Pipeline Gating:
Speculation Control For Energy Reduction

Srilatha Manne
University of Colorado
Dept. of Electrical and Computer Engineering
Boulder, CO 80309
srilatha.manne@colorado.edu

Artur Klauser, Dirk Grunwald
University of Colorado
Department of Computer Science
Boulder, CO 80309
grunwald,klauser@cs.colorado.edu

Abstract

Branch prediction has enabled microprocessors to increase instruction level parallelism (ILP) by allowing programs to speculatively execute beyond control boundaries. Although speculative execution is essential for increasing the instructions per cycle (IPC), it does come at a cost. A large amount of unnecessary work results from wrong-path instructions entering the pipeline due to branch misprediction. Results generated with the SimpleScalar tool set using a 4-way issue pipeline and various branch predictors show an instruction overhead of 16% to 105% for every instruction committed. The instruction overhead will increase in the future as processors use more aggressive speculation and wider issue widths [9].

In this paper, we present an innovative method for power reduction which, unlike previous work that sacrificed flexibility or performance, reduces power in high-performance microprocessors without impacting performance. In particular, we introduce a hardware mechanism called pipeline gating *to control rampant speculation in the pipeline. We present inexpensive mechanisms for determining when a branch is likely to mispredict, and for stopping wrong-path instructions from entering the pipeline. Results show up to a 38% reduction in wrong-path instructions with a negligible performance loss ($\approx$ 1%). Best of all, even in programs with a high branch prediction accuracy, performance does not noticeably degrade. Our analysis indicates that there is little risk in implementing this method in existing processors since it does not impact performance and can benefit energy reduction.*

1 Introduction

There has been considerable work on *low power* processors. Most of this work focuses on reducing power in applications where battery life is paramount. The focus of our research is to reduce the energy demands of high performance microprocessors without compromising performance. Such reductions will greatly reduce packaging costs and will allow the computer architect to better balance an overall "power budget" across different parts of the chip.

Existing low power work has focused on reducing energy in the memory subsystem [3, 8, 4]. In embedded processors, such as the StrongArm [11], the memory subsystem is the dominant source of area and power because the rest of the processor has been simplified to reduce power. State-of-the-art microprocessors have a high degree of control complexity and a large amount of area dedicated to structures that are essential for high-performance, speculative, out-of-order execution, such as branch prediction units, branch target buffers, TLBs, instruction decoders, integer and floating point queues, register renaming tables, and load-store queues. For example, $\approx 30\%$ of the core die area on the DECchip 21264 is devoted to cache structures, while the StrongARM processor uses $\approx 60\%$ of the core die area for memory. Figure 1 shows a distribution of the power dissipated in a PentiumPro processor [6] during a test designed to consume the most power, which is when the processor is committing each instruction that it fetches. The fetch and decode stages, along with components necessary to perform dynamic scheduling and out-of-order execution, account for a significant portion of the power budget. Therefore, pipeline activity is a dominant portion of the overall power dissipation for complex microprocessors.

Performance is the primary goal of state-of-the-art microprocessor design. Architectural improvements for performance have centered on increasing the amount of instruction level parallelism

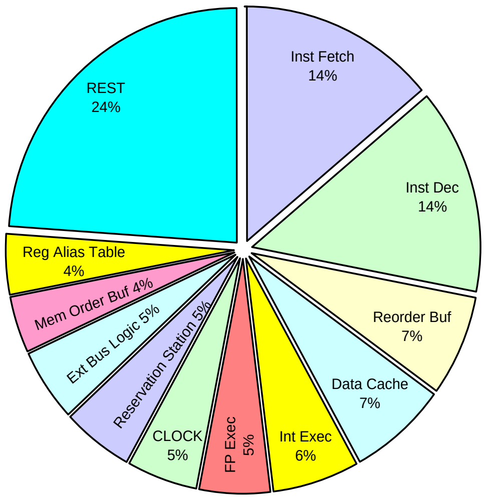

Figure 1: Power consumption for PentiumPro chip, broken down by individual processor components.

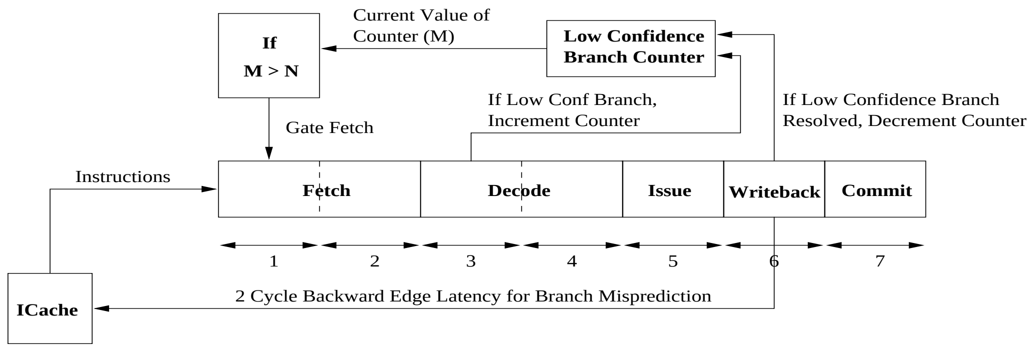
Current Value of Counter (M)
Low Confidence Branch Counter
If M > N
If Low Conf Branch, Increment Counter
If Low Confidence Branch Resolved, Decrement Counter
Gate Fetch
Instructions
Fetch
Decode
Issue
Writeback
Commit
1
2
3
4
5
6
7
ICache
2 Cycle Backward Edge Latency for Branch Misprediction

Figure 2: Pipeline with a two fetch and decode cycles, showing additional hardware required for pipeline gating. The low-confidence branch counter records the number of unresolved branches that reported as low-confidence. The counter value is compared against a threshold value ("N"). The processor ceases instruction fetch if there are more than N unresolved low-confident branches in the pipeline..

through aggressive speculation and out-of-order execution. Although these advances have increased the number of instructions per cycle (IPC), they have come at the cost of wasted work. Most processors use branch prediction for speculative control flow execution, and recent work has examined value and memory speculation [14]. Branch prediction is used to execute beyond the control boundaries in the code. With high branch prediction accuracy, most issued instructions will actually commit. However, many programs have a high branch misprediction rate, and these programs issue many instructions that never commit. Each of those instructions uses many processor resources. If we can decrease the percentage of uncommitted instructions actually issued, we can decrease the power demands of the processor as a whole.

