# Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors

Amirali Baniasadi
Electrical and Computer Engineering
Northwestern University
amirali@ece.northwestern.edu

Andreas Moshovos
Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

## ABSTRACT
We present a number of power-aware instruction front-end (fetch/decode) throttling methods for high-performance dynamically-scheduled superscalar processors. Our methods reduce power dissipation by selectively turning on and off instruction fetch and decode. Moreover, they have a negligible impact on performance as they deliver instructions just in time for exploiting the available parallelism. Previously proposed front-end throttling methods rely on branch prediction confidence estimation. We introduce a new class of methods that exploit information about instruction flow (rate of instructions passing through stages). We show that our methods can boost power savings over previously proposed methods. In particular, for an 8-way processor a combined method reduces traffic by 14%, 20%, 6% and 6% for the fetch, decode, issue and complete stages respectively while performance remains mostly unaffected. The best previously proposed method reduces traffic by 10%, 15%, 4% and 4% respectively.

## 1. INTRODUCTION
Successive high-performance processor generations have thus far relied on higher frequencies and more transistors to improve performance. Unfortunately, using more transistors clocked at higher frequencies requires more power. As a result, power dissipation in modern processors is now quickly approaching alarming levels jeopardizing further advances in performance. Finding ways to control further increases in power is imperative for future generation processors with billion transistors operating at multi-GHz frequencies.

In this work we are concerned with power reduction techniques at the *architectural* level that are complementary to low-level circuit techniques. Our goal is to revisit existing architectural decisions revising where necessary so that energy is used more efficiently. This is possible, as existing architectural decisions ignored power for the most part, focusing instead on performance, complexity and cost trade-offs.

A large fraction of power in modern high-performance processors is dissipated by the *front-end*, that is by instruction fetch and decode. For example, instruction decode and fetch dissipate 28% of overall (average) processor power in the Intel P6 [3] and as much as 36% for the less complex MIPS R3000 [1].

In this work, we are concerned with power-aware instruction fetch/decode designs for dynamically scheduled, superscalar processors. Previous work in this area exploits energy inefficiencies that are caused by incorrect control flow speculation (or, in the interest of space, *speculation*) [3,4]. Incorrect speculation does not improve performance[1] while it leads to extraneous instructions passing through the pipeline consuming additional energy (as we show in section 4, these instructions can be as much 4 times more than those executed by the program). Previous power-aware front-end proposals assign confidence to speculation decisions. They turn-off (or *gate* or *throttle*) the front-end on low confidence speculation decisions (i.e., when it is highly likely that the front-end is processing instructions down an incorrectly speculated control flow path). We explain their operation in detail in section 2. Power is reduced significantly since the number of extraneously processed instructions is reduced. Of course, currently it is not possible to always predict when the front-end has wondered off to an incorrect control flow path.

In this work we introduce a new class of power-aware front-end throttling methods that are complementary to previously proposed methods. Our methods are oblivious to control flow speculation. They rely on another well understood inefficiency of conventional front-end designs: Existing designs process instructions with the maximum possible rate since this maximizes the chances of exploiting as much parallelism as possible. However, this does not always improve performance significantly. Our methods exploit this inefficiency by slowing down the front-end (they turn it off for a couple of cycles) when they predict that this will *not* reduce performance. This prediction is based on local information about the instructions passing

1. Actually, in principle incorrect speculation may improve performance indirectly by prefetching memory data or instructions. However, as we will see in section 4, this rarely happens in practice.

through various pipeline stages, or *instruction flow* or *traffic*. Since our methods are oblivious to control flow speculation they also reduce the number of extraneous instructions appearing on miss-speculated control flow paths. For this reason, they also reduce power demands. We make the following contributions:

- A minor contribution of this paper is that we demonstrate that the previously proposed speculation confidence-based methods remain effective for future, highly-aggressive processor designs.
- A major contribution of this work is that we introduce a new class of power-aware front-end throttling mechanisms that rely on instruction-flow information. We demonstrate that these methods can *complement* existing speculation confidence-based methods, further decreasing power dissipation.

