Performance-Aware Speculation Control using Wrong Path Usefulness Prediction

Chang Joo Lee† Hyesoon Kim§ Onur Mutlu‡ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{cjlee, patt}@ece.utexas.edu

§School of Computer Science
Georgia Institute of Technology
hyesoon@cc.gatech.edu

‡Computer Architecture Group
Microsoft Research
onur@microsoft.com

Abstract

Fetch gating mechanisms have been proposed to gate the processor pipeline to reduce the wasted energy consumption due to wrong-path (i.e. mis-speculated) instructions. These schemes assume that all *wrong-path instructions are useless for processor performance and try to eliminate the execution of all wrong-path instructions. However, wrong-path memory references can be useful for performance by providing prefetching benefits for later correct-path operations. Therefore, eliminating wrong-path instructions without considering the usefulness of wrong-path execution can significantly reduce performance as well as increase overall energy consumption.*

This paper proposes a comprehensive, low-cost speculation control mechanism that takes into account the usefulness of wrong-path execution, while effectively reducing the energy consumption due to useless wrong-path instructions. One component of the mechanism is a simple, novel wrong-path usefulness predictor (WPUP) that can accurately predict whether or not wrong-path execution will be beneficial for performance. The other component is a novel branch-count based fetch gating scheme that requires very little hardware cost to detect if the processor is on the wrong path. The key idea of our speculation control mechanism is to gate the processor pipeline only if (1) the number of outstanding branches is above a dynamically-determined threshold and (2) the WPUP predicts that wrong-path execution will not be beneficial for performance. Our results show that our proposal eliminates most of the performance loss incurred by fetch gating mechanisms that assume wrong-path execution is useless, thereby both improving performance and reducing energy consumption while requiring very little (51-byte) hardware cost.

1. Introduction

Current high performance processors use speculative execution through branch prediction to maximize the number of useful instructions in the pipeline. If speculative execution turns out to be incorrect, the pipeline is flushed. Flushed wrong-path instructions unnecessarily consume power/energy *unless they are useful for performance*.

In order to reduce the wasted power/energy due to wrong-path instructions, several fetch gating mechanisms have been proposed [16, 4, 12, 2, 7, 8]. These mechanisms decide whether or not to gate (i.e. stall) the fetch engine of the processor based on branch prediction confidence [11], performance monitoring, or instruction utilization rates. They explicitly or implicitly assume that wrong-path instructions are NOT useful for performance and hence eliminating their fetch/execution will always save energy. However, *NOT* all wrong path instructions are useless. Previous research has shown that some wrong-path instructions can be very beneficial for performance because they might prefetch into caches data and instructions that are later needed by correct-path instructions [21, 18]. Thus, the execution of wrong-path instructions can not only improve performance but also lead to energy savings through reduced execution time. With increasing memory latencies and instruction window sizes, the positive performance impact of wrong-path instructions becomes more salient [18]. Therefore, effective fetch gating mechanisms need to take into account the usefulness of wrong-path instructions.

Figure 1 shows the performance and energy consumption of an "ideal" fetch gating scheme that immediately gates the fetch engine when a mispredicted branch is fetched, using oracle information. This scheme is impossible to implement, but shows the potential of previously proposed fetch gating schemes. Ideal fetch gating improves performance of most benchmarks (by 2.3% on average excluding *mcf* and *parser*).[1] Furthermore, ideal fetch gating results in significant energy savings for most benchmarks (18.0% on average excluding *mcf*). However, two benchmarks show opposite, undesirable behavior even with ideal fetch gating. For *mcf*, ideal fetch gating both reduces performance (by 29.5%) and increases energy consumption (by 15.0%). For *parser*, ideal fetch gating also reduces performance (by 5.3%) but saves energy (by 27.9%). As shown in [18], these two benchmarks take advantage of wrong-path memory references. In *mcf*, since many (36.9% of) wrong-path instances (or episodes) prefetch a large number of useful wrong-path L2 cache misses for later correct-path instructions (99.8% of L2 misses generated on the wrong path are useful), eliminating all wrong-path operations reduces both performance and energy efficiency. On the other hand, few (2.0% of) wrong-path episodes have significant prefetching benefits (37.3% of L2 misses generated on the wrong path are useful) in *parser* while many others do not. Therefore, ideal fetch gating reduces performance in *parser* but it still improves overall energy efficiency.

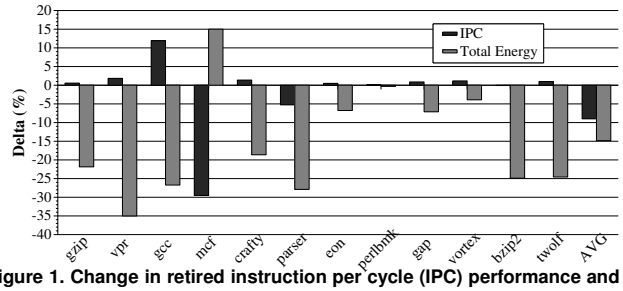

Figure 1. Change in retired instruction per cycle (IPC) performance and energy consumption with ideal fetch gating

Because the performance benefit of wrong-path memory references is significant for some applications, a speculation control (e.g. fetch gating) scheme that does not take into account the prefetching benefits of wrong-path instructions can hurt overall performance and result in increased energy consumption. As such, the net effect of speculation control can be exactly the opposite of what it is designed to achieve (i.e. reduced energy consumption). In order to overcome this problem, the goal of this paper is to propose *new speculation control techniques that predict the usefulness of wrong-path episodes* on top of a fetch gating mechanism that is implemented with low hardware cost. If a wrong-path episode is predicted to be useful for performance, the proposed mechanism does not gate the fetch engine.

Previously proposed fetch gating mechanisms [16, 2, 7, 8] have one other important limitation. They require a significant amount of additional hardware to decide whether or not to gate the fetch engine. For example [16, 2] require a branch confidence estimator, [7] requires significant changes to critical and power-hungry pipeline structures such as the instruction scheduler, and [8] requires a large (4KB) wrong-path predictor. The additional hardware not only increases the complexity of the processor but also consumes both dynamic and static energy, which can offset the energy savings from fetch gating. Therefore, simple and more power/energy-efficient speculation control mechanisms are very desirable. To this end, we propose a fetch gating technique that does not require large hardware structures or significant modifications to critical portions of the pipeline. The key insight of our technique is that the probability of having at least one mispredicted branch instruction in the pipeline increases as the number of outstand-

[1] Performance improvement of ideal fetch gating is mainly due to the elimination of the cache pollution caused by wrong-path memory references [18].

Benchmark	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf
L2 data misses	37407	2250	14374	4576397	20947	139112	694	43047	1076502	99205	245810	825
Total L2 misses	38427	4916	53824	4577214	25300	141214	4598	45720	1083592	115601	246765	3841
L2 Misses per 1K Inst	0.301	0.130	1.026	36.611	0.306	1.478	0.063	0.913	4.459	0.857	1.164	0.048
Memory intensive?	No	No	Yes	Yes	No	Yes	No	Yes	Yes	Yes	Yes	No

Table 1. Number of L2 cache misses for SPEC2000 integer benchmarks (A benchmark is memory intensive if its L2 MPKI > 0.5)

ing branch instructions increases. As such, branch-count based fetch gating gates the pipeline if the number of branch instructions in the pipeline exceeds a threshold value, which is determined based on the current branch misprediction rate. We show that simply adjusting the *number of outstanding conditional branches* based on branch prediction accuracy is effective at reducing the number of wrong-path instructions, without requiring costly confidence estimation or wrong-path prediction hardware.

Contributions: We make four major contributions in this paper:

1. We show that ignoring the performance benefits of wrong-path execution and thus using speculation control assuming wrong-path execution is always useless can significantly degrade performance and increase energy consumption. We describe via code examples why it makes sense to take into account the performance benefits of wrong-path execution.
2. We introduce the concept of *wrong path usefulness prediction (WPUP)* and propose two low-cost WPUP mechanisms that can be used with any previously proposed speculation control scheme. To our knowledge, no previously proposed speculation control scheme explicitly takes into account the usefulness of wrong-path instructions. We show that our new WPUP mechanisms eliminate almost all of the performance loss due to fetch gating, while requiring very little (only 45-byte) hardware cost.
3. We propose a new fetch gating mechanism, *branch-count based fetch gating*, that achieves the performance and energy benefits of previously proposed fetch gating schemes, while requiring much smaller hardware cost. The key idea of branch-count based fetch gating is to gate the pipeline if the number of branch instructions in the pipeline exceeds a threshold value, which is determined based on the current branch misprediction rate. As such, branch-count based fetch gating does not require a confidence estimator, a wrong-path predictor, or significant changes to pipeline structures.
4. We combine WPUP and branch-count based fetch gating to provide a comprehensive speculation control scheme that is aware of the benefits of wrong-path instructions. We show that our combined proposal provides the best performance and energy efficiency, while requiring very little (51-byte) hardware cost.

Our evaluations show that our comprehensive speculation control proposal that requires only 51 bytes of storage significantly reduces the performance loss incurred by fetch gating mechanisms that assume wrong-path execution is useless. On a relatively conservative processor with an 11-stage pipeline, our proposal improves performance by up to 8.1%, and reduces energy consumption by up to 4.1% compared to a previously proposed fetch gating scheme [16]. On a more aggressive baseline processor with a 30-stage pipeline, our proposal improves performance by up to 15.4% and 4.7% on average. As such, our proposal shows the value of taking into account the benefits of wrong-path execution, a concept largely ignored by previous research in speculation control.

