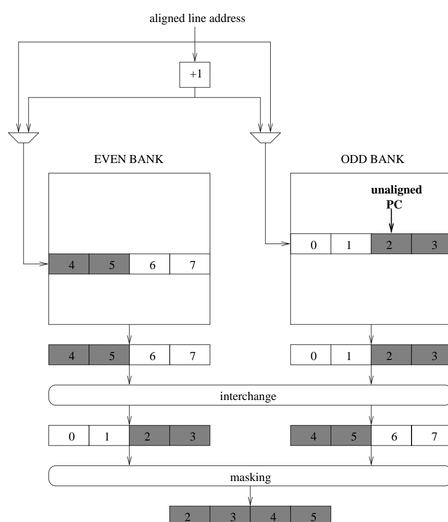


FIGURE 1.4: Two-way interleaved instruction cache.

The BTB detects and predecodes branches among the instructions fetched from the instruction cache, in the same cycle they are fetched. This enables the next fetch PC to be calculated in time for the next cycle, before the decode stage. If a control transfer is a conditional branch and the conditional branch predictor predicts the branch is taken, then the BTB also provides the precomputed taken target. If the control transfer is a return instruction, then the RAS is popped to predict the target of the return.

As stated earlier, the conditional branch predictor must be capable of generating multiple branch predictions per cycle, if it is to guide fetching traces from the trace cache. We will discuss the operation of the trace cache in the next subsection. But, first, note that the core fetch unit based on the instruction cache may also exploit multiple branch predictions. A conventional instruction cache can supply multiple *sequential* basic blocks (i.e., their branches are all predicted not-taken), up to the first predicted-taken branch. This aggressive core fetch unit has been referred to as SEQ.n [19, 21], highlighting the ability to fetch multiple sequential basic blocks in a single cycle, as opposed to SEQ.1 which can only fetch one basic block per cycle, even if its terminal branch is predicted as not-taken.

The instruction cache is indexed using the low bits of the current fetch PC. The upper bits of the PC are used in the usual way to detect a cache hit

or miss (compared against one or more tags). One or more hits in the BTB locate branches within the fetched instructions from the instruction cache. Branch predictions from the multiple-branch predictor are matched up with branches detected by the BTB. The *alignment* logic after the instruction cache swaps the two lines fetched from the banks, if instructions from the odd bank logically precede instructions from the even bank. *Masking* logic combines the BTB information (location of branches) and the multiple-branch predictions to select only instructions before the first predicted-taken branch.

1.2.3 Trace cache operation

In early trace cache implementations, such as the one shown in Figure 1.3, the instruction cache and trace cache are accessed in parallel. The current fetch PC plus multiple taken/not-taken predictions are used to access both caches. If the predicted trace is found in the trace cache, then instructions supplied by the instruction cache are discarded since they are a subset of the trace supplied by the trace cache. Otherwise, instructions supplied by the instruction cache are steered to the decoder. Fetching multiple sequential basic blocks from the instruction cache was explained in the previous subsection.

In one of the most basic trace cache designs, the trace cache is indexed with the low bits of the current fetch PC. Each trace in the trace cache is tagged with its trace id, i.e., start PC plus embedded branch outcomes. The low bits of the start PC are omitted since they are implied in the trace cache index (same concept applies to conventional cache tags). The embedded branch outcomes are called the *branch flags*. Each branch flag is one bit and encodes whether or not the corresponding branch is taken. In addition, a *branch mask* effectively encodes the number of embedded branches in the trace. The branch mask is used by the hit logic to compare the correct number of branch predictions with branch flags, according to the actual number of branches within the trace. Also, several bits are included to indicate whether or not the last instruction in the trace is a branch, and its type. Finally, two address fields are provided, a *fall-through address* and a *target address*. These are explained later. The content of a generic trace cache line is shown in Figure 1.5, including the various fields outlined above and the trace's instructions. For illustration, the maximum number of branches in a trace is 4 (the maximum number of instructions is not explicitly shown – 16 is fairly typical).

The trace cache hit logic compares the fetch PC (excluding index bits) plus multiple branch predictions with the trace id read from the trace cache. A hit is signaled if the fetch PC matches the trace's start PC and the branch predictions match the trace's branch flags. The branch mask provides a quick means for comparing only as many predictions/flags as there are branches internal to a trace. Only internal branches, and not a terminal branch (a branch that is the last instruction in the trace), distinguish the trace and are considered by the hit logic. The branch mask is a sequence of (zero or more) leading 1's followed by (zero or more) trailing 0's. The number of

leading 1's is equal to the number of internal branches in the trace (again, this excludes a terminal branch). Thus, there is a trace cache hit if (1) the fetch PC matches the indexed trace's start PC and (2) the following test is true: (branch_predictions & branch_mask) == (branch_flags & branch_mask).

