RC22668 (W0212-062) December 9, 2002
Computer Science

# IBM Research Report

## Energy Efficient Co-Adaptive Instruction Fetch and Issue

**Alper Buyuktosunoglu, [+]Tejas Karkhanis, [*]David Albonesi, Pradip Bose**

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[+]University of Wisconsin
716 Langdon Street
Madison, WI 53706

[*]University of Rochester
Rochester, NY 14627

**Research Division**
**Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Energy Efficient Co-Adaptive Instruction Fetch and Issue

## Abstract

*Front-end instruction delivery accounts for a significant fraction of the energy consumed in a dynamic superscalar processor. The issue queue in these processors serves two crucial roles: it bridges the front and back ends of the processor and serves as the window of instructions for the out-of-order engine. A mismatch between the front end producer rate and back end consumer rate, and between the supplied instruction window from the front end, and the required instruction window to exploit the level of application parallelism, results in additional front-end energy, and increases the issue queue utilization. While the former increases overall processor energy consumption, the latter aggravates the issue queue hot spot problem.*

*We propose a novel, complementary combination of fetch gating and issue queue adaptation to address both of these issues. We introduce an* issue-centric *fetch gating scheme based on issue queue utilization and application parallelism characteristics. Our scheme attempts to provide an instruction window size that matches the current parallelism characteristics of the application while maintaining enough queue entries to avoid back-end starvation. Compared to a conventional fetch gating scheme based on flow-rate matching, we demonstrate 20% better overall energy-delay with a 44% additional reduction in issue queue energy. We identify Icache energy savings as the largest contributor to the overall savings and quantify the sources of savings in this structure. We then couple this issue-driven fetch gating approach with an issue queue adaptation scheme based on queue utilization. While the fetch gating scheme provides a window of issue queue instructions appropriate to the level of program parallelism, the issue queue adaptation approach shuts down the remaining underutilized issue queue entries. Used in tandem, these complementary techniques yield a 20% greater issue queue energy savings than the addition of the savings from each technique applied in isolation. The result of this combined approach is a 6% overall energy-delay savings coupled with a 54% reduction in issue queue energy.*

## 1 Introduction

Power dissipation limits constitute one of the primary design constraints in future processors, both in the embedded and general-purpose, high performance segments [5], [12], [13]. Current generation high-end processors like the dual-core IBM POWER4$^{TM}$ [21] are performance-driven designs where overall power densities are reportedly [4] still below acceptable limits, even though the net chip power is well over 100 watts [10]. However, as reported in [4], localized hotspots in regions like the out-of-order issue queues may experience ungated power densities as high at 70 watts/cm$^2$. These levels would be hard to sustain at the broader chip level without a considerable boost in the packaging/cooling cost of the chip.

The front-end instruction delivery path - consisting of the fetch, decode, rename, dispatch and issue stages - consumes about 20% of the net chip system power and about 35% of each processor core, based on data reported in [4]. Thus, it is necessary to devise microarchitectural techniques that can reduce power and power density without sacrificing performance for high-end systems. In the embedded processor segment, on the other hand, it is often acceptable to implement a energy reduction scheme that significantly improves the energy-delay product, but with a small performance degradation.