The rest of the paper is organized as follows, In section 2, we explain the rationale of our approach and discuss the various heuristics. In section 3, we discuss and report both performance and power related results. In section 4, we review related work. Finally in section 5, we summarize our findings and offer concluding remarks**.**

## 2. POWER-AWARE FRONT-END THROTTLING

Conventional front-end designs rely heavily on control flow speculation to forge ahead into the dynamic instruction stream. Speculation allows a processor to guess the target of a branch without waiting for it to execute. Consequently, the processor can speculatively fetch and start executing the target instructions. Eventually, when the branch executes, the processor verifies its guess. While speculation greatly improves performance, it also increases power dissipation. In particular, instructions that execute due to incorrect speculation consume power while they do not contribute to performance[2]. As we also show in section 3, the number of extraneous instructions can account for as much as 80% of all instruction passing through the front-end stages. Unfortunately, not using speculation in most modern processors is not an option as the performance penalty is too high. Moreover, as we move toward deeper pipelines and wider/deeper instruction windows, the need for speculation increases.

Previous work on power-aware front-end designs focused on using speculation confidence mechanisms to reduce the number of extraneously executed instructions. Speculation confidence mechanisms predict when the front-end is fetching instructions down an incorrectly speculated control flow path. If so, they turn the front-end off until confidence is regained, an action referred to as *gating* or *throttling*. Of course, if a perfect confidence mechanism was possible, then perfect speculation would be possible too.

In this work we propose a new class of front-end throttling methods that are *orthogonal* to existing confidence-based methods. Our methods attack another well known power inefficiency of modern processor designs. In particular, conventional processors strive to fetch and decode as many instructions as possible

2. Some performance improvements are possible by side-effects, such as prefetching of data and instructions.

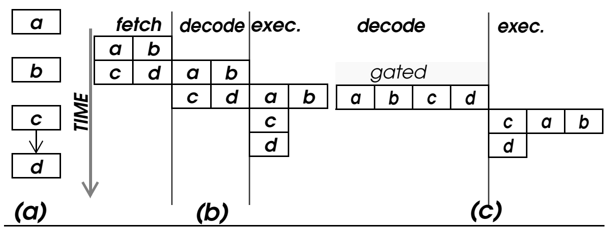

**Figure 1: Decoding as fast as possible may not improve performance. (a) Four instructions in program order. (b) Fetch, decode and execute in a conventional 4-way processor. Fetching and decode proceeds as quickly as possible. (c) Decode is gated for one cycle. It still takes 5 cycles to process all four instructions even though some of them are delayed.**

and as quickly as possible. However, it is well understood, that the additional parallelism that may be exposed (if any) does not always improve performance. Often times, the processor is stalled waiting for some other instructions to complete. To understand this, let us consider the example of figure 1 and let us focus on decode power. Shown in part (a) is sequence of four instructions, *a* through *d*. Instructions *a* and *b* are independent, while *d* depends on *c*. In part (b) we see how these instructions may be fetched, decoded and executed in a conventional 4-way processor. Even though the processor has the decode resources to process all four of them in a single cycle, the fetch stage delivers them in pairs in two consecutive cycles. This can happen, for example, if the four instructions appear on two different cache blocks (which are accessed in separate cycles). As a result it takes a total of 5 cycles to execute these instructions. In part (c) we turn off decode for one cycle (we omit the fetch stage in the interest of space). As a result, we decode all four instructions at the same time (3rd cycle). Even though *a* and *b* are decoded a cycle later than they did in part (b), notice that it still takes a total of 5 cycles to process all four instructions.

We can exploit this phenomenon to throttle the front-end whenever we predict that fetching/decoding as quickly as possible will not lead to significant performance improvements. As a result, instructions are fetched just in time for exploiting parallelism. This applies to *both* correctly and incorrectly speculated instructions. In the latter case, since fetch/decode will proceed at a slower pace, we essentially reduce the number of incorrectly speculated instructions that enter the pipeline and hence we reduce power. Our methods work by selectively disabling fetch and decode on a cycle by cycle basis. These methods reduce *average dynamic power.* In the sections that follow we review the heuristics we have investigated.

## 2.1. Flow-Based Heuristics

Our methods are *instruction flow-based* or *flow-based*. They rely on information about the instructions that flow through each pipeline stage. They use this information to estimate the amount of instruction level parallelism currently present in the processor. They stall fetch/decode when they estimate that sufficient parallelism is already available. In this case, the assumption is that introducing additional instructions would not impact performance significantly. The decision to throttle the front-end is done on a cycle by cycle basis. When so determined, we simply disable fetch and decode for 3 cycles (the decode stage actually occupies 3 pipeline stages in our configuration which is heavily pipelined). Among the various method we tried, we report the best three: *Decode/Commit Rate, Dependence-based* and *Adaptive*.

