Protean: Resource-efficient Instruction Prefetching

Muhammad Hassan
Dept. of IT, Uppsala University
Uppsala, Sweden
hassan.muhammad@it.uu.se

Chang Hyun Park
Dept. of IT, Uppsala University
Uppsala, Sweden
chang.hyun.park@it.uu.se

David Black-Schaffer
Dept. of IT, Uppsala University
Uppsala, Sweden
david.black-schaffer@it.uu.se

ABSTRACT

Increases in code footprint and control flow complexity have made low-latency instruction fetch challenging. Dedicated Instruction Prefetchers (DIPs) can provide performance gains (up to 5%) for a subset of applications that are poorly served by today's ubiquitous Fetch-Directed Instruction Prefetching (FDIP). However, DIPs incur the significant overhead of in-core metadata storage (for all workloads) and energy and performance loss from excess prefetches (for many workloads), leading to 11% of workloads actually losing performance. This work addresses how to provide the benefits of a DIP without its costs when the DIP cannot provide a benefit.

Our key insight is that workloads that benefit from DIPs can tolerate increased Branch Target Buffer (BTB) misses. This allows us to dynamically re-purpose the existing BTB storage between the BTB and the DIP. We train a simple performance counter based decision tree to select the optimal configuration at runtime, which allows us to achieve different energy/performance optimization goals. As a result, we pay essentially no area overhead when a DIP is needed, and can use the larger BTB when it is beneficial, or even power it off when not needed.

We look at our impact on two groups of benchmarks: those where the right configuration choice can improve performance or energy and those where the wrong choice could hurt them. For the benchmarks with improvement potential, when optimizing for performance, we are able to obtain 86% of the oracle potential, and when optimizing for energy, 98% of the potential, both while avoiding essentially all performance and energy losses on the remaining benchmarks. This demonstrates that our technique is able to dynamically adapt to different performance/energy goals and obtain essentially all of the potential gains of DIP without the overheads they experience today.

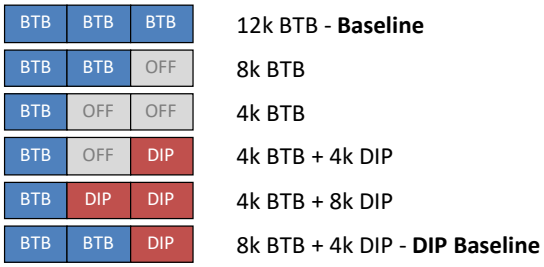


Figure 1: Our proposed dynamic reconfiguration of the BTB to share space with a Dedicated Instruction Prefetcher (DIP) or save energy. We assume a 12k-entry, 3-bank baseline BTB storage, leading to 6 configurations.

1 INTRODUCTION

Several studies have shown that the increasing code footprint of datacenter workloads cause frequent instruction cache misses [8, 9, 16, 18, 25, 50, 54, 59]. Even with well-provisioned frontends (effective Branch-Prediction-Unit (BPU), large BTBs and I-caches, etc.), instruction supply often falls short for modern server and cloud applications. Dedicated Instruction Prefetchers (DIPs)[1] have been proposed to alleviate this bottleneck, and a plethora of works has explored both hardware and software techniques. Common hardware based approaches include record and replay based prefetchers ([6, 9, 10, 17, 27, 28, 31, 49, 50]) and branch predictor directed prefetchers ([7, 12, 34, 35, 43, 58]), while software based techniques often use profiling to change the code layout ([5, 39, 40, 44]) or insert prefetch instructions ([4, 9, 29, 37, 38]). While there has been extensive work on DIPs, it is crucial to evaluate them together with the baseline instruction prefetcher that already exists in nearly all modern processors.

Essentially all modern processors employ *decoupled frontends* for effective instruction prefetch [1, 3, 11, 15, 20, 41, 56]. This Fetch Directed Instruction Prefetching (FDIP) [46–48] leverages the existing Branch Predictor Unit (BPU) and BTB to run-ahead and identify future basic blocks to prefetch into the L1I. As modern processors include large BTBs and highly accurate BPUs, FDIP is extremely effective for the majority of applications, and particularly those that fit in the existing frontend structures. Further, as FDIP leverages the existing BTB and BPU resources, it requires very little additional metadata storage compared to most DIPs.

The Potential and Peril of DIPs. While today's well-provisioned frontends make FDIP highly effective in most cases, workloads with particularly large instruction footprints can overwhelm the BTB and BPU and make FDIP ineffective [8, 9, 25, 54]. In these cases, DIPs have been shown to deliver performance benefits on top of FDIP ([6, 28, 33–35, 49]). According to our simulations of 2122 benchmark traces, a DIP can provide more than 2% performance improvement for 1% of the benchmarks, with a maximum benefit of 5%. However, these benefits come with the cost of extra in-core metadata storage (area), cache pollution and contention (performance), and excess prefetch requests (energy). Specifically, while we observe that 5% of benchmarks see a >1% performance improvement, almost 12% suffer from >1% performance loss, and 33% see a >1% energy increase. In addition, the 84% of benchmarks that see no performance/energy gains or losses still pay the in-core area overhead. These results show that with a representative (aggressive) FDIP baseline [23, 24] and including wrong path instruction prefetching, the overall benefit of DIPs is limited, as are the number of benchmarks that benefit. However, as pointed out in many studies, instruction footprint is growing, and this gives reason to

[1]We use "dedicated" to contrast with the "integrated" default prefetching that the BTB and BP provide through FDIP, and to identify the large class of proposed designs that only do instruction prefetching.

believe that the performance difference between FDIP and DIP will increase in future workloads.

Goal. Our goal is to provide the performance and energy benefits of DIPs for those workloads that benefit from them while avoiding the costs (performance, energy, and area) for the majority of workloads that do not. To achieve this goal, we introduce Protean, which dynamically allocates the *existing* BTB storage between the BTB (to support the baseline FDIP) and the DIP metadata (for applications where the DIP is effective), or power gates it to save energy (for applications that do not require the large BTB capacity). (See Figure 1.) This allows us to match the instruction prefetching needs of the application or program phase[2] without extra storage or the dynamic costs of the DIP when it is not beneficial.

Insight. We observe that most benchmarks that benefit from a DIP can tolerate BTB misses owing to Post Fetch Correction (PFC) *of direct branches*. This is because PFC [24] is able to re-steer the frontend early on a BTB miss to a direct branch, since direct branches contain the target address encoded in the instruction. As a result, such BTB misses do not wait until the execute stage for re-steering and only need to flush the Fetch Target Queue (FTQ), not the ROB and pipeline, making them far less performance critical. We observe that applications that benefit from a DIP are also insensitive to increased (direct) BTB misses for this reason, making it effective to share BTB capacity.

Solution. Protean dynamically allocates the existing BTB storage between BTB and DIP metadata or power-gated modes to obtain the best benefit according to the specified performance/energy optimization goal. For our baseline design with a three-bank BTB, we can choose from among the six configurations shown in Figure 1. To do so, we need: compatible BTB and DIP metadata layouts, to allow us to share the same storage; a mechanism that enables us to define a desired performance/energy goal; and a method to dynamically and efficiently choose the correct configuration at runtime. Our final design uses a scoring metric that enables us to vary the desired performance/energy optimization goal, which, in turn, allows us to train a machine learning algorithm to choose the correct configuration for the goal at runtime. The result is a system that can efficiently and accurately share BTB storage between the BTB and DIP to achieve a given performance/energy optimization goal, and thereby achieve nearly all of the potential benefits of DIPs, while avoiding nearly all of their cost where they do not provide a benefit.

Our work makes the following contributions:

- We show that BTB and instruction prefetching interact in a useful way: benchmarks that require DIP can tolerate direct BTB misses due to effective Post Fetch Correction.
- We show how the storage used for the BTB can be repurposed for storing DIP metadata, leading to a range of BTB/DIP configurations for essentially the same area cost as today's standard frontend.
- We propose a scoring methodology that allows us to evaluate and train for a range of performance/energy goals.
- We show how a decision tree can be used to accurately choose the best BTB/DIP configuration for a range of performance/energy goals.
- We demonstrate that this system is able to achieve nearly all of the potential DIP performance/energy benefits for the applications where there is a potential gain, while avoiding nearly all of the potential losses.
- We note that while our work enables the limited benefits available from DIPs today, it also provides a framework for obtaining the benefits of future DIPs and addressing future workloads by simply re-training the decision tree.

[2]Hence Protean, referring to the shape-shifting Greek god, Proteus.

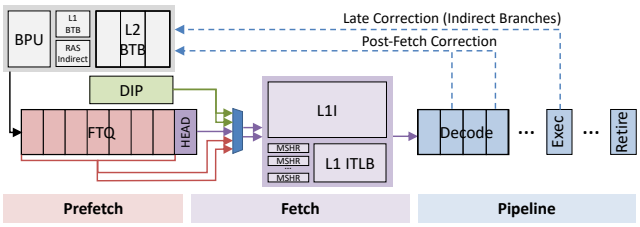


Figure 2: Baseline design with the Post-Fetch-Correction (PFC) in the decode stage. The head of the FTQ (demand), FDIP prefetches, and DIP prefetches share 2 ports to the instruction cache.