Note that, if the maximum branch limit is b branches overall, then the maximum number of embedded branches is no more than $b-1$. Thus, there are $b-1$ branch flags and the branch mask is $b-1$ bits long. (This assumes a trace selection policy that terminates a trace once the bth branch is brought into the trace. A subtle alternative is to try extending the trace after the bth branch until the maximum instruction limit is fulfilled, but stopping just before the next branch so that the maximum branch limit is still observed. We assume the former policy.)

In the case of a trace cache hit, the branch mask determines how many of the predictions supplied by the multiple-branch predictor are actually "consumed" by the matching procedure. If the trace ends in a conditional branch, then one additional prediction is consumed, for predicting this terminal branch (more on this when we explain the two address fields). (Since the maximum branch limit is observed regardless of specific nuances of trace selection policies, the multiple-branch predictor provides enough branch predictions.)

Like conventional caches, the trace cache can be set-associative. In this case, multiple trace ids are read from the trace cache, equal to the number of ways. The trace cache hit logic is replicated to match the fetch PC plus predictions against all of the trace ids read from the trace cache.

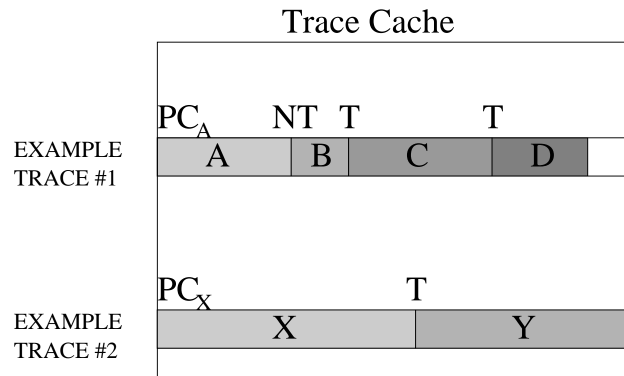
The trace's fall-through and target addresses are required because the BTB cannot generate the next PC following the trace, in the case of a trace cache hit. The trace typically extends beyond the scope of the current BTB access, by virtue of fetching past multiple taken branches. The current BTB access extends only through the first predicted-taken branch, supplying that branch's taken target, corresponding to an internal target within the trace and not the overall trace's target. (Moreover, for a less aggressive core fetch unit implementation than the one described in subsection 1.2.2, the BTB access only detects and provides a target for the first branch, whether taken or not-taken.)

The trace's fall-through and target addresses are managed according to the last instruction in the trace (terminal instruction). If the terminal instruction is a conditional branch, then the two address fields are equal to the branch's fall-through and target addresses, respectively. If the trace hits and the terminal branch is predicted taken, then the next PC is set equal to the trace's target address. If the terminal branch is predicted not-taken, then the next PC is set equal to the trace's fall-through address.

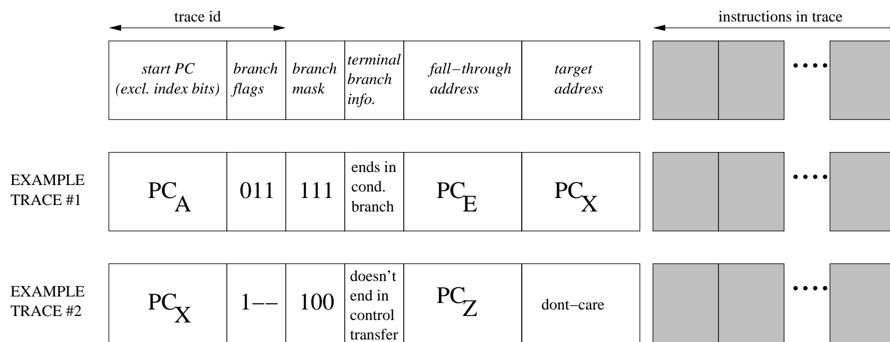
If the last instruction is not a control transfer instruction, then the trace's fall-through address is equal to the last instruction's PC plus one. If the trace hits, then the next PC is set equal to the trace's fall-through address, unconditionally (the trace's target address is ignored).