2. Motivation: Benefits of Wrong-Path Execution

Wrong-path instructions affect execution of the correct path by changing the state of the processor's memory subsystem. Wrong-path memory references generated by the wrong-path instructions or the prefetcher can be beneficial for performance if they fetch cache lines that will later be needed by instructions on the correct program path. On the other hand, wrong-path memory references can be detrimental to performance if they fetch cache lines that will not be needed by instructions on the correct program path, if they fetch cache lines that evict the cache lines that will be needed by correct-path instructions, or if they tie up bandwidth and resources in the processor or the memory system that are needed to service the correct-path references. Previous research [18, 19] has shown that both positive and negative effects of wrong-path memory references are mainly due to the changes (prefetching or pollution) they cause in the L2 cache (as opposed to the changes they cause in L1 instruction/data caches and other memory system resources). Therefore, we focus our analyses on effects of wrong-path execution on the L2 cache.

We first provide a brief analysis of the usefulness of wrong-path memory references to motivate why speculation control techniques should be aware of the usefulness of wrong-path execution. Table 1 shows the number of L2 cache misses for each benchmark in the SPEC CPU 2000 integer suite and Figure 2 shows the distribution of total L2 cache misses based on whether the miss is generated on the wrong path and whether or not the cache line allocated in the L2 cache is used by a correct-path memory instruction with the processor model described in Section 4. *Correct-path miss* indicates the number of L2 cache misses that are generated by correct-path instructions. *Unused, partially used*, and *used wrong-path miss* indicate the number of L2 cache misses that are generated by wrong-path instructions but never used, used when missing cache lines are still outstanding in the miss status holding registers (MSHRs) (i.e. not in the L2 cache yet), and used when the cache lines are in the L2 cache, respectively. In other words, *partially used* and *used* wrong-path misses together quantify the useful prefetching effect of wrong-path memory references into the L2 cache.

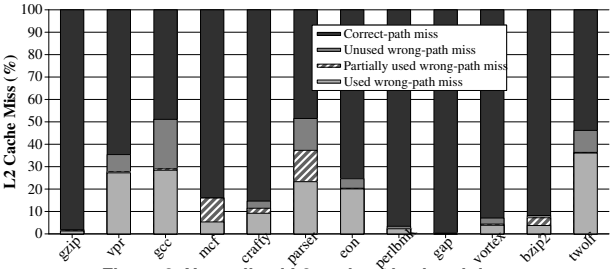

Figure 2. Normalized L2 cache miss breakdown

Clearly, a larger number *unused* wrong-path misses indicates a larger amount of disturbance (pollution) caused by wrong-path memory references in the L2 cache. Hence, if a processor correctly gates wrong-path instructions, it can potentially achieve performance improvement as well as power/energy savings if the fraction of unused cache lines is large. This is the case for most benchmarks except *mcf* and *parser*. For example, for *gcc*, the performance improvement of fetch gating can be significant (11.9% in Figure 1) because many wrong-path L2 cache misses are never used (23.4% out of all cache misses).

On the other hand, a larger number of *used/partially-used* misses indicates that wrong-path instructions are prefetching useful data into the L2 cache. For *mcf*, most of the wrong-path misses are used (99.8% of all wrong-path misses are either used or partially-used). Due to the large number of total L2 cache misses (almost all of which are useful), ideally eliminating wrong-path instructions hurts performance significantly in *mcf* as shown in Figure 1. For *parser*, wrong-path misses are frequently used (37.3% of all misses are *used/partially-used* wrong-path misses). However, ideally eliminating all wrong-path instructions induces less performance degradation than seen in *mcf*. This is because *parser* has a higher portion of *unused* wrong-path misses (14.2% of all misses) and a smaller number of total L2 cache misses than *mcf*.

Our Goal: Thus, wrong-path instructions are not always harmful to performance as commonly (and perhaps implicitly) assumed by previous speculation control schemes. On the contrary, they can be quite beneficial for performance. Therefore, it is important to distinguish when wrong-path references are beneficial for performance in order to design a more intelligent, performance-aware speculation control mechanism that does not gate the fetch engine when wrong-path memory references provide prefetching benefits. Note that the hardware cost for the mechanism must be small enough to guarantee that the achieved energy savings is not offset by the energy consump-

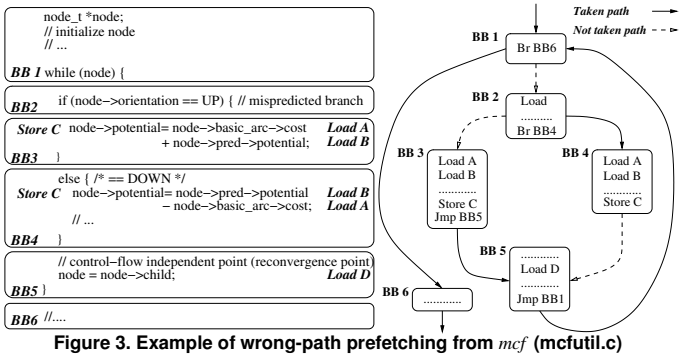

Figure 3. Example of wrong-path prefetching from *mcf* (mcfutil.c)

tion of the additional hardware. Our goal in this paper is to design such a low-cost speculation control technique that is aware of the usefulness of wrong-path execution.

Motivation for Detecting Wrong-Path Usefulness in the MSHRs: We observe in Figure 2 that *partially used wrong-path misses* account for a significant portion of the total number of useful wrong-path misses (*used + partially used wrong-path miss*) for both *mcf* and *parser* (66.6% and 37.5% respectively), the only two memory-intensive applications where wrong-path misses significantly affect the L2 cache. As such, MSHRs [14], bookkeeping registers where outstanding misses are buffered until they are fully serviced by the memory system, can be a good candidate for detecting the usefulness of wrong-path execution by detecting *partially-used* wrong-path misses. It is also more cost-effective to track the usefulness of a wrong-path reference within MSHRs than within the L2 cache because MSHRs have much fewer entries than the L2 cache. Therefore, the wrong-path usefulness predictors (WPUP) we will propose use the MSHRs to detect whether a wrong-path memory reference is useful for later correct-path instructions and use this information to train the WPUP structures dynamically.

2.1. Why Can Wrong-Path Execution Be Useful?

We briefly describe two code examples to provide insights into why wrong-path execution can provide prefetching benefits for correct-path execution. We found that there are two major code constructs that lead to prefetching benefits on the wrong path: (1) *Hammocks:* control-flow hammocks that use the same data on both sides of the hammock, (2) *Control-flow independence:* control-flow independent program portions that are executed twice, once before a misprediction and once after.[2]

Figure 3 shows a program segment from *mcf* and its control flow graph that takes advantage of wrong-path prefetching due to both *hammocks* and *control-flow independence*. The conditional branch instruction in basic block (BB) 2 is a frequently mispredicted branch. The load and store instructions (*Load A, B* and *Store C*) in both BB3 and BB4 refer to the same load and store addresses. Therefore, regardless of whether or not the branch in BB2 is mispredicted, the cache lines for the data of the loads and the store in either BB 3 or 4 are touched. Hence, the basic block that is executed on the wrong path of the branch in BB2 *always* provides prefetching benefits (due to the fact that same data is used on both sides of a hammock).

Note that in Figure 3 the load instruction in BB5 (*Load D*) is control-independent of the branch in BB2. Moreover, the data address of the load is not dependent on any operation in BB3 or BB4. Hence, *Load D* loads the same data regardless of the direction of the branch at BB2. If the branch at BB2 is mispredicted, the miss generated by *Load D* (executed on the wrong path) would later be needed when the processor recovers from the misprediction and executes *Load D* on the correct path. Hence, wrong-path execution of a control-independent program portion can provide prefetching benefits for its later correct-path execution.

[2]We refer the interested readers to Mutlu et al. [18] for a detailed analysis of code structures that cause wrong-path prefetching benefits. Our characterization of the code constructs that lead to prefetching benefits on the wrong path is a subset of the code constructs described in [18].

Figure 4 shows a code section from the `put_into_match_table` function of the *parser* benchmark to illustrate a control-flow hammock structure that causes a useful wrong-path memory reference. This function adds a node to the appropriate (left or right) hash table depending on the value of the `dir` (direction) parameter passed to the function (lines 4-7). Depending on the value of `dir`, two different functions are called. The arguments passed to the called functions, `m` and `t[h]`, are the same regardless of the value of `dir`. In other words, instructions in the `if` block (line 5) and instructions in the `else` block (line 7) use the same data. Therefore, when the branch of the `if` statement (line 4) is mispredicted, a wrong-path load instruction generates a request for `t[h]`. Shortly after the mispredicted branch is resolved and the processor starts execution on the correct path, a correct-path load instruction will generate a request for the exact same data, which would already be in the cache or in flight.

```
1:   void put_into_match_table (... , t, dir, ...) {
2:      // compute h
3:      // initialize m
4:      if (dir == 1) {
5:         t[h] = add_to_right_table_list(m, t[h]);
6:      } else {
7:         t[h] = add_to_left_table_list(m, t[h]);
8:      }
9:   }
```

Figure 4. Example of wrong-path prefetching from *parser* (fast-match.c)

In our analysis, we found that most of the code structures that take advantage of the prefetching effect of wrong-path instructions are repeatedly executed (e.g. the code structures in Figures 3 and 4 are located and called within frequently-executed loop bodies). Therefore, it is conceivable to design a history based prediction mechanism that estimates the usefulness of wrong-path execution. The goal of the WPUP mechanism we will propose in the next section is to detect useful wrong-path prefetching provided by frequently-executed code structures (similar to those shown in Figures 3 and 4), and to disable fetch gating when wrong-path execution is predicted to provide prefetching benefits.

