PROBABILITY-BASED MEMORY ACCESS CONTROLLER (PMAC) FOR ENERGY REDUCTION IN HIGH PERFORMANCE PROCESSORS.

A Dissertation

by

VAMSI KRISHNA KODATI

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Paul V. Gratz
Committee Members,	Daniel A. Jimenez
	Peng Li
Head of Department,	Jose Silva Martinez

December 2015

Major Subject: Computer Engineering

ABSTRACT

The increasing transistor density due to Moore's law scaling continues to drive the improvement in processor core performance with each process generation. The additional transistors are used to widen the pipeline, increase the size of the out-of-order instruction scheduling window, register files, queues and other pipeline data structures to extract high levels of instruction level parallelism and improve upon single-threaded performance. Such dynamically scheduled superscalar processor cores speculatively fetch and execute several instructions far ahead in a program, along the program path predicted by its branch predictors. During branch mispredictions, the architectural state of high performance processor cores can be restored at cost of high latency penalties, but the speculative memory requests sent by data memory access instructions on the mispredicted paths cannot be revoked. Such memory requests alter the data arrangement across memory hierarchy and result in wasted memory transactions, bandwidth and energy consumption. Even with low branch misprediction rates, these processor cores spend significant time on mispredicted program paths. In this thesis, we propose a probability based memory access controller to curb the data memory requests sent along mispredicted paths and achieve energy and memory bandwidth savings with minimum impact on performance. It computes path probability of instructions and throttles memory access instructions with low probability of execution. A deterministic or dynamically varying probability value is used as a threshold to control speculative memory requests sent to the memory hierarchy. The proposed design with a dynamic threshold reduces up to 51% of wrong path memory accesses and maximum of 31% of wrong path execution while achieving power savings up to 9.5% and maximum of 6.3% improvement in IPC/Watt in

a single core processor system.

DEDICATION

To my Parents, Teachers and the Almighty

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor Dr. Paul V. Gratz for his continuous guidance and support through my academic journey at Texas A&M University. I express my sincere gratitude to my advisory committee members: Dr. Daniel Jimenez, and Dr. Peng Li for their invaluable feedbacks.

Special thanks to my colleagues in the CAMSIN (Computer Architecture, Memory Systems and Interconnection Networks) research group for their help, insights and fun times.

Thanks to the Department of Electrical and Computer Engineering at Texas A&M University for their financial support in the form of one time graduate merit scholarship, graduate student employment and teaching assistantship.

I cannot thank my family and the Almighty enough for their unwavering support, belief in me and blessings for all the endeavours in my life.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

High performance dynamically-scheduled superscalar processor cores have deep pipelines, aggressive branch predictors, large instruction windows and wide issues to exploit high levels of instruction level parallelism. The number of transistors available per unit area continues to increase with each process generation due to CMOS scaling. In recent years, the additional transistors are dedicated to improve upon single-threaded performance of processor cores and integrate multiple such cores in a single die to make powerful multicore processors. In every generation, the single-threaded performance of the processor core is improved by widening the pipeline and increasing the size of the out-of-order instruction scheduling window, register files, queues and other pipeline data structures to extract more parallelism.

Dynamically-scheduled superscalar processor cores achieve high performance by speculative execution. They speculatively fetch and execute multiple instructions in every cycle along the program path predicted by their branch predictors. The average accuracy of branch predictors used in such processor cores is generally high. However, a deeply speculating core, even with low branch misprediction rates, can a spend significant number of cycles in fetching and executing instructions along the mispredicted program path [17], also known as the wrong path. They fetch, execute and issue memory requests along the wrong path until the mispredicted branch is resolved.

Upon misprediction, the architectural state of the processor can be recovered, but the data arrangement affected along the memory hierarchy cannot be restored. The impact of wrong-path references is significant in processors with longer memory latencies and larger instruction windows [17]. They impact performance positively by

indirect prefetching or negatively by causing pollution in caches [2][18][5][17]. Wrong path memory references result in increased memory transactions [21]. When multiple programs are run in multiple cores, the demand for memory increases in multicore processor systems. In such cases, the memory bandwidth and energy consumption by wrong path memory references from each core are significant in shared memory resources such as last level cache and main memory. The wrong path instructions are completely flushed from the pipeline following mispredictions. The work done by the processor core in fetching and executing wrong path instructions is wasted. Hence, wrong path instructions result in unnecessary power dissipation and energy consumption across various pipeline resources and the memory hierarchy.

Prior works [2][18][5][17][16] studied the effects of wrong path memory accesses on cache behaviour and performance of single core out-of-order superscalar processors. They provided varied solutions to mitigate cache pollution and exploit the benefits of indirect prefetching caused by wrong path memory references [19][15][20]. Earlier works [13][11][1][3] proposed designs for improving performance per watt by gating front end of the processor core when it is more likely to be on the wrong path. In our thesis, we propose a design to achieve energy and memory bandwidth savings with minimum impact on performance by controlling speculative data memory requests sent to the memory hierarchy.

We implement a probability-based throttling mechanism to curb the speculative data memory requests that are more likely to be on a mispredicted path. We use the composite confidence estimator proposed by Jimenez [9] to compute confidence of branch predictions. The ideology proposed by Malik et al [12] is used to compute path probability of instructions from branch confidence values. The path probability of an instruction depends on number of outstanding branches in the pipeline older than it. It is a dynamic value which increases as branches older than the in-

struction resolve with correct predictions. In our design, we track this continuous change in probability for all instructions in the pipeline. Using a deterministic or dynamically varying probability value as threshold, we classify instructions as low and high confidence. We leverage an observation that low confidence instructions are more probable to be mispredicted than high confidence instructions. The proposed throttling mechanism stalls low confidence speculative data memory instructions in Load-Store Queue (LSQ) until they become those of high confidence due to correct predictions. In the event of a misprediction, the stalled instructions are flushed from LSQ and thus prevent wrong path memory requests from being sent to the memory hierarchy. Reducing wrong path memory requests reduces the number of cache and memory accesses. This decreases energy consumption and power dissipation along memory hierarchy.

The rest of the thesis is divided into following sections. In the motivation section, we present the motivation behind our work. In the related works section, we discuss about works related to effects of memory accesses and execution in the wrong path by various authors in the past. We also explain the need and use of branch confidence estimation and path confidence in the background section. In the design section, we elaborate the design of our probability based memory access throttling mechanism. We explain the methodology followed to evaluate our design in the evaluation section. The results and analysis on performance, power, performance per watt, prefetching/pollution effects, wrong path memory accesses and wrong path execution of the proposed design are also presented here. In the last section, we conclude with our findings and observations.

2. MOTIVATION

Out-of-order superscalar processor cores fetch and execute instructions whose data dependencies are satisfied even before their control dependencies are resolved. They exploit instruction level parallelism by speculative execution to achieve high performance. The aggressive out-of-order processor cores have deep pipelines on the order of 15-19 stages. The branch instructions are executed at execute stage which is 13-15 cycles after fetch. Control dependent instructions cannot be committed until the control instruction is executed. In order to achieve higher performance, the instructions are speculatively fetched and executed ahead of conditional branches by predicting their outcomes. Correct predictions benefit the processor performance by eliminating the delay that would have been incurred in fetching and executing control dependant instructions due to late branch resolutions. On the other hand, the misprediction penalties can be costly and negatively impact the processor performance and power dissipation. In general, these processors have high accuracy in their predictions.

2.1 Speculative Execution

Conditional branches alter the control flow of a program. They occur regularly in the majority of program codes. Figure 2.1 shows the percentage of instructions that are branches in SPEC2006 benchmarks. On an average, every 7^{th} instruction fetched is a conditional branch instruction. The high performance processor cores have aggressive branch predictors to predict outcome of multiple such branches. The large instruction window allows them to speculatively fetch and execute instructions beyond them. As branches resolve with correct predictions, they speculate deeply along the predicted program path. They have large number of instructions that are

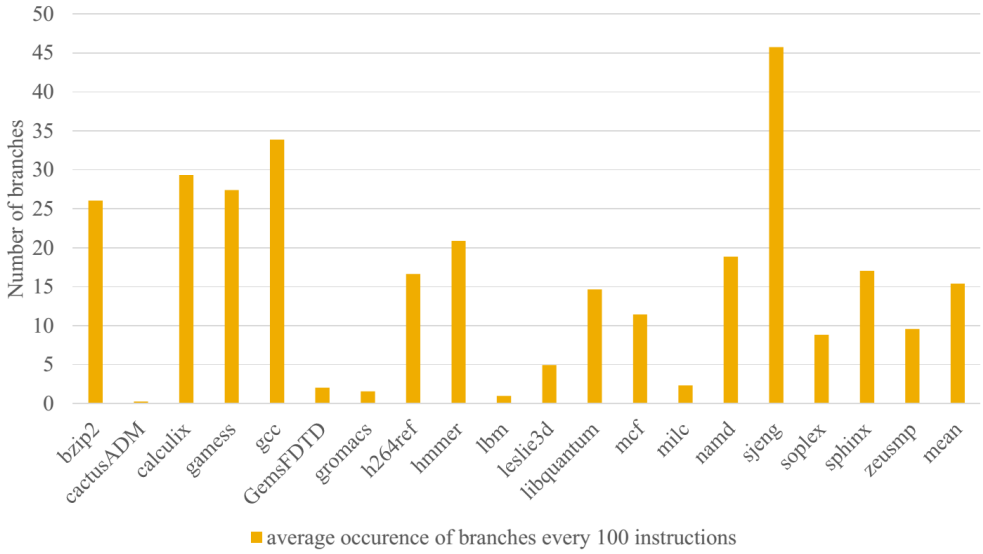


Figure 2.1: Average occurrence of branches every 100 instructions by baseline processor running SPEC2006 benchmarks.

in-flight and speculatively executed in the pipeline.

Consider a baseline processor; an 8-wide processor, with a 15-stage deep pipeline, 256-entry instruction window, 32K L1 I-Cache, 32K L1 D-Cache, 256K Unified L2 Cache, 2MB Unified L3 Cache and a 8K tournament branch predictor. Figure 2.2 shows the number of unresolved branches in flight together with the branch misprediction rate for applications in SPEC2006 suite. The figure shows that the baseline processor has an average misprediction rate of 2.8% and that, on average, nine branches are speculatively in flight at any given time.

The baseline processor speculates and fetches instructions ahead of nine conditional branches on an average relative to its point of recent commit. It is speculatively executing at least nine conditional branches and respective control dependant instructions in the pipeline. In the event of any misprediction, the results of specula-

Figure 2.2: Average number of unresolved branches and average conditional branch misprediction rate in baseline processor for SPEC2006 benchmarks.

tively executed instructions are discarded and flushed from the pipeline. Significant amount of work is wasted in fetching and executing instructions speculatively along the mispredicted path, also known as wrong path.

Figure 2.3 shows the percentage of fetched and executed instructions that are on the wrong path in the baseline processor running SPEC2006 benchmarks. A high performance processor core, even with low misprediction rates (2.8%), can fetch and execute significant number of instructions in the wrong path. In the baseline processor, on an average, 31.6% of all fetched instructions and 18.6% of all executed instructions are on the wrong path. The wrong path instructions are eventually flushed from the pipeline after the misprediction is known. Hence, 31.6% of the work done in fetching and 18.6% of work done in executing instructions is wasted work. The wrong path instructions cause unnecessary power dissipation and energy consumption in the processor core. 8% of all executed instructions are wrong path

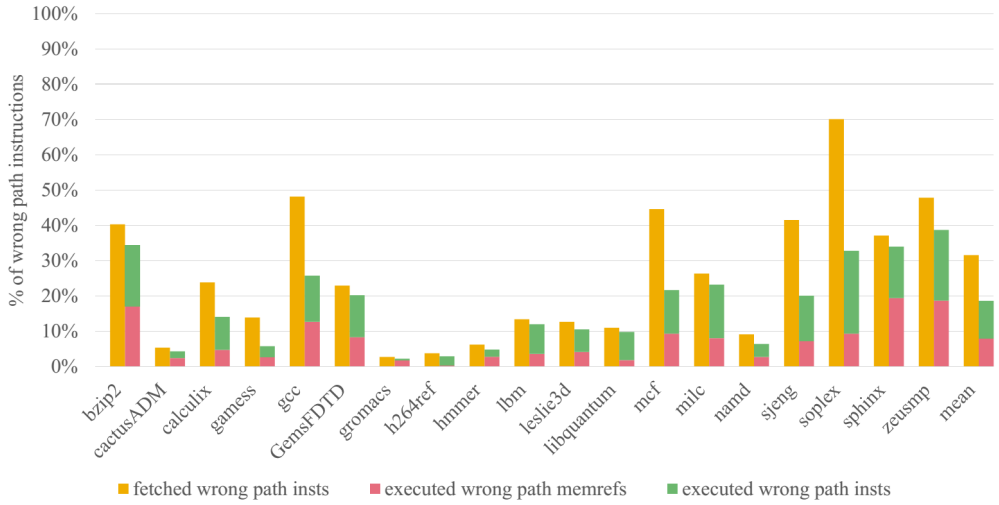


Figure 2.3: Percentage of instructions fetched and executed (memory and non-memory) on wrong path in baseline processor for SPEC 2006 benchmarks.

data memory access instructions.