**Decode/Commit Rate (DCR):** This technique estimates that sufficient parallelism exists when the number of instructions passing through decode *exceeds* significantly the number of instructions that commit. Intuitively, branch miss-speculations will result in more instructions being decoded than committed. Also, when there is little parallelism we will temporarily observe many instructions being decoded (while filling up the window) and few being committed.

This technique simply compares the number of instructions decoded and committed during each cycle. Best results where obtained when we throttled the front-end when three times as much instructions were decoded than committed.

**Dependence-Based (DEP):** As another estimate of available parallelism this method simply inspects the instructions currently being decoded counting the dependences among them. Whenever the number of dependences exceeds a pre-specified threshold we stall the front-end during the next three cycles. We tried different threshold values and the best results were possible with the threshold was set to half the decode width. The intuition is that a high number of dependences is an indication of a long and probably critical computation path. Consequently, it is unlikely that introducing additional instructions as quickly as possible will have a significant impact on performance. We have also experimented with taking into account dependences with the instructions already in the window. However, the results were not promising. This heuristic is also fairly straightforward to implement. Since dependence checking is done during decode, the necessary information is already available. Accordingly, the power overhead should be negligible.

**Adaptive (ADAP)**: As we show in section 3, DEP works best for some benchmarks whereas DCR works best for some others. The adaptive method attempts to exploit the best of both techniques. We observed that in most cases DEP works best when the commit rate is relatively low. Accordingly, we developed a method that measures the commit rate over short periods of time and selects DEP when the commit rate is low, or DCR when it is high. In particular, we count the number of committed instructions over a period of 1024 cycles. Then we divide this number by 1024 (ignore the lower ten bits). If the number exceeds a pre-specified threshold (4 for our 8-way processor) we revert to DCR for next 1k cycles. Otherwise, we use DEP. We also reset the counter and start counting over again.

## 2.2. Confidence-Based Heuristics

Previously proposed *control flow confidence-based* heuristics, or simply *confidence-based* heuristics aim at reducing the number of erroneously executed instructions due to control flow miss-speculations. Of course, it is not possible to identify miss-predictions early enough to avoid fetching and decoding instructions down the miss-predicted path. Otherwise, perfect branch prediction would be possible and there wouldn't be any miss-speculations to start with. For this reason, these methods rely on *confidence* mechanisms to identify low confidence speculation decisions.

In investigating these heuristics we draw from the suggestions and experience reported in [3]. We investigate the following heuristics: C*Last, Cjrs* and *CboS*. These heuristics employ different mechanisms for identifying *low confidence* branches. These are branches for which we have experienced low prediction accuracy. To throttle the front-end we count the number of low confidence branches in the pipeline [3]. We stall the front-end when this number exceeds a pre-specified threshold (2 in our case chosen after experimenting with various values).

**Clast:** This heuristic uses an PC-indexed table with a single bit per entry. This table is accessed for branches during fetch. The bit indicates whether the corresponding branch was miss-predicted last time it was encountered. Updates are done at commit. A branch is deemed low-confidence if the bit is set.

**Cjrs:** This is the JRS confidence estimator proposed by Jacobsen *et al.* [2]. This estimator uses a table of miss distance (saturating) counters (MDC) to keep track of branch prediction correctness. Correctly predicted branches increment the related MDC while the miss-predicted ones reset it. A branch is considered high confidence if its MDC exceeds a threshold (here 12).

**CboS:** This is the "Both Strong" heuristic suggested by Manne *et al.* [3]. It deems a branch as low confidence when any of the two underlying branch predictors (we use a combined, McFarling predictor) is not strongly biased. This mechanism leverages information that is readily available and hence it power and complexity requirements are negligible.

Manne *et al.*, suggest additional confidence based methods. However, we present the ones that perform the best for our base configuration and branch predictor.