3. Performance-Aware Speculation Control: WPUP and Branch-count Based Fetch Gating

Our performance-aware speculation control technique consists of two prediction components as shown in Figure 5: 1. a *wrong-path usefulness predictor (WPUP)* and 2. a new *fetch gating scheme: branch-count based fetch gating*. The fetch gating scheme predicts if the processor is on the wrong path. WPUP predicts whether wrong-path execution would provide *useful* prefetching benefits. The speculation control scheme gates the fetch engine only if the processor is predicted to be on the wrong path *and* wrong-path is predicted *not* to provide prefetching benefits.[3]

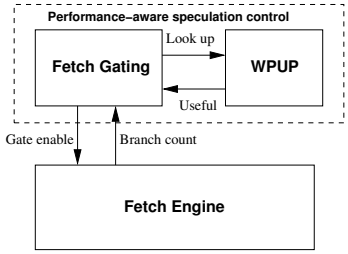

Figure 5. Performance-aware speculation control mechanism

3.1. Wrong Path Usefulness Prediction (WPUP)

We propose two techniques to detect the usefulness of wrong-path episodes. These mechanisms work at different granularities: 1. *Branch PC-based WPUP* is a fine-grained scheme that predicts wrong-path usefulness for each mispredicted or wrong-path branch instruc-

[3]This mechanism can simply be implemented by looking up the both predictors in parallel and ANDing the predictions. In our implementation, to reduce power/energy consumption, WPUP is looked up only when the fetch gating scheme predicts that the processor is on the wrong-path.

tion, 2. *Phase-based WPUP* predicts wrong-path usefulness in a more coarse-grained fashion during different program phases.

3.1.1. Branch Program Counter(PC)-based Wrong Path Usefulness Prediction Branch PC-based WPUP uses a set-associative cache structure (WPUP cache) to store the PCs of branches that lead to useful wrong-path prefetches.[4] For example, the branch in BB2 in Figure 3 will lead to useful prefetches (loads in BB3 or BB4, and BB5) if it is mispredicted. When the processor encounters such a useful mispredicted or wrong-path branch, it trains the WPUP cache with the program counter of the branch. The fetch engine keeps track of the PC of the latest branch instruction in a register (LBPC) by updating it whenever a branch is fetched. When the fetch gating mechanism decides to gate the fetch engine, the fetch engine looks up the WPUP cache with the PC of the latest fetched branch. If the PC of the fetched branch is present in the WPUP cache, then the processor predicts that the wrong path of the fetched branch (if mispredicted) would provide prefetching benefits and therefore, discards the gating decision.

In contrast to conventional cache structures, the WPUP cache does not require a data store. The tag store contains the tags (higher bits of the PC) of the branches that are found to provide wrong-path prefetching benefits along with the LRU replacement information.

Detecting the Usefulness of Wrong-Path Memory References: In order to know *exactly* whether or not a wrong-path memory reference is useful and to find the corresponding latest branch PC that resulted in the useful wrong-path reference, the processor either requires separate storage or needs to store the PC address of the mispredicted branch that caused the wrong-path memory request along with each L2 cache line. Unfortunately, the amount of this extra information can be prohibitive. For example, assuming a 1024-line cache, storing a 16-bit partial PC address with each L2 cache line would require 2KB extra storage to detect the latest branch PC that leads to wrong-path useful memory references. In order to eliminate the need for such extra storage, we propose a simple scheme that detects the usefulness of wrong-path memory references using the existing MSHRs [14] and extending them with a few additional fields. Since MSHRs have a small number of entries, storing information with each MSHR entry is more cost-effective than storing the same information with each L2 cache line.[5]

The WPUP mechanism uses the L2 MSHRs to detect useful branches that will train the predictor. The scheme detects two properties: (1) whether an outstanding cache miss in the MSHRs is generated by a wrong-path instruction and (2) whether it is useful (i.e. used by a correct-path instruction while the miss is being serviced). If a wrong-path miss in the MSHRs is determined to be useful, the branch that lead to the wrong-path miss is marked as useful in the WPUP cache (i.e. the PC of the branch is inserted into the WPUP cache).

Hardware Support: As described below, we augment several hardware structures in a conventional out-of-order processor to support the detection of branches that result in wrong-path prefetches. Note that, to reduce hardware cost, we use the lower 16 bits of the PC to identify a branch instruction, instead of its full 64-bit PC:[6]

1. Fetch engine:
   (a) Latest branch PC register (LBPC, 16 bits): is added to the fetch engine. It stores the PC of the latest fetched/predicted branch.
2. Inter-stage pipeline latches (decode, rename, and issue):
   (a) Branch PC field (BPC, 16 bits): A branch PC is associated with every instruction packet fetched in the fetch stage, indicating the youngest branch before the packet. This field is used to send the latest branch PC of the packet through the pipeline.
3. Load/store queue (LSQ) entries:
   (a) Branch PC field (BPC, 16 bits): A branch PC is associated with every load/store instruction, indicating the youngest branch before the load/store. This field in the LSQ stores the latest branch PC at the time the load or store was fetched.
4. L2 MSHRs:
   (a) Branch PC field (BPC, 16 bits): stores the latest branch PC from the branch PC field of the LSQ entry of an L2-miss-causing load/store instruction.
   (b) Branch ID field (BID, 10 bits): stores the branch ID from the branch ID field of the LSQ entry of an L2-miss-causing memory instruction. Branch ID is conventionally used for branch misprediction recovery in some out-of-order execution processors.
   (c) Wrong Path field (WP, 1 bit): is set when the corresponding memory request is known to be generated on the wrong path (when a branch older than or equal to the associated branch in the MSHR entry is resolved and found to be mispredicted).

Operation: When a branch is fetched, LBPC is updated with the PC of the branch. This LBPC is transferred through the front-end pipeline stages (i.e. using the BPC field in each pipeline latch) along with the corresponding instruction packet. Both BPC and branch ID are recorded in the LSQ when an entry is allocated for a load/store, and are transferred to the L2 MSHRs if the load/store's memory request misses in the L2 cache. Once a branch is resolved as mispredicted, the corresponding branch ID is sent to both the L2 MSHRs and the branch misprediction recovery mechanism. The ID of the mispredicted branch is used to search the MSHRs for entries that are allocated after the resolved mispredicted branch. The MSHR control logic compares the ID of the resolved branch to the branch ID fields of the MSHR entries. If the resolved branch ID is older than the branch ID of the MSHR entry, the MSHR entry is known to be allocated on the wrong path and its WP field is set.[7] With this mechanism, a wrong-path memory request can be detected as long as it has not been fully serviced before the mispredicted branch is resolved. We found that this scheme can detect 94% of all wrong-path L2 cache misses.

Whenever an outstanding wrong-path MSHR entry is hit (i.e. matched) by a later memory request to the same cache line, our mechanism estimates that the MSHR entry is useful for correct-path instructions. Thus, the WPUP cache is updated with the BPC (Branch PC) field stored in the MSHR.[8] Note that this scheme is inexact in the sense that the later memory request to the same cache line may not necessarily be generated by a correct-path instruction. However, we have found (and our evaluation shows) that this simplification does not affect the performance of WPUP.[9]

If the fetch gating mechanism (described in Section 3.2) predicts that the processor is on the wrong path, the processor accesses the WPUP cache with the current LBPC. If there is a hit in the WPUP cache, the processor does not gate the fetch engine, predicting that wrong path would provide prefetching benefits. Otherwise, the processor gates the fetch engine to save power/energy. Unlike conventional caches, the LRU information in the WPUP cache is not updated on a hit because frequent lookups by the fetch engine do not mean that the corresponding branch results in useful wrong-path prefetches.

Note that none of the additional hardware changes made to support wrong-path usefulness detection significantly increase the complexity of the structures they augment. The only time-critical structure that is

[4] Note that most fetch gating mechanisms do not know *exactly which branch* among all fetched conditional branches is mispredicted. Therefore, it is difficult to keep track of only the PC of the mispredicted (or wrong-path causing) branch in the fetch engine to look up the WPUP cache with. For example, if branch A is fetched and later branch B is fetched on the predicted path of branch A, the fetch engine does not know whether branch A or branch B is mispredicted until they are resolved later. For this reason, in the WPUP cache, we decide to keep track of the PC of the *latest branch* fetched before a useful wrong-path memory instruction is fetched.

[5] Our experiments show that detecting useful wrong-path memory references using both MSHRs and L2 cache lines only negligibly (by 0.01%) improves the performance of our WPUP mechanisms.

[6] We found that using only 16 bits of the PC does not affect the performance of our proposed speculation control scheme.

[7] Depending on the choice a microarchitecture makes in handling wrong-path memory requests, this mechanism might already be implemented in current processors. A processor that cancels wrong-path memory requests after a branch misprediction resolution requires a similar mechanism to detect and invalidate wrong-path memory requests.

[8] If the corresponding set in the WPUP cache is full, the LRU entry in the set is overwritten.

[9] We found that if a wrong-path memory request is later referenced by another memory request while in the MSHR, the latter request is very likely (with 95% probability) on the correct path.

augmented is the LSQ. However, the branch PC field added to each LSQ entry is used only for bookkeeping and therefore it does not participate in the content-addressable search of the LSQ entries.