As noted in prior work [3], [16], [17], [22], instruction delivery power is higher than necessary because of the performance focused design strategy at the high end. In such designs, the front-end fetch mechanism provides instructions using the peak architected bandwidth, as early as possible, by making use of sophisticated branch prediction algorithms. This strategy often wastes energy because instructions - whether they are on correct or mis-speculated paths - are frequently fetched earlier than necessary. These instructions spend many needless cycles in the issue queue waiting for dependencies to be resolved (or to be aborted following a misprediction event). In particular, we observe from wide-issue superscalar processor simulations that in programs or program phases where instruction level parallelism (ILP) is limited, there is often a significant mismatch between the front-end decode rate and the back-end completion rate, leading to high utilization in the issue queue. Figure 1 depicts an execution profile for the SPEC2000 benchmark gzip assuming an 8-wide superscalar machine with a 32-entry issue queue. The vertical (Z) axis records the percentage of cycles during which x instructions were decoded and y instructions were committed, where the X- and Y-axes record the number of decoded instructions and the number of committed instructions respectively. We see that in 10% of the total execution cycles, there are no instructions committed while the decoding unit is utilized to the full decode bandwidth of eight; and the issue queue utilizations (shown in the right side of Figure 1) are very high, indicating that the front-end is operating faster than necessary to match the commit rate. Also, as described in [4], the instruction fetch unit in a POWER4-like processor is idle during as many as 47% of the execution cycles for TPC-C like commercial workloads.

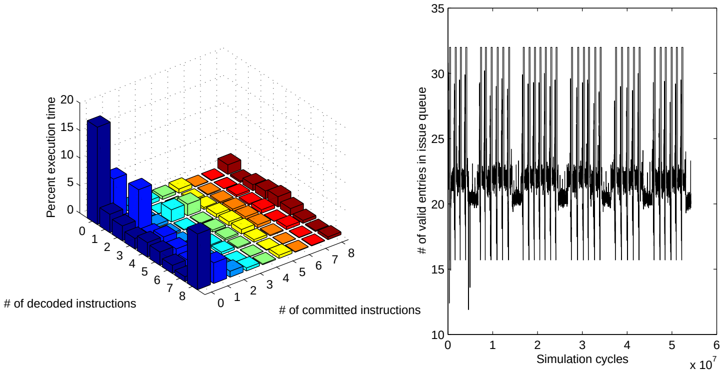


**Figure 1. Simulation statistics for SPEC2000 gzip benchmark**

Of course, much of the idle energy waste in the front-end (including the issue queue) is attributable to incorrect control flow speculations. This can be reduced by investing into more accurate branch prediction schemes or by using confidence estimation as a control mechanism for fetch-gating [17]. However, reducing mis-speculated fetches alone would not necessarily reduce the large component of idle energy that results from earlier-than-needed fetch of instructions in the correct path.

Fetch gating [3], [16] attempts to reduce idle energy by making the fetch mechanism more demand-driven; that is, instruction fetch is throttled (or gated off) when the downstream utilization is high or the flow rate mismatch (between front-end decode and back-end commit) is high. In this context of flow rate matching, the issue queue plays a central role for two reasons. First, it bridges the front and back ends of the processor and second, it serves as the window of instructions for the out-of-order engine. A mismatch between the front-end producer rate and back-end consumer rate, and between the supplied instruction window from the front end and the required instruction window to exploit the level of application parallelism, results in additional front-end energy, and in addition, unnecessarily increases the issue queue utilization. While the former increases overall processor energy consumption, the latter exacerbates the issue queue power-density problem [4]. The goal of any fetch gating mechanism should therefore be to maintain only that number of instructions in the issue queue that matches the window size that can be exploited by the level of application parallelism, without allowing the queue utilization to become so dangerously low that back-end starvation results. For these reasons, information regarding the level of application parallelism and the utilization of the issue queue should drive the fetch gating control mechanism.

Even with such a mechanism in place, the full energy-saving potential of the issue queue cannot be realized without additional techniques. While fetch gating may provide a level of issue queue utilization appropriate for the application, unused entries will still consume energy. Dynamic adaptation of the issue queue [8], [9], [11], [19] is one technique for saving energy in an underutilized issue queue. In this approach, the issue queue is sized to match its level of utilization or the necessary instruction window demanded by the application. Thus, unnecessary entries are shut down, saving considerable energy. The combination of issue-aware fetch gating and dynamic issue queue adaptation, in which the queue is appropriately utilized and unused entries consume negligible energy, has the potential for both good overall chip and issue queue energy savings. However, because issue queue adaptation increases the likelihood of a full issue queue, it stalls the front-end more often, achieving some degree of fetch gating for free. Therefore, it is unclear how the combination of this technique impacts the effectiveness of fetch gating in reducing overall energy-delay.