2 BACKGROUND

We start by introducing FDIP (Section 2.1) and demonstrating how FDIP provides nearly all of the potential benefits, which corroborates previous results (Section 2.2). We then look at how a state-of-the-art dedicated instruction prefetcher (DIP) can improve on FDIP (Section 2.3), and see that it pays significant penalties in area, energy, and performance for the majority of applications where it provides no benefit (Section 2.4). This background motivates our goals of obtaining the benefits of the DIPs without their costs, which we then address in the remainder of the paper.

2.1 Fetch Directed Instruction Prefetching

Fetch-Directed Instruction Prefetching (FDIP) is a ubiquitous technique that takes advantage of the Branch Prediction Unit (BPU) and the Branch Target Buffer (BTB) to predict future control flow and run-ahead using that information to issue instruction prefetches [46–48]. Essentially all current high-performance processors employ a variant of FDIP, including IBM [1], AMD [11], Samsung [55], Arm [3], Intel [42], Marvell [56], etc.

FDIP places predicted instructions in a Fetch Target Queue (FTQ, see Figure 2), which decouples the generated instruction prefetches from instruction fetch, and enables the branch predictor to run-ahead during fetch stalls from instruction cache misses. While the FTQ logically holds predicted instructions, in practice they are stored at a coarser granularity, e.g., 32 byte [24] or 64 byte [32] basic blocks. Because FDIP leverages the existing frontend resources it is almost metadata-free (requiring only 186 bytes for the FTQ [24]) and benefits from improvements to the BPU and BTB.

With FDIP, the pipeline takes the entry at the head of the FTQ as a *demand* fetch while the other entries in the FTQ are issued to the instruction cache as *prefetches*. As long as FDIP can accurately run far enough ahead, basic blocks are likely to have been prefetched into the instruction cache by the time they reach the head of the

FTQ, thereby resulting in a cache hit at demand fetch. To avoid repeated cache searches when entries reach the head of the FTQ, Ishii et al. [24] propose issuing all FTQ entries as demand loads into the instruction cache such that the loaded lines are locked, and then storing the resulting cache way in the FTQ's metadata. The way information then allows them to avoid a tag search when the FTQ entry makes it to the head.

One challenge with FDIP is how far ahead to prefetch. Since the predicted control flow accuracy decreases with each predicted branch, prefetching further ahead increases the likelihood of wrong path prefetching, leading to cache pollution and port contention. To address this, Reinmann et al. advocated limiting prefetching to 10 basic blocks [47] while Perais et al. limit in-flight prefetches to 4 [43].

One particularly important FDIP detail is Post Fetch Correction (PFC) [24], which re-steers the frontend as early as possible, thereby minimizing the impact of direct BTB misses and branch-mispredictions. PFC detects if an unconditional *direct* branch is mispredicted as not-taken or if a predicted taken direct branch misses in the BTB (dashed arrow in Figure 2 from decode). These cases can be detected early in decode/pre-decode stage, which means that only the FTQ needs to be flushed after such a miss, as opposed to waiting until branch resolution in the execute stage, which requires costly flushing of ROB entries. This is particularly important to model, as BTB misses and mispredicted direct branches appear far more costly without PFC, which can result in an over-emphasis of the impact of BTB size and BTB prefetching.

2.2 FDIP Delivers Excellent Prefetching

Current processors with large BTBs and highly accurate branch predictors provide a solid foundation for FDIP. Figure 3 shows that a realistic FDIP implementation [24] comes very close to a state-of-the art DIP, the Entangled Instruction Prefetching (EIP) [49], and even does well against an infinite capacity instruction cache. Indeed, the majority of our 2122 benchmark traces show that FDIP is within 1% of the ideal instruction cache (51%) or state-of-the-art DIP (95%). As discussed by Ishii et al. [24], this demonstrates how effective FDIP implementations are, but also shows that there is a small subset (5%) of applications where a DIP is beneficial. Our work not only enables the benefits of DIPs for this subset today, but also provides a framework for leveraging future DIPs and for addressing future workloads through simply re-training our decision trees.

2.3 EIP: A Dedicated Instruction Prefetcher

While our approach is independent of any particular DIP, we evaluate our approach with the winner of the 1st Instruction Prefetching Championship [22], the Entangled Instruction Prefetcher (EIP) [49].

EIP is a state-of-the-art Dedicated Instruction Prefetcher that has been shown to deliver considerable performance improvement over FDIP baseline for a subset of applications [49]. EIP is a "record and replay" based DIP, which works by finding earlier basic blocks and using them to trigger timely prefetches for later misses. EIP achieves this by keeping a history buffer of the start addresses of recent basic blocks and the time at which they were demanded. On an instruction cache miss, EIP uses the history to find a basic block that was sufficiently far before the miss, and then "entangles" the

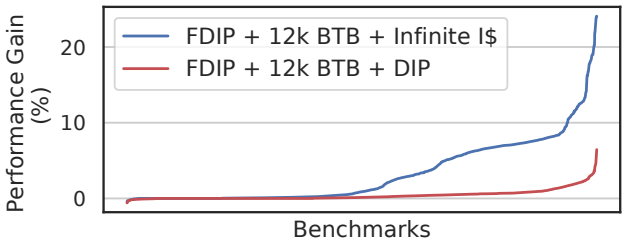


Figure 3: Benefits of an infinite instruction cache or a state-of-the-art DIP over the baseline FDIP. Across our 2122 benchmark traces we see that FDIP instruction prefetching is within 1% of an infinite instruction cache for 51% of the benchmarks and within 1% of a state-of-the-art DIP for almost 95% of the benchmarks. This makes it hard to justify the area and dynamic costs of a DIP by itself.

start address of the earlier basic block with the desired prefetch address via the Entanglement mapping table. For demand access at the head of the FTQ, EIP checks the entanglement table to see if there are any entangled prefetches and issues them. If a prefetch is late, EIP finds an even earlier basic block in the history and entangles the prefetch with it instead. To improve efficiency, EIP uses a compressed storage format that allows multiple cache lines to be entangled to each basic block and stores the length of the basic block for better prefetching coverage. EIP uses this information to both issue prefetches for the subsequent cache lines in the current basic block and for basic blocks which are entangled to the current basic block.

EIP has two main advantages over FDIP. First, because EIP is able to adjust how early it initiates prefetch requests over a very large range, as opposed to FDIP, which can only run ahead in the predicted program flow, it is able to generate prefetches far earlier. This allows it to schedule prefetches earlier enough to handle BTB misses or BPU mispredictions when the frontend is stalled. Second, because it uses a separate metadata storage (the entanglement table), EIP is not affected by the BPU and BTB capacity limitations that FDIP suffers from for applications with large code footprints. As a result, the state-of-the-art EIP provides a maximum of 5% improvement, with >1% and >2% improvements for only 5% and 2% of benchmarks, respectively[3].

2.4 The cost of a DIP

Although EIP is effective for a some applications, it also entails significant costs, particularly for the applications that see no benefit on top of the FDIP baseline. These costs include increased energy consumption due to excess prefetches and performance loss due to cache pollution and data cache misses being delayed due to port contention. The complexity of this trade-off is shown in Figure 4. We see that almost 5% of our 2122 benchmark traces show >1%

[3]While EIP is the best-performing DIP, we note that our performance numbers are lower than what was reported in the original paper [49]. This is largely due to our using a baseline that implements PFC, has a larger BTB (12k-entry vs. 8k-entry) and larger FTQ, a more competitive BPU, and a simulator that models wrong path instruction-prefetching.

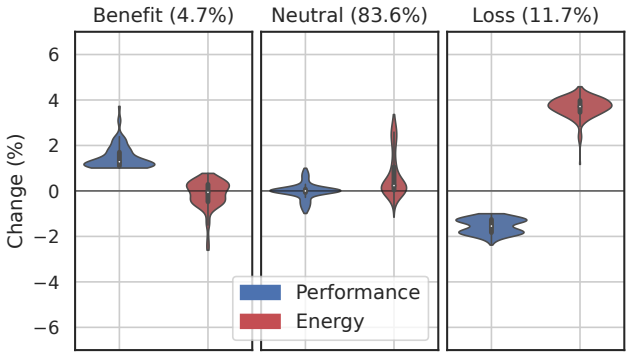


Figure 4: The impact of a specific DIP (the Entangled Instruction Prefetcher) normalized to the FDIP baseline across our 2122 benchmark traces on performance (blue) and energy (red). Left: For the 5% of benchmarks that exhibit an performance gain of >1% with the DIP. Middle: For the 84% of benchmarks that have less than a 1% performance change. Right: For the rest 11% of benchmarks that show a performance loss of >1%.

performance gain with EIP (left), while almost 12% show a performance loss of >1% (right). The remaining 84% (middle) see no loss or benefit with EIP. For energy, the impact ranges from a savings of 0.15% (up to 2.7%) for those with a performance benefit (left) to an increase of 3.6% (up to 4.5%) for those with a performance loss (right). And for the vast majority which do not benefit from EIP (middle), we see an energy increase of 0.6% (up to 3.3%). This demonstrates that while EIP can benefit a subset of applications, overall it is hard to justify its area and dynamic performance and energy costs. (See Section 7 for simulation details.)