Goals and Contributions It is the goal of this paper to control speculation and reduce the amount of unnecessary work in high-performance, wide-issue, super-scalar processors. We accomplish this by using a particular form of speculation control, called *pipeline gating*, to limit speculation and reduce energy consumption. In many processor implementations, functional units and clocks are gated to restrict spurious signals from producing unnecessary activity in circuits. Similarly, the pipeline can also be gated to restrict spurious or wrong-path instructions from entering the pipeline. Although a thorough power analysis is beyond the scope of this paper, the reduction in fetch and decode activity resulting from pipeline gating can clearly be exploited to reduce the power needs of a complex microprocessor. This paper makes the following contributions:

- We present *pipeline gating*, a method to reduce the number of speculatively issued instructions, and demonstrate the benefits of that method using a detailed pipeline-level simulation of a wide-issue, out-of-order, super-scalar microprocessor. By reducing the number of instructions fetched, decoded, issued and executed, we reduce the average activity in the processor without reducing performance, and thus reduce the total energy.

- We compare the effectiveness and cost of this design using various *confidence estimation* mechanisms, and show how to increase the effectiveness of these confidence estimation mechanisms for pipeline gating.

- We present results which show a significant reduction in unnecessary work with a negligible performance loss.

The rest of the paper discusses work reduction and the pipeline gating method in more detail. Section 2 describes the gating method and the work reduction metric used throughout the paper. An overview of the pipeline model, confidence estimators and characterization of the estimators for pipeline gating are presented in Section 3. Section 4 presents results for pipeline gating and Section 5 concludes the papers.

2 Processor Pipeline Gating for Work Reduction

The energy consumed by a processor is a function of the amount of work the processor performs to accomplish a given task. In a non-speculative processor all work performed is necessary. In a speculative, multi-issue, dynamically scheduled processor, a large amount of extra work is performed without realizing any performance benefits. We define the *Extra Work* of a given pipeline stage to be $Ew = \left(\frac{SeenInsn - CommittedInsn}{CommittedInsn}\right)$. There is a different Ew value for each stage of the pipeline. For example, if only 100 out of 130 instructions fetched by the processor actually commit, the Ew of the fetch stage is 30%. If 120 of the 130 instructions actually execute, the Ew of the execution stage is 20%. The Ew parameter has a lower bound of zero when no extra work is performed, but has no upper bound.

The goal of pipeline gating is to reduce the amount of extra work performed to complete a task without affecting the overall performance of the system. Since performance drives the market for these processors, it is difficult to justify a performance loss without extraordinary savings in power. Secondly, overall energy consumption is dependent on performance. Since $Energy = Power \times Time$, simply reducing the power in a processor may not decrease the energy demands if the task now takes longer to execute. In [3], Fromm *et al* noted a correlation between energy and performance. Reducing performance does not always reduce the overall energy consumed by the processor because of the quiescent energy consumed in the system [1]. In this paper, we reduce work while retaining performance and thus reduce the overall energy consumption of the processor.

2.1 Pipeline Gating

We will use the schematic of the processor pipeline shown in Figure 2 to describe pipeline gating. Like many high-performance processors, such as the DEC AXP-21164 or Intel PentiumPro, our sample pipeline uses two fetch and decode cycles to allow the clock rate to be increased. We assume the fetch stage has a small instruction buffer to allow instruction fetch to run ahead of decode. Branch prediction occurs when instructions are fetched to reduce the misfetch penalty. The actual instruction type may not be known until the end of decode. Conditional branches are resolved in the execution stage, and branch prediction logic is updated in the commit stage. Since the processor uses out-of-order execution, instructions may sit in the issue window for many cycles, and there may be several unresolved branches in the processor.

We use a *confidence estimator* to assess the quality of each branch prediction. A "high confidence" estimate means we believe the branch predictor is likely to be correct. A "low confidence" estimate means we believe the branch predictor has incorrectly predicted the branch. We use these confidence estimates to decide when the processor is likely to be executing instructions that will not commit; once that decision has been reached, we "gate" the pipeline, stalling specific pipeline stages.

In our study, we vary a number of parameters, including the branch predictor, the confidence estimator, the stage at which a gating decision is made, the stage that is actually gated and the number of outstanding low-confident branches needed to engage gating. The decision to gate can occur in the fetch, decode or issue stages. Equally important is the decision about *what* to gate and *how long* to gate. Gating the fetch or decode stages would appear to make the most sense, and we examined both cases. We used the number of unresolved low-confident branches to determine when and how long to gate. For example, if the instruction window includes one low-confident branch, and another low-confident branch exits the fetch (or, alternatively, decode or issue) stage, gating would be engaged until one or the other low-confident branch resolves. Figure 2 illustrates this process for a specific configuration. We add a counter that is incremented whenever the decode encounters a low-confident branch and is decremented when a low-confident branch resolves. If the counter exceeds a threshold, the fetch stage is gated. Instructions in the fetch-buffer continue to be decoded and issued, but no new instructions are fetched.

We have found that gating the processor typically stalls the processor for a very short duration. Figure 3 shows the number of times a specific configuration of our pipeline model is stalled while executing different programs. Generally, gating stalls occur for about 2-4 processor cycles. Most processor configurations exhibit a similar distribution, and indicate that our mechanism is exhibiting fine control over the speculative state of the processor.

2.2 Confidence Estimation Metrics

A complete comparison of confidence estimation mechanisms [5] is beyond the scope of this paper, but we implement several confidence estimation methods and compare their performance for pipeline gating. There are two important metrics to characterize the performance of confidence estimators used by pipeline gating: *specificity* SPEC and the *predictive value of a negative test* (PVN). The specificity (SPEC) is the fraction of all mispredicted branches actually detected by the confidence estimator as being low confidence. The PVN is the probability of a low-confidence branch being incorrectly predicted. A larger SPEC means that more mispredicted branches are marked as "low confidence". A larger PVN means that a given low-confidence branch is more likely to be mispredicted. A confidence estimator could have a perfect specificity by marking *all* branches as low confidence, but the PVN would then be no more than the branch misprediction rate.

In practice, a confidence estimator must balance SPEC *vs.* PVN to provide a good quality confidence estimate for many branches. The confidence estimators we examined have an average SPEC between 17%-77%, and an average PVN between 19%-40%; typically, estimators with a higher SPEC have a lower PVN. If we simply used the PVN of a single branch to control pipeline gating, we would stall the pipeline too frequently, compromising performance. However, if there were N low-confident branches in the pipeline, the probability that *at least one* of those branches is mispredicted becomes $1 - (1 - \text{PVN})^N$. Thus, if the average PVN is 30% and we gate when there are two or more low-confident branches in the pipeline, the probability of at least one misprediction becomes 51%. Since any subsequently fetched instructions would be control dependent on both branches in the pipeline, this "boosting" improves our gating decision.