In this paper, we explore such *issue-centric fetch gating* and investigate the coupling of fetch gating and issue queue adaptation. As such, we make the following contributions:

- We present a new issue-centric fetch gating algorithm that is readily implementable with modest hardware overhead. We demonstrate that issue-centric fetch gating in comparison to a conventional fetch gating scheme based on flow-rate matching achieves 20% better overall energy-delay and 44% additional issue queue energy savings.

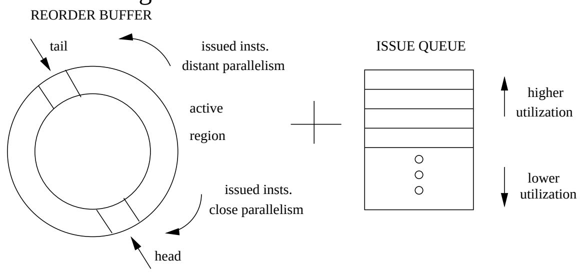
- We provide a detailed breakdown and analysis of the savings due to fetch gating and issue queue adaptation. In doing so, we identify L1 Icache energy savings as a major contributor to the overall energy gains of fetch gating. We quantify the degree to which reduced mis-speculation and clustered access (fetches of larger instruction groups) contribute to this savings, and identify the latter as the larger effect.

- We demonstrate how the combination of fetch gating and issue queue adaptation produces an issue queue energy savings that is much greater than the combination of savings of each technique in isolation. We quantitatively attribute this to the aforementioned way that these techniques synergistically combine to adjust the issue queue utilization to the appropriate level, and to virtually eliminate the energy cost of underutilized entries.

The rest of this paper is organized as follows. In the next section, we present PAUTI, an issue-centric fetch gating algorithm and discuss its implementation complexity. For comparison purposes, we also present a flow-rate-based fetch gating algorithm that performs well yet can be reasonably implemented. In Section 3, we discuss the selection of an appropriate issue queue adaptation algorithm to work in conjunction with these fetch gating schemes. Our evaluation methodology is presented in Section 4 followed by our results in Section 5. Finally, we present related work in Section 6, and conclude in Section 7.

## 2 PAUTI: An Issue-Centric Fetch-Gating Algorithm

Unlike previously published fetch gating approaches [3], [16], our approach is to drive instruction fetch based on issue queue information [1]. Our approach, called PAUTI (**Pa**rallelism and **Uti**lization Based Fetch-Gating), detects mismatches between the size of the instruction window (group of instructions being examined for possible execution) in the queue and the size necessary to match the parallelism characteristics of the program. The former is obtained by tracking the occupancy of the queue, while for the latter, we monitor how deep in the Reorder Buffer (ROB) instructions are being issued from. A general depiction of the scheme as well as possible scenarios are summarized in Figure 2. The parallelism in the program can be close to the head of the ROB (*close parallelism*), or it could be towards the tail of the ROB (*distant parallelism*). Fetching is stopped when there is close parallelism and the issue queue utilization is high.



| parallelism | IQ utilization | fetch gating |
|---|---|---|
| distant | high | X |
| distant | low | X |
| close | high | ✓ |
| close | low | X |

