Trace cache miss tolerance for deeply pipelined superscalar processors

G. Reinman and G. Pitigoi-Aron

Abstract: The trace cache is a technique that provides accurate, high bandwidth instruction fetch. However, when a desired instruction trace is not found in the cache, conventional instruction fetch and decode must be used to satisfy the trace request. Such auxiliary fetch hardware can be expensive in terms of energy, area and complexity. An approach to combine a trace cache and conventional instruction fetch hardware using a decoupled design is explored. The design enables the processor to dynamically switch between trace ID and PC-based prediction methods and helps to hide the latency associated with the instruction memory path. The decoupled design with accelerated slow path instruction delivery and no instruction cache is able to provide comparable benefit to a front-end with an 8 kB instruction cache (within 2% of the instructions per cycle with the cache). High tolerance can be demonstrated for both trace table misses and increased memory latency when scaling down the size of the trace table and scaling up the L2 access latency.

1 Introduction

The instruction fetch unit of a microprocessor is responsible for providing a stream of decoded instructions to an execution core, which is responsible for executing and retiring the stream of instructions provided by the instruction fetch unit. This producer/consumer relationship limits the performance of even the most aggressive execution core: the processor can only execute instructions as quickly as the instruction fetch unit can provide them.

Instruction fetch is a critical component of any aggressive out-of-order superscalar processor, but is particularly critical for deeply pipelined processors that may carry substantial branch misprediction penalties. Moreover, deeper pipelines can impact the size of structures that make up the processor pipeline [1].

This paper explores one highly scalable branch-prediction mechanism, the trace cache and trace prediction architecture presented in the work of Jacobson et al. [2]. Although the trace cache provides a high bandwidth and accurate branch prediction, it must be filled by conventional methods, and can miss, requiring the processor to be fed from the slow path [3] (a conventional branch-prediction architecture). Such a miss can carry substantial performance implications, especially if the slow path has been optimised to reduce hardware requirements for a relatively infrequently used component (as in the P4 [4]). Moreover, conventional branch-prediction architectures [5–9] predict instruction addresses using instruction addresses, whereas the trace cache predicts trace IDs using trace IDs. Coordinating two paths on the basis of mechanisms with different interfaces and very different producible bandwidths is challenging and can introduce latency into the instruction fetch unit.

In this paper, a mechanism for combining a trace cache with a conventional branch-prediction architecture is presented, along with techniques to help reduce the latency of dynamically switching from one predictor to another and the latency and hardware requirements of the slow path. In particular, we address the latency of the slow path memory hierarchy without an instruction cache. We investigate two techniques to accelerate instruction delivery without an instruction cache and provide comparable performance to an architecture with an instruction cache. We demonstrate that this architecture scales gracefully even when the L2 latency increases or when the slow path pressure increases.

2 Prior work

The front-end architecture is critical to the performance of a processor as a whole, as the processor can only execute instructions as fast as the front-end can supply them.

2.1 High bandwidth instruction fetch

Architectures for efficient instruction throughput include the two-block ahead predictor [7], the collapsing buffer [8] and the trace cache [10].

Yeh and Patt proposed using a basic block target buffer (BBTB) [5, 6]. The BBTB is indexed by the starting address of the basic block. Each entry contains a tag, type information, the taken target address of the basic block and the fall-through address of the basic block. If the branch ending the basic block is predicted as taken, the taken address is used for the next cycle’s fetch. If the branch is predicted as not taken, the fall-through address is used for the next cycle’s fetch. If there is a BBTB miss, then the current fetch address plus a fixed offset is fetched in the next cycle. In this paper, the BBTB is assumed to work in conjunction with a return address stack (RAS) [11, 12] and a gshare [9] branch predictor.
Reinman et al. [13] proposed a fetch architecture that decouples branch-prediction architecture from the instruction fetch unit (including the instruction cache). The branch predictor and instruction fetch unit are separated by a queue of fetch addresses (branch predictions) called the fetch target queue (FTQ) [14]. In our baseline architecture, we make use of a FTQ to decouple branch prediction from instruction delivery.