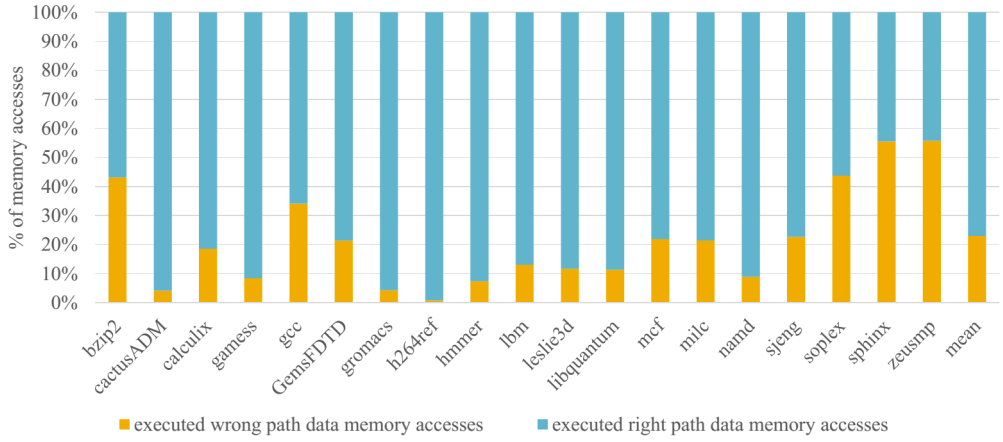


Figure 2.4: Percentage of data memory access on wrong path in baseline processor for SPEC2006 benchmarks.

Figure 2.4 shows the percentage of data memory requests that are on the wrong path in the baseline processor. 23% of all data memory accesses are wrong path data memory accesses. The additional memory accesses caused by wrong path data memory instructions change the data arrangement and increase contention and energy consumption along the memory hierarchy.

2.2 Wrong Path Execution

The speculative program path taken by processor core from the point of misprediction is referred to as the mispredicted or wrong path, and that of the actual path can be considered as the correct or right path. The architectural state of the processor can be recovered following a misprediction, but the memory requests sent by the data memory access instructions in the mispredicted path cannot be revoked. They affect the arrangement of data in the memory hierarchy and also increase memory transactions. Wrong path memory references can either result in prefetching of data for correct path instructions and increase the performance, or pollute the cache and reduce the performance. Wrong path memory references will result in prefecthing if the cache lines allocated are referenced by memory access instructions on the correct path before they are evicted. Otherwise, they cause cache pollution, unnecessary evictions, memory transactions and bandwidth usage.

There are program scenarios where wrong path execution can result in prefetching for correct path execution [17]. A mispredicted conditional branch inside a loop during an iteration can prefetch data for the correct path execution in the same iteration. Two different loops working on the same data structure can prefetch data for each other. Misprediction of hammock branch can prefetch data for instructions on the correct path if both paths of the hammock need the same data. If a portion of the code is common for the mispredicted and correct path, then the wrong path

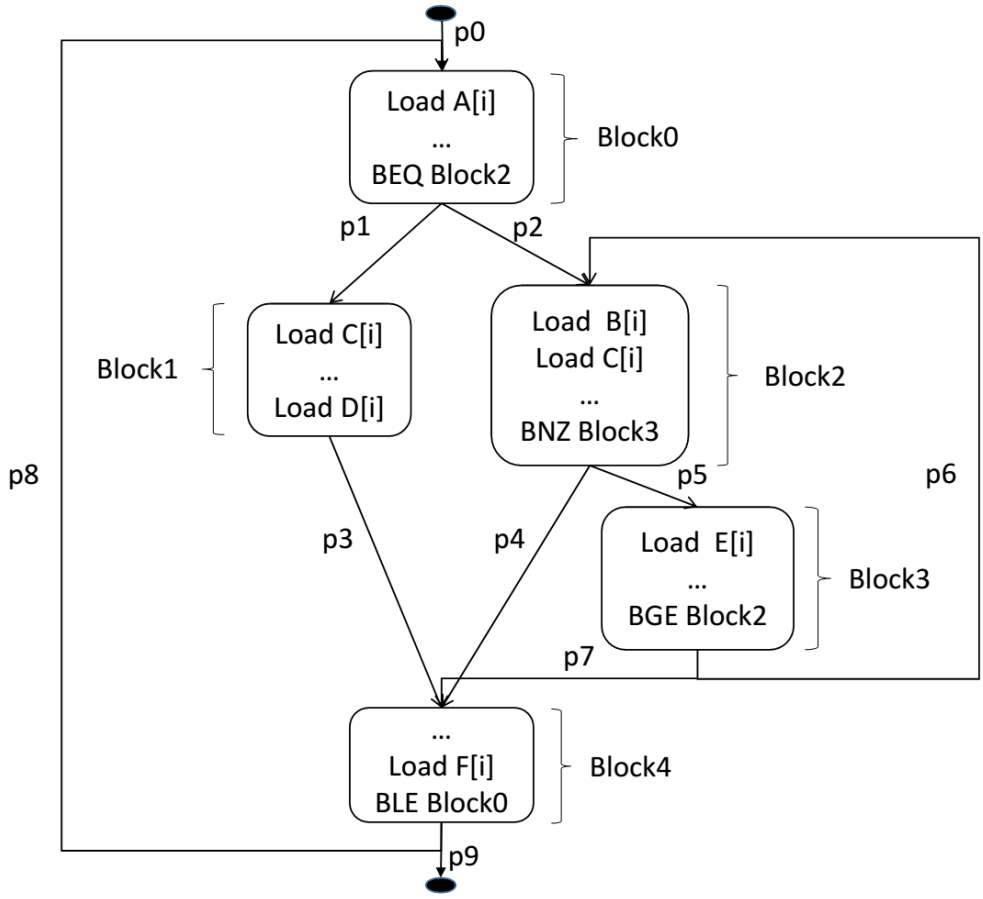


Figure 2.5: Program flow snippet.

memory references prefetch data for later correct path execution. If the correct path execution has no code or data structure in common with the wrong path execution, the wrong path memory references result in cache pollution and increase memory transactions.

Figure 2.5 shows snippet of a program code with different paths that can be taken by a processor. In the figure, the path $p0 \rightarrow p2 \rightarrow (p5 \rightarrow p6)^i \rightarrow p7 \rightarrow p9$ prefetches data for $C[i]$ in path $p0 \rightarrow (p1 \rightarrow p3 \rightarrow p8)^i \rightarrow p9$ and vice versa. If the branch instruction in $Block0$ is mispredicted, the loads for $C[i]$ are prefetched but

the loads for $D[i]$ or $B[i]$ can cause cache pollution depending upon the mispredicted path. The paths $p0 \rightarrow (p1 \rightarrow p3 \rightarrow p8)^i \rightarrow p9$ and $p0 \rightarrow (p2 \rightarrow p4 \rightarrow p8)^i \rightarrow p9$ execute common code in $Block4$ and $Block0$ every iteration, hence either of the path prefetches data $A[i]$ or $F[i]$ for the other. The loads for $B[i]$ or $D[i]$ cause cache pollution based on the load instructions on the mispredicted path. From Figure 2.5, we can say that memory references to $A[i]$, $C[i]$ and $F[i]$ are more likely to result in prefetching while $B[i]$, $D[i]$ and $E[i]$ are more likely to cause cache pollution during mispredictions.

Eliminating the wrong path memory accesses can bring significant energy savings by reducing data accesses in the memory hierarchy and execution of wrong path data dependent instructions. This can impact performance in one of the two ways. It can either improve processor performance by reducing cache pollution, or decrease the performance by taking away the indirect data prefetching for correct path instructions. We have observed from the code snippet in Figure 2.5 that, not all wrong path memory accesses results in prefetching. We believe that the power savings achieved by reduction in wrong path memory accesses are more significant and deterministic than the indirect prefetching. Our goal is to improve the performance per watt of the processor by reducing the wrong path memory accesses.

3. RELATED WORK

Several works in the past studied the effects of wrong path instructions on cache behavior and performance of out-of-order processor cores. Varied solutions were provided to reduce cache pollution and exploit benefits of indirect prefetching caused by wrong path memory references. Fetch gating mechanisms to reduce the number of instructions fetched and executed in the wrong path for achieving energy reduction and power savings were also proposed in the past. The branch confidence estimators were commonly used in these techniques to estimate the likelihood of the processor being on the wrong path.

3.1 Wrong Path Pollution And Prefetching

Pierce and Mudge [18] did one of the earliest studies on the effects of speculative execution on cache performance. Combs et al [5] also studied the effects of wrong-path memory references on cache behaviour and processor performance. They showed that wrong path memory references can positively impact performance by prefetching data for correct path instructions and can also negatively impact performance by causing pollution in caches. Bahar and Albera [2] proposed a branch confidence predictor based filtering to reduce cache pollution caused by wrong path memory references. The state of prediction counters in the McFarling [14] hybrid branch predictor tables is used to determine the confidence of branch predictions. They label the branch predictions as high confidence or low confidence based on the state of counters predicting the branch outcome. If the number of unresolved low confidence branches in the pipeline exceeds a certain pre-set threshold, the processor is estimated to be on the wrong path. The cache refills of probable wrong path memory accesses from second level cache are placed in a separate fully-associative 16-entry buffer

known as a confidence buffer instead of the first level cache. The confidence buffer is accessed in parallel with L1 cache whenever a memory request is serviced. It reduces pollution caused by wrong path memory instructions in caches and provides prefetching benefit, if any, for the later memory references.

Mutlu et al. [17] analyzed performance impact of wrong path memory references in uniprocessor systems with large instruction windows and longer memory latencies. They studied the effects of indirect prefetching and pollution caused by wrong path memory references. The importance of modelling wrong path memory references in out-of-order processor simulations was emphasized. Different code structures that cause prefetching by wrong path memory references were examined. They found that the pollution caused by wrong path references in L2 cache has a significant negative impact on performance than that of L1 cache. From this observation, Mutlu et al. proposed a technique [15] to filter useless speculative memory references to the L2 cache and reduce pollution. They made an observation that if a speculatively fetched cache block in the L1 cache is not used, then it is more likely that the block will not be used before it is evicted from the L2 cache. The L1 cache is used as filter to predict usefulness of speculative memory references. Two policies to control the writes of speculatively fetched blocks to L2 cache and reduce the L2 pollution were proposed. One of the policies is to not write these fetched blocks into L2 cache, while the other policy writes them into LRU positions in the L2 cache. In both the policies, speculatively fetched cache blocks unreferenced by non-speculative memory instructions are not written into L2 cache during L1 cache evictions. IPC improvement was observed in a few of the benchmarks by reduction of L2 cache pollution caused by wrong path memory references.

Sendag et al. [21] studied effects of wrong path memory references in shared memory multiprocessor (SMP) systems. They noticed significant number of unnecessary

cache line state transitions, replacements, writebacks, invalidations, coherence traffic and resource contention due to memory accesses along wrong path. A cache filtering and wrong path aware cache replacement policies was proposed to reduce wrong path memory accesses. They make use of the observation [16] that branch mispredictions are usually resolved before most of the wrong path L1 misses complete. By using speculative tags, misses by speculative memory accesses were tracked and identified in MSHR entries. The cache blocks brought by them are marked as wrong path blocks after the branch misprediction is known. They filter and reduce pollution in the L2 cache by not writing evictions of wrong path blocks unreferenced by correct path instructions from the L1 cache. Wrong path blocks were further filtered from being brought into L2 cache by cancelling the miss requests of wrong path memory accesses after the misprediction was known. In wrong path aware replacement policy, the blocks marked as wrong path are evicted first on an LRU basis. When the blocks marked as wrong path are referenced by correct path instructions, they get marked as correct path instructions. The evicted wrong path blocks are placed in LRU position in the higher level cache. Their cache filtering and replacement policy showed performance improvement in a few of the benchmarks in the SMP systems.