While the dynamic performance and energy costs of EIP could be avoided by deactivating it when not needed, the area cost is fixed. The proposed 4K-entry entanglement table (which represents 97% area overhead of EIP) requires roughly 40KB of metadata storage [49]. As few applications benefit from EIP, an industrial research team (Ishii et al. [23, 24]), concluded based on an ISO-area comparison that a DIP, such as EIP, on top of FDIP is difficult to justify. It is further worth noting that not all area in a processor is of equal cost: adding a 40KB of metadata to a multi-MB cache is indeed a small overhead, but adding a similar amount inside the frontend of the core is a much more daunting challenge from a timing point of view. As DIPs such as EIP are tightly integrated into the core, and they benefit only a small set of applications, this makes them hard to justify in practice[4].

2.5 Challenge: The Benefits of a DIP without its cost

The preceding analysis of FDIP and DIP shows us that DIPs can provide a meaningful benefit for a subset of applications but that they come with a significant dynamic and static cost. This leads us to the two main questions of this work:

[4]We are unable to find a published reference to this and can only relate that the authors have had discussions with industrial design teams in multiple companies that have emphasized this point.

BTB Metadata Storage Format

Tag	PID	Type	Target	SRRIP	Conf.
12	1	3	57	3	2-3

DIP Metadata Storage Format (EIP)

Tag	BB Size	Format, Destination, Confidence
10	6	63

Figure 5: BTB [53] and EIP [49] metadata storage formats are largely compatible.

(1) Can we avoid the *static area costs* by sharing existing frontend metadata storage? (Section 3)
(2) Can we obtain the benefits of DIP without its *dynamic performance and energy costs* by dynamically enabling/ disabling it at runtime? We take a step further to identify the best configuration when enabling/disabling the DIP (Section 4).

3 AVOIDING THE STATIC COSTS

To reduce the static area cost of a DIP, we propose sharing metadata storage with another frontend structure. For this to work, we need to identify a structure that is compatible (in terms of the physical storage format) and complementary (i.e., not needed at the same time). There are a range of frontend storage structures one could consider: TLBs, BPU, BTBs, etc. The first-level TLBs and BTBs are latency-critical, and, even though the logic changes needed to dynamically re-purpose their storage are small, doing so might have a significant impact on the cycle time. First-level TLBs also have low capacities, for example, 256 4kB pages in Intel's Alder Lake [2], and translation is still required to issue FDIP prefetches. The BPU is also latency-critical and its costs of mispredictions are high as they are detected late in the execute stage and require ROB/FTQ flushes.

Sharing the second-level BTB storage, however, is more promising. Not only are L2 BTBs larger (up to 12k entries in Intel's Alder Lake [2]), but there is reason to believe that their performance is complementary to that of a DIP. This is because a DIP is needed when FDIP is unsuccessful, and, when that happens, it is either because the BTB or the BPU are not effective. While this might suggest sharing the BPU storage, the cost of BTB misses are far lower than that of BPU misses, as they can often be corrected early in the pipeline with PFC.

3.1 Are BTB and DIP data compatible?

To dynamically share the BTB storage with a DIP, the existing BTB storage must have a readout width, associativity, and capacity that is compatible with those required for the DIP. We examine EIP and focus on sharing storage for its entanglement table, as it is almost 98% of its metadata storage[5]. While the exact details of BTB implementations are proprietary, we compare a typical BTB [53] to EIP. We assume a baseline 12-way 12k-entry BTB, as in Intel's Alder Lake, built of three 4-way, 4k-entry banks, with each bank

[5]EIP also proposes small changes to the instruction cache, but can function at lower performance without them. We model EIP's history buffer in our evaluation, but as it is only 1KB in storage, we do not propose sharing it with other frontend resources.

separately configurable as either DIP metadata storage, BTB storage, or power-gated. This is consistent with Intel's recent designs which allow adjusting BTB capacity on-demand [21].

Figure 5 shows that the BTB entry is 79 bits, which is compatible to the EIP entanglement entry. However, we see that the number of tag bits varies, indicating that the lookup logic (indexing and tag comparisons) will require a configurable shifter on the index and tag comparators. While the rough width of each entry is a good match, specific designs may end up wasting a few bits of each entry if they do not align perfectly. This significant storage compatibility indicates that it is plausible to dynamically allocate the three storage banks between BTB storage, DIP metadata storage, and powered-off states. As we assume a minimum of 4k-entries for BTB storage, we end up with the six configurations shown in Figure 1. Alternatively, one could allow DIP and BTB entries to compete for space within the banks, although we have not investigated this.

While adding reconfigurability to the BTB will incur some overhead, it is unlikely to significantly affect the critical path as we are modifying the 2nd-level BTB and Intel has already demonstrated similar reconfigurability. If added latency did become an issue, we could readily support a design where one bank (the one is always assigned to BTB entries) retains the baseline latency, as it is never reconfigured. This would allow our decision tree (Section 5.1) to choose a low-latency configuration as appropriate.

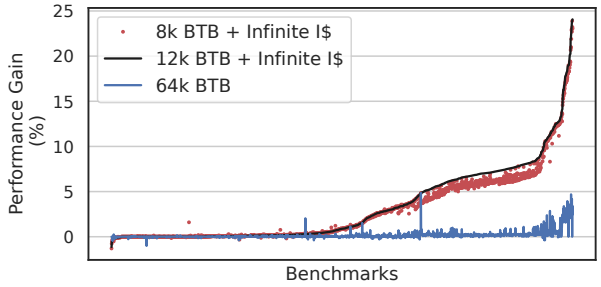
3.2 Is the BTB complementary to the DIP?

For the performance loss of sharing the BTB to be minimal, we need the effects of the BTB and the DIP to be complementary. That is, the performance loss of reduced BTB capacity must be negligible when the DIP is beneficial. To investigate this, we look at the impact of reducing the BTB capacity on the *potential benefit* of improved instruction delivery: e.g., how much the potential benefit a DIP could deliver would be reduced by a smaller BTB. Specifically, we use an infinite instruction cache with our baseline 12k BTB to show the limit of what an ideal DIP could achieve over FDIP, and consider whether the applications that could benefit from a DIP suffer from reduced BTB capacity.

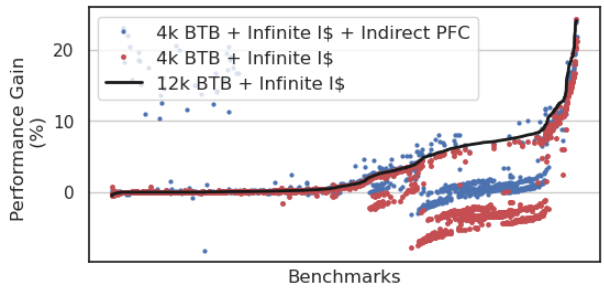
Figure 6a shows that approximately half of our benchmarks could benefit from a DIP as their performance goes up with the infinite instruction cache, e.g., the black line is above zero. More interestingly, the vast majority of the benchmarks see little loss from reducing the BTB capacity from 12k to 8k entries (red dots are close to the black line). However, Figure 6b shows that some benchmarks suffer significantly from reducing the BTB further to 4k entries (red benchmarks below 0 on the right). This suggests that using 4k entries of BTB capacity for a DIP (e.g., 8k BTB + 4k DIP) will not significantly reduce the potential benefit, but that there are many benchmarks for which giving up 8k entries may be problematic (e.g., 4k BTB + 8k DIP). We also explored the potential benefits of a larger 64k-entry BTB directly in Figure 6a, but found that there was more to be gained from better instruction fetch.

To understand why some benchmarks are particularly sensitive to the 4k BTB vs. the 8k BTB, we look at the impact of the type of BTB misses they experience. We observe that benchmarks that see a significant loss in performance potential with a smaller BTB do so because they have significantly more *indirect branch BTB misses* with the smaller BTB size (red dots on Figure 7, left). This leads to a performance loss, as the PFC can eliminate most of the performance loss from *direct branch BTB misses*, but not the *indirect* ones. We confirmed this by modifying the simulator to provide the same PFC benefit for indirect BTB misses (which is not possible to implement), and observed that about half of the performance loss is recovered (blue dots in Figure 6b).