## 2.3. Combining Confidence and Flow

The last method we report, combines the best confidence- and flow-based heuristics. This is the *Cadap* method which combines *CboS* and *ADAP*. Here we throttle the front-end when either of the underlying heuristics instructs us to do so. We will show that this method outperforms all others, demonstrating that our flow-based methods can indeed complement existing confidence-based methods.

## 3. METHODOLOGY AND RESULTS

In this section, we present our analysis of the various front-end throttling methods. We report performance results in section 3.2. We report power-related measurements in section 3.3.

We used programs from the SPEC'95 suite compiled for the MIPS-like architecture used by the Simplescalar v3.0 simulation tool set. We used GNU's gcc compiler (flags: -O2 –funroll-loops –finline-functions). Table 1 reports the dynamic instruction count and control flow prediction accuracy per benchmarks (notice that this include all control flow instructions and takes into account not only direction prediction but also target prediction). In the interest of space, we use the abbreviations listed

**Table 1: Benchmarks, dynamic instruction count (committed) and control flow prediction accuracy (direction and target).**

| Benchmark | Ab. | Inst. Count | BP Acc. |
|---|---|---|---|
| ***gcc*** | gcc | 265M | 90% |
| ***go*** | go | 132M | 80% |
| ***compress*** | com | 152M | 90% |
| ***li*** | li | 202M | 88% |
| ***ijpeg*** | ijp | 136M | 92% |
| ***m88ksim*** | m88 | 209M | 93% |
| ***perl*** | per | 185M | 92% |
| ***vortex*** | vor | 294M | 98% |
| ***fpppp*** | fpp | 150M | 94% |
| ***swim*** | swm | 213M | 98% |

**Table 2: Base configuration details.**

| ***Base Processor Configuration*** | |
|---|---|
| ***Branch Predictor*** | 64K GShare+64K bi-modal with 64K selector |
| ***Scheduler*** | 256 entries, RUU-like |
| ***Fetch Unit*** | Up to 8 instr. per cycle. Max 2 branches per cycle 64-entry Fetch Buffer |
| ***Load/Store Queue*** | 128 entries 4 loads or stores per cycle Perfect disambiguation |
| ***Issue, Decode, Commit Bandwidth*** | any 8 instructions / cycle |
| ***Functional l Unit Latencies*** | same as MIPS R10000 |
| ***L1 - Instruction /Data Caches*** | 64K, 4-way SA, 32-byte blocks, 3 cycle hit latency |
| ***Unified L2*** | 256K, 4-way SA, 64-byte blocks, 16-cycle hit latency |
| ***Main Memory*** | Infinite, 100 cycles |

under the "Ab." column to refer to these benchmarks.The base configuration we used is detailed in table 2.

## 3.1. Estimating Power

Measuring power dissipation for a modern processor is challenging in part due to the complexity of the underlying structures. Furthermore, different design styles (e.g., pre-charged or static cells) may significantly impact power dissipation. For these reasons, in this work we use *traffic* as an indirect, approximate metric of dynamic power dissipation. We measure traffic per pipeline stage and define it as the number of instructions passing through the stage. Given that we are suggesting high-level optimizations that may be applicable to a variety of implementations, traffic is a high-level design-independent, albeit possibly inaccurate, way of estimating the potential power savings.

## 3.2. Performance

As explained in section 2, reducing the rate of fetch/decode can negatively impact performance. Figure 2 reports relative performance for all methods. In part (a) we show performance for the confidence-based methods, while in part (b) we do the same for the flow-based methods. Performance is reported relatively to the base configuration. Numbers lower than 1 represent slowdowns. In part (a) we include an ideal confidence mechanism (Cideal) that never fetches miss-speculated instructions. While this is impractical, it provides us with an upper bound. We also include Round-Robin (RR), a simplistic method that throttles the front-end on and off every three cycles. We include this method as we believe that it is a fairly obvious alternative and hence could provide an indication (not a proof) on whether more elaborate methods are necessary. As shown, RR reduces performance by 25% on average and as much as 47% for ijpeg. This suggests that the additional cost and complexity of the other methods may be worthwhile. Another fairly obvious alternative is to use a narrower processor. We have experimented with a 4-way processor and found that it too significantly lacks in performance. Interestingly, Cideal outperforms the base suggesting that rarely miss-speculated instructions have positive side-effects (e.g., prefetching). Cjrs performs significantly worse than Clast and CboS for most programs. While Clast performs slightly better than CboS, the slowdowns are on the average relatively small (1.3% and 1.5% respectively).