3.2 Wrong Path Fetch Gating

Manne et al. [13] proposed fetch gating mechanism based on a confidence estimator to save power/energy by reducing the number of extra executed instructions along the wrong path. The branches with low confidence were more likely to be mispredicted than those with high confidence. They engage gating of fetch unit of the pipeline when the number of unresolved low confidence branches exceeds a threshold value. The gating is disengaged when the number of unresolved low confidence branches are less than or equal to the threshold. The pipeline gating technique was

evaluated with different confidence estimators. Although the gating removes extra work done by the wrong path instructions, it introduces performance loss.

Lee et al [11] proposed a performance aware fetch gating mechanism. The gating mechanism takes into account the possible prefetching benefits of wrong path instructions. They introduced a wrong path usefulness predictor (WPUP) to predict if mispredicted branches lead to useful prefetches. They proposed two schemes at different granularities to predict the usefulness of wrong path memory references. A branch PC based WPUP, a fine grained mechanism, predicts the usefulness of mispredicted branch. It uses a set associative cache like structure to store PCs of mispredicted branches causing prefetches as tags. The WPUP cache doesn't store any data. A phase based WPUP, a coarse grained mechanism, predicts usefulness of wrong path memory references during different program phases. L2 MSHRs are used to detect the usefulness of mispredicted branches and wrong path memory references. The branch PC and branch ID of recently fetched branch are stored along with every load/store memory request missing in L2 in the MSHRs. After the branch misprediction is known, the ID of the mispredicted branch is sent to L2 MSHRs. All the MSHRs entries with the same or later ID as the mispredicted branch are marked as wrong path. If an outstanding wrong path MSHR entry is hit by later memory requests to the same cache line, the mispredicted branch associated with the memory request is identified as useful. In branch based WPUP, the branch PC in the MSHR entry is stored in WPUP cache upon wrong path MSHR entry hit. While in phase based WPUP, the wrong path usefulness counter (WPUC) is incremented. In the WPUP mechanism, a memory request can be successfully identified as a wrong path request if it's not fully serviced when the misprediction is resolved. The usefulness of this request can be detected if the correct path instructions send memory requests to the same cache line before the request is serviced.

Using WPUP and a number of in-flight unresolved branches information, a fetch gating mechanism to achieve power savings with minimum impact on IPC is modelled. If the number of unresolved branches in the processor pipeline is larger than a certain threshold, then they enable fetch gating. This threshold is set based on the branch prediction accuracy in a given time interval. The threshold is high when the branch prediction accuracy is high. The threshold is low when the branch prediction accuracy is low. In branch based WPUP, the fetch unit tracks the PC of the latest branch in a register; if it is present in the WPUP cache, then the gating decision is discarded. In phase based WPUP, if the value in WPUC exceeds a certain threshold, then the gating decision is discarded. The WPUC counter is reset every 100K cycles in order to detect the phase behaviour of wrong path usefulness. When the number of unresolved branches in the pipeline is less than the threshold, the WPUP mechanism disables gating to fetch useful wrong path instructions.

Aragon et al. [1] proposed a power aware branch confidence based selective throttling of mis-speculated instructions to achieve energy reductions. They classify branches based on predictions as very-high confidence (VHC), high confidence (HC), low confidence (LC) and very-low confidence (VLC) branches. The fetch and decode bandwidth of the pipeline is reduced to half, if a VLC branch is fetched, and further reduced to quarter. The units are stalled if more VLC and LC branches are fetched following the VLC branch. They avoid selection of instructions in the pipeline wakeup logic that are control dependant on low confidence branches. They aggressively throttle instructions following VLC branches and less aggressively so on instructions that follow LC branches.

Malik et al. [12] proposed a path confidence based fetch gating mechanism to reduce fetch and execution of wrong path instructions in the pipeline. The previous techniques use the number of unresolved branches or low confidence outstanding

branches in the pipeline as a rough measure for probability of processor is on the correct path. They do not consider branch misprediction rates. The authors propose a more accurate method for measuring the likelihood of the processor fetching instructions on the correct path, also known as path confidence. The branch confidence estimator associates every branch prediction with a confidence value. On a per confidence value basis, the number of branch mispredictions in a given time interval are counted. The misprediction rates are tracked for the whole range of branch confidence values and then used to compute misprediction probabilities per branch confidence value. The path confidence is computed as a product of misprediction probabilities of all unresolved branches in the pipeline. They use a deterministic path confidence value as a threshold to control gating. The fetch gating is engaged when the path confidence of the processor goes below the threshold value. The probability based path confidence estimate is more accurate than a counter based estimate. Hence, the fetch gating showed better results.

The above mentioned gating mechanisms throttle the fetch and execution of all low confidence instructions. A deeply speculating core with a high accuracy branch predictor can spend significant amounts of time in fetching low confidence instructions. These low confidence instructions eventually become those of high confidence when branches in flight resolve with correct predictions. The fetch gating mechanisms prevent wrong path memory accesses in the event of a misprediction, but they cause performance losses in the processor due to reduced speculative execution of instructions. In our approach, the wrong path memory accesses are reduced by only throttling low confidence memory access instructions instead of all low confidence instructions. We allow speculative fetching and execution of low confidence data independant instructions. Thus, we can reduce wrong path memory accesses with lesser impact on the performance than the fetch gating mechanisms.

4. BACKGROUND

The architectural designs focussed on reducing wrong path execution must distinguish wrong path instructions from the dynamic stream of fetched instructions. Identifying wrong path instructions in an out-of-order processor is not straightforward. We cannot tell with absolute certainty if an instruction is on the correct or wrong path until all the branch predictions older than the instruction are resolved. However, we can tell if an instruction is on the correct or wrong path with a probability based on the information about outstanding branches in the pipeline. Malik et al. [12] used a path confidence predictor to estimate the probability that the processor is on the correct path, also known as path confidence. Conventionally, the number of unresolved, low-confidence branches is used as an estimate for likelihood that the processor is on the correct path. They showed that this conventional approach is inaccurate because it assumes that all low-confidence branches have the same misprediction rates and high-confidence branches never mispredict. They proposed a more accurate method of computing path confidence from the misprediction rates of all unresolved branches having different confidence values. We extend the concept of path confidence to compute and track the probability that an instruction is on the correct path during its lifetime in the pipeline in our design.

The path confidence predictor [12] classifies branches based on their confidence values and computes misprediction rates for all branch classifications. We use the composite confidence estimator proposed by Jimenez [9] to compute branch confidence values. Manne et. al [6] emphasizes that SPEC and PVN should be high for confidence estimators employed in architectures focussed for energy reduction. SPEC and PVN are two among four statistical metrics proposed by Manne et al. [6] for eval-

uating and comparing the performance of confidence estimators. Specificity (SPEC) is the probability that a mispredicted branch has a low confidence value. Predictive value of negative estimate (PVN) is the probability that a low confidence branch is mispredicted. Having high SPEC and PVN implies that most of the mispredicted branches have low confidence values and most of the low confidence branches are mispredicted. The likelihood of a processor to be on a wrong path can be estimated from the unresolved low confidence branches which can be used for some form of speculation control. The $Composite\ Up/Down + JRS + Self$ confidence estimator [9] has high and wide range of SPEC and PVN values than individual estimators for hybrid branch predictors.

4.1 Composite Up/Down+JRS+Self Confidence Estimator

The $Composite\ Up/Down + JRS + Self$ confidence estimator [9] is built by combining Enhanced JRS, Up/Down and Self branch confidence estimators. If C_{jrs}, C_{ud} and C_{self} are branch confidence estimations of Enhanced JRS, Up/Down and Self estimators respectively, then the branch confidence estimation of composite estimator C_{comp} is given as,

$$C_{comp} = C_{jrs} + C_{ud} + C_{self} \tag{4.1}$$

The branch confidence estimators use tabular structures and history registers either similar to or that of hybrid branch predictor for confidence estimation. Branch confidence tables and branch prediction tables are looked up and updated in parallel when branches are fetched and resolved respectively.

4.1.1 Enhanced JRS estimator

The JRS estimator [8] uses miss distance counter (MDC) table in addition to branch predictor tables for confidence estimation. The miss distance counter counts

number of correct branch predictions since a misprediction. It is a saturating counter which is incremented when a branch prediction is correct and reset upon misprediction. The global MDC table is similar to GAg prediction history table (PHT). It consists of miss distance counters indexed by a global history register. In the enhanced JRS estimator, the branch confidence estimation is done after the branch prediction. The branch prediction is updated in the global history register and then the miss distance counters are referenced.

4.1.2 Up/Down estimator

The Up/Down estimator [10] uses up/down counter (UDC) table in addition to branch predictor tables for confidence estimation. The up/down counter also counts number of correct branch predictions since a misprediction similar to the JRS estimator. However, the counter is decremented instead of resetting upon misprediction. It is a saturating counter which is incremented when a branch prediction is correct. The local UDC table is similar to second level PAg prediction history table (PHT) structure. The first level PAg predictor table containing local history registers indexed by the branch address is used to index the local UDC table. The local UDC is a second level table table which consists of up/down counters.

4.1.3 Self estimator

The self estimator [6] computes confidence estimation from the saturating counters in the branch prediction history tables. If c is the value in saturating counter and n is its width in bits, then the confidence estimate for the branch which is taken is c and for the branch which is not taken is $2^n - c - 1$. The self estimator does not need any additional tabular structures. The global branch confidence can be computed directly from the counter value referred by the branch in global GAg PHT. Similarly the local branch confidence is computed from the counter value referred

by the branch in local PAg PHT table. The total confidence is a sum of global and local confidence values.

4.2 Path Confidence Estimation

The path confidence [12] is the probability that the processor is fetching instructions on the correct path. It considers contributions of all unresolved branches, both high and low confidence. The path confidence is the product of correct prediction rates of all unresolved branch instructions in the pipeline. In other words, it is the product of the probability of correct prediction of all outstanding branches. If there are n outstanding branches in the pipeline, the path confidence of the processor is,

$$PathConf = \prod_{i=0}^{j} P_{br_i} \qquad 0 \leq P_{br_i} \leq 1 \quad \forall j \; in \; Unresolved \; Branches \tag{4.2}$$

where P_{br_i} refers to the probability of correct prediction of i^{th} unresolved branch in the pipeline.

The branches are classified into different confidence buckets based on the confidence value obtained from the branch confidence estimator. If the composite confidence estimator has n confidence estimators and w_i is the width of confidence counters in i^{th} confidence estimator, then the confidence value c is bounded by,

$$0 \leq c \leq \sum_{i=0}^{n} 2^{w_i} - n - 1 \tag{4.3}$$

The total number of confidence buckets is the range of confidence values which is $\sum_{i=0}^{n} 2^{w_i} - n$. In each confidence bucket we count the number of branches with confidence c that resulted in correct predictions and mispredictions in last N cycles. If R_c and W_c are number of branches with confidence c that had correct predictions and

mispredictions respectively in last N cycles, then the probability of correct prediction of a branch with confidence c is given as,

$$P_{br_c} = \frac{R_c}{R_c + W_c} \tag{4.4}$$

For example if a composite confidence estimator has enhanced JRS with 3-bit miss distance counters, Up/Down estimator with 5-bit up/down counters and the Self estimator with 3-bit saturating counter, then the confidence value of the composite confidence estimator is an integer c such that $0 \leq c \leq 44$. So, we have 45 confidence buckets in total, one bucket per confidence value.

5. PROBABILITY-BASED MEMORY ACCESS CONTROLLER (PMAC) DESIGN

In this section we elaborate on the architectural design of Probability-Based Memory Access Controller (PMAC) to reduce the wrong path data memory accesses. Our proposed design aims to identify probable wrong path memory access instructions and stall them until the branch mispredictions are known. We extend the concept of path confidence introduced by Malik et al. [12] to compute path probability of instructions. The path confidence predictor [12] computes the probability that the processor is on the correct path. Here, we use similar technique to compute and track path probability of dynamic instructions in the pipeline. The correct path instructions are eventually committed in-order while wrong path instructions are flushed by the processor core. The probability of an instruction getting executed and committed depends on the program path it lies on. It is the same as the probability of the processor taking the path containing the instruction from its point of recent commit. We refer to this as path probability of an instruction.