If the last instruction is a return instruction, then both address fields are

irrelevant. If the trace hits, then the next PC is obtained from the RAS, like usual. (If there are one or more calls embedded within the trace and the trace ends in a return, then a RAS "hazard" exists: the last call's target must be forwarded to the return instruction, effectively implementing a RAS bypass.)



(a) Example traces in trace cache.



(b) Contents of trace line, and representations for two example traces.

FIGURE 1.5: Contents of trace cache line (with two examples).

1.3 Trace Cache Design Space

1.3.1 Path associativity

The basic trace cache design presented in the previous section uses only the fetch PC to index into the trace cache. Thus, if the trace cache is direct-mapped, then two different traces rooted at the same start PC cannot be cached simultaneously.

Path associativity refers to the ability to cache multiple traces rooted at the same start PC. Note that a set-associative trace cache provides some degree of path associativity, since it can cache multiple traces with the same start PC within a set. Traces with the same start PC always map to the same set and thus this implicit approach for achieving path associativity is not totally flexible. Nonetheless, research has shown that conventional 2-way and 4-way set-associativity is effective for providing path associativity [21, 14].

A different indexing strategy is needed to endow a direct-mapped trace cache with path associativity, as discussed in subsection 1.3.2.

1.3.2 Indexing strategy

In the basic trace cache design described in the previous section, the trace cache index is formed using only PC bits. However, additional information from the predicted trade id can also be used, i.e., one or more branch predictions (taken/not-taken bits). One or more predictions can either be concatenated or exclusive-or'ed with the low bits of the PC. This has the advantage of mapping multiple traces with the same start PC to different sets. In this way, even a direct-mapped cache can provide path associativity.

A potential flaw with this approach is that the number of embedded branches within the trace to be fetched, indicated by the branch mask, is not known until it is fetched, too late for forming the index. A one-size-fits-all approach must be used, i.e., a fixed number of predictions is used to form the index. For traces that have fewer embedded branches than the number of branch predictions hashed into the trace cache index, redundant copies of the trace may be created within the trace cache.

Thus, this approach is best used with a predictor that produces trace ids explicitly (in contrast, a conventional multiple-branch predictor always generates the maximum number of branch predictions without regard for trace ids explicitly). A predictor that predicts trace ids explicitly is called a *trace predictor* [5, 21].

1.3.3 Partial matching

Partial matching is another facet of the trace cache design space. Partial

matching refers to the ability to supply part of a trace if there is no cached trace that wholly matches the predicted trace, but there is a trace with a matching prefix.

Partial matching can soften the impact of not providing path associativity. If two traces with the same start PC cannot be cached simultaneously, then one can be pinned in the cache such that one of the traces always hits entirely, and the other trace partially hits due to a common prefix. (To implement this policy, a partial match should be treated as a hit instead of a miss, so that the conflicting traces do not repeatedly replace each other and one trace is preferred. It is not clear how to determine the preferred trace automatically.)

Implementing partial matching may be more trouble than it is worth. Partial matching is more expensive due to the need to include extra fall-through and target addresses for embedded branches, since the BTB can only supply the target for the first embedded branch. Without partial matching, only the overall trace fall-through and target addresses are needed, as described in Section 1.2.3. Moreover, accessing the instruction cache in parallel with the trace cache achieves some of the benefits of partial matching, in that the instruction cache may be able to supply instructions up to the first predicted-taken branch if the trace cache misses. In any case, if partial matching is implemented, it should be designed efficiently so that it supplements rather than competes with the partial matching already provided by the instruction cache.

Of course, there are arguments one way or the other for shifting complexity away from the core fetch unit and focusing efforts on the trace cache, for instance: only access the instruction cache if the trace cache misses, do not provide an L1 instruction cache at all, etc. (these alternatives are explored in subsections 1.3.9, 1.3.10, and 1.3.11). In this case, partial matching in the trace cache may be more than just a luxury.

1.3.4 Coupling branch prediction with the trace cache

In some sense, traces in the trace cache reflect recent histories of branches. This can be exploited to tie branch prediction to the trace cache [16], rather than use a separate branch prediction mechanism.