2.2 Trace cache

Peleg and Weiser [15] originally led the patent on the trace cache idea. Rotenberg et al. [10] further explored the use of a trace cache to improve instruction fetch throughput. The trace cache holds traces of possibly non-contiguous basic blocks within a single-trace cache line. A start trace address and multiple branch predictions are used to access the trace cache. If the trace cache holds the trace of instructions, all instructions are delivered aligned to the processor core in a single access. A separate engine, called the fill unit, is used to provide dynamic instruction snapshots to the trace cache, avoiding the need to decode instructions stored in the cache.

Rotenberg et al. [3] demonstrated a fetch unit that combined a conventional branch prediction and fetch architecture (including an instruction cache) with a trace cache. The conventional (slow path) fetch architecture is used to fill the trace cache in the event of a trace cache miss.

Jacobson et al. [2] proposed a path-based next-trace predictor to efficiently predict multiple branches in a single cycle and improve overall branch-prediction accuracy. They use trace IDs instead of PCs to index the trace cache, and predict at the trace index granularity. This reduces the problem to a single prediction per cycle, rather than to predict multiple branches with a program-counter-based (PC-based) predictor. Their correlated predictor uses old trace IDS to form a history-hashing function that will predict the next trace to be executed. We extend this architecture by tagging the secondary table as well as the correlating table. This further avoids collisions due to duplication or poorly hashed traces.

Jacobson et al. [2] also examined a RAS to store some trace IDs from the trace history at a procedure call site. This history was then used with some of the history of the return from the call to generate the next trace after a return. The trace cache of our baseline architecture features their path-based next-trace predictor and RAS.

Intel’s Pentium 4 processor [4] makes use of a trace cache. On a trace cache miss, their architecture does make use of a conventional branch-predictor architecture, but they do not use an instruction cache. Instead, they have a dedicated bus to the L2 cache for instruction retrieval. They only have a single decoder for instructions arriving from the L2 cache, taking advantage of the relatively low frequency of trace cache misses to reduce the silicon required by their fetch architecture.

Both Rotenberg et al. [3] and Hinton et al. [4] experience the full impact of instruction decode and retention latency on a trace misprediction. This paper attempts to minimise this latency with minimal impact to silicon area.

3 Integrated fetch architecture

A trace cache [3] has the potential to provide high bandwidth instruction fetch to the execution core of a superscalar processor. However, when the trace cache misses, the processor must rely on conventional fetch mechanisms. This is straightforward when using a conventional branch-prediction engine on the trace cache [3], as both the trace cache and the conventional fetch hardware (i.e. the fetch unit slow path – the BBTB and instruction cache) are indexed via the instruction PC. However, the path-based trace cache [2] is indexed by a hash of trace IDs. This paper presents one approach to alternate between a path-based trace cache and a conventional PC-indexed BBTB and examines the hardware requirements of the fetch unit slow path.

3.1 Integrated fetch prediction

In this paper, we consider two paths: the fast path and the slow path. The fast path is the higher bandwidth trace table and trace cache, and the slow path is the lower bandwidth BBTB and instruction memory. When the fast path misses, the slow path must take over instruction fetch. Switching dynamically between predictors is complicated, as the trace history and RAS of the fast path must be updated during slow path execution to keep prediction accuracy high. This is further complicated by the fact that the tra ce table and BBTB make use of different prediction granularities and bandwidths. The simplest way to transition back to the fast path would be to wait for the next branch misprediction, where the RAS and trace history of the fast path would be recovered from non-speculative data. However, this has a dramatic performance impact, as will be described in Section 5.1. We describe an architecture that provides fast switching between the two paths, and explores techniques to accelerate the slow path, even when there is extra latency on this path (i.e. if there is no instruction cache).

Fig. 1 illustrates the integration of the path-based trace table and BBTB. Only one predictor is active in a given cycle. When the trace table is active, it provides a single trace ID per cycle to the later stages of the processor pipeline. The trace table is indexed with a hash of the last eight trace IDs (stored in the trace history register). As in Jacobson et al. [2], a RAS adjusts the trace history on a return.