Figure 5.1 illustrates the overall architecture of the Probability-based Memory Access Controller (PMAC) in an out-of-order processor core. The unshaded regions constitute the major components in an out-of-order CPU pipeline. The light shaded regions correspond to existing components in the CPU pipeline which are modified to implement PMAC. The dark shaded regions are the newly added components in the out-of-order core parallel to the main CPU execution pipeline. They operate in parallel with main CPU pipeline. The BlockID counter is used for assigning unique identification number for every branch fetched by the CPU fetch engine. The conditional branch and its control dependent instructions are assigned with

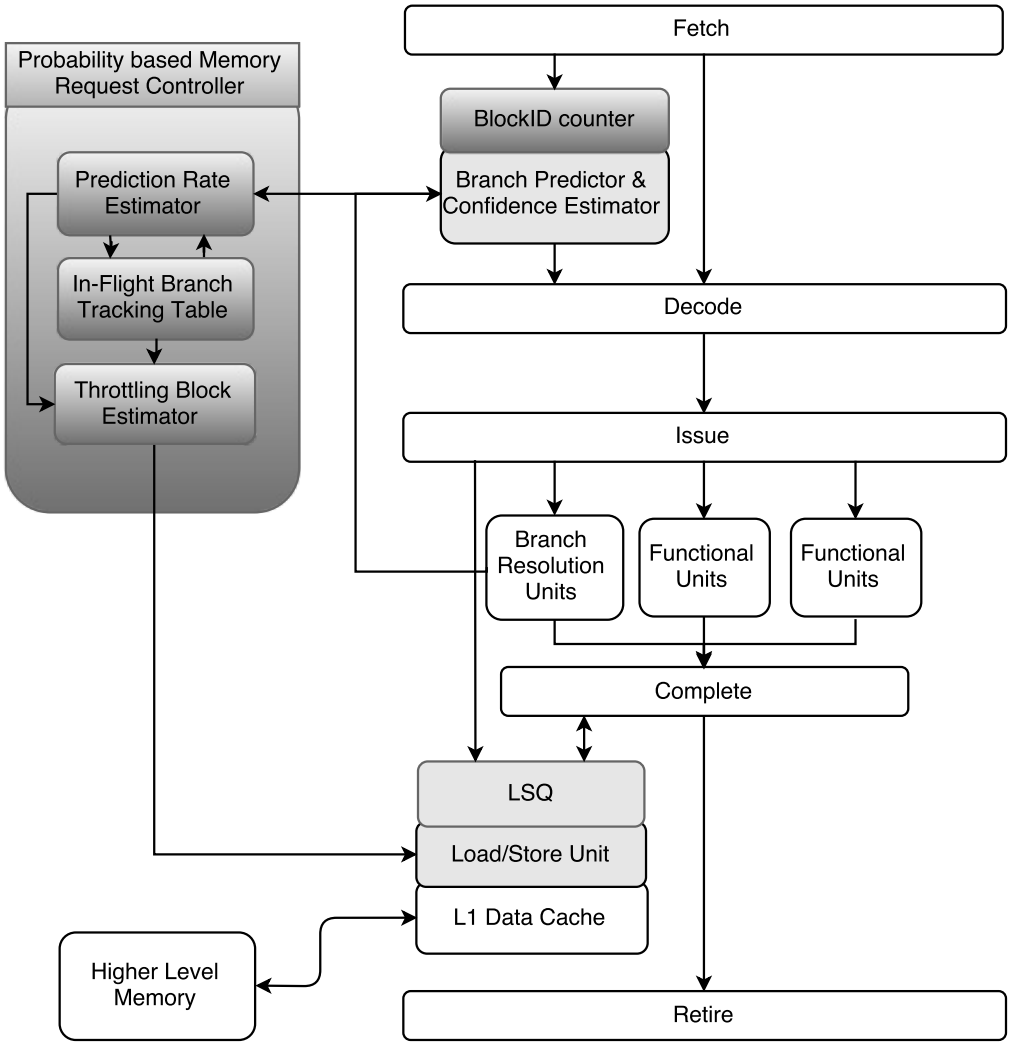


Figure 5.1: Microarchitecture of probability based memory throttle

same BlockID. The branch predictor and confidence estimator provides prediction and confidence values respectively for the fetched branches. The prediction rate estimator provides correct prediction rate for the fetched or resolved branch. The

in-flight branch tracking table keeps track of unresolved branches in the pipeline. The throttling block estimator identifies the BlockID of oldest program block in the pipeline with path probability less than the threshold. It provides the BlockID of memory instructions to be throttled at the load store unit. The Load-Store Queue (LSQ) structure in the main CPU pipeline is modified to store BlockID and a stall bit for every entry. The Load-Store Unit (LSU) is modified to control the issue of memory instructions. It does not issue memory requests for loads and stores with BlockID greater than or equal to the BlockID provided by the throttling block estimator. In the remainder of this section we describe each of these components in detail, followed by a detailed discussion of their operation together.

5.1 BlockID Counter

The control instruction altering the dynamic flow of a program and its control dependant instructions constitute a program block. All the instructions in a program block are identified with a unique block identification number (BlockID). The BlockID counter at the fetch unit is used to keep track of BlockID of currently fetching program block. The processor is fetching next dynamic program block in the program code whenever a new control instruction is encountered. Hence, the BlockID counter is incremented whenever a new control instruction is fetched. The fetched instructions are assigned to BlockID of recently fetched control instruction. Thus, all the instructions in a program block are assigned with the same BlockID.

5.2 Branch Predictor And Confidence Estimator

The branch predictor predicts the outcome of conditional branches fetched by the processor. The branch confidence estimator provides a confidence level for the predictions made by the branch predictor. Higher the value, higher is the confidence in its prediction and vice-versa. Tracking and maintaining correct prediction rates for

every branch in the program is difficult. We determine and manage correct prediction rates for branches having same confidence value. The branch confidence value is used to track and maintain the correct prediction rates of branches encountered in a program. This technique was initially proposed by Malik et al. [12]. Wider the range of branch confidence estimator, greater is the number of branch confidence levels, finer is the granularity and greater is the accuracy of correct prediction rate of branches. The McFarling hybrid branch predictor [14] with $JRS + Up/Down + Self$ composite confidence estimator [9] provides a wide range of confidence values than the individual confidence estimators. Hence we employ a composite confidence estimator in our design.

5.3 Prediction Rate Estimator

The prediction rate estimator contains a branch probability table which provides correct prediction rates for branches with different confidence values. It classifies branches based on their confidence value into different confidence buckets of size 1. It tracks the number of branches that are fetched and committed over a period of N cycles for all confidence buckets in a per confidence branch fetch count (BFCT) and commit count table (BCCT). The BFCT table entry is incremented following every branch fetch while the BCCT table entry is incremented following every correct branch resolution. After every N cycles, the per confidence branch probability table (BPT) is updated with $\frac{Number\ of\ Branch\ Commits}{Number\ of\ Branch\ Fetches}$ value for every confidence bucket. The BPT table contains the correct prediction rate of branches for all confidence buckets. All the per confidence based tables in the prediction rate estimator are accessed using branch confidence value.

5.4 In-flight Branch Tracking Table

The in-flight branch tracking table tracks the outstanding branches in the pipeline. It consists of a per BlockID Branch Confidence Table (BBCT) which contains branch confidence values of in-flight branches. It is a direct mapped structure where table entries are accessed using lower bits of the BlockID of the branch. Whenever a branch is fetched, the table entry directly mapped to its BlockID is updated with its confidence value and marked as valid. Whenever a branch is resolved or flushed, the table entry directly mapped to its BlockID is invalidated. Unlike direct mapped cache, this structure does not store any tags.

5.5 Throttling Block Estimator

The throttling block estimator identifies the BlockID of oldest program block with path probability less than the threshold. The path probability of an instruction is dependant on the correct prediction rates of in-flight branches older than the instruction. All instructions in a given program block have same path probability as they all lie on same speculative program path determined by the in-flight branches older than them. We refer to this as path probability of a program block.

5.5.1 Calculating the path probability of program blocks

If there are n outstanding branches in the pipeline, the path probability of any j^{th} program block is,

$$P_j = \prod_{i=0}^{j} P_{br_i} \qquad 0 \leq P_{br_i} \leq 1$$

$$\forall j \in Unresolved\ Branches \quad | \quad BlockID_{branch} \leq BlockID_{instruction} \tag{5.1}$$

where P_{br_i} refers to the probability of correct prediction of i^{th} older and unresolved branch in the pipeline. The path probability of a younger program block cannot be greater than the path probability of any older program blocks.

$$P_{j-1} \leq P_j \leq P_{j+1} \qquad \forall j \; in \; Unresolved \; Branches \tag{5.2}$$

The throttle block estimator has a path probability register (PPR) to hold the path probability of program block under consideration.

5.5.2 Determining the program blocks to be throttled

If P_{Thresh} refers to the pre-set or dynamically varying path probability threshold, we need to identify a program block j whose path probability is such that,

$$P_{j-1} > P_{Thresh} \geq P_j \tag{5.3}$$

If $\exists$ an outstanding program block j with path probability satisfying Equation 5.3, then this block is known as throttling block. From Equation 5.2, all the instructions in and after the program block j have path probability less than P_{Thresh}. These instructions are referred as low confidence instructions. All the instructions before the j^{th} program block are referred as high confidence instructions.

5.5.3 Block ID based throttling

The block identification number of any outstanding program block j in the pipeline is such that,

$$BlockID_{j-1} > BlockID_j > BlockID_{j+1} \;\; \forall j \; in \; Outstanding \; Program \; Blocks \tag{5.4}$$

If $BlockID_T$ is the block identification number of program block satisfying Equation 5.3, then all the memory access instructions with block identification number greater than or equal to $BlockID_T$ will be stalled in the Load-Store Queue (LSQ). All the memory access instructions with block identification number less than $BlockID_T$ will go ahead to issue data memory requests. The $BlockID_T$ is used to demarcate the low confidence and the high confidence instructions. The low confidence memory access instructions are stalled while the high confidence memory access instructions are allowed to issue memory requests.

As the outstanding branches in the pipeline resolve with correct prediction, the throttling block satisfying the Equation 5.3 is recovered. The $BlockID_T$ of recovered throttling block will be greater than or equal the old $BlockID_T$. The stalled memory access instructions with block identification number less than the recovered $BlockID_T$ are released. The low confidence memory instructions become high confidence when branches resolve with correct prediction and goes ahead to issue data memory requests.

When an outstanding branch in the pipeline is resolved with misprediction, then all the instructions fetched after the mispredicted branch are flushed from the pipeline. All the stalled memory access instructions younger than the mispredicted branch are flushed from the Load-Store Queue, thus avoiding the issue of wrong path memory requests.

The throttle block estimator has a throttling block register (TBR) and a throttle control (TC) bit. When the value in path probability register (PPR) is less than or equal to the threshold, then the TBR holds the BlockID of oldest program block with path probability less than the threshold. When the value in PPR is greater than the threshold, then the TBR holds the BlockID of currently fetching program block. TC bit is used to specify whether the memory request throttling has to be

engaged or disengaged at the Load-Store Unit in the pipeline.

5.5.4 Determining the path probability threshold

We define two approaches for determining the path probability threshold. The first approach is a static methodology, where the path probability threshold is a pre-set empirical value and remains constant throughout the program execution. The path probability threshold indirectly controls the depth of speculation or amount of speculative execution in an out-of-order processor core. A high threshold value decreases the speculation depth of the processor and can inhibit the progress of correct path instructions. While, a low threshold value can miss out opportunities to reduce wrong path execution. For any given program, a suitable static threshold can only be determined experimentally. Also, having a constant pre-set threshold value may not be beneficial at all executing conditions. When the average execution time of instructions and the number of stalled memory access instructions in the processor core is low, it is favorable have a higher threshold. Similarly, when the average execution time of instructions in processor core and the number of stalled memory access instructions is high, it is preferable to have a lower threshold. It is desirable to have different threshold values at different circumstances. Hence our second approach is a dynamic methodology, where the path probability threshold is varied during the program execution. In the dynamic approach, we vary the threshold based on number of free entries in the instruction window, current threshold value and the number of stalled memory access instructions. We use the average execution time of instructions indicated by the number of free entries in the instruction window to determine whether the threshold value has to be increased or decreased. The current threshold value is used to decide the step size for incrementing or decrementing the threshold value. While, the number of stalled memory access instructions is used to

select the frequency of threshold update.