If there is no external branch prediction mechanism, the trace cache is indexed using only the PC and it returns a predicted trace. This implies the trace cache supplies only the most recent trace rooted at a given PC, implicitly predicting a path through the control-flow region. From the standpoint of branch prediction, this is equivalent to using a one-bit counter to predict branches. (Nonetheless, redundant copies of static branches in the trace cache enables specialization of static branches, to some extent, exploiting limited context.) Alternatively, two-bit counters could be associated with each embedded branch within a trace. A trace is displaced by an alternate trace rooted at the same PC, only if the two-bit counter of the first differing branch switches polarity.

A potential drawback of tying branch prediction to the trace cache is degraded accuracy compared to using a separate state-of-the-art branch predictor, since an integrated approach is typically limited to simple bimodal prediction, as discussed above.

A hybrid approach, called branch promotion [13], exploits the trace cache for predicting highly biased branches but still uses an external predictor to predict unbiased (yet predictable) branches. A branch is promoted when its outcome is perceived to be invariant, in which case the branch predictor stops predicting it. Implicitly, the trace cache predicts promoted branches by virtue of only caching traces reflecting their favored directions. While promotion reduces the cost of the branch predictor and/or improves its accuracy (by offloading biased branches to the trace cache), it does not altogether eliminate the external branch predictor and thus does not achieve the simplicity of fully coupling branch prediction with the trace cache. Nonetheless, promotion provides a subtle example of exploiting the trace cache to predict branches.

1.3.5 Trace selection policy

The trace selection policy determines how the dynamic instruction stream is delineated into traces. A base policy is dictated by hardware constraints – maximum trace length, maximum branch prediction throughput, hardware support (or lack thereof) for embedding indirects/returns, etc.

More sophisticated policies build on top of the base policy and provide leverage for affecting various performance factors of the trace cache. The key factors that affect overall instruction fetch bandwidth are average trace length, trace cache hit rate, and trace prediction accuracy. Trace selection significantly affects these performance factors, sometimes quite subtly [21]. Yet, trace selection has yet to be systematically explored. One reason is that a policy choice typically affects all factors simultaneously, making it difficult to discover more than just basic rules of thumb that have already been observed through ad hoc means and experience.

Typically, there is a tradeoff between maximizing trace length and minimizing trace cache misses. Reducing constraints in order to grow the length of traces also tends to increase the number of frequently-used unique traces, since there are more embedded branches and thus more possible traces (although, sometimes, fewer traces may be created if there is good coverage by the longer traces). One approach for reducing trace pressure, even with very long traces, is to terminate traces at branches [14, 13], i.e., avoid splitting a basic block if possible. This prevents scenarios where nearly identical traces are created except for their start points being slightly shifted.

On the other hand, it has been observed that a large working set of traces can be exploited for higher trace prediction accuracy [21]. A trace predictor that uses a history of trace ids seems to benefit from the shifting effect described above (although this has not been directly confirmed – at present, it is only a hypothesis). The shifting effect indirectly conveys a tremendous

amount of history and provides a very specific context for making predictions, by revealing specific information regarding the path that was taken up to the current point in the program.

As another common example of trading trace length for lower miss rates, terminating traces at call instructions significantly limits the average trace length. Nonetheless, stopping at call instructions significantly reduces the number of unique traces by "re-synchronizing" trace selection at the entry points of functions, regardless of the call site.

1.3.6 Multi-phase trace construction

When fetching and executing a particular sequence of dynamic instructions, there is a corresponding sequence of traces, according to the trace selection policy. However, the same sequence of dynamic instructions can be covered by a different trace sequence, that is slightly shifted or "out of phase" with respect to the current trace sequence. (In fact, there are many different trace sequences, having slightly different phases, that cover the same dynamic instructions.)

Typically, the trace cache is only updated with traces from the current sequence of traces. Yet, it is possible to update the trace cache with other potential trace sequences, which have different phases but otherwise cover the same dynamic instructions. For example, we can begin constructing a new trace after every branch [16], or even after every instruction. This approach requires multiple trace constructors, for simultaneously assembling multiple traces that are slightly shifted with respect to each other. This technique is tantamount to prefetching (preconstructing [6]) traces, based on the heuristic that the present control-flow region will be revisited in the future, only with a slightly shifted phase. This may reduce the perceived trace cache miss rate. On the other hand, a potential negative side-effect is polluting the trace cache with trace sequences that will never be observed.

1.3.7 Managing overlap between instruction cache and trace cache

In one school of thought, the trace cache is viewed as a fetch "booster" that can be kept small if all that can be done is done to improve the instruction cache's performance.