The flow-based DEP and DCR are not robust. For example, DEP does not perform well for the two floating point programs while DCR fails for jpeg and m88ksim. Fortunately, the ADAP method indeed captures the best of both offering good performance across the board. In some cases (e.g., perl) ADAP performance is worse than DEP or DRC, however, the differences are minor. On the average, ADAP, DEP and DCR are within 3.6%, 3.8% and 2.8% of the base case respectively. This does not compare favorably to the confidence-based methods. However, the combined method Cadap, performs within 2% of the base which is close to CboS and Clast.

## 3.3. Traffic Reduction

Figure 3 reports the traffic due to control miss-speculation for the fetch, decode, issue and complete stages. Reported is the number of extra instructions passing through those stages expressed as a fraction of all instructions passing through that stage (e.g., 0.7 means that 3 in 10 instructions are committed). As expected, extra traffic is higher for integer codes and it is correlated to branch prediction accuracy. In addition, extra traffic is higher for decode and fetch since fewer miss-speculated instructions get to issue or complete. Interestingly, in most integer codes, the vast majority of fetched and decoded instructions belong to miss-speculated paths (e.g., 72% of fetched and 56% of decoded instructions in gcc).

Figure 4 reports *traffic reduction* for the confidence methods. Reported is the total number of instructions passing through each stage measured as fraction of the total number of instructions passing through the same stage in the original non-power-aware configuration. Both correctly speculated and extraneous

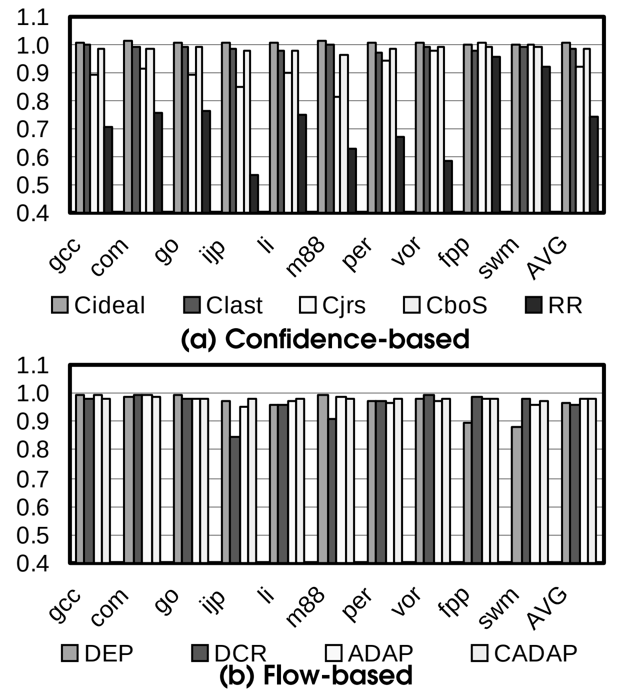

**Figure 2: Relative performance with various front-end throttling methods. (a) Confidence-based, (b) Flow-based. We also include the simplistic round-robin (RR) method and an ideal confidence-based method (Cideal). Higher is better.**

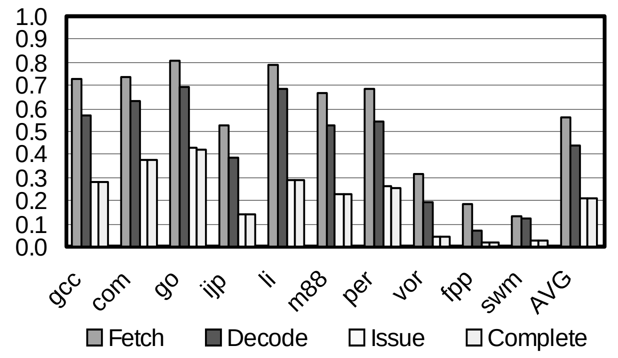

**Figure 3: Traffic (instructions) per stage due to control miss-speculation measured as a fraction of all instructions passing through that stage.**

instructions are included in this metric. For example, ideally we could see a fetch traffic reduction of up to 72% in gcc. Parts (a) through (c) show the results for Clast, Cjrs and CboS respectively. The Cjrs method does exceptionally well, however, as we have seen it also impacts performance significantly. On the average, Clast reduces extra traffic by 6.8%, 10.7%, 2.3% and 2.6% at the fetch, decode, issue and complete stages. CboS performs better by reducing extra traffic by 10.5%, 15.2%, 3.9% and 3.9% for the aforementioned stages.