3.1.2. Phase-based Wrong Path Usefulness Prediction Figure 6 shows the phase behavior of wrong-path usefulness for each 100K-cycle interval of *mcf* over the whole execution time. There are two distinct phases for the usefulness of wrong-path memory references. Until 75M cycles, wrong-path episodes do not result in useful memory references. In contrast, after 75M cycles, wrong-path episodes result in very useful references. We found that this phase behavior is due to execution of large loops whose bodies result in wrong-path prefetching benefits as discussed in Section 2.1. Loops that provide wrong-path prefetching benefits are executed in some phases, while others that do not are executed in other phases. As such, it might not be necessary to distinguish wrong-path usefulness on a per-branch basis because branches that do not provide wrong-path prefetching benefits might not be executed during the same phase as branches that do. To exploit such phase behavior in wrong-path usefulness, we would like to design a mechanism that can estimate wrong-path usefulness based on coarse-grained phase behavior.

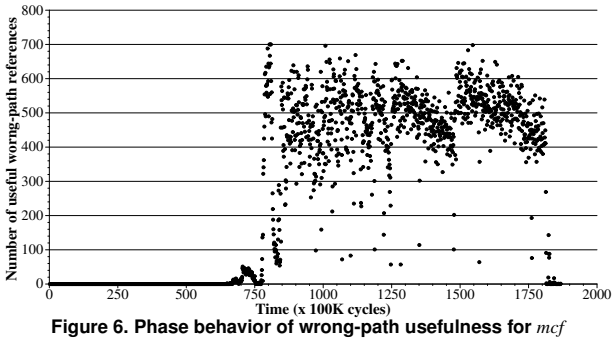

Figure 6. Phase behavior of wrong-path usefulness for *mcf*

We can simply detect the phase behavior of wrong-path usefulness using a counter (wrong-path usefulness counter - WPUC) and the MSHR-based wrong-path usefulness detection mechanism described in Section 3.1.1. We use a time-interval of 100K cycles to update and reset the WPUC counter in order to detect the phase behavior of wrong-path usefulness. At the beginning of a time-interval, WPUC is reset to 0. During a time-interval, WPUC is incremented by 1 whenever a wrong-path MSHR is hit by a later memory request. At the beginning of the next time interval, the value of WPUC is tested. If the value of WPUC is greater than a certain threshold ($phase\_wpup\_threshold$), the processor disables the fetch gating mechanism in the interval, predicting that the wrong-path episodes in the interval would be useful for performance.

This mechanism is advantageous because it does not require a hardware structure similar to the WPUP cache used in PC-based WPUP. All it requires is a simple hardware counter in addition to the support in the MSHRs required for detecting useful wrong-path requests. Note, however, that this is a coarser-grained WPUP scheme than the branch PC-based WPUP and hence may mispredict finer grained changes in wrong-path usefulness behavior. Nevertheless, we found that in most cases wrong-path usefulness is a coarse-grained function of program phase behavior rather than branch PC addresses: in other words, phase-based WPUP can outperform PC-based WPUP because it can better predict the usefulness of wrong-path memory references for the SPEC 2000 integer benchmarks.

3.2. Fetch Gating Mechanism: Branch-count Based Fetch Gating

We propose a low-cost fetch gating mechanism which leverages the observation that the probability of having a mispredicted branch instruction increases as the number of outstanding unresolved branch instructions in the pipeline increases. This mechanism requires (1) a count register that counts the number of outstanding branch instructions (branch count register, BCR) and (2) logic modifications in the branch resolution unit. Once a branch instruction is fetched (or decoded), the processor increments BCR by 1. When a branch instruction is resolved in the branch resolution unit, the processor decrements BCR by 1. If the BCR value is larger than a certain threshold T at any given time, the fetch engine stops fetching instructions. Due to the phase behavior of branch misprediction rate [23], a constant threshold value for T can inhibit correct-path instruction fetches significantly (if T is too low) or miss opportunities to remove wrong-path instruction fetches (if T is too high). Therefore, we adjust the threshold T dynamically based on the average branch prediction accuracy in a given time interval. If the average branch prediction accuracy is high, then T is set to a high value. Setting T to a high value makes it more difficult to gate the pipeline, which is desirable when the prediction accuracy is high. If the average branch prediction accuracy is low, T is set to a low value. Setting T to a low value makes it easier to gate the pipeline, which is desirable when the prediction accuracy is low. For example, if branch prediction accuracy is 99% in an interval, the threshold is set to 18 on our baseline processor. If branch prediction accuracy is 95%, the threshold is set to 13. In our study, we use 7 discrete values for T depending on the branch prediction accuracy (shown later in Table 5). These threshold values are determined empirically through simulation.

4. Methodology

4.1. Simulation Methodology

We use an execution-driven simulator of a processor that implements the Alpha ISA to evaluate our proposal. Our processor faithfully models the fetch and execution of wrong-path instructions and branch misprediction recoveries that occur on both the correct path and the wrong path. The memory system models bandwidth limitations, port contention, bank conflicts, and queuing effects at every level in the memory hierarchy. The baseline processor does not invalidate any memory requests from wrong-path instructions so that the wrong-path memory requests are eventually serviced and installed into the cache.[10] Our baseline processor includes a very aggressive stream prefetcher that was shown to improve the performance of our system significantly [24]. The parameters of the baseline processor are shown in Table 2.

Front end	I-cache: 64KB, 4-way, 2-cycle; fetch up to 2 branches per cycle; 11-stage pipeline (fetch, decode, rename, and execute)
Branch predictors	hybrid : 64K-entry gshare [17] and 64K-entry PAs predictor [26] with 64K-entry selector; 4K-entry BTB; 64-entry return address stack; minimum misprediction penalty: 11 cycles
Execution core	8-wide fetch/issue/execute/retire; 128-entry reorder buffer; 32-entry load-store queue; 128 physical registers
On-chip caches	L1 D-cache: 64KB, 4-way, 2-cycle, 2 read ports, 1 write port; L2 unified cache: 1MB, 8-way, 8 banks, 10-cycle, 1 port LRU replacement and 64B line size, 32 L2 MSHRs
Buses and memory	memory latency: 300 cycles; 32 memory banks; 32B-wide core-to-memory bus at 4:1 frequency ratio; bus latency: 40-cycle round-trip
Prefetcher	stream prefetcher: 32 streams and 16 cache line prefetch distance (lookahead) [25]

Table 2. Baseline processor configuration

We modified the Wattch power model [5] and augmented it to our simulator for power/energy simulation. We used a 0.10μm process technology at 1.2V V_{dd} and 2GHz clock frequency. We model the power consumption of all processor units faithfully so that the energy benefits of our speculation control mechanism is not over-estimated. Additional hardware structures used by the evaluated techniques (e.g. the WPUP cache structure, confidence estimator) are faithfully modeled in the power model. All our experiments use Wattch's aggressive clock-gating (CC3) option, where the power consumption of units is scaled linearly with port usage but unused portions of units still consume 10% of their maximum power.

We also model a more aggressive processor that is able to perform runahead execution [20] to evaluate the effect of our speculation control mechanism. Table 3 shows the parameters of this aggressive processor.

[10] Previous studies [21, 22, 18] have shown that this option provides better baseline performance than squashing wrong-path requests when a mispredicted branch is resolved.

Benchmark	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf
Performance (IPC)	2.38	1.98	1.59	0.59	2.83	1.97	3.23	2.59	2.76	3.21	1.75	2.34
Branch prediction accuracy (%)	94.17	92.22	94.95	96.89	96.12	95.65	98.87	99.96	98.94	99.52	92.08	94.91
Fraction of wrong path among all fetched inst. (%)	41.87	60.24	44.96	35.22	37.06	48.94	16.27	0.37	19.32	9.41	49.62	48.72
Fraction of wrong path among all executed inst. (%)	14.21	26.17	14.73	10.06	10.48	17.39	4.79	0.09	4.75	2.23	17.23	14.16

Table 4. Characteristics of baseline processor for SPEC2000 integer benchmarks

Branch predictors	minimum misprediction penalty: 30 cycles (30-stage front end)
Execution core	512-entry reorder buffer; 128-entry load-store queue; 512 physical registers; 512-byte runahead cache for runahead mode
Buses and memory	memory latency: 400 cycles; bus latency: 50-cycle round-trip

Table 3. Aggressive processor configuration

We use the SPEC 2000 integer benchmarks compiled for the Alpha ISA with `-fast` optimizations and profiling feedback enabled. The benchmarks are run to completion with a reduced input set [13] to reduce simulation time. Table 4 shows the baseline performance (in terms of IPC), branch prediction accuracy of the evaluated benchmarks, and the fraction of fetched/executed instructions that are on the wrong path. Table 1 has already shown information about the memory behavior of the evaluated benchmarks. All results presented in this paper are normalized to the baseline unless otherwise specified.

5. Results

5.1. Evaluation of the Branch-count Based Fetch Gating Mechanism

Figure 7 shows the change in performance and energy consumption with ideal (*ideal*), Manne's (*fg-manne*), and our branch-count based fetch gating mechanisms (*fg-br*). We used 14 as the miss distance counter (MDC) threshold for a 4K-entry, 4bit-MDC JRS confidence estimator [11] and 3 as the gating threshold for Manne's fetch gating (These thresholds were optimized to favor Manne's scheme). The thresholds used for branch-count based fetch gating as a function of the branch prediction accuracy are shown in Table 5. Branch prediction accuracy is measured and evaluated every 100K cycles.