Fig. 1 also contains the structures described in Section 2.1. This will be the slow path of the fetch unit, only to be used on a trace table miss. The BBTB is augmented to include a field for a trace ID for every entry in the predictor. The BBTB and trace table are both updated concurrently, and when a trace table entry is added, meaning that we have a new trace ID, the current BBTB entry is augmented with this trace ID. The BBTB entry for the last block in a trace will have the trace ID associated with it. If no trace is added to the trace table (i.e. the trace is still being formed), no trace ID is added to the BBTB.

On each successful trace table prediction, the branch history and PC are updated on the basis of the trace table prediction. Each trace table entry is augmented to track the branch history bits that need to be shifted for a given trace table. These bits are the pattern of taken/not taken bits for a predicted trace ID and are used to update the branch history of the slow path. The trace table is also augmented to track the PC of the head of the last basic block in the trace. This is used to update the current PC of the slow path. Additional hardware or extra space in the trace table could be added to update the RAS of the BBTB. To implement this, we would need the PC of all procedure calls in a given trace table entry. However, we found that there was no significant performance difference when performing this update. Therefore the RAS of the BBTB is not updated by the trace table in the remainder of this paper.
On a trace table miss, the slow path is activated and the BBTB guides instruction fetch. To start the slow path, one of two PCs must be sent to the BBTB: either the fallthrough PC of the last predicted trace or the target of the branch at the end of the last predicted trace if it ended in a branch. However, the trace table and BBTB use fundamentally different styles of prediction: the trace table predicts the current block on the basis of trace ID history and the BBTB predicts the next block on the basis of the current block to be fetched. Therefore the trace table has not made a prediction to determine the PC we need for the BBTB prediction in this cycle. One approach to solving this problem is to have the BBTB generate a prediction every cycle on the basis of the current PC, but only update the current PC with this prediction when the slow path is active or if the trace table misses. When the trace table misses, the PC predicted by the BBTB in that cycle will be the prediction for the instruction at the end of the last trace predicted by the trace table. In the cycle where the trace table misses, no branch predictions are inserted into the FTQ. The BBTB prediction is only made to set up the correct PC for the following cycle. After a single-cycle bubble, the fetch engine can then begin filling the FTQ again from the slow path (BBTB). Fig. 2 demonstrates the action taken on a trace table miss.

When the slow path is active during a trace table miss, if the BBTB hits and there is a valid trace ID in the predictor entry, the valid trace ID is shifted into the trace history. The BBTB also updates the RAS of the trace table when the slow path is active. For each call or return encountered, the RAS is updated accordingly. In parallel with each BBTB access, the trace table checks for a hit with the current contents of the trace history. Once the trace table registers a hit, the current BBTB prediction is disregarded, and the trace table begins to predict the fetch stream in the next cycle once the slow path is deactivated. This way, the trace table can miss and then resort to conventional fetch until the trace table can again make a successful prediction if the fetch stream returns to a path stored in the trace table. This is especially useful considering the fragmentation and duplication problems that can impact the performance of a trace cache fetch architecture [16].

The RASs and history structures of the BBTB and trace table are recovered on branch mispredictions using the techniques described in Jacobson et al. [2], Skadron et al. [12] and Reinman et al. [13].

Prior work [13] has demonstrated the benefit of decoupling fetch address prediction from instruction delivery, particularly as a means to hide latency and to provide a lookahead at the future fetch stream. The fetch-prediction architecture experiences latency because of the single-cycle stalls between switching from one form of prediction to the other. Moreover, the lower bandwidth provided by the conventional (BBTB) branch-prediction architecture can be hidden by the higher bandwidth trace table. The processor can actively be consuming enqueued larger bandwidth trace predictions, whereas the conventional branch-prediction architecture covers for a trace table miss.