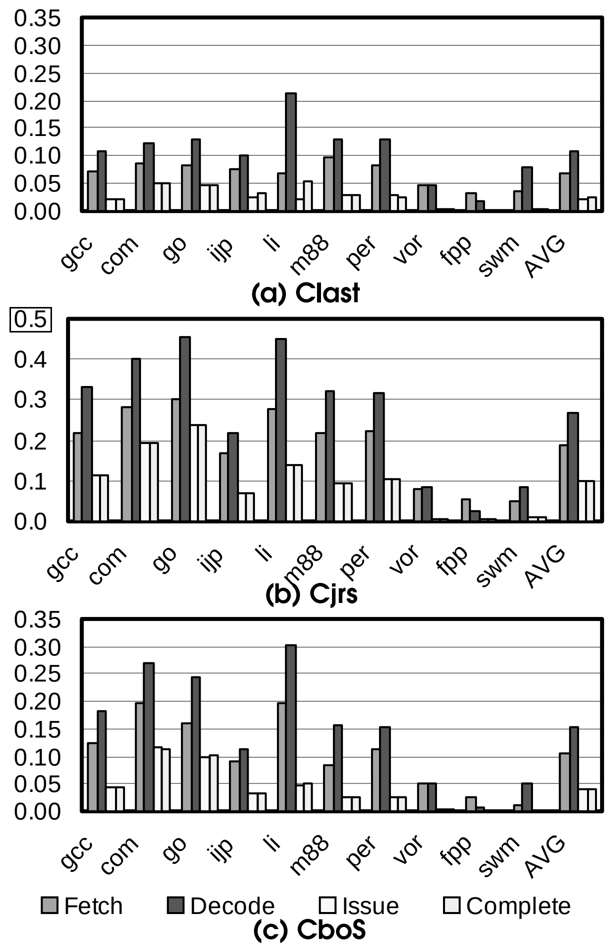

**Figure 4: Traffic reduction for the confidence-based methods. Higher is better.**

Figure 5 reports traffic reduction for the flow-based methods. Parts (a), (b) and (c) are for the DEP, DCR and ADAP methods respectively. DCR tends to perform much better than DEP with some exceptions (e.g., li). Fortunately, the ADAP method offers the best of DEP and DCR in most cases. In some cases (e.g., go) it evens outperforms both DEP and DCR suggesting that it alternates between the two during program phases. In m88ksim, however, it fails to perform as well as DCR (however, it still reduces traffic significantly). The average traffic reductions are for ADAP 10.8%, 15.2%, 4.2% and 4.4% for the fetch, decode, issue and complete stages.

Finally, figure 5(d) shows traffic reduction for the combined method Cadap. On the average, the reductions for the fetch, decode, issue and complete stages are 14.9%, 20.1%, 6.2% and 6.0%. These represent an increase of 4.1%, 4.9%, 2.3% and 2.2% in absolute terms over the best confidence-based method.

## 4. RELATED WORK

Many architectural power optimizations have been proposed. In this section we restrict our attention to front-end throttling techniques for high-performance processors.