**Figure 2. Overall view of PAUTI fetch gating scheme**

In our particular implementation of PAUTI, fetch gating occurs in a given cycle if over half of the instructions that issue are located in the lower half of the ROB (nearest the head) and the issue queue is at least half full. The rationale for the last restriction (queue being at least half full) is as follows. Many dynamic superscalar processors aggressively release issue queue resources upon instruction issue in order to keep queue sizes modest for timing and power reasons. For instance, the integer issue queue in the Alpha 21264 has only 15 entries despite the processor's aggressive microarchitecture. (The SimpleScalar-based microarchitecture that our simulations are based on requires a 32-entry queue, but this is still fairly modest for an 8-way issue machine.) A queue with such a small number of entries must be managed carefully. For instance, fetch-gating to the point where the queue is much less than half full runs the risk of back-end starvation and produces an extremely small instruction window, especially in the presence of data cache misses. Thus, we do not attempt to fetch-gate if the queue is less than half full. An issue queue that is more than half full should only be gated if distant parallelism is lacking; otherwise, a significant reduction in CPI may result. With such a small queue, we determined that the single criterion described above is sufficient to meet this requirement.

Our implementation, shown in Figure 3, leverages the index number of the ROB entry assigned to instructions each time they are dispatched into the circular ROB. Every cycle, the index numbers of each of the issued instructions (up to eight in our design) are subtracted from the index number of the instruction that is at the head of the ROB and the most significant bit of each result is extracted. Then a ones-count is performed on the eight bit result, yielding the number of instructions issued from the half closer to the head of the ROB. Next, the bits are shifted to the left by one and subtracted from the total number issued. If the final result is negative, it means that more than 50% of the issued instructions came from the half closer to the head of the ROB. In parallel, the issue queue utilization is checked using the scheme described in Figure 3 and a bit is set if the queue is at least half full. An AND of these two outcomes produces the fetch-gate condition for the next cycle.

The above hardware can easily operate within the cycle time of a machine with a 64-bit integer add operation (assumed to have a delay of 16 FO4 – based on a representative current generation superscalar microprocessor [14]). The eight 7-bit subtractors operate in parallel with a delay of about 1/8 that of the 64-bit adder, or 2 FO4. The Ones-count is a 3-to-2 compression widely used in Wallace trees. Compressing from 8 to 4 bits requires 2 levels of 3-to-2 elements, each of which is 2 FO4 (based on communications received from high-end processor designers), producing a total Ones-count delay of 4 FO4. The shift operation simply requires tapping of the wire a bit to the left, or 0 FO4. Finally, the last 4-bit subtractor has a single FO4 gate delay which makes the total for this path 7 FO4. For the other

[1] We focus only on fetch gating based on the integer issue queue for our integer application suite. In general, information from both the integer and floating point issue queues could be used.

parallel path that checks issue queue utilization, the ones-count is a 32 to 6 bit compression which requires 5 levels of 3-to-2 elements, producing a total Ones-count delay of 10 FO4. The ORing of the two most significant bits from the 6 bit number (output of Ones-count block) results in a signal, which if asserted, would mean that the count is greater than or equal to 16. This would indicate utilization of 50% or more in the issue queue. Finally, the last OR bit has a single FO4 gate delay which makes the total delay of this path 11 FO4. Thus, the logic to produce the fetch gate signal with the final AND gate has approximately 12 FO4 delay.

Based on an analysis of the logic structure, we computed the energy overhead of the PAUTI scheme to be at worst, 0.075% of the total chip power. The analysis was derived by counting the number of adder cells and using primitive adder macro power numbers from representative designs [10].