To achieve this, both predictors feed into a common FTQ [13], where predictions are buffered until they are consumed by the instruction delivery mechanism. The predictions share a common queue to maintain in-order semantics even when switching between predictors. Each entry in the queue can either hold a trace ID (if the prediction originated from the BBTB) or an RAS (if the prediction originated from the trace table).
in the trace table) or a starting and ending fetch address (delineating a stream of fetch addresses either from a BBTB entry or from a BBTB miss). This requires a bit in each FTQ entry to indicate whether the prediction is to be consumed by the trace cache or the instruction cache.

An alternative design might use two distinct queues, one for fetch addresses leading to the instruction memory hierarchy, and the other for trace IDs leading to the trace cache. However, this complicates the in-order delivery of instructions because either both queues must be ordered relative to one another or the fetch architecture would have to wait for one queue to completely drain before switching fetch predictors.

3.2 Accelerated instruction delivery

The instruction delivery mechanism of the architecture takes the predicted instruction stream from the trace table and BBTB and provides instructions to the execution core (Fig. 3). The architecture can support both the trace cache and the conventional instruction memory hierarchy. There may be an instruction cache present as in Rotenberg et al. [3] or absent as in Hinton et al. [4]. In the latter case, requests to instruction memory go directly to the unified L2 cache.

The trace cache and instruction memory need not have the same latency. If there is no instruction cache, and instructions must tolerate an access to the L2 cache (or further down the hierarchy), then the latency of the path accessing instruction memory will likely be longer than that of the trace cache. Moreover, because the trace cache stores predecoded instructions, the instruction memory path will also involve additional pipeline stages to perform instruction decode. This can be particularly significant in processors like the Pentium 4 [4], where the decoder bandwidth is limited. The predictor architecture can dynamically switch from trace to fetch address prediction and back, so there can be a mix of block IDs and fetch addresses entering the instruction delivery stage.

The cost of reducing the hardware on the slow path (like the instruction cache or effective decode bandwidth) is the impact on slow path performance. We consider two approaches to tolerate this latency: out-of-order fetch (OOOF) [18] and fetch-directed prefetching (FDP) [17]. The former is shown with dashed lines in Fig. 3, and the latter is shown with dotted lines. Both are optional mechanisms, which we will consider independently and together. Both also result in added pressure to the L2 cache, which we will see in the performance impact in the following sections.

3.2.1 Out-of-order fetch: OOOF [18] is an approach to avoid stalls caused by mismatched pipeline latencies between the block cache and instruction memory paths. We make use of an instruction fetch queue (IFQ) allocator that creates a place holder for the incoming fetch address or trace ID prediction in the IFQ. This placeholder is communicated to the trace cache or instruction memory paths to write the instructions to the appropriate entry in the IFQ. The allocator must therefore have some notion of how many IFQs are required by the prediction. In this architecture, the allocator is assumed to be able to compute the size of the prediction, but this could also have been communicated through the fetch-prediction architecture and stored in the FTQ. As opposed to what is stated in the work of Stark et al. [18], this mechanism must handle two distinct instruction delivery paths.

Even though delivery can be out-of-order, instructions must still be provided in-order to the execution core. If the next instruction to be provided by the IFQ is still in-flight (i.e. there is a placeholder for the instruction, but the instruction has not returned from the instruction memory hierarchy), then no further instructions are supplied by the IFQ until the in-flight instruction is ready.

There is an added complexity cost to this implementation, of course, in addition to the overhead of the allocator itself: the IFQ becomes more complex as instructions can now enter this structure out of order.

3.2.2 Fetch-directed prefetching: As in Reinman et al. [17], the stream of predictions from the fetch unit can be used to provide a look-ahead instruction prefetch mechanism. This was first explored by Chen and Baer [19] for data cache prefetching. Instruction prefetch requests for fetch addresses are directly queued into a prefetch instruction queue (PIQ) from the BBTB and are prefetched from the L2 cache (or main memory in the event of an L2 miss) into a small, 32-entry, fully associative prefetch buffer. If an instruction cache is present, these structures are probed in parallel. The instruction cache tag array can be replicated or dual-ported to provide an effective means of filtering prefetches already present in the L1 cache (cache probe filtering [17]). If no instruction cache is used, the prefetch buffer is accessed alone, potentially hiding the full latency to the L2 cache. In that case, the only form of prefetch filtering possible is using the prefetch buffer itself to avoid prefetches already present in the buffer.