Br prediction accuracy(%)	99+	97-99	95-97	93-95	90-93	90-85	85-
Threshold	18	16	13	12	11	7	3

Table 5. Branch-count based fetch gating thresholds

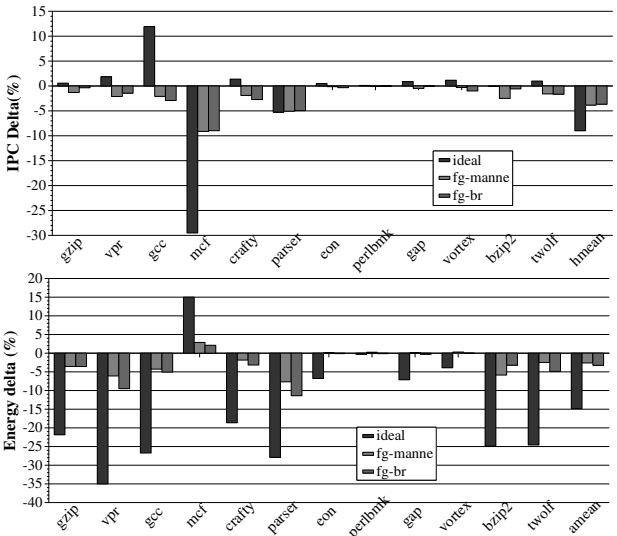

Figure 7. Performance and energy consumption of branch-count based fetch gating and Manne's fetch gating

As shown in Figure 7, the average performance and energy savings our branch-count based fetch gating scheme provides are better than Manne's fetch gating even though our scheme requires significantly less hardware cost (i.e. no need for a confidence estimator). This is because the accuracy and the coverage of our branch-count based fetch gating mechanism are higher (15.5% and 29.3% respectively) than those of Manne's fetch gating (13.2% and 19.9% respectively).[11] Our mechanism achieves better energy savings for *vpr, gcc, mcf, crafty, parser* and *twolf*. Overall, the energy savings of the branch-count based scheme is higher than Manne's (by 0.6%) while its performance loss is lower (by 0.2%). Hence, the branch-count based fetch gating scheme more efficiently eliminates wrong-path instructions than Manne's scheme by eliminating the hardware cost and design complexity introduced by a confidence estimator.

Figure 7 also shows that we cannot expect fetch gating to save significant energy in *eon, perlbmk, gap*, and *vortex*. Even the ideal fetch gating scheme reduces energy consumption by only 6.8%, 0.3%, 7.1%, and 3.9% respectively in these benchmarks. Because the branch prediction accuracy is very high (98.9%, 99.9%, 98.9% and 99.5% respectively as shown in Table 4), these benchmarks do not fetch or execute as many wrong-path instructions as the other benchmarks do, as shown in Table 4. Therefore, realistic fetch gating mechanisms achieve almost no energy savings for these benchmarks.

Note that both our and Manne's fetch gating mechanisms result in significant performance loss in *mcf* ($\sim$9%) and *parser* ($\sim$5%) because neither of the schemes takes into account the usefulness of wrong-path references. This shortcoming also results in increased energy consumption in *mcf* with both schemes due to the increased execution time. Next, we present results when our wrong-path usefulness prediction techniques are used in conjunction with the fetch gating schemes to make speculation control performance-aware.

5.2. Evaluation of Wrong Path Usefulness Prediction Mechanisms

As we showed in Section 5.1, both idealized and realistic fetch gating mechanisms hurt performance significantly for *mcf* and *parser*. We apply our WPUP techniques to the branch-count based fetch gating mechanism to recover the performance loss.

5.2.1. Branch PC-Based Wrong Path Usefulness Prediction Figure 8 shows the change in performance and energy consumption when branch PC-based WPUP is used in conjunction with branch-count based fetch gating. We vary the size of the WPUP cache from 8 to 128 entries and fix its associativity to 4. As the number of WPUP cache entries increases up to 32, *mcf*'s performance improves compared to the fetch gating mechanism without a WPUP.[12] With a 32-entry WPUP cache, a wrong-path usefulness predictor improves performance by 8.0% on *mcf* while also reducing energy consumption by 3.4%. Hence, utilizing WPUP eliminates almost all the performance loss incurred in *mcf* due to fetch gating.

Note that for benchmarks other than *mcf*, PC-based WPUP does not significantly affect performance or energy consumption. This is because wrong-path execution in these benchmarks (other than *parser*) does not provide significant prefetching benefits. In *parser*, we found that PC-based WPUP does not work well because the usefulness of wrong-path memory references is not a function of which branches are mispredicted but rather a function of program phase behavior.

5.2.2. Phase-based Wrong Path Usefulness Prediction Figure 9 shows the change in performance and energy consumption when phase-based WPUP is used in conjunction with branch-count based fetch gating. We vary the $phase\_wpup\_threshold$ from 5 to 20. For

[11] Accuracy of fetch gating is calculated by dividing the number of fetch-gated cycles when the processor is actually on the wrong path by the total number of fetch-gated cycles. Coverage of fetch gating is calculated by dividing the number of fetch-gated cycles when the processor is actually on the wrong path by the total number of cycles when the processor is on the wrong path.

[12] For the baseline configuration, a 32-entry WPUP cache is optimal for performance and energy savings. A larger WPUP cache leads to storing some stale branches that do not lead to useful wrong-path references any more, which results in incorrect prediction of some useless wrong-path episodes as useful. One other way of overcoming this "information staleness" problem is to flush the WPUP cache periodically.

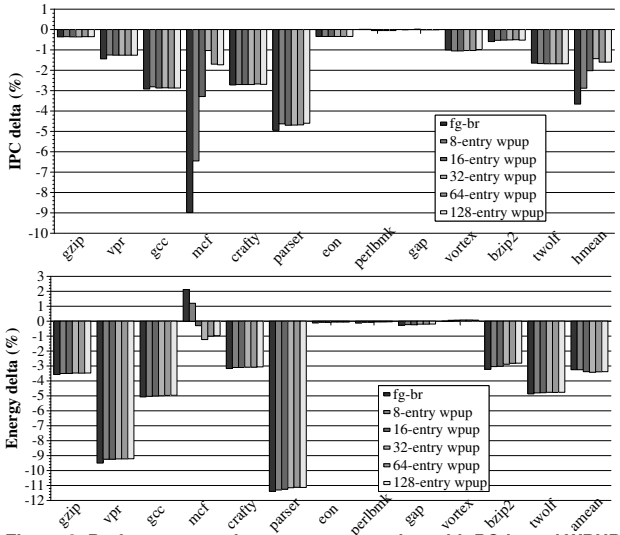

Figure 8. Performance and energy consumption with PC-based WPUP

comparison, we also show the performance and energy consumption of the best-performing PC-based WPUP with a 32-entry WPUP cache.

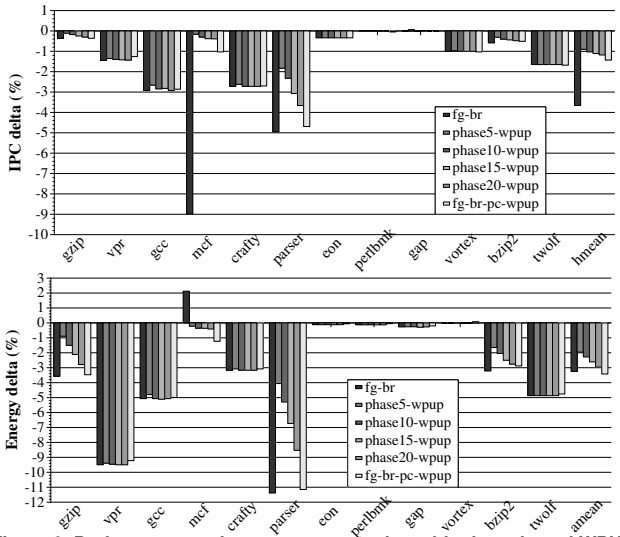

Figure 9. Performance and energy consumption with phase-based WPUP

Phase-based WPUP improves the performance of *mcf* and *parser* significantly compared to fetch gating without a WPUP. With a $phase\_wpup\_threshold$ of 5, performance improves by 8.8% for *mcf* and by 3.1% for *parser*. Phase-based WPUP eliminates almost all of the negative performance impact of fetch gating in *mcf*. On average, the best phase-based WPUP reduces the performance degradation of fetch gating from -3.7% to -0.9%. For *mcf*, energy consumption also reduces by 2.3% compared to branch-count based fetch gating without a WPUP.

Results with various $phase_wpup_threshold$s show the trade-off between performance and energy consumption for *parser*. As the threshold increases, performance decreases and energy savings increases because a larger threshold reduces the likelihood that a wrong-path episode is predicted to be useful. In *parser*, the energy reduction obtained due to the execution time improvement caused by useful wrong-path references does not outweigh the energy increase due to the execution of more wrong-path instructions. Hence, as fewer and fewer wrong-path episodes are predicted to be useful (i.e. as the threshold increases), energy consumption reduces because the fetch engine is gated to prevent the execution of wrong-path instructions.

Why does Phase-based WPUP Perform Better than PC-Based WPUP? The average performance of phase-based WPUP is higher than that of the PC-based WPUP, mainly because wrong-path usefulness is better predictable using program phase behavior rather than branch PCs in the *parser* and *mcf* benchmarks. This is because phase-based WPUP also takes into account the L2-miss behavior in a given interval whereas PC-based WPUP has no notion of either phases or L2-miss behavior in phases. If no wrong-path L2 misses happen in an interval, wrong-path periods are *not* predicted as useful by the phase-based scheme whereas they may be predicted as useful by the PC-based scheme because the PC-based scheme relies only on the past behavior of the wrong-path periods caused by a branch instruction.

Even though phase-based WPUP saves slightly less energy, it is simpler to implement than PC-based WPUP because it does not require a cache structure. Since phase-based WPUP provides higher performance while requiring less complexity, we believe it provides a better implementation trade-off than the PC-based WPUP.