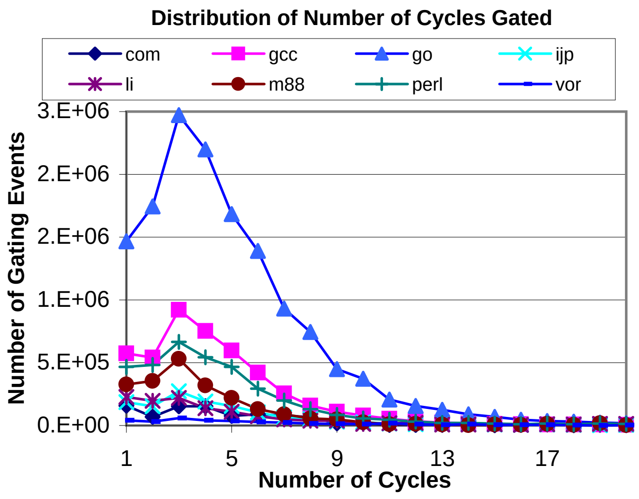

Figure 3: Distribution of gating events and the number of cycles gated per event.

3 Empirical Evaluation of Pipeline Gating

To properly understand the effects of stalling the pipeline, we used the SimpleScalar tools [2] to develop a pipeline model of an out-of-order, speculative, wide-issue processor. We modified the *sim-outorder* processor model to produce the machine configuration listed in Tables 1 and 2. Table 3 shows the latency of the different operation types. Although we used a 32kByte instruction cache, it is effectively equivalent to a 16kByte instruction cache because the SimpleScalar instruction set uses 8-byte instructions. The processor can fetch, issue, and commit four instructions each cycle.

We used both McFarling combining branch predictor and Gshare branch predictor to characterize the effect of branch predictor accuracy on pipeline gating. The McFarling combining predictor uses gshare and bimodal branch component predictors along with a meta predictor. The meta predictor chooses one of the branch predictors as the correct prediction for the branch. We chose the combination of gshare and bimodal because McFarling [10] indicated this combination had the best performance for the predictor sizes used in this paper. In both the Gshare and McFarling predictors, the branch prediction counters are updated at commit, and both predictors speculatively update the global history register, but not the prediction counters. The penalty for a branch misprediction is a minimum of seven cycles. Five of the cycles are incurred in the pipeline stage for the new instruction to travel to the point of execution, and the other 2 cycles are incurred for sending the misprediction signal to the rest of the pipeline and to calculate a new target address. The penalty will be larger than seven cycles if the new instruction is not available in the L1 instruction cache.

			McFarling			Gshare		
Name	Comm Inst	Inst/ Branch	Exec Cycles	Fetch Inst	MisPred Rate (%)	Exec Cycles	Fetch Inst	MisPred Rate (%)
compress	80.4	5.6	44.7	120.9	9.9	44.6	119.8	9.8
gcc	250.9	5.0	249.3	413.3	12.2	282.0	461.3	21.4
go	548.1	6.8	508.8	1043.2	23.9	544.1	1127.3	32.2
ijpeg	252.0	12.6	112.7	316.6	10.4	114.4	320.3	12.2
li	183.3	4.4	100.3	275.2	6.9	106.6	286.3	9.4
m88ksim	416.5	4.6	282.5	544.9	4.6	290.1	555.6	6.5
perl	227.5	5.2	276.8	361.3	11.3	305.0	403.4	21.3
vortex	180.9	6.2	119.7	192.8	1.7	127.7	211.1	5.0

Table 4: Baseline performance for McFarling and Gshare predictors. Instruction count and execution cycles are given in millions. Also shown in the number of instructions fetched (in millions) for each branch predictor.

Parameter	Configuration
L1 Icache	256:64:2 (32 kB) - 1 cycle*
L1 Dcache	256:32:2 (16 kB) - 1 cycle
L2 Combined Cache	512:64:2 (64 kB) - 6 cycles
Memory	128 bit wide - 18 + 2 X chunks cycles
Branch Pred. (McFarling)	2k gshare + 2k bimodal + 2k meta
Branch Pred. (Gshare)	8k gshare entries
BTB	1024 entry, 4-way set associative
Return Address Stack	32 entry queue
ITLB	64 entry, fully associative
DTLB	128 entry, fully associative
Ifetch Queue	8 instructions

Table 1: Machine configuration parameters. Cache configurations are described as Lines:Block Size:Associativity. $*$ The 32kByte instruction cache is equivalent to a 16kByte cache because the SimpleScalar Tool Set uses 8 byte instructions.

Parameter	Units
Fetch/Issue/Commit Width	4
Integer ALU	3
Integer Mult/Div	1
FP ALU	2
FP Mult/Div/Sqrt	1
Memory Ports	2
Instruction Window Entries	64
Load/Store Queue Entries	32
Minimum Mispredict Latency	7

Table 2: Resource and pipeline configuration for simulated architecture.

Resource	Latency	Occupancy
Integer ALU	1	1
Integer Mult	3	1
Integer Div	20	19
FP ALU	2	1
FP Mult	4	1
FP Div	12	12
FP Sqrt	24	24
Memory Ports	1	1

Table 3: Function unit configuration in terms of execution latency and occupancy.

We used the SPECint95 applications to evaluate the different pipeline gating techniques. The applications were compiled with the Gcc compiler with full optimization. We used scaled down inputs to reduce the runtime of some applications, but each application was run to completion. Relevant information for the benchmarks, along with the conditional branch misprediction rates for Gshare and McFarling branch predictors, are shown in Table 4. The misprediction measurements use the base processor configuration with no pipeline gating. The misprediction rate across our applications ranges from 2% to 32%. We used the SPECint95 benchmarks for our performance evaluation and did not simulate the SPECfp95 since those programs typically pose few difficulties for branch predictors.

A schematic model of the pipeline was given in Figure 2, and both fetch and decode take two cycles to complete. This model should highlight flaws in pipeline gating, because the time to recover from an incorrect pipeline gating decision is a function of the number of cycles it takes for the gated instructions to reach the issue stage. Hence, the longer the front end of the pipeline, the larger the penalty for incorrect gating. Figure 2 also shows the signals for the pipeline gating mechanism we found to be most effective. The decision to gate and the actual gating is performed during the first fetch cycle. Our performance results show that most of the extra work in the pipeline occurs at the fetch and decode stages, and gating at the fetch stage will have the largest impact. The number of unresolved, low-confidence branches were measured at decode. This insures some "slip" between the fetch and decode stages if we made an incorrect gating decision. This increases the extra work (Ew) of the stages beyond fetch, but also reduces the performance loss by providing the issue stage with a few instructions from the correct-path while the pipeline catches up from an incorrect gating decision.

Pipeline gating is engaged when the number of low confidence branches exceeds the *gating threshold (N)*. As mentioned, this is used to improve the likelihood that at least one mispredicted branch is being processed. Gating is disengaged when the number of low confidence branches is less than or equal to the gating threshold. As was shown in Figure 3, gating is triggered a number of times, but for very few cycles each time. Therefore, pipeline gating effectively slows the injection of instructions into the pipeline rather than stopping instructions altogether.