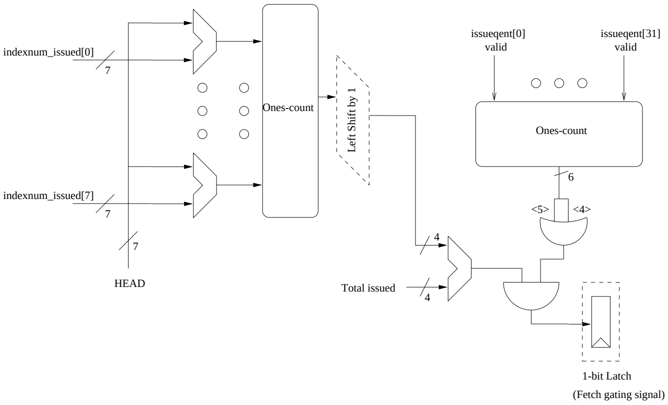


**Figure 3. PAUTI hardware implementation**

## 2.1 Decode/Commit Rate (DCR) Fetch-Gating

For comparison purposes, we investigated a wide variety of alternative previously proposed fetch gating schemes [3], [16], with a dual emphasis on performance and compactness of hardware implementation. We chose a variation of one of the best-performing schemes that we determined could be reasonably implemented. The scheme, called Decode/Commit Rate (DCR) fetch gating, is an instruction flow-rate based approach that attempts to match the bandwidths of the front and back-ends. One such DCR approach by Baniasadi and Moshovos [3] uses instruction flow information to estimate the amount of instruction-level parallelism (ILP) currently present in the machine. If introducing additional instructions will not increase ILP, fetch and decode are stalled. In one of their approaches, when the number of instructions passing through decode significantly exceeds the number of instructions that commit, fetch and decode are disabled for 3 cycles.

Our DCR approach differs from [3] in that we do not delay the execution of an instruction by stalling it in the decode stage. As shown in Figure 4, we simply observe the number of committed instructions versus decoded instructions on a cycle-by-cycle basis and compare these to determine the fetch-gating condition. If the condition is met, no fetching is performed in the next cycle.

We obtained the best results when we fetch-gated whenever the number of decoded instructions exceeded the number of committed instructions in a given cycle. The intuition behind this approach is as follows. In a machine with perfect branch prediction, the front and back-end flow rates should be identical over the long run. A lower front-end flow rate will compromise performance, while an excessive front-end flow rate will needlessly waste energy as instructions reside in the issue queue for more cycles than necessary. As application parallelism characteristics change, so does the commit rate and the decode rate is compensated appropriately. Performing comparisons at a cycle-by-cycle granularity tends to balance out the rates over the long run, despite the time dilation between commit and decode.

With imperfect branch prediction, more instructions are executed than committed. Thus, matching the instruction decode and commit rates has the effect of throttling the front-end with respect to instruction execution. This has the following effects [17]:

- Fewer mis-speculated instructions are executed, which saves energy and also increases performance by reducing the number of branch mispredicts (discussed in more detail in Section 5).

- The instruction cache warmup benefits of executing mis-speculated instructions are not fully realized, decreasing performance.

- The fetching of correctly speculated instructions is delayed, also decreasing performance.

Application phases with high branch mispredict rates (and thus many more fetched/executed than committed instructions) will experience greater fetch gating with respect to execution. The resulting benefits of executing fewer mis-speculated instructions greatly overrides the other effect of fetching fewer correct-path instructions in a timely manner. Phases with low mis-predict rates will have little fetch gating, which is beneficial overall since most of the instructions are correct-path instructions. The throttling rate automatically adjusts to the mispredict rate. The net effect is a significant reduction in energy with only a small performance impact (performance increase in some applications) over a range of branch mis-predict rates with this simple DCR fetch-gating mechanism.

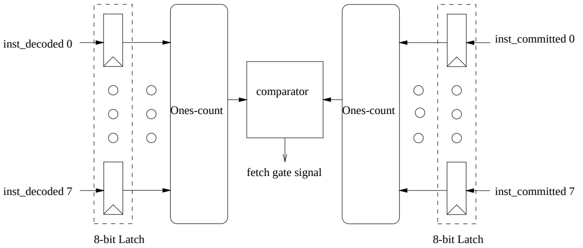


**Figure 4. DCR hardware implementation**

## 3 Fetch Gating Combined with Dynamic Issue Queue Adaptation

In combining fetch gating and issue queue adaptation, we need to select an appropriate issue queue adaptation algorithm, with the two general alternatives being to adapt based on issue queue utilization [8], [9], [19] and based on application ILP characteristics [8], [11]. Recall that PAUTI adjusts the level of instruction flow in order to build the appropriate window of instructions in the queue. Thus, unused entries based on the metric of issue queue utilization, should be shut down by the adaptive algorithm. Similarly, because DCR adjusts the front-end flow rate to match the back-end commit rate, the issue queue should be sized based on the number of entries needed to accommodate those rates, based again on utilization information.