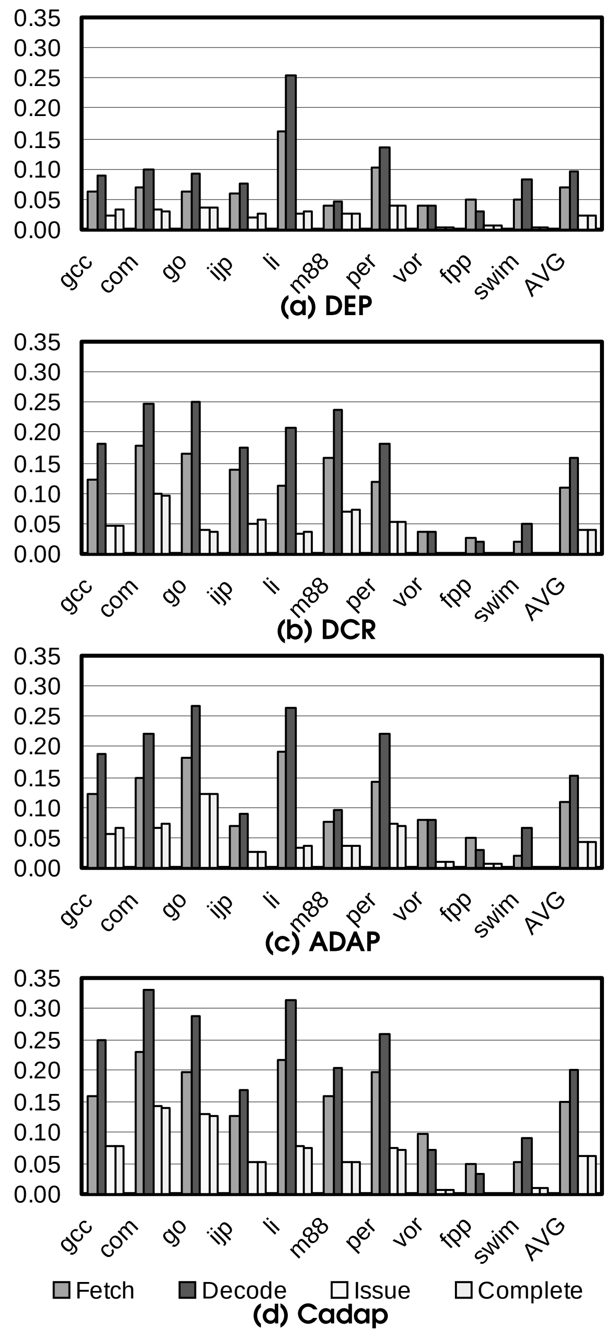

**Figure 5: Traffic reduction for the flow-based methods. Higher is better.**

Jacobsen, Rotenberg and Smith suggested confidence mechanisms for branch predictors [2]. In this work we utilized some of the more accurate ones for the purposes of throttling a highly-aggressive front-end.

The idea of using confidence for speculation control and power reduction is not new. It was introduced by Manne, Klauser and Grunwald [3,4]. H. Sanchez et al., describe a fetch stage throttling mechanism for the G3 and G4 PowerPC processors [5]. They use this throttle mechanism in response to thermal emergencies. This is a fail-safe mechanism. Our throttling mechanisms are fine grain and aim at reducing average power during normal operation.

## 5. CONCLUSION

We have studied various front-end throttling methods including previously proposed methods that exploit branch prediction confidence information. We also proposed methods that exploit information about the instructions flowing through the various pipeline stages. We affirmed that a confidence-based method significantly reduces pipeline traffic with minimal performance impact. We showed that similar benefits are possible with a simple, instruction flow based heuristic. Finally, by combining both confidence- and instruction-flow-based methods, further power benefits were possible while maintaining competitive performance (within 2% of original core).

In this work we limited our attention to reducing power due to control miss-speculation. Our techniques are oblivious to control speculation and it may be possible to use them to reduce power even on correctly speculated paths.

## ACKNOWLEDGMENTS

We thank Farid Najm and the anonymous reviewers for their comments. This work was supported by DARPA under contract F33615-00-C-1631, by funds from the University of Toronto and by research and equipment grants from NSERC.

## REFERENCES

[1] T. Burd and B. Peters. *A Power Analysis of a Microprocessor: A Study of the MIPS R3000 Architecture.* ERL Technical Report, University of California, Berkeley, May 1994.

[2] E. Jacobsen, E. Rotenberg, and J. E. Smith. *Assigning Confidence to Conditional Branch Predictions.* In Proc. Intl. Symposium on Microarchitecture, December 1996.

[3] S. Manne, A. Klauser and D. Grunwald. *Pipeline Gating: Speculation Control For Energy Reduction.* In Proc. Intl. Symposium on Computer Architecture, Jun, 1998.

[4] D. Grunwald, A. Klusser, S. Manne and A. Plezkun, *Confidence Estimation for Speculation Control,* In Proc. Intl. Symposium on Computer Architecture, Jun, 1998.

[5] H. Sanchez *et al. Thermal management systems for high performance PowerPC microprocessors*. In Proc. COMPCON '97, February, 1997.