What is most interesting to note is that half the performance loss is from the indirect BTB misses, despite the fact that the direct BTB misses increase enormously more with the smaller BTB size. Specifically, Figure 7 shows that the indirect MPKI for the benchmarks that are sensitive (red dots) increases by 1 while the direct



(a) Reducing the BTB to 8k-entries (red) has little impact on the potential benefits of improved instruction fetch (black).



(b) Many benchmarks see a significant loss with a 4k-entry BTB (red), but providing PFC for indirect branches addresses most of this (blue).

Figure 6: Exploring whether BTB and DIP effects are complementary. For most applications (top, red dots) reducing the BTB size to 8k-entries has very little impact on the potential improvements of better instruction fetch (infinite instruction cache, black line). Reducing the BTB size to 4k-entries causes a significant number of benchmarks to no longer be able to reach that potential (bottom, red dots below 0). By applying PFC to indirect branches, which is not realistic, we see that those benchmarks regain half of their potential (bottom, blue dots). This, combined with the data showing the dramatically greater increase in direct BTB misses in Figure 7, indicates that much of our ability to re-purpose the BTB storage for DIP without losing performance potential is due to the benefit of effective PFC. We also see that the benefits of increasing BTB size alone are far smaller (top, blue line).

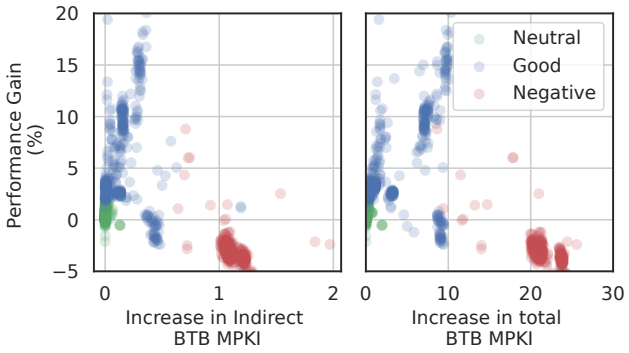


Figure 7: Increase in indirect (left) and overall (right) BTB MPKI for benchmarks with an infinite instruction cache and 4k BTB over the baseline of 12k BTB and normal instruction cache. Red dots show benchmarks sensitive to reducing the BTB capacity, from Figure 6b.

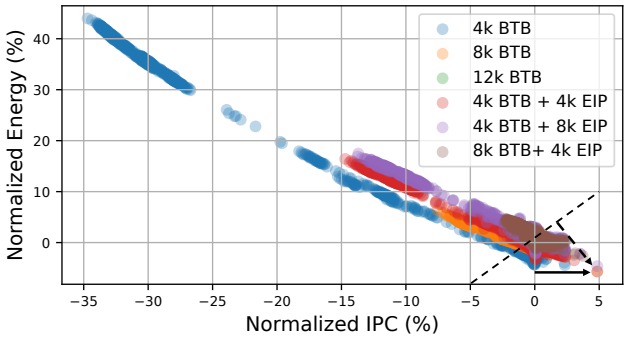


Figure 8: Configuration space normalized to the 12k BTB FDIP baseline with 16 MSHRs. The score for the furthest red point shown for a performance-only objective function (solid black line) and an equal performance-energy objective function (dashed black line).

BTB MPKI increases by over 20. And yet, both indirect and direct contribute a similar amount to the performance loss. This indicates that our ability to re-purpose the BTB for DIP is heavily enabled by PFC's ability to cheaply correct direct BTB misses and indicates that L1I hits are more important than BTB hits, as re-steering can happen quickly even if the FTQ goes down the wrong path. Finally, this suggests that for the applications that suffer increased indirect branch misses, the BTB size should not be decreased, as these applications are limited by the BTB and not the cache.

4 AVOIDING THE DYNAMIC COSTS

To explore the potential of dynamically sharing metadata storage between the BTB and a DIP, we first look at the normalized performance and energy for each benchmark and configuration in Figure 8. We see that there is no single configuration (color) that provides the best performance (right) *and* energy (bottom) for all benchmarks. This demonstrates that the optimal configuration varies depending on the benchmark and the desired performance-energy trade-off.

To choose the best configuration we need to define an objective function that allows us to trade-off performance and energy. In this work we use the perpendicular distance to a configuration from a line through the origin of the performance-energy space as the *score*. Example scores are shown in Figure 8 by the solid black line (Performance-only goal, e.g., distance in the performance dimension only) and the dashed line (Balanced performance/energy goal, e.g., distance equally in the performance and energy dimensions). The further the configuration is to the lower-right of the line, the higher the score. This allows us to change the performance-energy *optimization goal* by simply varying the slope of the line as shown in Figure 9 from 4:1 (**Performance** over energy) to 1:1 (**Balanced** energy equal to performance) to 1:4 (**Energy** over performance). Figure 9 shows that the configurations with the best scores vary with the goal.

As previously discussed, only a minority of the benchmarks see a *performance increase* with a DIP. More generally, only a minority of the benchmarks see a potential *score increase* for a given optimization goal, and this subset varies with the goal. For example, Figure 9 shows that if we define *sensitive* benchmarks as those where the best configuration choice can increase the score by more than 1.0 (dashed line)[6], only 8% of the benchmarks are sensitive for the Performance goal, while that number is 68% for the Energy goal, and 66% for the Balanced goal. Conversely, a large number of benchmarks can be severely impacted by bad configuration choices. These configurations are to the upper-left of the dashed lines in Figure 9, and continue further in Figure 8.

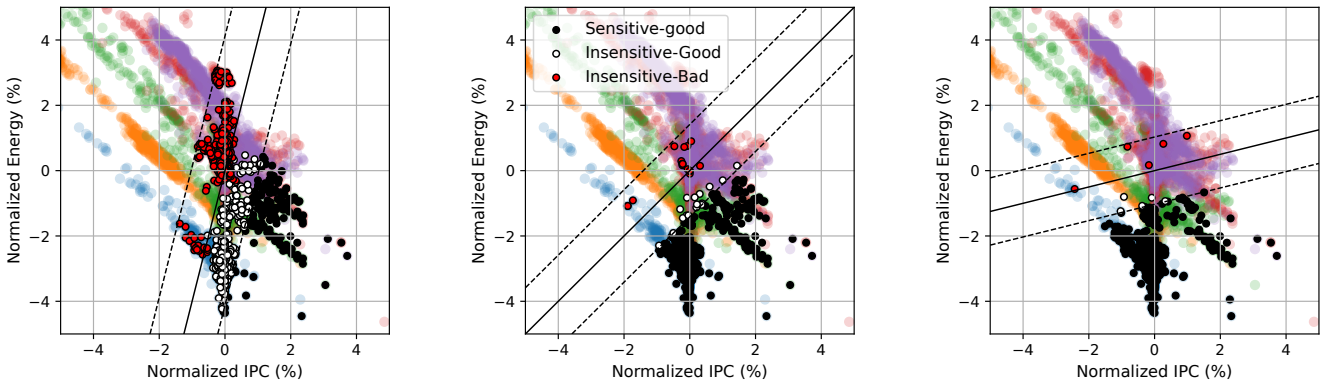
We define three groups of benchmarks: *Sensitive-good* have a best configuration that improves their score by 1.0 or more, *Sensitive-bad* have a worst configuration that hurts their score by more than 1.0 and no configuration that improves the score by more than 1.0, and *Insensitive* have a best configuration that improves the score by less than 1.0 and a worst configuration that hurts the score by no more than 1.0. We found that 41/123/193 of the Performance/Balanced/Energy Sensitive-good applications (2-10% of total) also had a configuration that could hurt their score by more than 1.0.

Table 1: Benchmark sensitivities by optimization goal

	Sensitive-good	Insensitive	Sensitive
Performance	171 (8%)	1212 (57%)	739 (35%)
Balanced	1387 (65%)	16 (0.75%)	719 (34%)
Energy	1443 (68%)	5 (<0.3%)	674 (32%)

The distribution of benchmark sensitivity for the different optimization goals is shown in Table 1. The Performance goal has many fewer Sensitive-good benchmarks as the slope of its trade-off line (Figure 9a) is steep enough that the large number of 4k BTB configurations (blue points) fall close within the dashed lines, leading to more benchmarks being classified as Insensitive. As the slope of the

[6]The meaning of the score changes with the optimization slope, but in the performance-only or power-only extremes it would be percent change in IPC or energy, respectively.



(a) Performance (4:1 Perf:Energy)

(b) Balanced (1:1 Perf:Energy)

(c) Energy (1:4 Perf:Energy)

Figure 9: Impact of optimization goal on configuration choice. Performance/Energy trade-off is shown by the slope of the lines. *Insensitive* benchmark's best and worst configurations are marked in white and red respectively. The best configuration for the *Sensitive-good* applications are marked in black.

optimization goals is lowered (more emphasis on energy, Figure 9b and Figure 9c), more of the 4k BTB configurations are classified as the Sensitive-good, resulting in less Insensitive benchmarks for these two optimization goals.