3.1 Confidence Estimators

Although branch predictors have been widely studied, confidence estimators have only recently been discussed [7, 5]. Thus, we will describe the mechanics of confidence estimation and the confidence estimators we used in more detail. Confidence estimation is a diagnostic test that attempts to classify each branch prediction as having "high confidence", meaning that the branch was likely predicted correctly, or "low confidence", meaning the branch was likely mispredicted. We used the SPEC and PVN metrics defined in the previous section to classify the confidence estimators discussed below.

Perfect Confidence Estimation: Although a perfect confidence estimator is unattainable in practice, we used precise information from the pipeline state to evaluate the potential of pipeline gating, and to determine how much of that potential performance was exploited by other configurations.

Static Confidence Estimation: Static confidence estimation associates a confidence estimate with each conditional branch instruction. The confidence is determined by running the program through a branch prediction simulator and recording the branch misprediction rate of individual branch sites. Branch instructions with a misprediction rate above a specified threshold were considered to have low confidence. Static confidence estimation has the benefit that it can be "customized" for a specific SPEC and PVN. For the experiments in this paper, we wanted to demonstrate the best performance that a static confidence estimator could provide. Thus, we use the same input to select and evaluate the static confidence sites, and we varied the selection threshold across each program to report the best performance. We used the static method for both Gshare and McFarling predictors.

JRS Confidence Estimation: Jacobsen *et al* [7] proposed a confidence estimator that paralleled the structure of the gshare branch predictor. This estimator uses a table of *miss distance counters* (MDC) to keep track of branch prediction correctness. Each MDC entry is a "saturating resetting counter". Correctly predicted branches increment the corresponding MDC, while incorrectly predicted branches set the MDC to zero. A branch is considered to have "high confidence" only when the MDC has reached a particular confidence threshold value referred to as the *MDC-threshold*. For this simulation, we used a table of 4096 entries of 2-bit saturating/resetting counters. We also discuss the effectiveness of different JRS configurations for pipeline gating in future sections. We use the JRS method for both Gshare and McFarling predictors.

Saturating Counters: Most branch predictors use some form of saturating counters to predict the likely branch outcome. Smith [13] mentioned that it may be possible to use these counters as branch confidence estimators. We used this mechanism with the McFarling predictor to produce the "Both Strong" estimation method which marks a branch as high confidence only if the saturating counters for both gshare and bimodal predictors are in a strong state and have the same predicted direction (taken or not-taken). We tried a number of other variants with the McFarling counters and found that the "Both Strong" configuration provided the best results for our needs because it produced a high SPEC value with a reasonable PVN. The saturating counters method did not work well for Gshare.

Distance: In [5], we found that branch mispredictions were clustered and that this clustering could be used to build an inexpensive confidence estimator. The conditional probability of a misprediction for branches that issue d branches after a mispredicted branch is resolved is higher for smaller values of d. Varying the distance d affects the SPEC and PVN – smaller values increase the PVN (but reduce the SPEC). We found a value of $d = 4$ worked best for pipeline gating in our model. We used the Distance method as an inexpensive confidence mechanism for Gshare.

Gshare		
Conf Pred	SPEC	PVN
static	87.5	27.5
JRS	72.8	37.1
distance=4	71.9	25.8
McFarling		
Conf Pred	SPEC	PVN
static	88.4	26.3
JRS	65.9	30.7
Both Strong	77.2	20.3

Table 5: Assorted confidence estimators with the Gshare and McFarling branch predictors. Values given are the arithmetic mean of all committed branches for SpecInt95 benchmarks.

Table 5 shows the performance of the different confidence estimators in terms of SPEC and PVN using the Gshare and McFarling branch predictors. A complete comparison of different confidence estimation methods is beyond the scope of this paper. Instead, we wanted to compare the performance of pipeline gating using inexpensive implementations and more expensive implementations. Unlike the JRS estimator, which has a considerable overhead, the Distance estimator is very inexpensive to implement. Likewise, the "Both Strong" method simply uses existing processor state, and introduces negligible additional hardware cost. Although we tried other estimators with the branch predictors, we found that the ones presented in Table 5 performed the best by producing a high SPEC and a reasonable PVN.

As we will see in later sections, it is more important, within reason, to select an estimation mechanism with a good SPEC value as opposed to one with just a good PVN value. Effectively, using a gating threshold boosts the effective PVN, and it becomes more important to see *more* low-confident branches (*i.e.*, a higher SPEC) than to know that the low confident branches were truly mispredicted (*i.e.*, a higher PVN).

4 Results

The basic configuration used for pipeline gating is given in Figure 2. We evaluated the McFarling and Gshare branch predictors using a variety of modifications. Analysis is performed across different confidence estimators, gating threshold values, and pipeline configurations.

Figures 4 and 5 show the amount of extra work being performed with the McFarling and Gshare predictors, respectively for the base case with no pipeline gating. The bars represent the amount of extra work (EW) performed in each stage of the pipeline. Most of the extra work occurs in the front stages of the pipeline, at fetch and decode. As we progress down the pipeline, the amount of extra work decreases dramatically. This is because most mispredicted branches resolve in a reasonable amount of time, and the probability is small that an instruction from the wrong-path has progressed deep into the pipeline. As expected, the amount of unnecessary work is generally correlated to the misprediction rate. For example, *vortex* has a low misprediction rate, and there is very little extra work being done for this program. On the other hand, the pipeline performs twice the amount of necessary work for *go*, which suffers from a high misprediction rate. Fortunately, confidence mechanisms inherently do better on programs with a large misprediction rate [5], and are most effective in reducing the amount of extra work in programs that have the largest overhead.

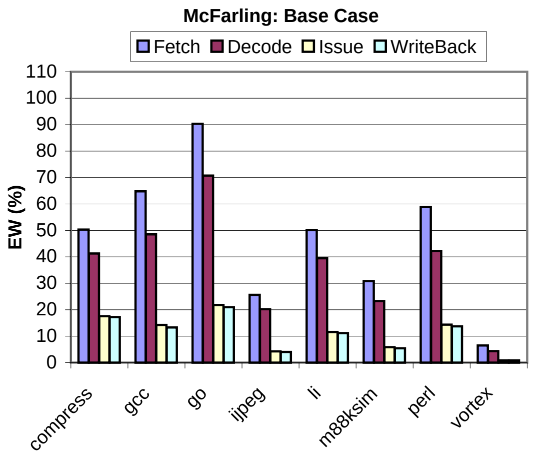

Figure 4: Extra work for base case with the McFarling predictor.