The path probability threshold is initially set to 0.01. The threshold is increased in steps of 0.1 for every n cycles when the instruction window is less than 75% full and the current threshold value is less than 0.5. When the instruction window is less than 75% full and the current threshold is between 0.5 and 0.95, the threshold is increased in steps of 0.01 for every n cycles. The threshold remains unchanged when the instruction window is 75% to 85% full. If the instruction window is more than 85% full and the memory access instructions are stalled, then the threshold is decreased in steps of 0.1 every n cycles. The threshold is decreased to 0.01 when the instruction window becomes full and the memory access instructions are stalled. The interval n is a variable which varies exponentially with the number of stalled memory instructions m. When more than 25% of the instruction window is free,

$$n = K_1 * 2^{\lfloor m/C_1 \rfloor} \tag{5.5}$$

where K_1 and C_1 are constants. The n is doubled for every C_1 number of stalled memory access instructions. When 85% of the instruction window is free,

$$n = K_2 * 2^{\lfloor -m/C_2 \rfloor} \tag{5.6}$$

where K_2 and C_2 are constants. The n is halved for every C_2 number of stalled memory access instructions. Table 5.1 shows how the path probability threshold is varied under various scenarios.

5.6 Logarithmic encoding and storing of probability values

Malik et al. [12] has proposed the use of logarithmic encoding and scaling of probability values between 0 and 1 to ease the hardware implementation. Simi-

Table 5.1: Varying path probability threshold

Free Entries in Inst. Window	Current Threshold	No. of Stalled Insts.	New Threshold	Interval (n)
$\geq 25\%$	≤ 0.50	≥ 0	$\uparrow$ *by* 0.10%	$K_1 * 2^{\lfloor m/C_1 \rfloor}$
$\geq 25\%$	> 0.50 & < 0.95	≥ 0	$\uparrow$ *by* 0.01%	$K_1 * 2^{\lfloor m/C_1 \rfloor}$
$\leq 15\%$	≤ 0.95	> 0	$\downarrow$ *by* 0.10%	$K_2 * 2^{\lfloor -m/C_2 \rfloor}$
0%	≤ 0.95	> 0	$\downarrow$ *to* 0.01%	0

lar logarithmic encoder and scaler can be employed in our design. The multipliers required for computing the path probability in Equation 5.1 can be replaced with simple adders. When the log is taken in both sides of Equation 5.1, it results in following equation which requires only additions instead of multiplications,

$$\log_2(P_j) = \sum_{i=0}^{j} \log_2(P_{br_i}) \quad \forall j \; in \; Unresolved \; Branches \tag{5.7}$$

The probability value lies between 0 and 1. $\log_2(P_j)$ is a negative number, which is scaled by multiplying with -1024 and rounded off to the closest integer. We can logarithmically encode and scale all the probability values in our design.

$$Enc(P_j) = -1024 * \log_2(P_j) \quad \forall j \; in \; Unresolved \; Branches \tag{5.8}$$

Due to logarithmic encoding and negative scaling done in Equation 5.8, the higher the probability value, the lower the final encoded value. For example the encoded value of 0.25 will be 2048, the encoded value of 0.75 will be 425. Hence, the instructions with encoded probability greater than or equal to the encoded path probability threshold will be low confidence instructions. The instructions with encoded probability less than the encoded path probability threshold will be high confidence

instructions. We never need to convert the encoded probabilities back into real probabilities. The hardware complexity and implementation of logarithmic encoder and scaling is discussed by Malik et al in [12].

5.7 Load Store Queue (LSQ) And Load Store Unit

The Load Store Queue (LSQ) contains the information of in-flight loads and stores until the memory requests are serviced. The LSQ structure is modified to store BlockID and a stall bit for each entry. The BlockID in TBR and TC bit is used to search and set/clear stall bits for loads and stores in the LSQ. The Load Store Unit picks eligible loads and stores from LSQ to issue memory request to the memory hierarchy. It is modified to ignore picking up and issuing memory requests for eligible loads and stores with stall bit set when throttling is engaged (or TC is set).

5.8 PMAC Operation

The components proposed in our design execute in parallel to the main CPU pipeline. They process branch instructions after they are fetched and executed in the main CPU pipeline to compute path probabilities of in-flight program blocks. The BlockID of program block with path probability less than the threshold is identified and communicated to Load-Store Unit (LSU) of the pipeline. The LSU is modified to throttle low confidence memory requests. Figure 5.2 shows a flow chart briefing the operation of PMAC during different phases.

5.8.1 Fetch phase

Whenever the fetch unit encounters a branch instruction, the global BlockID counter (BIDC) is incremented and assigned as the BlockID of the branch. The branch prediction and confidence tables are looked up for the prediction and confi-

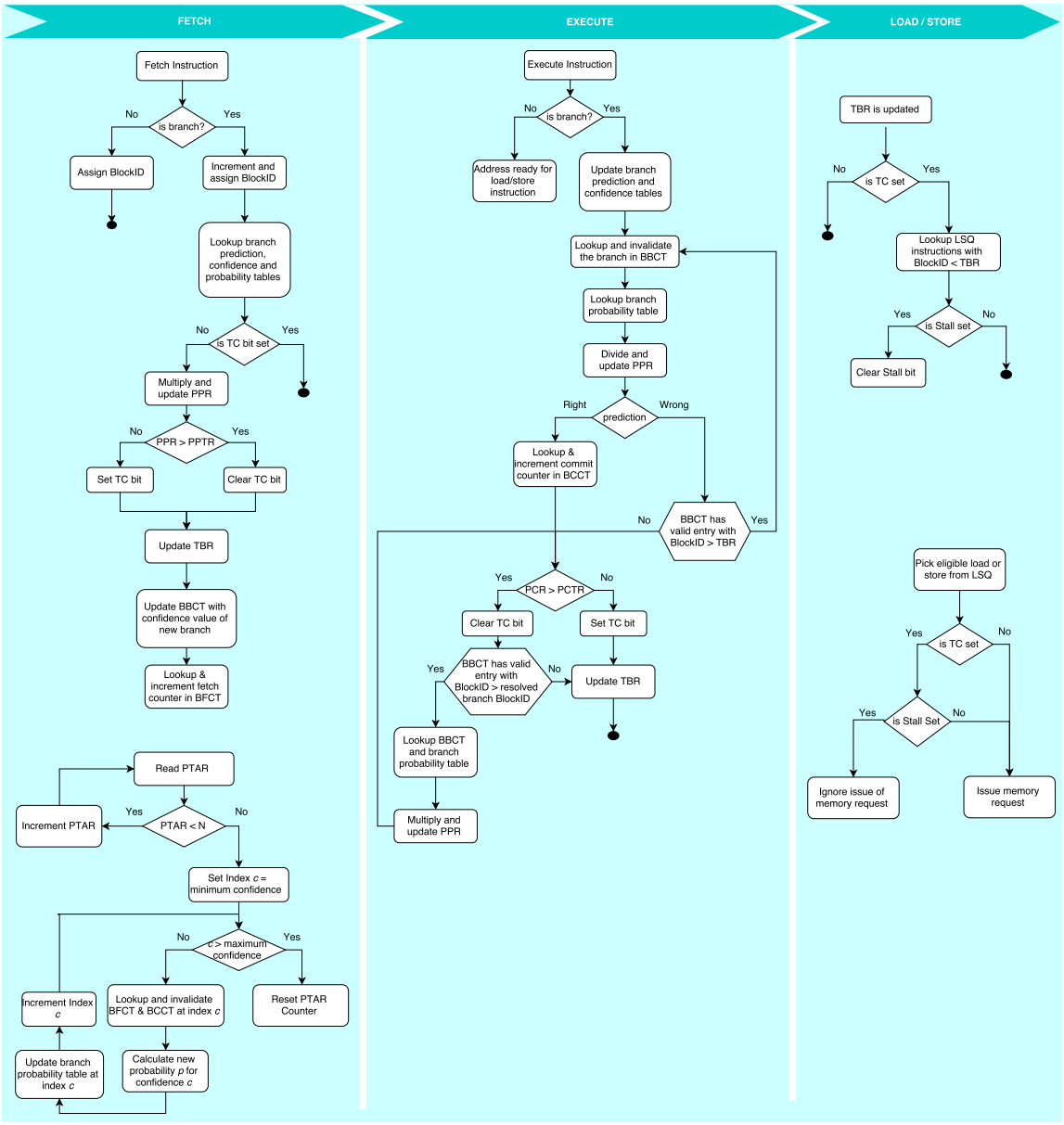


Figure 5.2: Flow chart of operation of PMAC during Fetch, Execute and Load/Store phases.

dence values respectively. Using the BlockID of the fetched branch, the per BlockID branch confidence table (BBCT) is updated with its confidence value and marked as valid. The correct prediction rate of the fetched branch is obtained by looking up

the per confidence branch probability table (CBPT) with its confidence value. When the throttle control (TC) bit is not set, the value in path probability register (PPR) is multiplied with the correct prediction rate of the fetched branch and the result is stored in PPR. The TC bit is set if the value in PPR exceeds the path probability threshold in PPTR, otherwise the TC bit is cleared. When the TC bit is not set, the throttling block register (TBR) is updated with the BlockID of the fetched branch, otherwise TBR is not updated. The per confidence branch commit table (BCCT) is looked up with the fetched branch confidence value and the commit counter is incremented.

We have a probability table age register (PTAR) which keeps track of number of cycles since the last update of per confidence branch probability table (CBPT). PTAR in incremented every cycle until N which is the time interval for updating probability values in CBPT. When the value in PTAR counter equals N, the fetch counter and commit counter for every confidence bucket is looked up in the branch fetch count (BFCT) and branch commit count table respectively. Then the probability value for each confidence bucket is calculated as $\frac{Number\ of\ Branch\ Commits}{Number\ of\ Branch\ Fetches}$. Respective CBPT entries are updated with the new probability values and the PTAR counter is reset to 0.

5.8.2 Execute phase

Whenever a branch is resolved with correct prediction, the tables in branch predictors and composite confidence estimator are updated. The direct mapped entry of the resolved branch in the per BlockID branch confidence table (BBCT) is looked up for its confidence value and it is invalidated. The correct prediction rate of the resolved branch is obtained by looking up the per confidence branch probability table (CBPT) with its confidence value. When the TC bit is set and the BlockID

of the resolved branch is less than or equal to the BlockID in TBR, the value in PPR is divided by the correct prediction rate and the result is stored in PPR. The per confidence branch commit table (BCCT) is looked up with the resolved branch confidence value and the commit counter is incremented.

If the throttle control (TC) bit is not set and the BlockID in TBR is not same as the BlockID of the recently fetched branch, then the path probability of program blocks fetched since the last TBR update is computed iteratively. The computation is done until the program block with path probability less than threshold in PPTR is identified. The branches with BlockID greater than the BlockID in TBR are looked up in BBCT one after the other until the TC bit is set or until the most recently fetched branch is encountered. Following every look up in BBCT, the correct prediction rate of the corresponding branch is obtained from the per confidence branch probability table (CBPT). Then the value in PPR is multiplied with the correct prediction rate of the branch and the result is stored in PPR. The TC bit is set when the value in PPR exceeds the path probability threshold in PPTR, otherwise the TC bit is cleared. When the TC bit is not set, the throttling block register (TBR) is updated with the BlockID of the in-flight branch under consideration.

Whenever a branch is resolved with misprediction, the tables in branch predictors and composite confidence estimator are updated. The BBCT entries of in-flight branches in the mispredicted path are looked up and invalidated. When the BlockID of mispredicted branch is less than or equal to the BlockID in TBR, the value in PPR is divided by the correct prediction rates of all branches with BlockID less than or equal to the BlockID in TBR iteratively. The result is stored in PPR and TC bit is cleared when the value in PPR exceeds the threshold in PPTR.