Thus, we implemented a variation of the utilization-driven approach of [9]. In this scheme, the issue queue is broken down into eight-entry *chunks*, each of which can be disabled on-the-fly at runtime. A hardware-based monitor measures issue queue utilization over a *cycle window* period by counting the number of valid entries in the queue, after which the appropriate control signals disable and enable queue chunks [8]. The approach in [9] resizes the issue queue at chunk granularity based on the utilization of the issue. The algorithm that we use for our 32-entry issue queue is a simpler version of [9] and is as follows:

```
{utilization-based algorithm}
        if (valid_entries <= threshold1)
                issue_queue_size=8;
        else if (valid_entries <= threshold2)
                issue_queue_size=16;
        else if (valid_entries <= threshold3)
                issue_queue_size=24;
        else
                issue_queue_size=32;
```

Here, at the end of every cycle window we determine the appropriate issue queue size for the next interval based on the number of valid entries relative to pre-set threshold values. A range of threshold values were investigated to find the best values for each approach.

| | |
|---|---|
| Fetch unit | up to 8 instr. per cycle<br>max 2 branches per cycle<br>16 entry fetch buffer |
| Branch predictor | comb of bimodal and 2-level global<br>chooser size: 1024 entries<br>bimodal size: 4096 entries<br>2 level size: 4096 second level table entries<br>12 bits history width |
| Branch target buffer | 8192 sets, 4-way |
| Branch mispredict penalty | 8 cycles |
| Dispatch, issue, commit width | 8 instructions |
| ROB and Ld/St queue size | 128 and 32 |
| Issue queue size | 32,16 (int and fp, each) |
| Integer ALUs/Multipliers | 4/4 |
| Flt Pt ALUs/Multipliers | 2/2 |
| Memory Ports | 4 |
| L1 Icache, Dcache | 64KB 2-way, 32-byte lines, 2 cycles |
| L2 unified cache | 2MB 4-way, 64-byte lines, 15 cycles |
| TLB | 128 entries, 8KB page size |
| Memory latency | 75 cycles for the first chunk |

**Table 1. SimpleScalar simulator parameters**

## 4 Evaluation Methodology

For microarchitectural simulations, we use the SimpleScalar toolset [7] with the Wattch power extensions [6] to simulate an aggressive 8-way superscalar out-of-order processor. The simulation parameters are summarized in Table 1. The simulator has been modified to model separate integer and floating point queues. In Wattch, we use the activity-sensitive power model with aggressive conditional clocking (cc3 mode). The rationale for this choice is to compare our fetch gating plus the adaptive issue queue schemes to a baseline that is already power efficient.

We use a number of integer benchmarks from the SPEC95 and SPEC2000 suites with different data cache miss rates and branch predictor accuracy. Warmup times are determined for each benchmark, and the simulation is fast-forwarded through these phases. Table 2 summarizes the benchmarks and their L1 data cache miss rate, branch prediction accuracy, and CPI (cycles per instruction) and EPI (energy per instruction) values with the default simulation parameters. We focused on the integer issue queue and consequently SPECint codes, because that particular queue is a known hot spot identified in prior published work [4]. However, for floating point codes with high branch prediction rates, although branch-related stalls would be rarer, high cache miss rates and dependence-related stalls also cause non-uniform behavior that our approach can exploit.