5.2.3. Effect of Wrong Path Usefulness Prediction on Manne's Fetch Gating Technique Our WPUP techniques can be used in conjunction with not only our branch-count based fetch gating scheme but also other fetch gating mechanisms. We evaluate the WPUP techniques with Manne's fetch gating mechanism. Figure 10 shows the performance and energy consumption of Manne's fetch gating without a WPUP (*fg-manne*), with the PC-based WPUP (*fg-manne-pc-wpup*) and with the phase-based WPUP (*fg-manne-phase-wpup*). We used a 32-entry, four-way set-associative cache structure for the PC-based WPUP configuration and a $phase\_wpup\_threshold$ of 20 for the phase-based configuration.

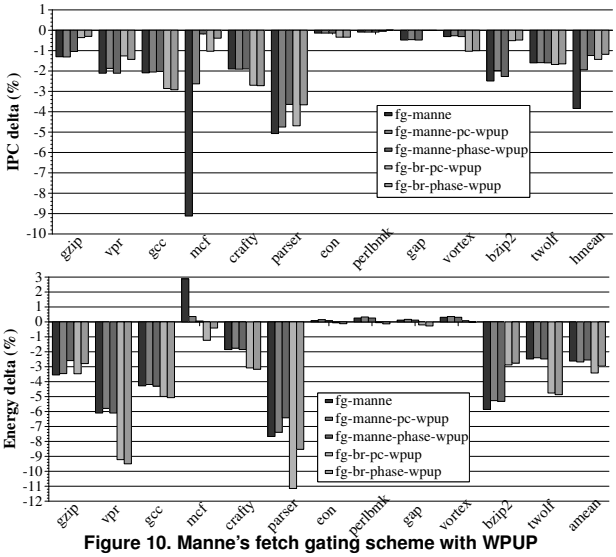

Figure 10. Manne's fetch gating scheme with WPUP

Phase-based WPUP improves both the performance and energy savings of Manne's fetch gating mechanism. In *mcf* performance improves by 9.8% while energy consumption reduces by 2.8%. In *parser*, performance improves by 1.5% while energy consumption increases by 1.3%. With phase-based WPUP, the average performance loss of Manne's fetch gating scheme is reduced to only -1.2% from -3.9% while its energy savings are preserved. We conclude that employing wrong-path usefulness prediction is also effective at improving the performance of Manne's fetch gating.

Figure 10 also shows that our branch-count based fetch gating mechanism with WPUP achieves better overall energy efficiency than Manne's fetch gating mechanism with WPUP. This is because branch-count based fetch gating achieves better energy efficiency (without WPUP) as we discussed in Section 5.1. Compared to Manne's fetch gating mechanism without a WPUP, our proposed techniques (branch-count based fetch gating with PC-based WPUP) provide up to 8.1% (2.5% on average) performance improvement along with up to 4.1% (1.0% on average) reduction in energy consumption.

5.3. Effect on Fetched and Executed Instructions

Figure 11 shows the reduction in fetched and executed instructions using our and Manne's schemes. Branch-count based fetch gating removes fetched and executed instructions by 11.0% and 1.4% on average respectively while Manne's fetch gating does so by 9.8% and

Benchmark	gzip	vpr	gcc	mcf	crafty	parser	eon	perlbmk	gap	vortex	bzip2	twolf
base	0.000	0.000	0.002	1.124	0.001	0.022	0.000	0.011	0.001	0.005	0.007	0.000
fg-br	0.000	0.000	0.001	0.991	0.001	0.020	0.000	0.010	0.001	0.004	0.007	0.000
fg-br-pc-wpup	0.000	0.000	0.001	1.114	0.001	0.020	0.000	0.010	0.001	0.005	0.007	0.000
fg-br-phase-wpup	0.000	0.000	0.001	1.123	0.001	0.022	0.000	0.010	0.001	0.004	0.007	0.000

Table 6. Useful wrong-path L2 caches misses per wrong-path episode (UMPW)

2.1% respectively. This explains why branch-count based fetch gating achieves better energy-efficiency as shown in Section 5.1.[13]

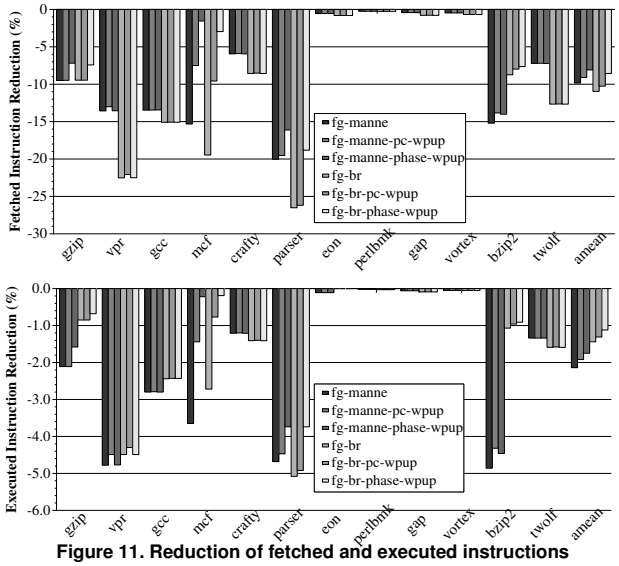

Figure 11. Reduction of fetched and executed instructions

When WPUP is used in conjunction with branch-count based fetch gating, fetched and executed instructions reduce by 8.6% and 1.1% on average (up to 22.5% and 4.49%), respectively. Hence, using WPUP slightly increases the fetched and executed instructions (especially in *mcf* and *parser*) because it disables fetch gating for useful wrong-path episodes. Even so, the reduction in fetched instructions is significant, leading to the energy savings shown in previous sections.

5.4. Effect on the Usefulness of Wrong-Path Episodes

We provide more insight into the performance improvement provided by wrong-path usefulness prediction by analyzing wrong-path usefulness with and without our techniques. To quantify wrong-path usefulness, we define a new metric, *Useful wrong-path L2 cache Misses Per Wrong-path episode* (UMPW) as follows:

$$UMPW = \frac{Total\ Number\ of\ Useful\ Wrong\ Path\ L2\ Cache\ Misses}{Total\ Number\ of\ Wrong\ Path\ Episodes}$$

UMPW quantitatively shows the efficiency of wrong-path episodes.[14] If wrong-path prefetching is not salient in an application, UMPW for that application will be close to zero. For an application that takes advantage of wrong-path prefetching, a fetch gating mechanism that does not take into account the usefulness of wrong-path execution might reduce the number of useful wrong-path L2 misses by executing fewer wrong-path instructions. Therefore, such a scheme that is unaware of wrong-path usefulness would decrease UMPW and therefore performance. On the other hand, a performance-aware speculation control scheme can increase UMPW (and hence performance) by allowing wrong path execution to occur when it is predicted to be useful and useful L2 misses to be generated on the wrong path. The larger the increase in UMPW a speculation control mechanism provides, the higher the performance improvement it can achieve.

Table 6 shows the UMPWs for the baseline and the baseline with our mechanisms (branch-count based fetch gating, PC-based WPUP, and phase-based WPUP). As expected, the UMPWs of all the benchmarks except for *mcf* and *parser* are close to zero and show very little change with our mechanisms, since these benchmarks have very little wrong-path prefetching effect. In contrast, the UMPW for *mcf* drops from 1.124 to 0.991 when the branch-count based fetch gating is applied to the baseline, resulting in a 9% performance loss (as shown in Figure 7). However, both PC-based WPUP and phase-based WPUP recover the loss in UMPW to 1.114 and 1.123 respectively. The improvement in UMPW explains why wrong-path usefulness prediction improves performance significantly for *mcf* in Figures 8 and 9. A similar effect is observed for *parser* with phase-based WPUP, which improves the UMPW metric compared to branch-count based fetch gating. We conclude that wrong-path usefulness prediction improves performance by increasing UMPW.

5.5. Effect on the Energy-Delay Product

Figure 12 shows the Energy-Delay Product (EDP) comparison of different WPUP prediction techniques when they are employed with branch-count based and Manne's fetch gating schemes. The highest savings in EDP is provided by combining our branch-count based fetch gating with branch PC-based WPUP, which results in a 2.1% decrease in EDP compared to the baseline. Note that schemes that do not take into account wrong-path usefulness (both Manne's scheme and branch-count based fetch gating scheme without WPUP) do not result in significant savings in EDP; in fact Manne's scheme without WPUP increases EDP by 0.6%. We conclude that our wrong-path usefulness prediction techniques are very effective at not only improving performance but also finding the right balance between energy consumption and performance.

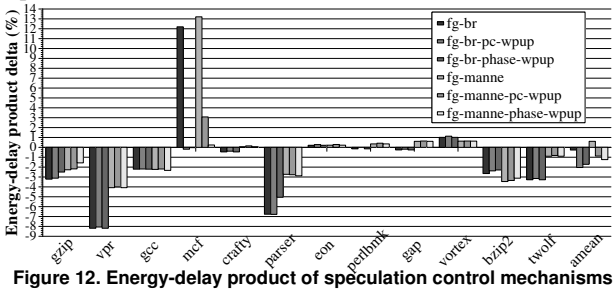

Figure 12. Energy-delay product of speculation control mechanisms

5.6. Hardware Cost and Power/Energy Consumption of Our Speculation Control Techniques

Table 7 shows the hardware cost of the two proposed WPUP techniques. The WPUP mechanisms do not add significant combinational logic complexity to the processor. Combinational logic is required for the update of WP-bits in the MSHRs and the update of WPUP cache and WPUC. None of the required logic is on the critical path of the processor.[15] The storage overhead of PC-based WPUP is only 260 bytes, which is less than 0.16% of the baseline front-end size (assuming a 64KB I-cache + 64KB branch predictor + 4K-entry BTB). The overhead of phase-based WPUP is almost negligible: only 45 bytes.