One way to reduce pressure in the trace cache is to avoid obvious cases of overlap with the instruction cache's capabilities. The instruction cache can be highly optimized to fetch multiple continguous basic blocks. Thus, the trace cache should not cache traces with zero taken branches, since the instruction cache can supply these traces very efficiently (and more efficiently than the trace cache, since the trace cache does not exploit spatial locality) [18].

Moreover, a profiling compiler can explicitly off-load more traces from the trace cache to the instruction cache, by converting frequently-taken branches

into frequently-not-taken branches [1, 17]. By increasing the total number of instructions supplied by the instruction cache vs. the trace cache, a smaller trace cache can be used to target only traces that cannot be addressed by any other means.

1.3.8 Speculative *vs.* non-speculative trace cache updates

Another choice is whether to construct new traces speculatively as instructions are fetched from the instruction cache, or non-speculatively as instructions are retired. Constructing traces and filling them into the trace cache speculatively has the advantage of enabling trace cache hits for new traces that are reaccessed soon after the first access. This scenario may occur in small loops. Moreover, if the trip counts are low for these loops, filling the trace cache with new traces at retirement time (non-speculatively) may be altogether too late. On the other hand, filling the trace cache with speculative traces runs the risk of polluting the trace cache with useless traces.

1.3.9 Powerful *vs.* weak core fetch unit

If the trace cache performs sufficiently well (hits often and supplies large traces), one can make a case for simplifying the core fetch unit since most instructions are supplied by the trace cache in any case. For instance, the instruction cache could be non-interleaved since the trace cache handles unaligned accesses just fine. Also, the instruction cache could limit bandwidth to only one basic block per cycle. Both of these suggestions reduce complexity of the logic after the instruction cache needed to swap and mask instructions, as well as the BTB and next PC logic. These simplifications reduce area, static and dynamic power, and latency of the core fetch unit, although these savings must be weighed against a more complex (e.g., larger) trace cache, which we used to justify the downsizing of the core fetch unit, in the first place.

The above argument rests on the tenuous case that the trace cache can always compensate for a weak core fetch unit, and more. The counterargument is that the trace cache is not always reliable, due to lack of spatial locality and occasional explosions in the number of frequently-used unique traces. Trace caches fail to exploit spatial locality because instructions are not individually accessible, only whole traces are. Poor trace selection coupled with a large working set of static instructions may cause temporary or perpetual trace cache overload, even if there is significant instruction cache locality. From this standpoint, the core fetch unit should be designed to maximize sequential-fetch performance, for an overall robust design that is tolerant of marginal trace cache performance.

1.3.10 Parallel *vs.* serial instruction cache access

Up until now, we have assumed that the instruction cache and trace cache are accessed in parallel. The advantage of this approach is that the penalty for missing in the trace cache is not severe. Since the instruction cache is accessed in parallel, part of the trace is supplied to the decoder with no delay, despite the trace cache miss.

A downside of parallel lookups is power consumption. If the trace cache hits, then the instruction cache is needlessly powered up and accessed. A solution to this problem is to access the instruction cache only when it is for certain that the trace cache does not have the trace. While this saves power, unfortunately, there is a performance hit due to delaying the instruction cache access when the trace cache misses.

To sum up, parallel access optimizes performance for the less common case of a trace cache miss, whereas serial access optimizes power for the more common case of a trace cache hit. Which is better depends on how common a trace cache hit is in actuality, almost certainly benchmark dependent (among other factors). It also depends on which of the two factors, power or performance, is deemed more important for the given situation. In a hybrid approach, one could try to predict the likelihood of a trace cache hit, and access the instruction cache either in parallel (trace cache hit unlikely) or serially (trace cache hit likely), accordingly [4].

1.3.11 L1 *vs.* L2 instruction cache

With a sufficiently high trace cache hit rate, one could perhaps make a case for eliminating the L1 instruction cache altogether. In this case, new (or displaced) traces are constructed (or reconstructed) by fetching instructions from the L2 cache. Some research provides evidence that trace construction latency is not critical to overall performance (e.g., even ten cycles to construct a trace and fill it in the trace cache has shown little effect on performance [14]).

The problem with eliminating the L1 instruction cache is that the trace cache may, at times, perform poorly. In this case, an L1 instruction cache can offset trace cache misses better than an L2 instruction cache can.