5.8.3 Load-Store phase

The BlockID in TBR and the throttle (TC) bit is used by the Load Store Unit to stall or allow memory access instructions to issue memory requests. Whenever the TBR and the TC bit is updated, the stall bits of loads and stores with BlockID greater than or equal to the BlockID in TBR are set while the the stall bits of loads and stores with BlockID less than the BlockID in TBR are cleared. When the TC bit is set the Load Store Unit ignores picking up and issuing memory requests for eligible loads and stores with stall bit set.

6. EVALUATION

In this section we outline our experimental methodology followed by a detailed exploration and analysis of the effects of our proposed technique.

6.1 Methodology

Gem5[4], a full system cycle accurate simulator is used to evaluate our technique. Table 6.1 shows the baseline configuration of the processor core and the memory hierarchy used in our evaluation. We use a subset of SPEC CPU2006 benchmarks to run simulations in a single-core processor system. McPAT1.3 [7] model is used to estimate energy consumed by the processor, caches and the DRAM memory controller for 65nm technology node.

Table 6.1: Baseline system configuration

CPU	2GHZ, 8-wide out-of-order processor 256-entry Instruction window
Branch Predictor	8KB tournament predictor 2.8% conditional branch mispredict rate
L1I & L1D Cache	64KB 8-way 3 cycle latency
L2 Cache	Unified 256KB 8-way 10 cycle latency
Shared L3 Cache	2MB/Core 16-way 20 cycle latency
Off-Chip DRAM	2GB 8-banks 200 cycle latency

We employ a 8K McFarling [14] hybrid branch predictor. The choice predictor table has 8K entries of 2-bit saturating counters. The prediction history table in

GAg has 8K entries of 2-bit saturating counters indexed by a 13-bit global history register. The first level per address branch history table in PAg has 2K entries of 11-bit local history registers indexed by branch PC address. The second level prediction history table in PAg has 2K entries of 3-bit saturating counters indexed by entries of first level table. A 4K global miss distance counter table of 3-bit saturating counters is used for JRS confidence estimation. It is indexed by the lower 12-bits of global history register in GAg. A 1K local Up/Down Counter table consisting of 5-bit saturating counters is used for Up/Down confidence estimation. It is indexed by lower 10-bits of local history registers in the first level PAg. The composite confidence estimator combines enhanced JRS, Up/Down and self estimators. The prediction rate estimator classifies branches into 45 confidence buckets. It has a 45 entry branch Fetch-Count and Commit-Count tables containing 16-bit saturating counters. The per confidence branch probability table is also a 45 entry table containing branch prediction rates in last 500000 cycles. In-flight branch tracking table is a 256 entry per block confidence table. The table entries hold 6-bit confidence values of in-flight branches. They are accessed using using lower 7 bits of the BlockID of a branch.

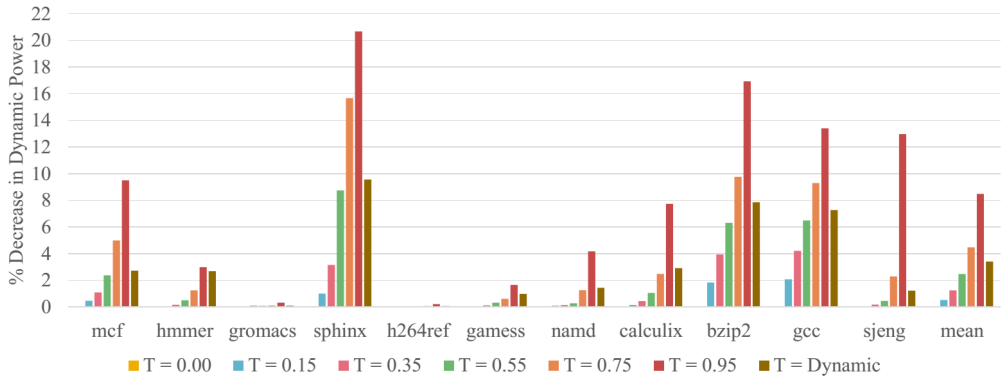
All simulations are fast-forwarded by 10 billion instructions, warmed-up for additional 1 billion instructions and then run in the out-of-order detailed mode for next 1 billion instructions. We ran simulations with path probability thresholds pre-set to a constant value and dynamically varying path probability threshold. The simulations with pre-set threshold are run for several values that are multiples of 0.05 between 0.05 and 0.95. In simulations with dynamic threshold, we vary the threshold by choosing following empirical values as constants,

$$K_1 = 128, K_2 = 32, C_1 = C_2 = 8 \tag{6.1}$$

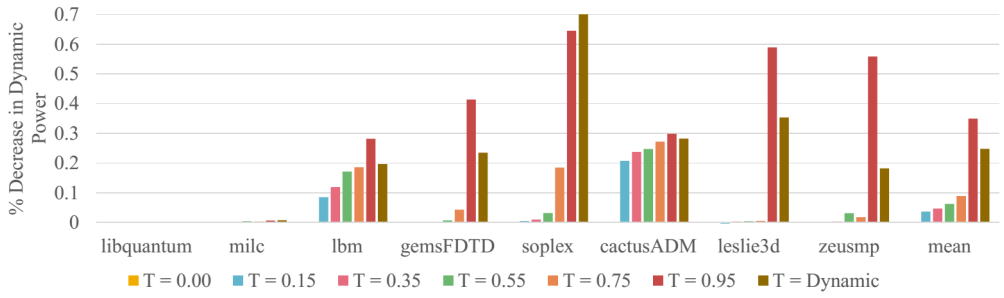
6.2 Results And Analysis

In this section, we present our results and analysis on the performance, average dynamic power consumption and the performance per watt of the processor. We also show the effects of memory access throttling on cache pollution, data prefetching and execution of instructions. The processor core with path probability threshold as zero is used as baseline for comparisons. The benchmarks showed in the graphs are sorted from left to right in the increasing order of misprediction rates.

6.2.1 Effect on dynamic power consumption of the processor



(a) Benchmarks with misprediction rates $\geq 1\%$



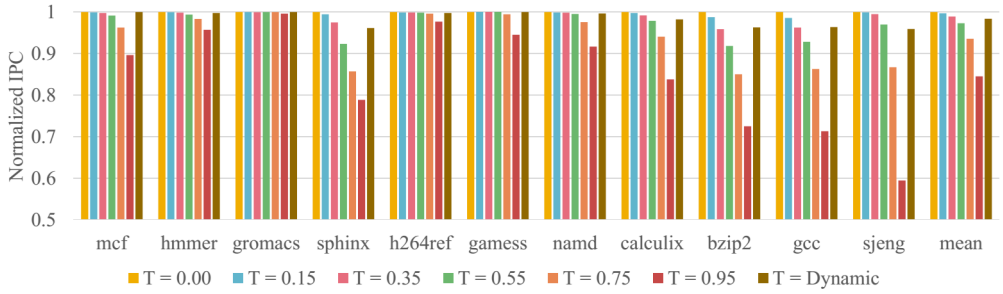
(b) Benchmarks with misprediction rates $< 1\%$

Figure 6.1: Decrease in dynamic power consumption of processor.

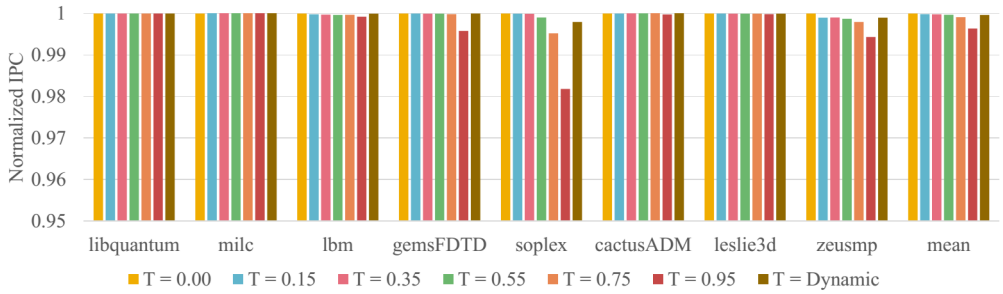
Figure 6.1 shows the decrease in dynamic power consumption of the processor which includes out-of-order core, caches and memory controller. In benchmarks with misprediction rates $\geq 1\%$, we observe that the dynamic power consumption of the processor decreases significantly at higher pre-set path probability thresholds. It is primarily due to the decrease in execution of wrong path memory access instructions and their data dependant instructions. When the threshold is sufficiently high, the wrong path memory access instructions are stalled until the mispredictions are known. They are flushed from the pipeline before the memory requests are issued and respective data dependant instructions are executed. At lower thresholds, the stalled wrong path memory requests are released even before the mispredictions are known when their path probability exceeds the threshold. This corresponds to missed out opportunities on reducing the wrong path memory accesses. With a dynamically varying threshold, in benchmarks with misprediction rates $\geq 1\%$, we observe that the dynamic power dissipation of the processor decreases by 3.4% (upto 9.55% in sphinx) on an average. The power savings are negligible in benchmarks with misprediction rates $< 1\%$.

6.2.2 Effect on performance of the processor

Figure 6.2 shows the decrease in performance (IPC) of the processor. In benchmarks with misprediction rates $\geq 1\%$, we observe a significant drop in IPC at higher pre-set path probability thresholds. The low confidence memory access instructions are stalled until they become those of high confidence. A high threshold value in a deeply speculating core will mark several correct path memory access instructions as low confidence. The issue of memory requests are delayed, consequently the execution of respective data dependent instructions are delayed and increases the overall execution time of instructions. The path probability threshold acts as a form of



(a) Benchmarks with misprediction rates $\geq 1\%$



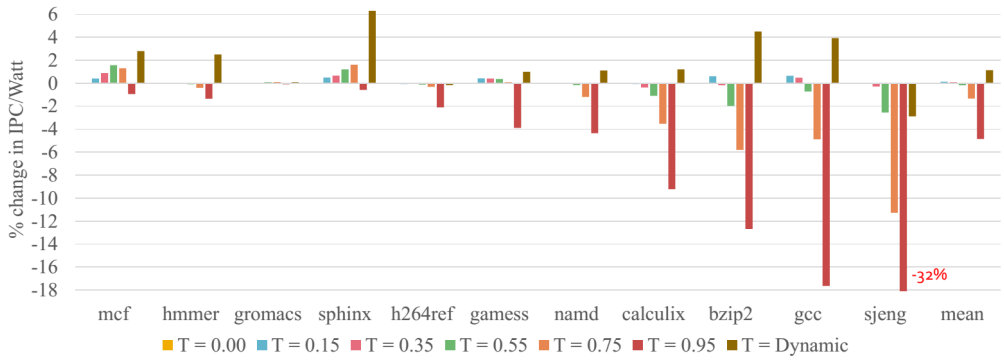
(b) Benchmarks with misprediction rates $< 1\%$

Figure 6.2: Decrease in IPC.

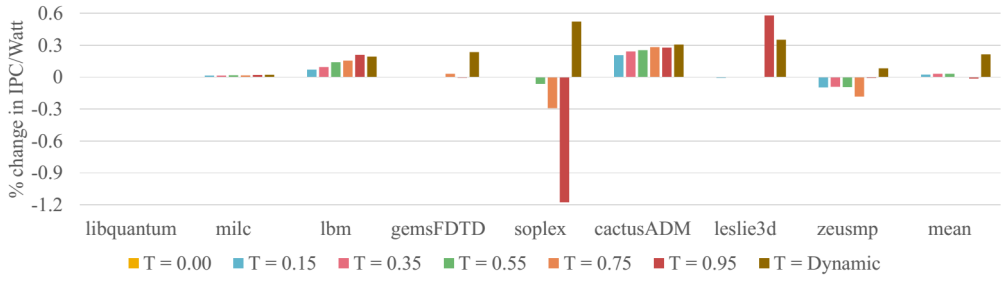
speculation control. It limits the amount of speculation done by the processor and inhibits the progress of correct path instructions. Hence, a higher threshold value restricts the speculative execution and decreases the performance. A lower threshold value allows more speculative execution in the processor but it misses out on opportunities to reduce wrong path execution. Having a pre-set threshold as a constant throughout the program execution is not beneficial. When the average execution time and number of stalled memory access instructions are low, increasing the threshold has less negative impact on the performance. When the average execution time of instruction and number of stalled is high, decreasing the threshold mitigates the performance loss. We vary the threshold accordingly, to reduce wrong path memory

accesses with minimum impact on performance (IPC). With this kind of dynamic threshold scheme, we observe that the performance (IPC) on an average decreases by 0.99% (upto 4.26% in sjeng) in benchmarks with misprediction rates $\geq 1\%$. The performance (IPC) loss is negligible in benchmarks with low misprediction rates ($< 1\%$). The dynamic threshold technique continuously adjusts the threshold according to executing conditions and minimizes the negative impact on performance.