Table 8 shows the hardware cost, power, and energy comparisons of Manne's fetch gating scheme (*fg-manne*) and our branch-count based fetch gating scheme with PC-based/phase-based WPUP (*fg-br-pc-wpup*/*fg-br-phase-wpup*). The total hardware cost of our two

[13] Note that the reduction in executed instructions is much lower than that in fetched instructions in both fetch gating schemes. This is natural because many wrong-path instructions are flushed before they are executed, as Table 4 also shows.

[14] Note that this metric is not perfect because it does not take into account the criticality and latency of useful wrong-path L2 cache misses. We only use it to provide insight with a single easy-to-understand metric. The actual performance improvement depends not only on the change in the number of useful wrong-path L2 misses, but also on their timing, criticality, and whether or not their latencies are hidden. However, defining a metric -separate from absolute performance- that takes into account all these aspects is very difficult.

[15] We varied the latency of the WPUP training mechanism from 1 to 500 cycles. The performance difference is negligible.

	Hardware cost			Dynamic access frequency	Max. power	Avg. power	Avg. energy
	Fetch-gating	WPUP	Total				
fg-manne	br.count (8bits) + confidence (2048B)	-	**2049B**	0.15/inst	94.54mW	30.16mW	1.40mJ
fg-br-pc-wpup	br.count (8bits) + bpred.accuracy (36 bits)	260B	**266B**	0.07/inst	25.13mW	4.55mW	0.22mJ
fg-br-phase-wpup	br.count (8bits) + bpred.accuracy (36 bits)	45B	**51B**	0.07/inst	1.20mW	0.24mW	0.01mJ

Table 8. Hardware cost, power and energy consumption comparison of Manne's and our speculation control schemes

	PC-based WPUP		Phase-based WPUP	
Fetch engine	LBPC	16bits	-	
Inter-stage latches	BPC	16bits*11stages	-	
LSQ	BPC	16bits*32entries	-	
MSHR	BPC BID WP	16bits*32entries 10bits*32entries 1bit*32entries	BID WP	10bits*32entries 1bit*32entries
Training storage	WPUP cache	(13bits(addr)+1bit(V) +2bits(LRU))*32entries	WPUC	5bits
Hardware cost	**260 bytes**		**45 bytes**	

Table 7. Hardware cost of wrong-path usefulness predictors

schemes is 266 bytes/51 bytes, which is much less than 2049 bytes, the cost of *fg-manne*.[16] With a much smaller hardware cost, our schemes are able to predict the usefulness of wrong-path instructions and provide better performance. Furthermore, the confidence estimator in Manne's scheme is accessed much more frequently than the WPUP in our scheme, requiring 0.15 accesses per every instruction because every branch should access the confidence estimator. However, our WPUP is accessed 0.07 times per instruction because it needs to be accessed only when the processor is predicted to be on the wrong path. Due to reduced hardware cost and reduced access counts to hardware structures, the extra hardware needed by our techniques (*fg-br-pc-wpup*/*fg-br-phase-wpup*) consumes only 27%/1.3%, 15%/0.8%, and 16%/0.9% of the maximum power, average power, and average energy of those of Manne's scheme. We conclude that our speculation control technique provides improved performance and energy-efficiency at very low hardware and energy cost.

5.7. Effect on a More Aggressive Processor

Table 9 shows gating thresholds for branch-count based fetch gating for the more aggressive processor and Figure 13 shows the performance and energy impact of our schemes and Manne's scheme on the more aggressive processor configuration. As the performance impact of wrong-path memory references becomes more salient on a processor with a large instruction window [18], fetch gating without WPUP results in even more significant performance degradation. For example, Manne's fetch gating results in an average performance loss of 5.1% in the more aggressive processor. Compared to Manne's scheme, our speculation control mechanism with branch-count based fetch gating and phase-based WPUP improves performance by up to 15.4% (4.7% on average) while increasing energy consumption by only 1.2% on average. The EDP reduction of our speculation control mechanism is 4.1% while that of Manne's fetch gating is 2.8%. Note that this is a good performance/energy trade-off [10]. Hence, our speculation control scheme becomes more effective in controlling the performance loss due to fetch gating as processors become more aggressive.

Br prediction accuracy(%)	99+	97-99	95-97	93-95	90-93	90-85	85-
Threshold	60	50	40	30	20	15	13

Table 9. Branch-count based fetch gating thresholds for a more aggressive processor

5.8. Effect on Runahead Execution

Figure 14 shows the performance and energy impact of our schemes and Manne's scheme on a runahead execution processor [9, 20]. The performance and energy results are normalized to when runahead execution is employed on the more aggressive processor without any fetch gating. Note that, on a processor employing runahead execution, the performance degradation with both our and Manne's schemes for *mcf* is much less than that on a processor without runahead, since the prefetching effect of runahead execution reduces the positive performance impact of the prefetching effect of wrong-path execution. Nevertheless, our speculative control mechanism improves performance by up to 2.6% compared to Manne's scheme. Hence, we conclude that our techniques are effective on runahead execution processors as well.

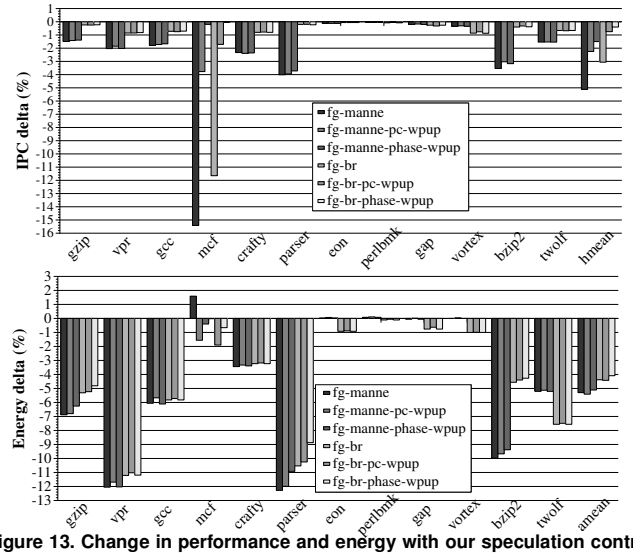

Figure 13. Change in performance and energy with our speculation control techniques on a more aggressive processor

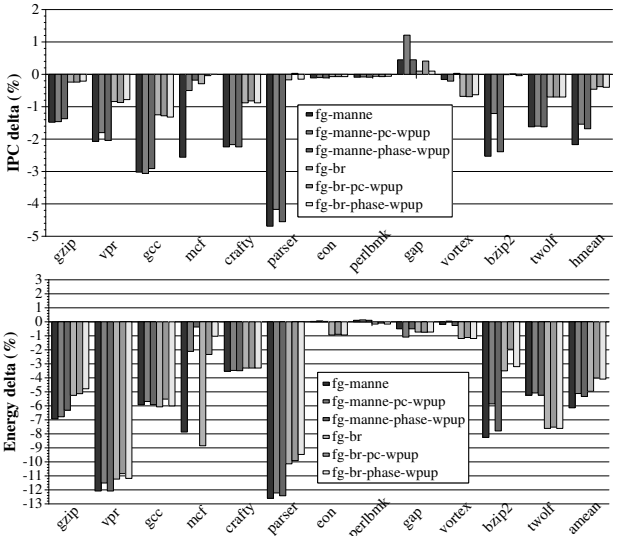

Figure 14. Change in performance and energy with our speculation control techniques on a runahead execution processor

5.9. Comparison with the Just In Time Instruction Delivery Mechanism

We compare our speculation control mechanisms with the Just In Time (JIT) instruction delivery mechanism [12]. We chose the best-performing JIT configuration in terms of energy-delay product among the 27 configurations we examined. We set JIT specific parameters,

[16] **Hardware cost of branch-count based fetch gating**: Branch counters (br.count) are used for counting the number of outstanding branches in our fetch gating scheme, and the number of outstanding low-confidence branches in Manne's scheme. They are 8-bit counters because the maximum number of instructions in the processor is not more than 2^8. Branch-count based fetch gating also requires two 18-bit counters (that store the number of correctly predicted branches and the number of total branches in an interval) in order to measure branch prediction accuracy. Because we use 100K-cycle intervals and the baseline processor can fetch up to two branches per cycle, 18 bit counters are sufficient.

minimum allowed instruction count in the pipeline, maximum instruction count increment unit, and noise margin to 128, 16 and 5% respectively.

As shown in Figure 15, JIT saves energy by only 1.6% and degrades performance by only 0.8% on average, leading to an EDP improvement of 0.7%. Note that since JIT adjusts fetch gating decisions based on the monitored IPC changes, it is able to impact performance less than branch-count based fetch gating. On the other hand, branch PC-based and phase-based WPUP with branch-count based fetch gating save energy by 3.4% and 3.0% while degrading performance by 1.4% and 1.2% respectively, resulting in EDP improvements of 2.1% and 1.7%. Note that both of our mechanisms save more energy than JIT on most of the benchmarks. These results demonstrate that our mechanisms achieve better energy efficiency (EDP) than JIT at the expense of slightly higher performance degradation.