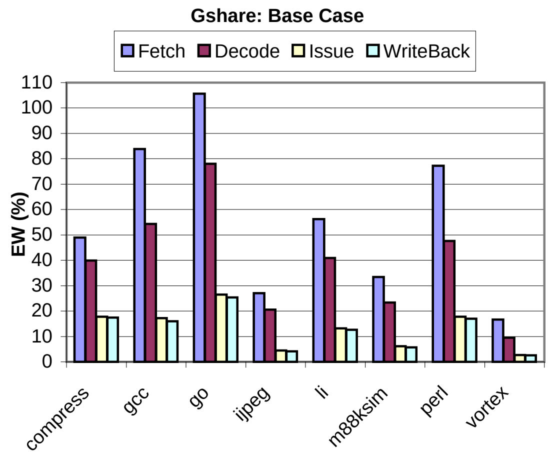

Figure 5: Extra work for base case with the Gshare predictor.

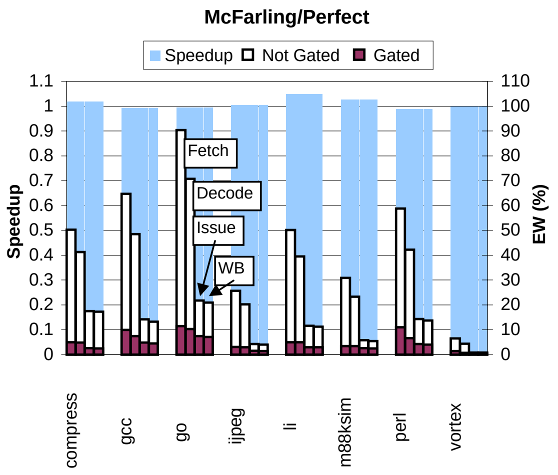

Figure 6: Extra work (Ew) and speedup for McFarling predictor with a perfect confidence estimator. The entire thin bar shows Ew with "No Gating" while the dark portion shows Ew with gating. The wide, gray bar represents relative speedup.

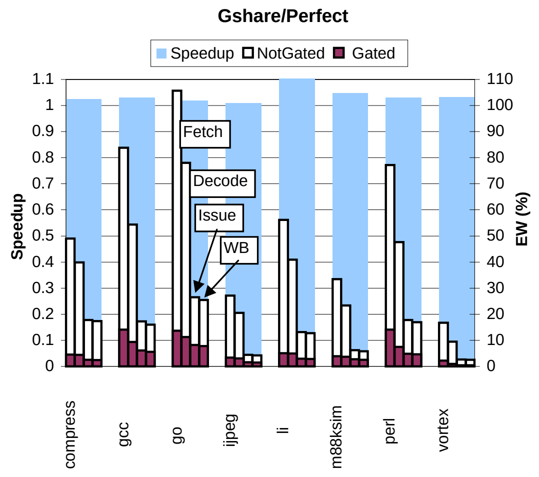

Figure 7: Extra work and speedup for Gshare predictor with a perfect confidence estimator. The entire thin bar shows Ew with "No Gating", while the dark portion shows Ew with gating. The wide, gray bar represents relative speedup.

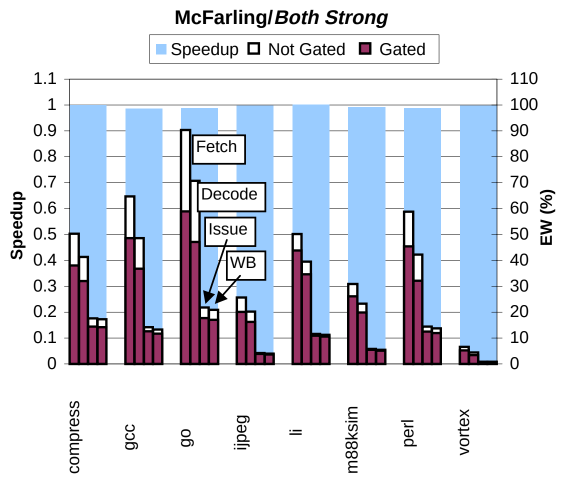

Figure 8: Results for McFarling and "Both Strong" using a gating threshold value of 2.

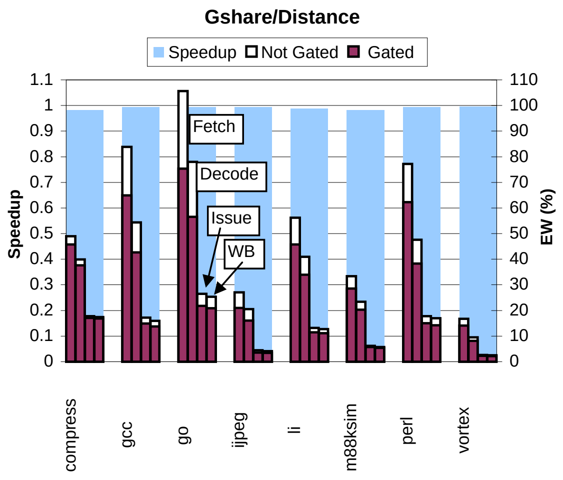

Figure 9: Results for Gshare and Distance using a gating threshold of 2.

4.1 Performance with Different Confidence Estimators

We first explore the effectiveness of pipeline gating as a function of the confidence estimation mechanisms. We present results using perfect confidence estimation, inexpensive dynamic estimation, static estimation, and a more expensive dynamic estimation based on the JRS estimator. For the analysis of different confidence estimators, we used the gating mechanism shown in Figure 2. The pipeline is gated at fetch, and the number of unresolved branches is measured at decode.

Perfect Confidence Estimation: Figures 6 and 7 show the extra work and speedup results when using McFarling and Gshare branch predictors, respectively, with a perfect confidence estimator. The dark portion of the thinner bars represents the amount of extra work with pipeline gating. The entire thin bar represents the amount of extra work without any pipeline gating. The four bars per group represent the four stages of the pipeline: fetch, decode, issue and writeback. We do not show the commit stage since the number of committed instructions is the same with and without pipeline gating. The wide, gray bars represent the speedup of the pipeline gating method relative to the base case. For EW, lower is better, whereas for speedup, higher is better. All speedup numbers above 1.0 represent a performance improvement from pipeline gating, while numbers below 1.0 represent a performance loss. All the data we present for pipeline gating is presented in a similar manner.

With a perfect confidence estimator, one would expect a 100% reduction in extra work. This does not happen with the pipeline gating configuration used because we do not "see" the low confidence branch until it reaches the *decode* stage. Therefore, some extra instructions will "leak" into the pipeline before gating is initiated. Pipeline gating with perfect confidence estimation can result in increased speedup for a number of programs, such as *li* and *m88ksim*. Performance improves in the gated pipeline because operations from the wrong path do not consume resources which correct path instructions might need. On the other hand, some programs, such as *perl* with the McFarling branch predictor, show a performance loss with perfect confidence estimation. Speculative execution has been shown to be beneficial for performance by warming up instruction caches [12], and gating the pipeline reduces the benefits of the warm-up effect. With more realistic confidence estimation mechanisms, we do not gate as many of the incorrectly predicted paths. Hence we still benefit from some of the warm-up effects in the instruction caches.