The complexity of this approach includes the addition of the fully associative prefetch buffer and PIQ, along with the associated logic that arbitrates between prefetch requests and demand misses from the L1 cache(s).

FTQ entries with trace IDs do not make use of any prefetch mechanism in this paper, but a multi-level trace cache hierarchy could certainly be used with such a decoupled organisation.

4 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [20], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator performs a detailed timing simulation of an aggressive four-way dynamically scheduled microprocessor. Simulation is execution-driven, including execution down speculative paths.

To perform our evaluation, results were collected for the SPECINT CPU2000 benchmarks. The programs were compiled under Alpha OSF/1 using the DEC C and C++ compilers at full optimisation (-O4 -if0). Programs...
were simulated using the ref set for that application for 100 million instructions after fast-forwarding past the initialisation portion of the benchmark (described in Sherwood et al. [21]).

4.1 Baseline architecture

The baseline architecture is a deeply pipelined, eight-way out-of-order superscalar processor, loosely based on the architecture of the Pentium 4 [4]. There is a 20-cycle branch misprediction penalty. The processor has four integer ALU units, two load/store units, one FP adder, one integer MULT and one FP MULT/DIV. The latencies are IntALU, 1 cycle; IntMULT, 7 cycles; Integer DIV, 12 cycles; FP adder, 4 cycles; FP MULT, 4 cycles and FP DIV, 12 cycles. Functional units are fully pipelined allowing a new instruction to initiate execution each cycle. The processor has a 128-entry instruction window and a 32-entry scheduling window.

The architecture features an 8 kB four-way set-associative dual-ported data cache, with 32-byte lines. Different architectures in Section 5 use different two-way set-associative first-level instruction caches (IS), most do not have an instruction cache at all. The L2 cache is a unified 1 MB eight-way set-associative cache, with 64-byte lines and a 12-cycle hit latency. The round-trip cost to memory is 180 cycles. The L2 cache has only one port. The L1–L2 bus can transfer 32 bytes/cycle. The bus to main memory can transfer 1 byte/cycle. The L1–L2 bus is shared between instruction and data cache block requests. The architecture has a 64-entry store buffer.

The conventional branch-prediction architecture used in this paper has a 512-entry, four-way set-associative BBTB, enhanced as described in Section 3. The architecture also features a 64-entry RAS and a 16 K-entry gshare indexed via a hash of the BBTB index (head PC of the basic block) and the branch history counter.

The path-based next-trace predictor used in this paper has a 16-instruction trace size; the next-trace predictor is four-way set-associative with 2048 sets. As in the Pentium 4 [4], we pipeline the trace cache and trace table to provide one prediction every two cycles, thus providing 16 instructions every two cycles (eight instructions per cycle) to match our issue bandwidth.

Unless otherwise specified, we use the architecture described in Section 3.1 (which we will refer to as fast switching) for our baseline architecture model.

5 Results

Results for the architecture proposed in Section 3 are presented here. We start by showing the benefit of fast switching back to using the trace cache after a trace table miss.

5.1 Fast switch benefits

Fig. 4 presents results for the baseline architecture with and without our fast switching technique.

On average, fast switching improves instructions per cycle (IPC) by 139% over waiting for a misprediction. For some individual benchmarks such as bzip, perl and vortex, the benefit is even more substantial. We realise that this method (or a similar one) is absolutely necessary in order to obtain good performance out of the trace cache. We show these results to emphasise that point and use fast switching in our baseline architecture for the rest of our experiments.