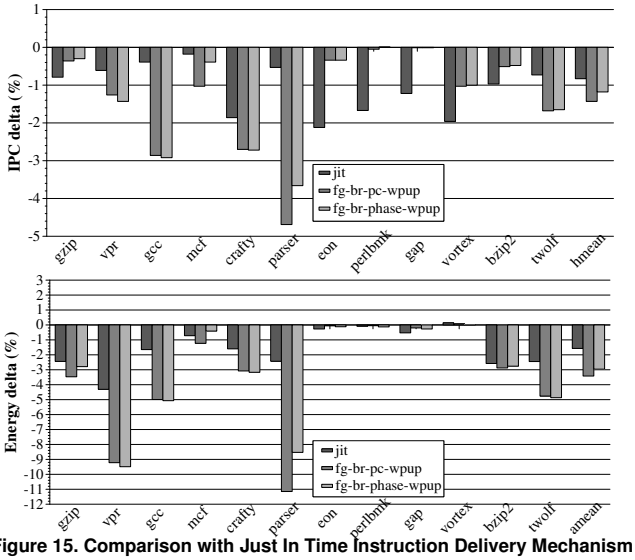

Figure 15. Comparison with Just In Time Instruction Delivery Mechanism

5.10. Sensitivity to Processor Parameters

We have examined the sensitivity of our techniques to branch predictor type, memory latency, and L2 cache size. For brevity, we do not present these results but refer the reader to [15], which contains the sensitivity analyses. Our results show that our techniques are effective with a wide variety of branch predictors, memory latencies, and L2 cache sizes.

6. Related Work

6.1. Related Work on Fetch Gating and Dynamic Reconfiguration Mechanisms

Fetch (pipeline) gating was first proposed by Manne et al. [16]. Their fetch gating mechanism counts the number of low confidence branches in the pipeline using a confidence estimator. If the number is more than a certain threshold, the fetch engine is gated to save power/energy. Our study shows that counting the number of in-flight branches and adjusting the threshold based on branch prediction accuracy are enough to save energy without losing significant performance and without requiring a separate confidence estimation structure. Furthermore, Manne et al.'s scheme does not take into account the prefetching effect of wrong-path instructions.

Many other fetch gating and dynamic reconfiguration schemes have also been proposed to reduce energy consumption [4, 6, 7, 12, 2, 8]. Baniasadi et al. [4] and Buyuktosunoglu et al. [6, 7] try to balance the front-end instruction supply rate and back-end execution rate. Baniasadi et al. point out that if there is enough instruction parallelism in the pipeline, having more instructions in the pipeline does not necessarily improve performance. They measure parallelism using decode/commit rate difference and data dependence count among the instructions being decoded. Adaptive issue queue (AIQ) [6] resizes the instruction scheduler based on its utilization to reduce power/energy consumption without significantly hurting performance. Buyuktosunoglu [7] proposes a scheme that performs fetch gating based on the utilization of the instruction scheduler and parallelism of the running application. If parallelism is low and the utilization of the instruction scheduler is high, the mechanism gates the fetch engine. Aragon et al. [2] extend the idea of fetch gating to multiple branch prediction confidence levels. If a branch is very low-confidence, the most aggressive fetch gating scheme is used. If a branch is relatively less low-confidence, the fetch engine is not fully gated but it is throttled so that it fetches fewer instructions than its maximum bandwidth allows.

Just in time (JIT) instruction delivery [12] dynamically adjusts the total number of instructions (not only branch instructions) in flight by monitoring IPC performance during certain intervals. BranchTap [1] also dynamically adjusts the number of instructions in the pipeline to reduce branch misprediction recovery cost in the presence of few global checkpoints. BranchTap keeps track of the number of low-confidence branches without an associated global checkpoint and gates the fetch engine if this number is greater than a certain threshold. The threshold is dynamically adjusted based on IPC during certain time intervals, just like in JIT. Both JIT and BranchTap require a tuning period to search for the optimal number of instructions in the pipeline to achieve the best performance. If the performance of an application is not stable for a long time, these mechanisms might not be able to stabilize, which could result in performance loss. Our WPUP mechanisms are partially orthogonal to and can be combined with BranchTap and JIT. For example, a combined mechanism could gate the fetch engine when the number of low-confidence branches without an checkpoint exceeds a dynamically-determined threshold (as in BranchTap) only if our WPUP mechanism predicts that wrong-path execution would not be useful.

Collins et al. [8] gate the fetch engine using their dynamic control-flow reconvergence prediction mechanism, utilizing the observation that a misprediction in their reconvergence mechanism very likely results from the fact that the processor is on the wrong path. Finally, Armstrong et al. [3] find that unusual or illegal pipeline events are strongly correlated with branch mispredictions and propose gating the fetch engine when the processor detects such events.

None of these previous works explicitly considers the performance impact of wrong-path memory references. Especially the positive performance effects of wrong-path execution are ignored in previous works, implicitly assuming that all wrong-path episodes are useless for performance. Our work improves the state-of-the-art by incorporating the usefulness of wrong-path execution into fetch gating decisions. Moreover, many of these previously proposed schemes require either large additional hardware structures (i.e. confidence estimators [16, 2] or wrong-path predictors [8]) that increase the complexity of the pipeline or significant changes to time-critical and power-hungry portions of the pipeline such as the instruction scheduler [4, 6]. We propose branch-count based fetch gating that eliminates the additional large hardware structures without requiring significant changes to time-critical and power-hungry portions of the pipeline.

Note that our WPUP mechanism can be incorporated into any of the previously proposed fetch gating mechanisms to make the mechanism performance-aware by taking into account the prefetching effect of wrong-path memory references. In this paper, we show the effectiveness of WPUP with both our novel branch-count based fetch gating scheme and Manne's confidence estimation based fetch gating scheme.

6.2. Related Work on the Usefulness of Wrong-path Memory References

Pierce et al. [21] study the effect of wrong-path memory references on cache performance using traces generated by an instrumentation tool. They show that the prefetching effect of wrong-path memory references is more dominant than their pollution effect for most of the SPEC92 C benchmarks. They also show that wrong-path instructions can prefetch useful instruction and data cache lines over 50% of the time.

Mutlu et al. [18] examine the effect of wrong-path memory references on the actual performance of a high performance out-of-order

processor using an execution-driven simulator. They show that not modeling wrong-path execution can lead to errors of up to 10 percent in performance. They find that wrong-path memory references are usually beneficial for performance in most SPEC2000 benchmarks but detrimental to performance in a few others. They also point out that the performance impact of wrong-path memory references gets larger memory latencies and instruction window sizes increase.

Building on this previous work on identifying the positive performance impact of wrong-path references, we show that speculation control schemes (i.e. fetch gating mechanisms) that do not take the wrong-path prefetching effect into account can hurt performance significantly. Unlike previous work that only evaluated the performance impact of wrong-path memory references, we propose an implementable mechanism that leverages and predicts the usefulness of wrong-path references in order to increase the performance and energy-efficiency of fetch gating.

7. Conclusion

This paper introduces the concept and low-cost implementations of wrong path usefulness prediction (WPUP) and its application to fetch gating mechanisms to improve the performance and energy-efficiency of speculation control, which has traditionally assumed that wrong-path execution is *always* useless for performance. We challenge this assumption and show that predicting the usefulness of wrong-path periods and not disabling wrong-path execution when it is estimated to be useful actually results in better performance and better energy-efficiency.

We propose simple PC-based and phase-based WPUP techniques that are applicable to previously proposed fetch gating schemes. In addition, we propose a branch-count based fetch gating scheme that eliminates the need for a confidence estimator to guess that the processor is likely to be on the wrong path. As such, we provide a very low-cost, comprehensive speculation control mechanism that is aware of the benefits of wrong-path execution. Our comprehensive mechanism significantly reduces the performance degradation of fetch gating while at the same time reducing energy consumption.

Acknowledgments

Many thanks to José A. Joao, Veynu Narasiman, Eiman Ebrahimi, other HPS members and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation. Chang Joo Lee is supported by an IBM Ph.D fellowship.

References

[1] P. Akl and A. Moshovos. Branchtap: Improving performance with very few checkpoints through adaptive speculation control. In *ICS-21*, 2006.

[2] J. L. Aragón, J. González, and A. González. Power-aware control speculation through selective throttling. In *HPCA-9*, 2003.

[3] D. N. Armstrong, H. Kim, O. Mutlu, and Y. N. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *MICRO-37*, 2004.

[4] A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *ISLPED*, 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-27*, 2000.

[6] A. Buyuktosunoglu, D. H. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. A circuit level implementation of an adaptive issue queue for power-aware microprocessor. In *Proceedings of 11th Great Lakes Symposium on VLSI*, 2001.

[7] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *ISCA-30*, 2003.

[8] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *MICRO-37*, 2004.

[9] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.

[10] S. Gochman, R. Ronen, I. Anati, A. Berkovits, T. Kurts, A. Naveh, A. Saeed, Z. Sperber, and R. C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, May 2003.

[11] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *MICRO-29*, 1996.

[12] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *ISLPED*, 2002.

[13] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[14] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA-8*, 1981.

[15] C. J. Lee, H. Kim, O. Mutlu, and Y. N. Patt. Performance-aware speculation control using wrong path usefulness prediction. Technical Report TR-HPS-2006-010, The University of Texas at Autin, Dec. 2006.

[16] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *ISCA-25*, 1998.

[17] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[18] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54(12):1556–1571, Dec. 2005.

[19] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt. Using the first-level caches as filters to reduce the pollution caused by speculative memory references. *International Journal of Parallel Programming*, 33(5):529–559, October 2005.

[20] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA-9*, 2003.

[21] J. Pierce and T. Mudge. The effect of speculative execution on cache performance. In *IPPS-8*, 1994.

[22] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *MICRO-29*, 1996.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X*, 2002.

[24] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *HPCA-13*, 2007.

[25] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.

[26] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ISCA-19*, 1992.