Inexpensive Dynamic Confidence Estimation: Figures 8 and 9 show results for the "Both Strong" and Distance confidence estimators, respectively. Gating is engaged when there are more than two low-confident branches in the pipeline. Gshare uses the Distance estimator, and McFarling uses the "Both Strong" estimator. These were determined to be the best and least expensive dynamic confidence mechanisms for pipeline gating for the respective branch predictors. The figures show the reduction in extra work and relative speedup for each SpecInt95 program. The dynamic confidence estimation mechanisms for both branch predictors perform well enough to reduce approximately 30% of the extra work in *go*, and yet not hurt performance in *vortex* through unnecessary gating.

Static Confidence Estimation: In Figures 10 and 11, we show results for gating when using a best-case static confidence estimator discussed in Section 3. The static confidence estimators do well for both McFarling and Gshare predictors. In the case of the McFarling predictor, a few programs, such as *compress*, do better with static profiling, but the results in general are about the same as the "Both Strong" estimation mechanism. For Gshare, on the other hand, there is marked reduction in extra work. For *gcc*, the EW is reduced from over 80% to just over 50% in the fetch stage. With the Distance estimator, we were only able to reduce this to 65%. The Distance estimator relies on the clustering behavior of mispredicted branches. Some programs, such as *go*, exhibit significant misprediction clustering while others, such as *compress* and *m88ksim* do not. Hence, the Distance method is not as consistent or accurate in its confidence estimations for Gshare as the Saturating Counters method is for McFarling.

Dynamic Confidence Estimation with JRS: Data for McFarling and Gshare predictors with a small JRS estimation mechanism is shown in Figures 12 and 13. We restricted ourselves to a JRS size of 1kByte or less because of area and power considerations. The results shown use a 128 entry, 4-bit JRS table for both branch predictors. The JRS estimator for McFarling used a MDC-threshold of 15, while the JRS estimator for Gshare used a MDC-threshold of 12. As mentioned earlier, a branch is considered to have "high confidence" only when the miss distance counters (MDC) have reached a specified MDC-threshold value. The results for McFarling with JRS are similar to those using the "Both Strong" estimator. Gshare results, on the other hand, improve significantly with the JRS estimator. For example, the reduction in EW for *compress* improves from 6% to 32%. Results produced are similar to those generated with the static estimation method. There are a couple of explanations for this. First, as discussed earlier, the Distance predictor does not do well for some types of programs. Secondly, the JRS estimator is tuned to work well with the Gshare predictor [7, 5], and does not perform as well with the McFarling predictor. If the hardware can be justified, a small, multi-bit, JRS confidence estimator will provide the best results of any dynamic estimation mechanism for Gshare.

JRS Configurations For Pipeline Gating: The JRS configurations that worked best for both Gshare and McFarling had a small table size and a large counter size. The question that remains is why such a small JRS table does so well. Figure 14 shows the geometric mean of EW and speedup for a variety of JRS table configurations with the Gshare predictor. The values of EW without pipeline gating does not change as a function of the JRS table, because the JRS estimator does not affect the "Not Gated" case. Although the first two sets of data (128–4bit, 256–2bit) use the same size JRS table, albeit different configurations, they show very different results. Furthermore, the larger tables shown do not produce significantly better results for the amount of hardware used. This is because even the largest JRS table suffers from a relatively low PVN value, and a gating threshold must be used to boost the effective PVN. As noted earlier, the best PVN values are around 0.4, and even a small increase in PVN requires considerable extra hardware. Therefore, it is far less expensive to target a high SPEC value and increase the accuracy of the estimation with the aid of the gating threshold.

To verify this hypothesis, we ran the 128 entry JRS table with different MDC-threshold values. Figure 15 shows the geometric mean of EW and speedup for a range of MDC-threshold values (which are labeled *T*) using a 128 entry JRS table with the Gshare predictor. The gating threshold was set to 2 for all of these simulations, which means that gating is engaged when there are 3 or more low confident branches in the pipeline. Figure 15 clearly shows the reduction in extra work with larger MDC-threshold values. As we increase the MDC-threshold, more branches are classified as "low confidence", resulting in a larger SPEC and lower PVN. With the lower PVN, we see a corresponding reduction in performance because the confidence estimation is less accurate. However, the in-

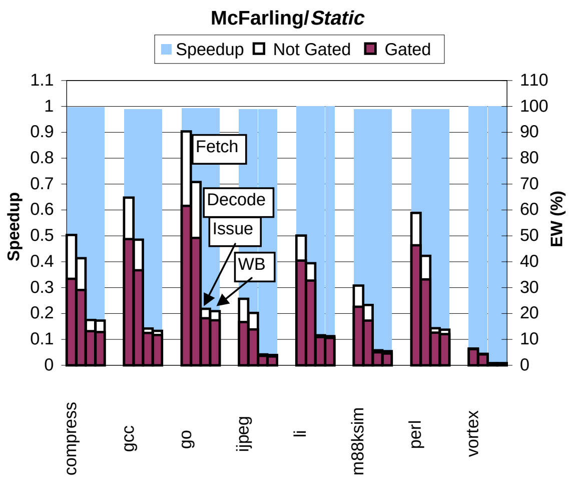

Figure 10: Extra work and speedup for McFarling with static confidence estimation.

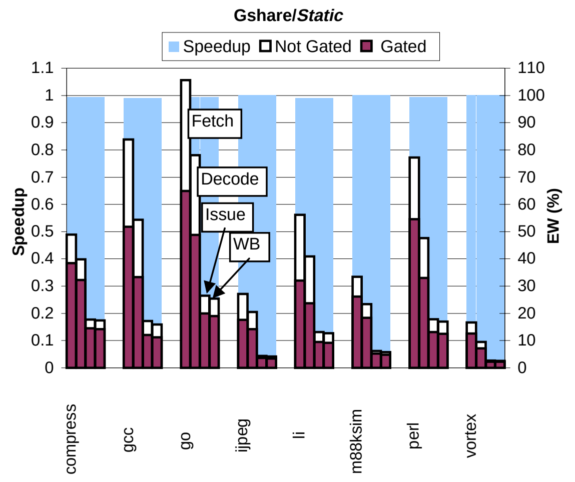

Figure 11: Extra work and speedup for Gshare with static confidence estimation.