Table 3 shows the fetch-gating, adaptive issue queue, and combined schemes that were simulated. The adaptive queue schemes both use an 8K cycle window but different thresh-

| Benchmark | Insts simulated | L1D miss rate (%) | Bpred accuracy (%) | Base CPI | Base EPI (x E-8) |
|---|---|---|---|---|---|
| compress (SPEC95int) | 2000M-2100M | 17.7 | 91.0 | 0.68 | 2.29 |
| go (SPEC95int) | 926M-1026M | 0.4 | 81.2 | 0.83 | 2.37 |
| li (SPEC95int) | 271M-371M | 0.8 | 91.9 | 0.61 | 2.16 |
| bzip2 (SPEC00int) | 1200M-1400M | 4.1 | 89.5 | 0.55 | 1.89 |
| gzip (SPEC00int) | 2000M-2100M | 1.9 | 90.4 | 0.54 | 1.86 |
| mcf (SPEC00int) | 1000M-1050M | 25.3 | 91.5 | 1.87 | 3.64 |
| parser (SPEC00int) | 1000M-1100M | 3.9 | 94.0 | 0.74 | 2.21 |
| vpr (SPEC00int) | 1000M-1100M | 0.8 | 95.0 | 0.46 | 1.70 |

**Table 2. Benchmark description, L1 D-cache miss rates, branch prediction accuracy, CPI, and EPI for the baseline configuration (without fetch gating or dynamic adaptation of issue queue)**

| | |
|---|---|
| ADQI | Adaptive issue queue - thresholds 6,13,21 |
| ADQII | Adaptive issue queue - thresholds 4,10,19 |
| DCR | Decode/Commit Rate fetch-gating |
| PAUTI | Parallelism+Utilization fetch-gating |
| DCR+ADQI | Decode/Commit rate based fetch gating with adaptive issue queue scheme I |
| PAUTI+ADQI | Parallelism+Utilization based fetch gating with adaptive issue queue scheme I |
| DCR+ADQII | Decode/Commit rate based fetch gating with adaptive issue queue scheme II |
| PAUTI+ADQII | Parallelism+Utilization based fetch gating with adaptive issue queue scheme II |

**Table 3. Fetch-gating, adaptive, and combined configurations**

old values. The ADQI scheme with larger threshold values favors smaller queue sizes (higher energy savings) while the ADQII scheme would tend to reduce the performance degradation. Note also that with the ADQI scheme there will be more instruction fetch stalls compared to ADQII due to the issue queue getting full more often.

## 5 Results

Figure 5 shows the CPI degradation, overall chip energy-delay product improvement, and the issue queue energy savings achieved by the various schemes relative to the baseline with no fetch-gating or issue queue adaptation.

First, we observe that the CPI degradation for all of the schemes is small (less than 2% in most cases). In fact, there are several notable performance improvements with some benchmarks with either fetch-gating or the combined scheme. We attribute this to the reduction in number of branch mispredicts (3.8% and 4.9% reduction on average for DCR and PAUTI, respectively) for those benchmarks compared to the baseline. Without fetch gating or issue queue adaptation, for the benchmarks that have small basic block sizes, it is likely that a less fresh (old instance) branch predictor table entry for a specific branch is accessed for subsequent branch accesses. On the other hand, with fetch gating, before subsequent branches access the branch predictor table (delayed due to fetch gating), the previous branches may have already resolved providing a more fresh copy (newer instance). The degree to which this greater prediction accuracy trades off with the added delay of fetch gating or dynamic adaptation determines whether there is a net performance benefit or degradation.

In comparing fetch gating with issue queue adaptation, we observe that the former has a much greater energy-delay impact, achieving roughly a 6% reduction versus 1% for issue queue adaptation. The breakdown of the energy results (Figure 6) shows that fetch gating has a significant impact on reducing the energy in many different units as opposed to the isolated impact of issue queue adaptation. In particular, an overall 32-36% (DCR-PAUTI) savings in Icache energy, 5-6% savings in rename, 6-7% in clock, and 11-15% savings in issue queue energy is achieved with fetch gating. Although some units experience an increase in energy (for instance, the register file energy increases due to less operand bypassing), these increases are small compared to the savings achieved in other power-density-sensitive areas such as the issue queue.