5.2 8 k-entry trace table results

The Pentium 4 [4] uses a slow path without an instruction cache. In this section, we consider the use of our accelerated instruction delivery mechanism described in Section 3.2 to improve the performance of an architecture that eschews the use of an instruction cache.

The following architectures are examined in the remainder of this section:

- **Base**: the Base architecture uses the fast switching architecture described in Section 3.1 to switch freely between the trace table and the BBTB. There is no OOOF and no instruction prefetching.
- **OOOF**: the OOOF is the Base architecture with the OOOF mechanism described in Section 3.2.1. There is no instruction prefetching.
- **FDP**: the FDP is the Base architecture with a FDP mechanism described in Section 3.2.2. There is no OOOF.
- **Both**: the Base architecture with OOOF as well as FDP.

Architectures with an instruction cache are annotated with the size of the cache (i.e. 8 or 2 kB); otherwise, there is no instruction cache.

Fig. 5 presents the IPC results for the earlier-mentioned architectures. These results show the need and the benefit in improving the slow path when there is no instruction cache. On average, OOOF provides a 9% improvement in IPC over the Base architecture. This is because we can continue to grab instructions from either the L2 or the trace cache in the face of the latency that is incurred when going to the L2. Using FDP gives an average 10% IPC speedup over the Base architecture. This method hides the latency of going to the L2 by accurately prefetching the slow path fetch stream. When using Both, there is an average 14% increase in IPC from Base, and a 3% increase over FDP. This shows that there is a great deal of latency that needs to be tolerated when accessing the L2S. Adding an 8 kB instruction cache improves the IPC of using Both by another 4%.

### Fast Switch Improvement

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Fast Switch OFF</th>
<th>Fast Switch ON</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>1.5</td>
<td>3.0</td>
</tr>
<tr>
<td>crafty</td>
<td>2.0</td>
<td>2.5</td>
</tr>
<tr>
<td>eon</td>
<td>1.0</td>
<td>1.5</td>
</tr>
<tr>
<td>gap</td>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>gzip</td>
<td>1.0</td>
<td>1.5</td>
</tr>
<tr>
<td>mcf</td>
<td>1.5</td>
<td>2.0</td>
</tr>
<tr>
<td>parser</td>
<td>2.0</td>
<td>2.5</td>
</tr>
<tr>
<td>perl</td>
<td>2.5</td>
<td>3.0</td>
</tr>
<tr>
<td>twolf</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>vortex</td>
<td>3.5</td>
<td>4.0</td>
</tr>
<tr>
<td>vpr</td>
<td>4.0</td>
<td>4.5</td>
</tr>
<tr>
<td>average</td>
<td>2.5</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Fig. 4 Fast switch improvement
Different applications experience different benefits from OOOF and FDP, and although there is some overlap in their benefit, Both does provide more benefit than either approach alone. Benchmarks like crafty and eon see more benefit from OOOF, improving 15% and 11%, respectively, over Base. These benchmarks see 8 and 5% benefit, respectively, from FDP, but 18% and 14% benefit, respectively, from Both. OOOF targets switch between paths and access latency. OOOF helps frequently switching applications by allowing the fast path or slow path to continue in the face of slow path latency. So long as there is sufficient occupancy in the IFQ, the latency of the slow path access is hidden.

Most of the benchmarks see more benefit from FDP than form OOOF, such as gzip, which sees 14% improvement from FDP and 5% improvement from OOOF. FDP targets access latency and can begin to hide the latency of consecutive cold path accesses even before they reach the instruction delivery mechanism (i.e. while the predictions are still in the FTQ). FDP is able to get further ahead of OOOF using the decoupling provided by the FTQ.

We continue to analyse the hardware needed on the slow path by varying the trace table size.

5.3 Tolerance to trace table size

The graph in Fig. 6 shows how our accelerated instruction delivery mechanisms adapt to the increased slow path pressure when the trace table size is varied. As shown in Fig. 6, we vary the number of sets in our four-way set associative trace table. A four-way trace table with 256 sets would have 1 k entries. We consider the same architecture as in Section 5.2, but also add measurements for a 2 kB instruction cache using Both, as well as 8 and 2 kB instruction caches using the Base architecture. The data presented is the average of all the SPEC INT 2000 benchmarks. Note that the y-axis is highly scaled (ranging from 1.1 to 1.45 IPC) to better show and differentiate each curve, but at the price of exposing noise in the results.