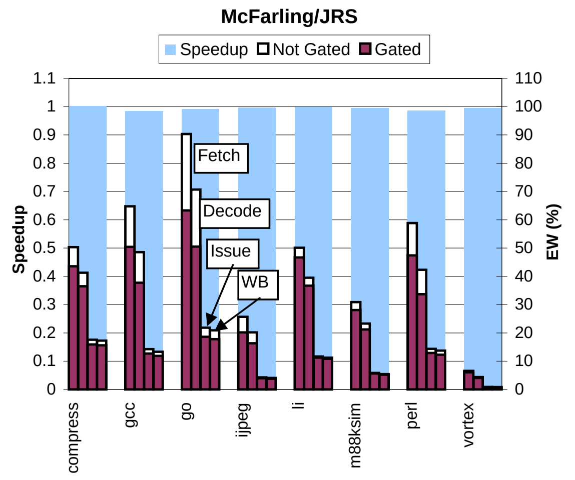

Figure 12: Results for gating using a 128 entry, 4-bit JRS table with McFarling. A gating threshold value of 3 was used.

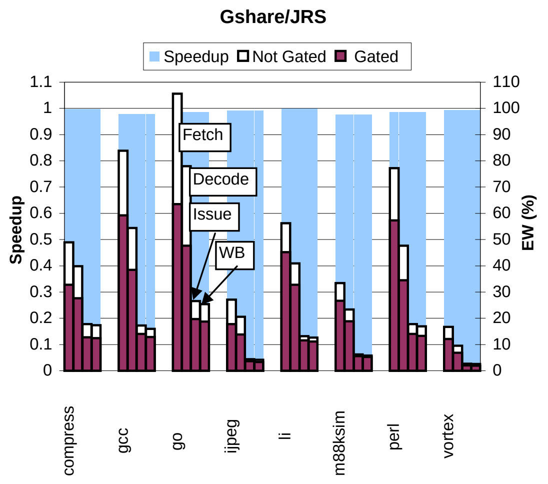

Figure 13: Results for gating using a 128 entry, 4-bit JRS table with Gshare. A gating threshold value of 2 was used.

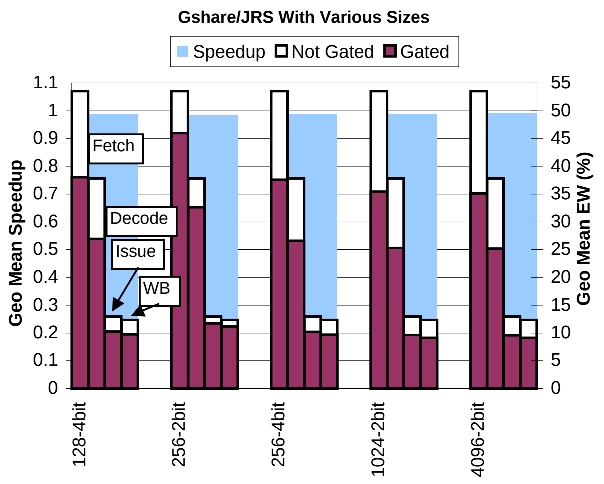

Figure 14: The effectiveness of various JRS table sizes for work reduction. The size of each table is given in <entries>–<bits per entry>.

crease in SPEC is much larger than the decrease in PVN. The SPEC increases from a value 34.4 to 93.10 as the MDC-threshold changes from 1 to 15, while the PVN decreases from 31.5 to 21.3. Since gating is engaged when there are 3 or more unresolved, low confident branches in the pipeline, the probability that at least one of the three low confident branches is mispredicted is $1-(1-.315)^3 = 68\%$ for a MDC-threshold of 1, and $1-(1-.213)^3 = 51\%$ for a MDC-threshold of 15. Although we are less accurate when the MDC-threshold is large, the short duration of gating events and the "slip" between instruction fetch and decode helps reduce the performance penalty due to incorrect gating.

4.2 Varying the Gating Threshold

Figures 16 and 17 show EW and speedup for the McFarling and Gshare predictors, respectively, as a function of the gating threshold value *N*. The gating threshold is used to determine the maximum number of low-confidence branches allowed in the pipeline before gating is triggered. The "Both Strong" confidence mechanism was used with McFarling, and the Distance mechanism was used with Gshare. The data given is the geometric mean of EW and speedup for different gating threshold values. Note that the value of EW without pipeline gating does not change as a function of *N*, since the gating threshold does not affect the "Not Gated" case.

The leftmost set of bars show the results for a configuration with a gating threshold of zero and all branches tagged as low confidence. This effectively reduces the pipeline to a super-scalar, non-speculative machine, which provides the best energy reduction albeit with a high performance penalty. This is not an exact replica of a non-speculative machine, which would see a EW value of zero. As with the perfect confidence estimation case, EW is not zero because we only "see" a low-confidence branch at decode. The speedup loss is over 35% for both predictors when approximating a non-speculative machine, although we achieve a substantial reduction in EW. With a reasonable confidence estimator and a gating threshold of zero, we still significantly reduce the amount of EW without the performance loss seen in a non-speculative machine. Although this loss in performance is not appropriate for power reduction, other applications, such as bandwidth multi-threading, might benefit from a zero gating threshold.

For work reduction with no performance loss, both figures clearly show the need for a gating threshold to compensate for a low PVN value. As the gating threshold (labeled *N*) increases, speedup improves but EW also increases. Ideally, as *N* increases, the improvement in speedup should be greater than the increase in EW. In both figures, this occurs for $N = 2$, given tight constraints on performance. Using a gating threshold value of two, we are able to reduce EW in the fetch and decode stages by approximately 25% and 23% for McFarling, and 18% and 17% for Gshare with a negligible performance loss.

4.3 Varying the Pipeline Structure

So far, we have investigated various confidence estimation mechanisms and gating threshold values, but have not changed the underlying structure of the gated pipeline. We decided to gate at fetch and measure at decode so that we could 1) capture a large portion of the wrong path instructions in fetch, and 2) allow some slip into the pipeline, respectively. We explored moving the point of gating to the decode and issue stages. All results in this section were generated for the McFarling predictor using the "Both Strong" estimation method. Table 6 shows results for no gating, for measuring at decode and gating at fetch, and measuring and gating at decode.

Gating at decode produces worse results for EW at the fetch stage than gating at fetch, although there is still an overall reduction

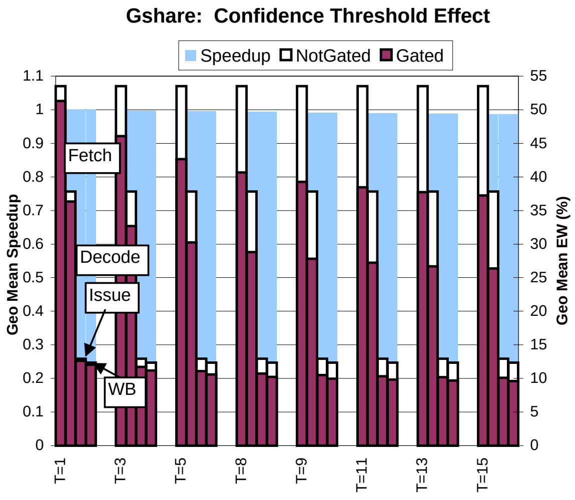

Figure 15: The effectiveness of a 128 entry JRS table as a function of MDC-threshold value. T denotes the MDC-threshold.