6.2.3 Effect on performance per watt of the processor



(a) Benchmarks with misprediction rates $\geq 1\%$



(b) Benchmarks with misprediction rates $< 1\%$

Figure 6.3: Change in IPC/Watt.

Figure 6.3 shows the variation in performance per watt of the processor. In

benchmarks with misprediction rates $\geq 1\%$, we observe that the performance per watt decreases significantly at higher pre-set path probability thresholds. The reduction in wrong path memory accesses comes at the cost of performance (IPC) of the processor. At higher thresholds, the IPC loss due to decrease in speculative execution outweighs the benefit of power savings due to reduction in wrong path memory accesses. In the dynamic threshold scheme, the threshold is varied to reduce wrong path memory accesses with minimum negative impact on performance (IPC). The achieved dynamic power savings outweighs the performance (IPC) loss in all benchmarks except sjeng. We observe an IPC/Watt improvement of 1.83% (upto 6.3% in sphinx). The variation in IPC/Watt is negligible in benchmarks with misprediction rates $< 1\%$.

6.2.4 Effect on prefetching and pollution

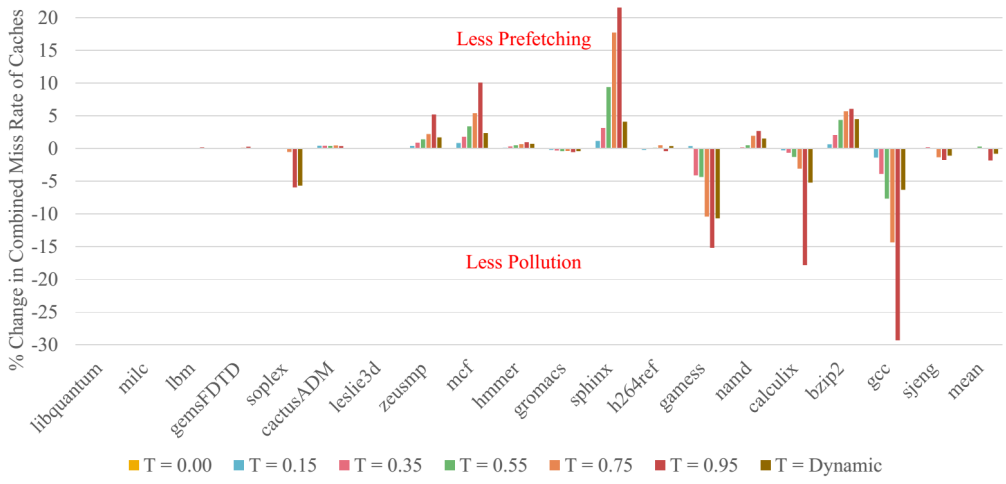


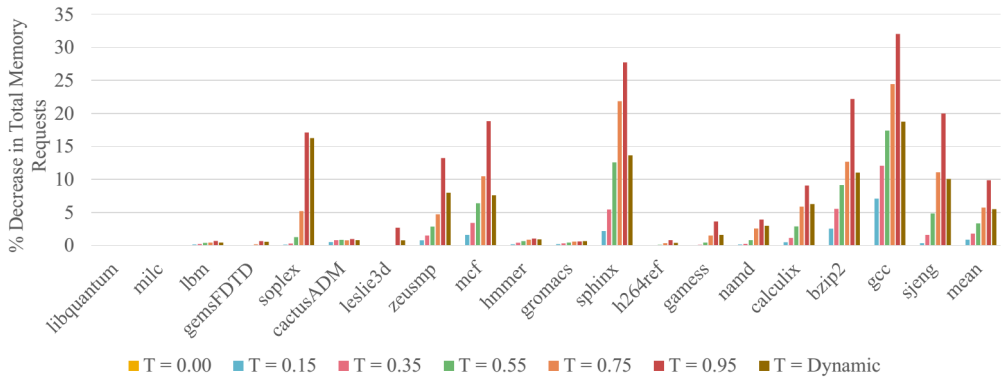
Figure 6.4: Effect on prefetching and cache pollution.

Figure 6.4 shows the variation in combined miss rate of L1 cache, L2 cache and

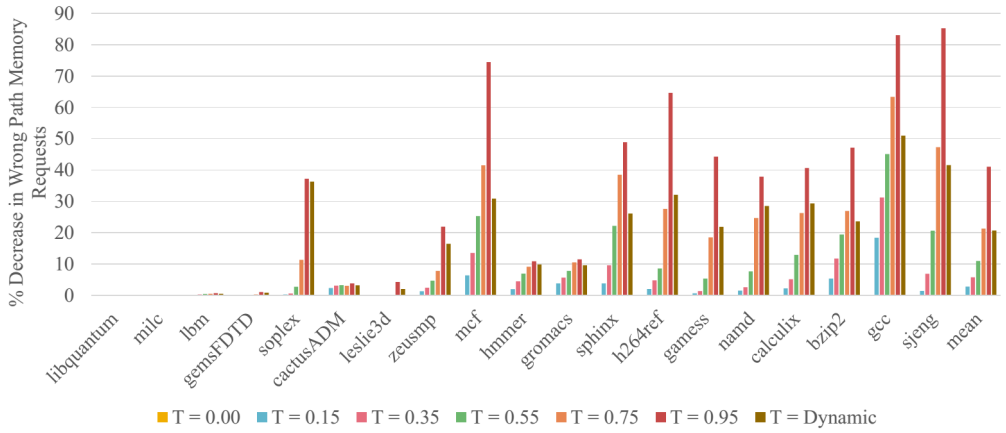
L3 cache of the processor. We use the combined miss rate of all caches which is product of L1, L2 and L3 cache miss rate to analyse the effect on prefetching versus pollution caused by wrong path memory references. In the bzip2, mcf, namd, sphinx and zeusmp benchmarks the combined miss rate increases at higher path probability thresholds. The increase in miss rate compared to baseline suggests that the decline in wrong path memory accesses has resulted in decrease of data prefetching for correct path instructions. We can observe from Figure 6.4 that, in sphinx benchmark the prefetching by wrong path memory access instructions is reduced by 21% when the path probability threshold is pre-set to 0.95. In the calculix, gamess, gcc, sjeng and soplex benchmarks the combined miss rate decreases at higher thresholds. The decrease in miss rate compared to baseline suggests that the decline in wrong path memory accesses has resulted in decrease of pollution in caches. From Figure 6.4 we can observe that, in gcc benchmark the cache pollution decreases by 29% when the path probability threshold is pre-set to 0.95. The wrong path memory references provide benefit of prefetching for correct path instructions in few benchmarks while it causes cache pollution in others. The dynamic threshold scheme decreases the cache pollution by 0.8% (upto 10.66% in gamess) on an average across all benchmarks.

6.2.5 Effect on the number of issued memory requests

Figure 6.5a shows the decrease in total number of issued memory requests due to a decrease in number of wrong path memory requests as shown in Figure 6.5b. We observe that the number of wrong path memory references decreases significantly at higher thresholds in benchmarks with higher misprediction rates. In gcc and sjeng benchmarks more than 80% of wrong path memory references are eliminated when the path probability threshold is pre-set to 0.95. We also observe significant decline in number of wrong path memory references in other benchmarks like bzip2,



(a) Decrease in number of issued memory requests



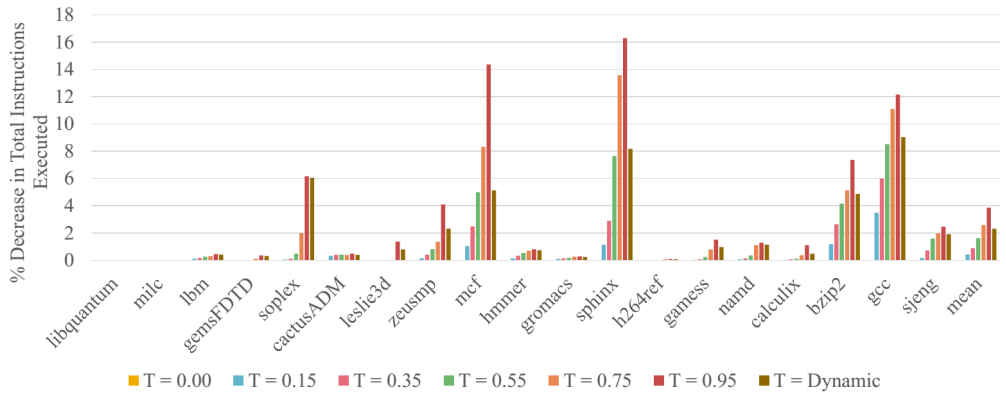
(b) Decrease in number of issued wrong path memory requests

Figure 6.5: Number of memory requests issued.

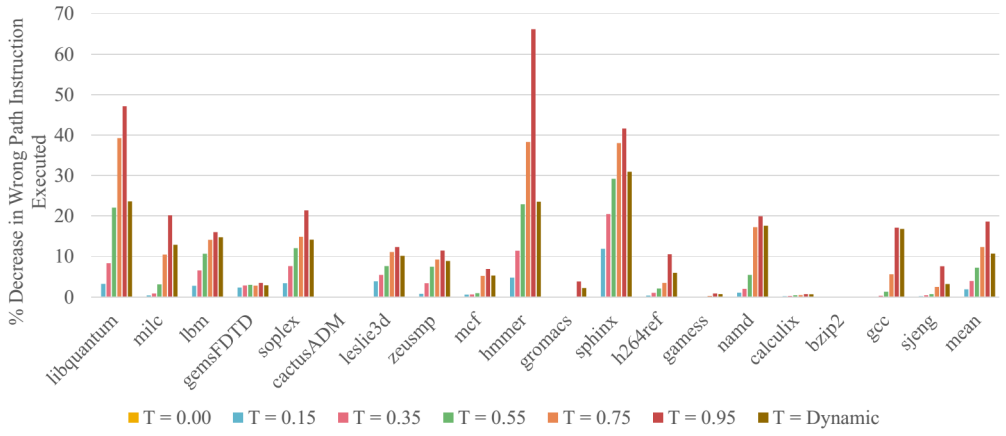
calculix, gamess, h264ref, mcf, namd, soplex and sphinx at higher thresholds. In dynamic threshold scheme, the threshold is varied to minimize the negative impact in performance. When we dynamically vary the threshold, we decrease the path probability threshold when the average execution time of instructions and number of stalled instructions is high. When the threshold is low, the wrong path memory requests are not stalled until the mispredictions are known. This corresponds to a lost opportunity in reducing the wrong path memory accesses. Due to such missed

out opportunities in order to minimize the IPC loss, we observe an average decrease of only 20.75% in wrong path references across all benchmarks (upto 51.02% in gcc). The decrease in wrong path memory accesses decreases the total number of issued memory requests on an average by 5.51% across all benchmarks (upto 13.76% in sphinx).

6.2.6 Effect on the number of dynamic instructions executed



(a) Decrease in number of executed instructions



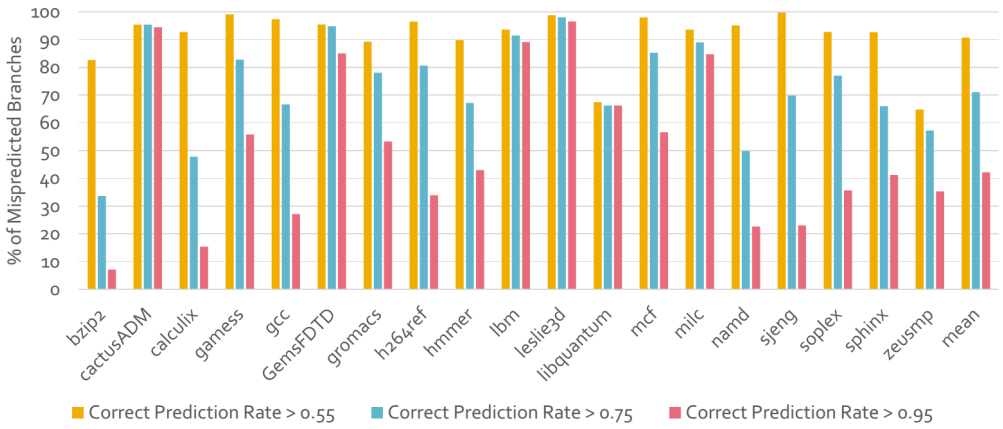
(b) Decrease in number of wrong path instructions

Figure 6.6: Number of instructions executed.