The potential benefit of dynamically choosing the configuration is shown in Figure 10, which compares an Oracle selection of the best dynamic policy for each Sensitive-good benchmark to the five Static configurations across all benchmarks, normalized to the 12k BTB baseline. We see that the Sensitive-good benchmarks have a potential 1.5% performance gain on average (max 5.5%) with a 1.5% average energy savings for the the Performance goal (a), and a 2.7% energy savings (max 6.1%) with 0% performance loss for the Energy goal (b). These results are better than any of the static single policy configurations shown to their right. (Note that we do not include the Sensitive-bad or Insensitive benchmarks in the Oracle configurations as they are essentially unchanged in the Oracle configuration, while they may see significant losses in the Static Configurations.) The impact on potential benefit across the different optimization goals is shown in Figure 10c, where the swing from performance to energy as the goal changes is shown in the marked averages. It is worth noting that the best configuration is the same for 80% of all benchmarks regardless of goal, as the majority have little potential for improvement, and the benefits are limited to only the sensitive-good subset.

5 DYNAMIC RUNTIME CONFIGURATION

There are two main challenges with dynamically choosing the configuration at run-time: First, building a sufficiently cheap, yet accurate, classifier, and second, choosing a reconfiguration period that is short enough to capture phase behavior while being long enough to amortize the warm-up incurred from configuration changes.

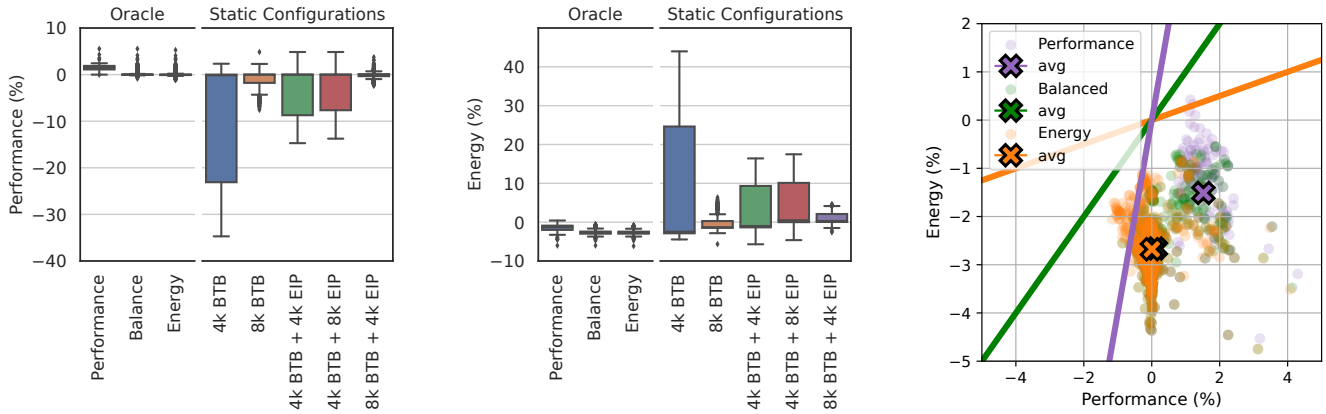
5.1 Classification

In this work we train a Decision Tree (DT) to select the configuration based on readily available performance counters. A DT is a tree structure that is walked to determine the final choice. Each non-leaf node in the tree contains the ID of the input to evaluate and its threshold value for deciding which way to go. The leaf nodes contain the final output, or choice of the DT. This makes DTs compact in storage (they store input IDs and thresholds at each node) and cheap to evaluate (the latency is determined by the depth of the tree).

To address our range of optimization goals, we simply train one DT for each goal. This results in 3 decision trees: 1 for each optimization goal (Performance/Balanced/Energy). At runtime, we periodically evaluate the DT for the current optimization goal using the current performance counter inputs to determine the next configuration. Our DT uses 15 different performance counters, with the three most important ones for each optimization goal shown in Table 2. We did not include the current configuration in the training as that would combinatorially explode the training. Despite this, the DT is very accurate on the chosen performance counters.

For training we simulate all six configurations for the benchmarks in the training set and calculate the score for each of the three optimization goals for each time period after 20M instructions of warm-up. We split the training benchmarks into windows of 20M instructions and used the performance counter values during those windows to identify the optimal configuration for next window. The optimal choice for each time period is determined by picking the configuration with the highest score. This approach does not include the cost of warming the BTB/DIP storage structures on configuration changes, and we find the final accuracy to be slightly lower (see Section 7).

To evaluate training robustness, we performed 20 random permutations of an 80/20 training/test split of the benchmarks. The resulting DTs provided accuracies between 97% to 98.8%, demonstrating robust training. We also explored 50/50 training/test splits with the random permutations and observed slightly reduced accuracies between 92% to 97%. Training on just 8% or 20% of benchmarks resulted in 90% DT accuracy for balanced and energy optimization goals, but lower for performance, showing the DT is robust even with little training data. Further, we found that random forests performed similarly, while the accuracy of a single-layer neural network was lower, at 87% to 95%, for the 80/20 split.



(a) Performance (b) Energy (c) Score

Figure 10: Potential improvements on the benchmarks of choosing the best configuration for each Sensitive-good benchmark (Oracle, left) vs. a single policy for all benchmarks (Static, right), for performance (a) and energy (b), and the average impact of changing the optimization goal on the score in the performance/energy configuration space(c).

The resulting DTs contain 1291/773/603 nodes and a maximum depth of 23/22/19 for performance/balance/energy trade-offs respectively, meaning they require approximately 4KB of storage (assuming 16 bits for threshold and remaining 1 byte for the ID of the feature to compare) and up to 23 memory accesses for each evaluation. Due to the simple nature of the DTs, either HW or SW could evaluate the DT in less than 2500 cycles (depth of 25 with at most one 100-cycle memory access per level), which is a negligible overhead compared to the 20M instruction windows between evaluations. The DT can always be made smaller (using cost-complexity pruning) for a trade-off in accuracy. However, evaluating this size decision tree has a negligible time and storage overhead so we did not look into this further.

Table 2: Top 3 Performance Counters

Performance Goal	Balanced Goal	Energy Goal
L1I total MPKI	Branch MPKI	L1I total MPKI
ITLB MPKI	L1I total MPKI	L2C total iMPKI
STLB iMPKI	ITLB MPKI	LLC total iMPKI

(All performance counters used: BTB MPKI direct/indirect/total, BP MPKI, L1I MPKI demand/prefetch/total, L2C iMPKI demand/prefetch/total, LLC iMPKI demand/prefetch/total, ITLB/STLB iMPKI)

5.2 Dynamic Reconfiguration Interval

To evaluate the potential gains from shorter reconfiguration intervals vs. the cost of more frequent BTB/DIP metadata zeroing and warm-up, we compared a system with no warm-up penalty to one which zeros the portion of the metadata that is reconfigured. In both cases we use an Oracle decision to isolate the potential of shorter intervals from the accuracy of the decision. The Performance, Energy, and Balanced results are shown in Figure 11.

The results show that without the warm-up cost, shorter intervals can improve the score significantly across all three optimization goals (blue, top line) as they can take the benefit of shorter phases. However, the warm-up cost from such frequent changes (red, bottom line) negates this benefit. Indeed, we see that the warm-up cost is sufficiently high that only when we use our full simulation trace (e.g., a single 100M instruction interval), do we amortize the warm-up cost. This suggests that intervals of ~100M instructions or longer between re-configurations would be preferred. However, as our simulation traces are on average only 100M instructions, we use 20M instruction intervals in the rest of the paper and include the cost of warm-up.

6 METHODOLOGY

We use ChampSim [19] with the architectural parameters in Table 3. Our L2 BTB has 12k entries, as with Intel's Alder Lake performance core [2], and we assume it is composed of 3 separate banks, each of which can be power-gated, as in [21, 26]. We use 2122 traces from the 1st Instruction Prefetching Competition [22] and the Championship Value Prediction Workshop [13]. The traces contain a diverse set of workloads including server, client, cryptography, compute integer/floating point, and SPEC.

We model a 24-entry FTQ, with each entry holding 32 aligned bytes (8 instructions), for a total capacity of 192 instructions, following [24]. The FTQ issues prefetches from the entries which are not at the head and a small prefetch queue tries to coalesce requests before issuing them to the L1I. The fetch unit issues a demand request for the entry at the head of the FTQ, which means that the instruction cache is potentially probed twice for each instruction: once on a prefetch and once on demand. However, prefetch reads are significantly lower-energy tag array reads, while the demand accesses also read the data. Writes to the tag and data array are similar in the case of either a demand or prefetch miss.