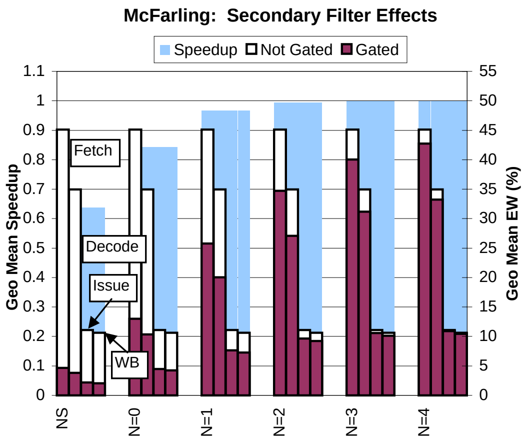

Figure 16: EW and speedup as a function of gating threshold values (denoted *N*) for the McFarling predictor. Also shown is the non-speculative version of the processor (NS).

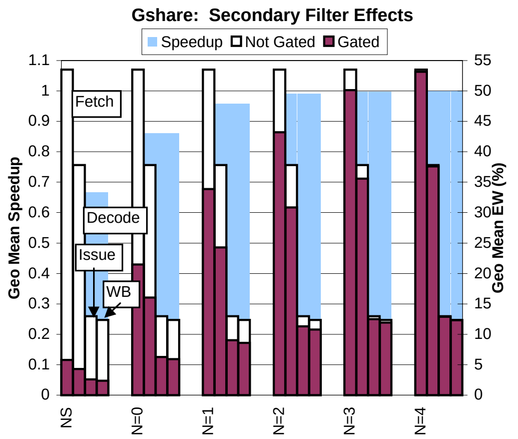

Figure 17: EW and speedup as a function of gating threshold values for Gshare. Also shown is the non-speculative version of the processor (NS).

		McFarling		
Case	Fetch	Decode	Issue	WriteBack
Base	45.12	34.96	11.11	10.64
Gate at Fetch	34.74	27.10	9.66	9.23
Gate at Decode	39.24	27.83	9.77	9.33

Table 6: Geometric Mean of Ew for base case (No Gating), gating at fetch, and gating at decode for McFarling.

in work when compared to the base case. This is reasonable since we are allowing the fetch stage to continue fetching until the fetch buffer is full. Therefore, more instructions will enter the pipeline; this would not happen if gating disabled instruction fetch. We expected to see an improvement in performance with gating at decode since the recovery penalty for incorrect gating would be less than gating at fetch. It takes only three cycles for an instruction to "catch up" and issue after an incorrect gating event with gating at decode as opposed to five cycles with gating at fetch. Results show no real performance benefit from moving the gating point from fetch to decode. As shown in Figure 3, the pipeline is generally not gated for more than a few cycles. The current pipeline model has a 64-entry register update unit, and results show that it usually has enough instructions in the issue queue to keep the execution units occupied while the pipeline catches up from gating.

We also tried other gating configurations such as measuring low confidence branches at the second decode cycle, and gating at issue. None of these configurations performed as well as gating at fetch and measuring at decode. Measuring at the second decode cycle did not change the results in any significant manner. Gating at issue resulted in very little savings since most of the wrong-path instructions do not reach the issue stage. Due to space limitations, we will not present results for these configurations. Of all the pipeline gating configurations attempted, gating at fetch and measuring at decode produced the best results.

5 Conclusion

We have looked at speculation control to reduce the amount of energy consumed in a speculative, multi-issue, out-of-order processor. We introduced a new mechanism, pipeline gating, which results in a reduction of instructions in the pipeline without significantly altering performance. We have shown results for different branch predictors and confidence estimators, and implemented inexpensive dynamic confidence estimation methods that do a reasonable job of reducing unnecessary work. Furthermore, we presented a practical configuration for the JRS confidence estimator that successfully reduces energy without a large hardware penalty. Most importantly, we showed that inexpensive, dynamic confidence estimation mechanisms exist which, at worst, do not impact performance for highly predictable programs, and at best, reduce work by a measurable amount for programs with a large misprediction rate.

Architectural level power reduction in high performance processors is a broad field and one that is in its infancy. We have presented an innovative method for reducing power, and there is much work left to be done in this area. With wider width processors and hyper speculation in the foreseeable future [9], pipeline gating methods will become even more essential for no-risk energy reduction in high performance processors.

Acknowledgments: We would like to thank Steve Gunther for many invaluable conversations on power dissipation in microprocessors, Doug Burger and Todd Austin for supporting SimpleScalar, and Todd Austin for his help in developing the ideas presented in this paper. We would also like to thank the referees for their helpful comments. This work was conducted on equipment provided by a Digital Equipment Corporation grant, and was partially supported by a grant from Hewlett-Packard, NSF grants No. CCR-9401689, No. MIP-9706286 and in part by ARPA contract ARMY DABT63-94-C-0029.

References

[1] Thomas D. Burd and Robert W. Brodersen. Processor design for portable systems. *Journal ov VLSI Signal Processing*, 13(2/3):203–222, August 1996.

[2] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. TR 1342, University of Wisconson, June 1997.

[3] Richard Fromm, Stylianos Perissakis, Neal Cardwell, Christoforos Kozyrakis, Bruce McGaughy, and David Patterson. The Energy Efficiency of IRAM Architectures. Technical report, May 1997.

[4] Ricardo Gonzalez and Mark Horowitz. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, September 1996.

[5] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew Pleszkun. Confidence esimation for speculation control. In *Proceedings 25th Annual International Symposium on Computer Architecture*, *SIGARCH Newsletter*, Barcelona, Spain, June 1998. ACM.

[6] Steve Gunther and Suresh Rajgopal. Personal communication.

[7] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning Confidence to Conditional Branch Predictions. In *International Symposium on Microarchitecture*, pages 142–152, December 1996.

[8] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *IEEE Micro*, December 1997.

[9] M. H. Lipasti and J. P. Shen. Superspeculative microarchitecture for beyond ad 2000. *IEEE Computer*, 30(9), 1997.

[10] S. McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[11] J. Montanaro and *et. all*. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. In *Digital Technical Journal*, volume 9. Digital Equipment Corporation, 1997.

[12] J. Pierce and T. Mudge. Wrong-Path Instruction Prefetching. *IEEE Micro*, December 1996.

[13] J.E. Smith. A Study of Branch Prediction Strategies. In *Annual International Symposium on Computer Architecture*, *SIGARCH Newsletter*, pages 135–148, May 1981.

[14] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Annual International Symposium on Computer Architecture*, *SIGARCH Newsletter*, pages 194–205. IEEE, June 1997.