The benefit gained by increasing the trace sets from 256 to 512 is significant, and scaling the number of trace sets beyond 4096 does not show any further performance improvement.

Both (without an instruction cache) is the same or slightly better than using a 2 kB IS with Base. The difference in improvement when going from a 2 kB IS to an 8 kB IS with Base is much greater than going from 2 kB IS to an 8 kB IS using Both. Moreover, the 2 kB Both case consistently performs better than the 8 kB Base case. These observations point to the fact that our techniques can really push the performance of adding even a small IS, without the need to add a larger IS.

5.4 Tolerance to instruction cache size

In Fig. 7, we continue to explore results using the labels defined in Section 5.2. The x-axis now represents the size of our L1 IS. The average results from the SPEC INT 2000 benchmarks are shown.

As stated in Section 5.3, one of the benefits of using Both is the amount of performance it can drive out of a small IS. The Base performance improves around 8% from no IS to a 2 kB IS and continues to increase at a noticeable rate. The Both curve also jumps from no IS to a 2 kB IS (though not as much as in the Base case, only around 3%), but benefit is minimal as the IS size continues to increase.

This figure stresses the ability of our accelerated delivery mechanism to make a slow path without an instruction cache a viable alternative. Both without an instruction cache at all is able to come within 2% of the performance of the Base 8 kB cache, and within 4% of the performance of the Base 32 kB cache.

However, a large instruction cache sees little benefit from the acceleration, as the cold path does not have sufficient pressure to justify the larger instruction cache footprint that could be tolerated. Future work could explore the impact of hyperthreading [22] on the cold path, as this may greatly increase the amount of state required when sharing fetch resources.
5.5 Latency tolerance

As the processor clock speeds continue to increase, the cycle penalty incurred for going to the L2 cache will continue to grow. In Fig. 8, we show how our system gracefully tolerates increasing latency to the L2$^2$. The experiments are all run without an IS and we plot the average results from the SPEC INT 2000 benchmarks.

Fig. 8 demonstrates that using Both with a long latency L2 gives slightly better performance (around 2% improvement) when compared with Base using an L2 cache with half the latency. As latency increases, the IPC for Both degrades more gracefully than IPC for Base. For the sizes we consider, Both degrades by 4, 14, and 27% relative to the 12-cycle L2 latency. Base degrades by 10, 24 and 42% relative to the 12-cycle L2 latency. Using either OOOF or FDP alone with a longer latency L2 performs slightly worse than Base using an L2 cache with half the latency. But both of these approaches are better able to scale to longer latencies when compared with Base. We can therefore use our techniques as the memory wall increases to hide the latencies associated with using the slow path.

6 Conclusion

The integrated trace cache and BBTB architecture presented in this paper features the ability to dynamically switch between fetch-prediction mechanisms. We demonstrated how this approach could be coupled with OOOF and FDP. These techniques used to accelerate instruction delivery enabled this architecture to tolerate longer latency in the conventional memory hierarchy, even in the absence of a first-level instruction cache.

For an architecture without an instruction cache, the combination of both of our acceleration techniques provide an average 14% improvement in IPC over a baseline architecture. And we are able to come within 2% of the performance of a baseline architecture with an 8 kB instruction cache. The ability of the architecture to tolerate latency increases even in the face of lower trace table coverage and in cases of longer L2 latency.

The architecture presented provides comparable performance to a similar processor with an instruction cache, but without the added hardware cost of the L1 instruction cache. However, there is an energy tradeoff here because of the increased L2 pressure through FDP. Future work will continue to explore this design space to determine the best combination of trace table size, BBTB architecture, instruction memory hierarchy support, predictor bandwidth and frequency and decoder bandwidth.

7 References