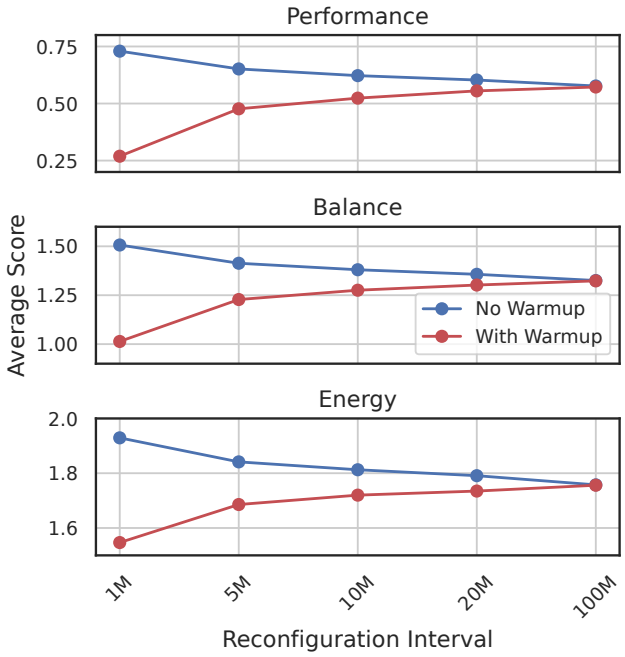


Figure 11: Impact of reconfiguration interval with and without the cost of warming the metadata, with an Oracle configuration selector. Without the cost of warm-up (blue lines) shorter intervals are able to take advantage of application phases. But with the warm-up costs (red line), longer intervals are required to amortize the warm-up overhead. As the traces are limited to 100M instructions, the 100M interval has only 1 re-configuration (after a warm-up interval of 20M instructions).

We implement Post fetch correction (PFC) for BTB misses (direct calls/jumps/returns) and branch mispredictions (direct calls/jumps). BTB misses are resolved earlier (1 cycle after fetch, e.g., pre-decode [24]), whereas branch-mispredictions for direct calls/jumps are resolved after full decode of 4 cycles, as shown in Figure 2[7]. We track returns and indirect branches in the BTB, but get their targets from the return address stack and indirect target predictor, respectively. BTB misses to indirect branches result in ROB flushes after execute, as they cannot be corrected early.

ChampSim does not natively model the impact of wrong-path execution on caches or TLBs, but only includes their pipeline execution delay. This means that FDIP generates prefetches until the FTQ is filled or an incorrectly predicted branch or BTB miss is encountered, thereby avoiding wrong-path prefetching. The EIP [49] and FDIP [24] evaluations use trace-driven simulators with this behavior. However, as wrong-path prefetching has been shown to be beneficial by bringing in lines that will be needed shortly [34, 45], the lack of it can hurt the baseline and thereby increase the apparent benefit of DIPs.

[7]To explore sensitivity to PFC latencies, we evaluated a 4-cycle re-steer on correctly predicted branches with BTB misses. We found that it had negligible impact for our energy and balanced optimization goals, but reduced the number of sensitive-good performance benchmarks by half and reduced their potential score by 21%. This demonstrates that the ability to tradeoff BTB space is dependent on the PFC latency and the optimization goal.

Table 3: Simulation Parameters

Core: 4GHz OoO, 8-wide fetch, 24-entry Fetch Target Queue, 8-wide decode, 60-entry Decode Queue, 352-entry ROB, 128-entry scheduler	
L1 BTB	128-entry, 2-way, 1-cycle latency
L2 BTB	12K-entry, 12-way, 2-cycle latency, 3 banks
Branch direction predictor	TAGE-SC [52], 8-table, 1024-entry/table
Indirect predictor	ITTAGE [51], 8-table, 512-entry/table
Return address stack	32-entry
Branch predictor bandwidth	Up to 8-instructions (32B) or 1-taken branch per cycle
ITLB/DTLB	256/128 entry 8-way
L2 TLB	2k entry 16 way - 50 cycle miss penalty
L1 Instruction Cache	32KB, 8-way, 16 MSHRs, 4-cycle
L1 Data Cache	48KB, 12-way, 16 MSHRs, 5-cycle, Next Line Prefetcher
L2 Unified Cache	512KB, 8-way, 32 MSHRs, 15-cycle, Signature Path Prefetcher (SPP) [30]
L3 Unified Cache	2MB, 16-way, 64 MSHRs, 35-cycle
DRAM	1 channel, 3200MT/s (25.6GB/s)

To address this, we added wrong-path instruction prefetching for BTB misses for predicted taken branches and branch mispredictions. When on the wrong-path, our frontend continues predicting basic blocks using the BP and BTB and issuing prefetches. As soon as the BTB miss is resolved (at decode) or the misprediction is resolved (at execute), execution returns to the correct path. With wrong-path prefetching we observed that the potential gain of a DIP over the baseline was reduced from 2% (max 9.8%) to 1.52% (max 5.3%) for Sensitive good benchmarks, but Protean's ability to take advantage of that potential was not significantly affected, showing the robustness of our approach.

With wrong-path prefetching for instructions, we found that FDIP could consume all L1I MSHRs, and thereby prevent the DIP from issuing prefetches. Indeed, some benchmarks had up to 12% of execution cycles where no MSHRs were available. We therefore limited FDIP to 8 of the 16 MSHRs and reserved the remaining 8 for the DIP. While MSHR allocation could be adjusted dynamically, we found that statically allocating 8 MHSRs to FDIP across all applications had negligible impact on performance (max 0.02% of cycles with no MSHRs available), and actually delivered a performance benefit of 0.038% (up to 2.3%) across all the applications. As a result, our proposed configurations all statically assign 8 L1I MHSRs to FDIP and the remaining 8 to the DIP when it is active, vs. our baseline 12k BTB which shares 16 MHSRs between the two.

As Protean can work with essentially any DIP whose metadata is largely compatible with the BTB storage, we chose the state-of-the-art Entangled Instruction Prefetcher (EIP) [49] and use the implementation provided by the authors [14]. We support two configurations sharing the BTB storage for the EIP entanglement table: 4k entries, 4-way (one bank) or 8k entries, 8-way (two banks). The original EIP used a 4k-entry, 16-way table. We use physical address prefetching in EIP to avoid the severe TLB contention EIP

causes, as observed by Vavouliotis et al. [57]. During execution, EIP accesses its entanglement table after every demand access to generate prefetches.

All simulations start with 20M instructions of warm-up in the baseline 12K BTB configuration followed by 100M instructions of simulation, or execution to the end for the shorter traces. After each 20M instruction period we evaluate the DT for the chosen optimization goal using the performance counter values from the previous period, and choose the next configuration based on its output. We zero the L2 BTB bank if it is reconfigured to run a different configuration (e.g. BTB to DIP, DIP to BTB, or BTB to off, etc.). We do not move entries or adjust the LRU ordering of the BTB entries across the ways.

We modeled both dynamic and static energy of the BTBs, ITLB, DTLB, L2 TLB, L1I, L1D, L2, and LLC using CACTI with a 22nm process [36]. Note that we have chosen to focus on on-chip storage structures related to instruction delivery in our energy evaluation to investigate the impact in the core itself. However, our decision tree could easily be retrained to include energy effects from other parts of the system[8].

7 RESULTS

We evaluate Protean's ability to achieve the benefits of a DIP without its cost for our three optimization goals (*Performance*, *Balanced*, and *Energy*) across three metrics: *Performance (IPC)*, *Energy*, and *Score* in Figure 12. Results are normalized to the baseline 12k BTB configuration without a DIP. For each optimization goal and metric we present results with perfect configuration selection (*Oracle*) and using our online Decision Tree (*DT*). Results are presented for the *Sensitive-good* and *Sensitive-bad* benchmarks and include metadata warm-up costs on configuration changes. Each column shows Score, Performance and Energy for a specific optimization labeled on the top of the column. For Score and Performance higher is better, and, for Energy, lower is better. This is seen in the Oracle extending further down in the bottom right plot.

Classification. The online classification accuracy of the DT is a bit lower than during training (95-97.5%, vs. 97-98.5%, see Section 5.1). This comes from the online DT evaluation using inputs from performance counters which have experienced warm-up effects from configuration changes, while our offline training data did not include warm-up. One extreme case of this is an outlier in the sensitive-bad set whose score is -16 (outlier in the energy optimization column of Figure 12. For this benchmark, the DT repeatedly switches between the 4/8/12K BTB configurations, while the Oracle choice is to stay with the 12k BTB. As a result, Protean reduces performance by 14% and increases energy by 13% for this case. Overall we note that the 6 similar outliers are a small subset of the 674 total in the sensitive-bad group, which demonstrates that we succeed in avoiding most of the downsides. The distribution of the configurations selected across the sensitive-good benchmarks (across individual windows of 20M instructions) is shown in Table 4.