If the trace cache is only backed by the L2 cache (no L1 instruction cache), the relatively high penalty of trace cache misses can be mitigated by preconstructing/prefetching traces into the trace cache [6].

1.4 Summary

High instruction fetch bandwidth is important as pipelines become wider and deeper. A wide execution engine is ultimately limited by the rate at which instructions are supplied to it. Therefore, raw instruction fetch bandwidth must increase to accommodate wider superscalar processors. Even relatively narrow processors can benefit from large instruction fetch bursts that quickly replenish an empty window, reducing the ramp-up delay after pipeline flushes.

The trace cache provides a means for overcoming the taken-branch bottleneck, which otherwise fundamentally constrains the peak instruction fetch bandwidth. A trace cache exceeds the bandwidth limit of conventional instruction caches, by enabling fetching past one or more taken branches in a single cycle. This is achieved by storing instructions in their dynamic program order, i.e., storing otherwise noncontiguous instructions in a single contiguous trace cache line.

In addition to describing the basic operation of a fetch unit with a trace cache, this chapter also highlighted many design alternatives and tradeoffs that trace caches present. Some of the design alternatives have not yet been fully explored, such as trace selection, suggesting areas for future research. Future adaptive policies may hold the key for reconciling the many tradeoffs among various design alternatives.

Very wide superscalar processors (e.g., 16-issue) – the initial impetus for exploring trace caches in the mid to late 90s – have not materialized commercially. And, for the most part, researchers have returned to modest superscalar cores as substrates for exploring solutions to other bottlenecks such as the memory wall. Nonetheless, ILP seems to come and go in cycles. The author cautiously predicts that wide processors will be revisited when high processor frequency becomes intractable from a complexity and power standpoint. Before that happens, researchers must crack the branch misprediction bottleneck with compelling solutions, in order to expose abundant ILP in programs.

References

[1] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. *22nd International Symposium on Computer Architecture*, pages 333–344, June 1995.

[2] D. Friendly, S. Patel, and Y. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. *30th International Symposium on Microarchitecture*, pages 24–33, Dec 1997.

[3] G. F. Grohoski. Machine organization of the ibm rs/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, Jan 1990.

[4] J. S. Hu, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Selective trace cache: A low power and high performance fetch mechanism. Technical Report CSE-02-016, Department of Computer Science and Engineering, Pennsylvania State University, Oct 2002.

[5] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. *30th International Symposium on Microarchitecture*, pages 14–23, Dec 1997.

[6] Q. Jacobson and J. E. Smith. Trace preconstruction. *27th International Symposium on Computer Architecture*, pages 37–46, June 2000.

[7] J. Johnson. Expansion caches for superscalar processors. Technical Report CSL-TR-94-630, Computer Science Laboratory, Stanford University, June 1994.

[8] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th International Symposium on Computer Architecture*, pages 34–42, May 1991.

[9] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, Jan 1984.

[10] M. Lipasti and J. Shen. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer, Billion-Transistor Architectures*, 30(9):59–66, Sep 1997.

[11] S. Melvin and Y. Patt. Performance benefits of large execution atomic units in dynamically scheduled machines. *3rd International Conference on Supercomputing*, pages 427–432, June 1989.

0-8493-0052-5/00/$0.00+$.50
© 2001 by CRC Press LLC

[12] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. *21st International Symposium on Microarchitecture*, pages 60–66, Dec 1988.

[13] S. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *25th International Symposium on Computer Architecture*, pages 262–271, June 1998.

[14] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan, 1997.

[15] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer, Billion-Transistor Architectures*, 30(9):51–57, Sep 1997.

[16] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. US Patent No. 5,381,533, Jan 1995.

[17] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *13th International Conference on Supercomputing*, pages 119–126, June 1999.

[18] A. Ramirez, J. L. Larriba-Pey, and M. Valero. Trace cache redundancy: Red & blue traces. *6th International Symposium on High-Performance Computer Architecture*, pages 325–336, Jan 2000.

[19] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *29th International Symposium on Microarchitecture*, pages 24–34, Dec 1996.

[20] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. Technical Report 1310, Computer Sciences Department, University of Wisconsin - Madison, Apr 1996.

[21] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers, Special Issue on Cache Memory*, 48(2):111–120, Feb 1999.

[22] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Oct 1996.

[23] J. E. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer, Billion-Transistor Architectures*, 30(9):68–74, Sep 1997.

[24] T-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *7th International Conference on Supercomputing*, pages 67–76, July 1993.