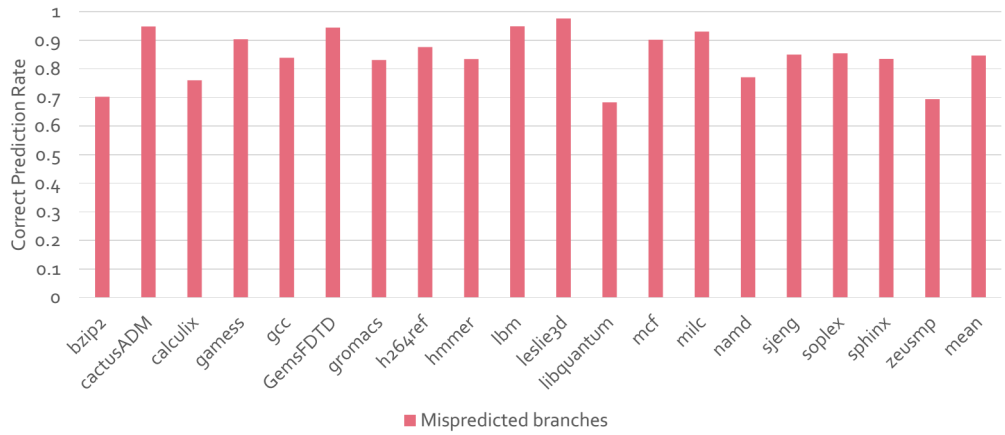
Figure 6.6a shows the decrease in total number of dynamic instructions executed due to a decrease in number of wrong path instructions executed as shown in Figure 6.6b. When wrong path memory access instructions are stalled, respective wrong path memory dependant instructions are also stalled. If the wrong path instructions are stalled until the branch mispredictions are known, they are flushed from pipeline without executing. At higher thresholds, the number of wrong path memory access instructions and data dependant instructions executed decreases which results in decrease of total number of executed dynamic instructions. In dynamic threshold scheme, the threshold is varied in order to minimize the negative impact in performance while reducing the wrong path execution. When the average execution time of instruction increases, the speculative execution which can result in wrong path execution is allowed to go ahead by decreasing the threshold. Hence, we observe an average decrease of only 10.73% in number of executed wrong path instructions across all benchmarks (upto 31% in gcc). The decrease in wrong path execution decreases the total number of dynamic instructions executed by 2.31% on an average across all benchmarks (upto 9.03% in gcc).

6.3 Effect Of Correct Prediction Rate Of Mispredicted Branches

Figure 6.7a shows the percentage of mispredicted branches with correct prediction rates greater than 0.55, 0.75 and 0.95. We can observe that, more than 40% of the mispredicted branches have correct prediction rate greater than 0.95. The probability that 40% of the mispredicted branches and respective control dependant instructions will be executed and committed by the processor is greater than 0.95. More than 70% of the mispredicted branches have correct prediction rate greater than 0.75. The cactusADM, lbm, leslie3d, milc have more than 80% and libquantum has 65% of mispredicted branches with likelihood of correct prediction greater than



(a) % of mispredicted branches with higher correct prediction rates



(b) Average correct prediction rates of mispredicted branches

Figure 6.7: Mispredicted branches with different correct prediction rates.

0.95. The processor has high accuracy of prediction (> 99%) in these benchmarks. The throttling mechanism does not reduce wrong path memory requests because the likelihood that the processor is on the correct path is very high while executing these benchmarks. Hence, they show no variation in IPC or power at high thresholds upto 0.95. More than 70% of the mispredicted branches in bzip2, gcc, sjeng, namd and calculix benchmarks have probability of correct prediction between 0.55

and 0.95. The processor has relatively lower accuracy of branch predictions ($< 98\%$) in these benchmarks. The throttling mechanism reduces significant number of wrong path memory requests for thresholds greater than 0.55 in these benchmarks. Rest of the benchmarks have moderate number of mispredicted branches (30%-60%) with probability of correct prediction between 0.55 and 0.95. They show modest variations in IPC or power at high thresholds upto 0.95 based on the accuracy of branch predictions.

The average correct prediction rate of mispredicted branches is 0.85 as shown in Figure 6.7b. The probability based memory access controller should have higher path confidence threshold in order to eliminate execution of significant number of wrong path instructions. But higher threshold inhibits speculative execution of processor and decreases the performance. Higher correct prediction rate of mispredicted branches makes the probability based memory access controller harder to reduce wrong path memory accesses and achieve power savings without impacting the performance negatively.

6.4 Design Overhead Estimate

The additional structures used in the design of probability based memory access controller require 2.8KB of storage. 2.1KB of storage is required by the 4K composite confidence estimator. The tables used in probability based memory access controller require 570B of storage. Tables 6.2 and 6.3 shows the storage required for tables and structures used in the composite confidence estimator and PMAC respectively. The Load-Store Queue requires 288B of storage for storing BlockID and a stall bit for each entry. The power consumption is $< 0.4\%$ of the total processor power.

The prediction rate estimator in our probability based memory request controller classifies branches across n confidence buckets for a given composite confidence es-

timator configuration. It employs n entry branch commit-count, squash-count and probability tables with 16-bit entries. The 256 entry per block confidence table has $\lceil log_2 n \rceil$ bit entries to hold confidence value of unresolved program blocks. Hence, the tables require additional storage of $(3 * n * 16 + 256 * \lceil log_2 n \rceil)$ bits.

Table 6.2: Composite confidence estimator storage overhead

Composite Confidence Estimator		
Estimator Tables	Global JRS	Local UDC
No. of Counters	4K	1K
Counter Size	3-bit	5-bit
Table Size	1.5KB	640B

Table 6.3: PMAC storage overhead

Probability-Based Memory Access Controller				
Tables	BCCT	BFCT	CBPT	BBCT
No. of Entries	45	45	45	256
Entry Size	16-bit	16-bit	32-bit	6-bit
Table Size	90B	90B	90B	192B
BlockID Counter	PPR	PPTR	TBR	PTAR
8-bit	64-bit	32-bit	9-bit	24-bit

7. CONCLUSION

In this thesis we proposed a probability-based memory access controller (PMAC) to reduce the wrong path data requests sent to memory. Branches are classified into different confidence buckets and correct prediction rates for each confidence bucket are computed. We used a composite confidence estimator proposed by Jimenez [9] to compute branch confidence values. The path probability of an instruction is computed as a product of correct prediction rates of all in-flight branches fetched before the instruction. Using a deterministic or dynamically varying probability value as a threshold the instructions are classified as low confidence and high confidence. PMAC stalls low confidence speculative memory access instructions and prevents data requests from being sent to the memory hierarchy. In the event of a misprediction, the stalled instructions are flushed. We achieve energy reduction and power savings by reducing wrong path execution. However, the threshold acts as a form of speculation control. It limits the speculative execution of the processor and negatively impacts its performance. A higher threshold value inhibits the progress of correct path instructions while a lower threshold value misses out on opportunities to reduce wrong path execution. A deterministic threshold is not beneficial. At higher thresholds, the IPC loss due to speculation control outweighs the benefit of power savings due to reduction in wrong path execution. We dynamically vary the threshold to minimize the negative impact on performance while reducing the wrong path memory accesses. It is desirable to have high threshold when the average execution time of instructions and the number of stalled memory access instructions is low and vice versa. The threshold is varied based on the number of free entries in instruction window, the current threshold value and the number of stalled memory

access instructions. Table 7.1 shows the summary of results for processor employing dynamic threshold. With the dynamic threshold scheme, we improved IPC/Watt by reducing wrong path execution and achieve dynamic power savings with minimum negative impact on performance.

The average correct prediction rate of mispredicted branches is 0.85 while 40% of them have correct prediction rate greater than 0.95. It means that the threshold should be as high as 0.85 or greater to identify and reduce at least 50% of the wrong path memory accesses. However, a high threshold decreases the performance of the processor significantly. Due to high correct prediction rates of mispredicted branches, the wrong path memory accesses cannot be eliminated without impacting the performance significantly. The probability-based memory access controller is more beneficial when the mispredicting branches have low correct prediction rates.

Table 7.1: Summary of results for processor with dynamic threshold

Effects of Dynamic Threshold	Average	Maximum
Dynamic Power Consumption	↓ by 2.08%	↓ by 9.55%
Performance (IPC)	↓ by 0.96%	↓ by 4.26%
IPC/Watt	↑ by 1.15%	↑ by 6.3%
Total Instructions Executed	↓ by 2.31%	↓ by 9.03%
Wrong Path Instructions Executed	↓ by 10.73%	↓ by 30.97%
Total Memory Requests	↓ by 5.51%	↓ by 18.76%
Wrong Path Memory Requests	↓ by 20.75%	↓ by 51.02%
Cache Pollution	↓ by 0.80%	↓ by 10.66%

REFERENCES

[1] Juan L Aragón, José González, and Antonio González. Power-aware control speculation through selective throttling. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 103–112. IEEE, 2003.

[2] R Bahar and Gianluca Albera. Performance analysis of wrong-path data cache accesses. In *Workshop on Performance Analysis and its Impact on Design*. Citeseer, 1998.

[3] Amirali Baniasadi and Andreas Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 16–21. ACM, 2001.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[5] Jonathan Combs, Candice Bechem Combs, and John Paul Shen. Mispredicted path cache effects. In *Euro-Par99 Parallel Processing*, pages 1322–1331. Springer, 1999.

[6] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence estimation for speculation control. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 122–131. IEEE Computer Society, 1998.

[7] HP. Mcpat1.3, http://www.hpl.hp.com/research/mcpat/.

[8] Erik Jacobsen, Eric Rotenberg, and James E Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 142–152. IEEE Computer Society, 1996.

[9] Daniel Jiménez et al. Composite confidence estimators for enhanced speculation control. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 161–168. IEEE, 2009.

[10] Artur Klauser, Srilatha Manne, and Dirk Grunwald. Selective branch inversion: Confidence estimation for branch predictors. *International Journal of Parallel Programming*, 29(1):81–110, 2001.

[11] Chang Joo Lee, Hyesoon Kim, Onur Mutlu, and Yale N Patt. Performance-aware speculation control using wrong path usefulness prediction. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 39–49. IEEE, 2008.

[12] Kshitiz Malik, Mayank Agarwal, Vikram Dhar, Matthew Frank, et al. Paco: Probability-based path confidence prediction. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 50–61. IEEE, 2008.

[13] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 132–141. IEEE Computer Society, 1998.

[14] Scott McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[15] Onur Mutlu, Hyesoon Kim, David N Armstrong, and Yale N Patt. Cache filtering techniques to reduce the negative impact of useless speculative memory references on processor performance. In *Computer Architecture and High Performance Computing, 2004. SBAC-PAD 2004. 16th Symposium on*, pages 2–9. IEEE, 2004.

[16] Onur Mutlu, Hyesoon Kim, David N Armstrong, and Yale N Patt. Understanding the effects of wrong-path memory references on processor performance. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, pages 56–64. ACM, 2004.

[17] Onur Mutlu, Hyesoon Kim, David N Armstrong, and Yale N Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *Computers, IEEE Transactions on*, 54(12):1556–1571, 2005.

[18] Jim Pierce and Trevor Mudge. The effect of speculative execution on cache performance. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 172–179. IEEE, 1994.

[19] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. In *Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on*, pages 165–175. IEEE, 1996.

[20] Resit Sendag, David J Lilja, and Steven R Kunkel. Exploiting the prefetching effect provided by executing mispredicted load instructions. In *Euro-Par 2002 Parallel Processing*, pages 468–480. Springer, 2002.

[21] Resit Sendag, Ayse Yilmazer, J Yi Joshua, and Augustus K Uht. The impact of wrong-path memory references in cache-coherent multiprocessor systems. *Jour-*

nal of Parallel and Distributed Computing, 67(12):1256–1269, 2007.