Impact. Across all benchmarks, the DT is able to achieve an absolute score difference vs. the Oracle of less than 0.25/0.50 for 90%/95% of all benchmarks, highlighting the robustness of the classifier. Table 5 demonstrates that Protean is able to come very close to achieving the full potential benefits of the state-of-the art DIP while Table 6 shows that this is achieved while avoiding nearly all of its costs. We see that when optimizing for Performance, Balanced, or Energy, Protean achieves an average of 86/96/98% of the maximum potential (Oracle) for the Sensitive-good benchmarks, without requiring extra in-core metadata storage and while avoiding nearly all of the DIP's run-time costs. For the sensitive-bad benchmarks, Protean avoids almost all of the downsides of static configurations. This demonstrates that Protean is successful in obtaining nearly all the benefits of a DIP while avoiding nearly all its costs.

[8]For example, we could re-train to include DRAM, GPU, networking, I/O, displays (for mobile devices), etc. However, in most of those cases the static energy of those system components would so outweigh the the actual instruction delivery as to bury the core-level impacts.

Table 4: Configurations for Sensitive-good Benchmarks

	Performance	Balanced	Energy
12k BTB	10%	2%	1.5%
8k BTB	1%	5%	5.3%
4k BTB	8.5%	84%	89%
4k BTB + 4k EIP	48%	6.5%	3.7%
4k BTB + 8k EIP	20%	1.5%	0.3%
8k BTB + 4k EIP	12%	0.6%	0.4%

Table 5: Protean's ability to achieve the potential benefits vs. the Oracle for sensitive-good benchmarks

	Perf. Goal	Balanced Goal	Energy Goal
Score	1.59 / 1.84	1.90 / 1.97 (*96% of potential*)	2.54 / 2.6
Perf.	1.31% / 1.52% (*86% of potential*)	0.07% / 0.12%	-0.036%/0.005%
Energy	1.33% / 1.51%	2.61% / 2.66%	2.63% / 2.67% (*98% of potential*)

Positive numbers indicate better results: increased score, increased performance, or energy savings.

Table 6: Protean's ability to avoid potential costs vs. the Oracle for sensitive-bad Benchmarks

	Perf. Goal	Balanced Goal	Energy Goal
Score	0.09 / 0.12	0.102 / 0.14	0.04 / 0.1
Perf.	0.08% / 0.11%	0.013% / 0.04%	-0.02% / 0.03%
Energy	0.04% / 0.07%	0.13% / 0.16%	0.05% / 0.09%

Positive numbers indicate better results: increased score, increased performance, or energy savings.

8 CONCLUSIONS

State-of-the-art Dedicated Instruction Prefetchers (DIPs) provide benefits for a very limited number of applications at significant static (area) and dynamic (energy and bandwidth) cost for most

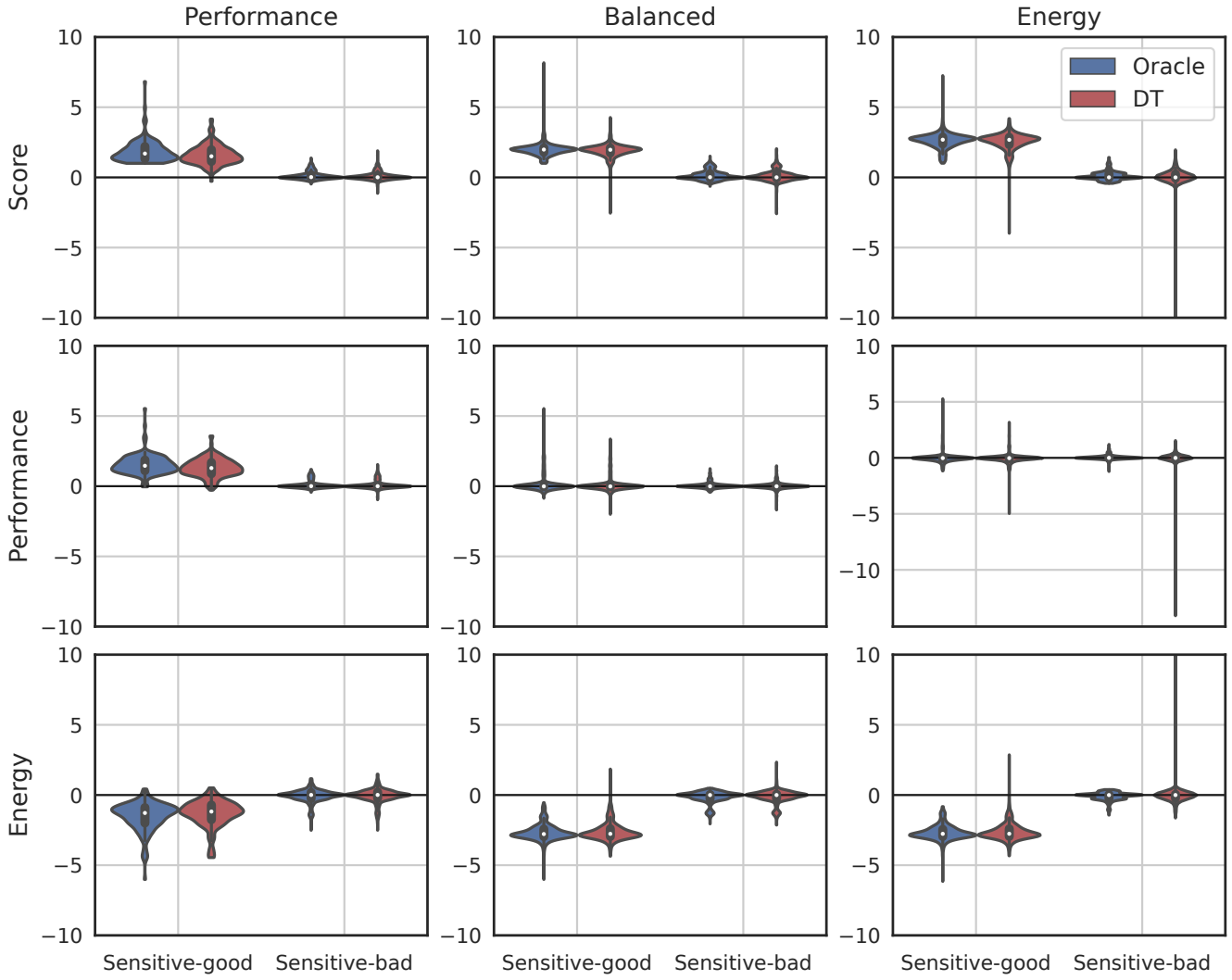


Figure 12: Ability of Protean to achieve the Performance and Energy benefits of a DIP (Sensitive-good) without suffering from its costs (Sensitive-bad), nor requiring its additional storage. Oracle (left, blue) shows the potential with perfect configuration selection compared to our online Decision Tree (right, red). Left column: Performance Goal, Middle column: Balanced Goal, Right column: Energy Goal. Results are normalized to the baseline 12k BTB with 16 shared L1I MSHRs. For Energy, 1 outlier extends beyond the plot as discussed in the text.

others. This work shows how we can achieve these benefits where possible without paying the costs.

Protean accomplishes this by sharing the existing frontend BTB storage between the DIP and the BTB. We demonstrate that this is practical because the applications that benefit from DIPs are less sensitive to BTB misses due to effective Post-Fetch Correction. This allows us to dynamically re-assign BTB storage banks based on the application behavior, resulting in six frontend configurations. We show that a Decision Tree can accurately and robustly pick a good configuration at run-time, and that we can target a range of energy/performance goals by simply training for the desired trade-off. While our use of aggressive, but realistic, baseline FDIP prefetching and Post-Fetch Correction limits the maximum benefits of the particular state-of-the-art DIP we considered, we were able to achieve 86% of the performance potential, 98% of the energy potential, and 96% of the balanced performance/energy potential, while avoiding essentially all of the penalties.

While the benefits today are limited by current-generation DIPs, Protean provides a general framework that can be easily re-trained for future advances. Indeed, with the development of more effective DIPs, and as applications become more challenging for FDIP, we expect Protean to be of even greater value in enabling designs that obtain the benefits of DIPs without paying their area and run-time costs.

ACKNOWLEDGMENTS

This work was supported by the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program (grant No. 2015.0153), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No. 715283), the Swedish Research Council (grant No. 2019-02429) and the Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (grant No. 23ZS1300).

REFERENCES

[1] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R. Prasky, and Anthony Saporito. 2020. The IBM z15 High Frequency Mainframe Branch Predictor Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 27–39. https://doi.org/10.1109/ISCA45697.2020.00014

[2] AnandTech. August 19, 2021. Intel Architecture Day 2021: Alder Lake, Golden Cove, and Gracemont Detailed. https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/. [Online; accessed 30-Oct-2022].

[3] AnandTech. May 27, 2019. Arm's New Cortex-A77 CPU Micro-architecture: Evolving Performance. https://www.anandtech.com/show/14384/arm-announces-cortexa77-cpu-ip/2. [Online; accessed 30-Oct-2022].

[4] Murali Annavaram, Jignesh M Patel, and Edward S Davidson. 2003. Call graph prefetching for database applications. *ACM Transactions on Computer Systems (TOCS)* 21, 4 (2003), 412–444.

[5] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Code Layout Optimization for Near-Ideal Instruction Cache. *IEEE Computer Architecture Letters* 18, 2 (2019), 124–127. https://doi.org/10.1109/LCA.2019.2924429

[6] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. Divide and Conquer Frontend Bottleneck. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 65–78. https://doi.org/10.1109/ISCA45697.2020.00017

[7] Truls Asheim, Boris Grot, and Rakesh Kumar. 2021. BTB-X: A Storage-Effective BTB Organization. *IEEE Computer Architecture Letters* 20, 2 (2021), 134–137. https://doi.org/10.1109/LCA.2021.3109945

[8] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory Hierarchy for Web Search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 643–656. https://doi.org/10.1109/HPCA.2018.00061

[9] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 462–473. https://doi.org/10.1145/3307650.3322234

[10] Gaurav Chadha, Scott A. Mahlke, and Satish Narayanasamy. 2014. EFetch: optimizing instruction fetch for event-driven webapplications. In *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, José Nelson Amaral and Josep Torrellas (Eds.). ACM, 75–86. https://doi.org/10.1145/2628071.2628103

[11] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. 2020. Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 52–64. https://doi.org/10.1109/ISCA45697.2020.00016

[12] I-Cheng K. Chen, Chih-Chieh Lee, and T.N. Mudge. 1997. Instruction prefetching using branch prediction information. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*. 593–601. https://doi.org/10.1109/ICCD.1997.628926

[13] CVP [n. d.]. Championship Value Prediction workshop. https://www.microarch.org/cvp1/index.html.

[14] EIP [n. d.]. The Entangling Instruction Prefetcher. https://github.com/alberto-ros/EntanglingInstructionPrefetcher.

[15] Mark Evers, Leslie Barnes, and Mike Clark. 2022. The AMD Next-Generation "Zen 3" Core. *IEEE Micro* 42, 3 (2022), 7–12. https://doi.org/10.1109/MM.2022.3152788

[16] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[17] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic (Eds.). ACM, 152–162. https://doi.org/10.1145/2155620.2155638

[18] Yu Gan and Christina Delimitrou. 2018. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters* 17, 2 (2018), 155–158.

[19] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. https://doi.org/10.48550/arXiv.2210.14324

[20] Brian Grayson, Jeff Rupley, Gerald Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 40–51. https://doi.org/10.1109/ISCA45697.2020.00015

[21] Tom's Hardware. August 19, 2021. Intel Architecture Day 2021: Alder Lake Chips, Golden Cove and Gracemont Cores. https://www.tomshardware.com/features/intel-architecture-day-2021-intel-unveils-alder-lake-golden-cove-and-gracemont-cores/4. [Online; accessed 30-Oct-2022].

[22] ICPC [n. d.]. The 1st Instruction Prefetching Championship (IPC1). https://research.ece.ncsu.edu/ipc/.

[23] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2020. Rebasing Instruction Prefetching: An Industry Perspective. *IEEE Comput. Archit. Lett.* 19, 2 (2020), 147–150. https://doi.org/10.1109/LCA.2020.3035068

[24] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2021, Stony Brook, NY, USA, March 28-30, 2021*. IEEE, 172–182. https://doi.org/10.1109/ISPASS51385.2021.00034

[25] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[26] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2016. Energy-efficient address translation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 631–643. https://doi.org/10.1109/HPCA.2016.7446100

[27] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2013. SHIFT: shared history instruction fetch for lean-core server processors. In *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, Matthew K. Farrens and Christos Kozyrakis (Eds.). ACM, 272–283. https://doi.org/10.1145/2540708.2540732

[28] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, Milos Prvulovic (Ed.). ACM, 166–177. https://doi.org/10.1145/2830772.2830785

[29] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 146–159. https://doi.org/10.1109/MICRO50266.2020.00024

[30] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path Confidence Based Lookahead Prefetching. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 60, 12 pages.

[31] Aasheesh Kolli, Ali G. Saidi, and Thomas F. Wenisch. 2013. RDIP: return-address-stack directed instruction prefetching. In *The 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, Davis, CA, USA, December 7-11, 2013*, Matthew K. Farrens and Christos Kozyrakis (Eds.). ACM, 260–271. https://doi.org/10.1145/2540708.2540731

[32] Jagadish B. Kotra and John Kalamatianos. 2020. Improving the Utilization of Micro-operation Caches in x86 Processors. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 160–172. https://doi.org/10.1109/MICRO50266.2020.00025

[33] Rakesh Kumar and Boris Grot. 2022. Shooting Down the Server Front-End Bottleneck. *ACM Trans. Comput. Syst.* 38, 3–4, Article 7 (jan 2022), 30 pages. https://doi.org/10.1145/3484492

[34] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the Front-End Bottleneck with Shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 30–42. https://doi.org/10.1145/3173162.3173178

[35] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. IEEE Computer Society, 493–504. https://doi.org/10.1109/HPCA.2017.53

[36] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 694–701. https://doi.org/10.1109/ICCAD.2011.6105405

[37] Chi-Keung Luk and T.C. Mowry. 1998. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 182–193. https://doi.org/10.1109/MICRO.1998.742780

[38] Priya Nagpurkar, Harold W Cain, Mauricio Serrano, Jong-Deok Choi, and Chandra Krintz. 2007. Call-chain software instruction prefetching in j2ee server applications. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE, 140–149.

[39] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 233–244. https://doi.org/10.1109/CGO.2017.7863743

[40] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 2–14.

[41] Irma Esmer Papazian. 2020. New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP). In *2020 IEEE Hot Chips 32 Symposium (HCS)*. 1–22. https://doi.org/10.1109/HCS49909.2020.9220434

[42] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62. https://doi.org/10.1109/MM.2020.2972222

[43] Arthur Perais, Rami Sheikh, Luke Yen, Michael McIlvaine, and Robert D. Clancy. 2019. Elastic Instruction Fetching. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 478–490. https://doi.org/10.1109/HPCA.2019.00059

[44] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 16–27. https://doi.org/10.1145/93542.93550

[45] Jim Pierce and Trevor Mudge. 1996. Wrong-Path Instruction Prefetching. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Paris, France) *(MICRO 29)*. IEEE Computer Society, USA, 165–175.

[46] Glenn Reinman, Todd M. Austin, and Brad Calder. 1999. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture, ISCA 1999, Atlanta, Georgia, USA, May 2-4, 1999*, Allan Gottlieb and William J. Dally (Eds.). IEEE Computer Society, 234–245. https://doi.org/10.1109/ISCA.1999.765954

[47] Glenn Reinman, Brad Calder, and Todd M. Austin. 1999. Fetch Directed Instruction Prefetching. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 32, Haifa, Israel, November 16-18, 1999*, Ronny Ronen, Matthew K. Farrens, and Ilan Y. Spillinger (Eds.). ACM/IEEE Computer Society, 16–27. https://doi.org/10.1109/MICRO.1999.809439

[48] Glenn Reinman, Brad Calder, and Todd M. Austin. 2001. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE Trans. Computers* 50, 4 (2001), 338–355. https://doi.org/10.1109/12.919279

[49] Alberto Ros and Alexandra Jimborean. 2021. A Cost-Effective Entangling Prefetcher for Instructions. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 99–111. https://doi.org/10.1109/ISCA52012.2021.00017

[50] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. 2022. Lukewarm Serverless Functions: Characterization and Optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 757–770. https://doi.org/10.1145/3470496.3527390

[51] Andre Seznec. CBP-3, 2011. A 64-kbytes ITTAGE indirect branch predictor.

[52] Andre Seznec. CBP-5, 2014. TAGE-SC-L branch predictors again.

[53] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 779–791. https://doi.org/10.1145/3466752.3480046

[54] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. 2019. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 513–526.

[55] David Suggs, Mahesh Subramony, and Dan Bouvier. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40, 2 (2020), 45–52. https://doi.org/10.1109/MM.2020.2974217

[56] Rabin Sugumar, Mehul Shah, and Ricardo Ramirez. 2021. Marvell ThunderX3: Next-Generation Arm-Based Server Processor. *IEEE Micro* 41, 2 (2021), 15–21. https://doi.org/10.1109/MM.2021.3055451

[57] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A Composite Instruction TLB Prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1138–1153. https://doi.org/10.1145/3466752.3480049

[58] Alexander V. Veidenbaum. 1997. Instruction Cache Prefetching Using Multilevel Branch Prediction. In *Proceedings of the International Symposium on High Performance Computing (ISHPC '97)*. Springer-Verlag, Berlin, Heidelberg, 51–70.

[59] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. 2015. Microarchitectural implications of event-driven server-side web applications. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 762–774. https://doi.org/10.1145/2830772.2830792