The significant savings in Icache energy with fetch gating is attributable to two effects. First, fewer mis-speculated instructions are fetched. Second, as is shown in Figure 7, with fetch gating, each Icache access fetches a larger group of instructions on average than with the baseline. Thus, the energy cost of an Icache fetch is amortized over this larger instruction group. The reason for this behavior is that the number of available entries in the fetch buffer determines the amount of instructions that can be fetched. A fetch gating event often causes the fetch buffer to be drained, leaving it able to accommodate a full fetch fairly often. With the baseline, an Icache fetch occurs so long as there are avail-

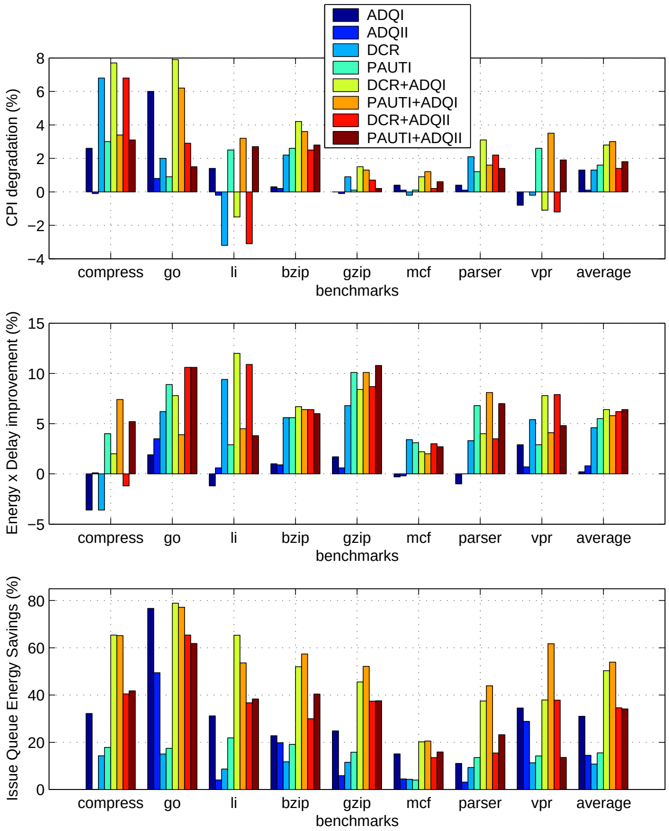


**Figure 5. CPI degradation, energy-delay product improvement, and issue queue energy savings for the individual and combined approaches**

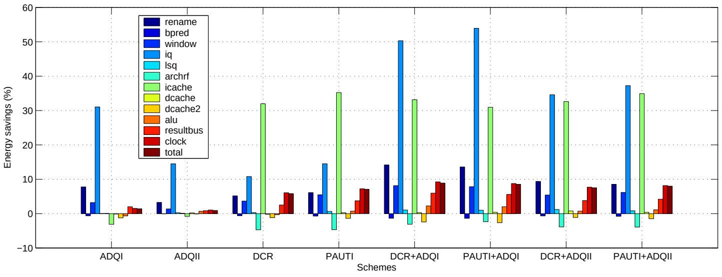


**Figure 6. Energy savings in various units averaged across all benchmarks. rename: rename logic; bpred: combined branch predictor and branch target buffer; window: ROB and physical register files; iq: issue queue; lsq: load-store queue; archrf: architectural register file. icache: instruction cache; dcache: level1 data cache; dcache2: level2 data cache; clock: global clock distribution and local clock units in each macro**

able fetch buffer slots, resulting in more instances in which a subset of the full fetch bandwidth is used. (Performance was insensitive to increasing fetch buffer size beyond 16, the size used in this analysis.) As is shown in Figure 8, on average roughly 40% of the Icache energy savings is due to reduced mis-speculation, while the larger remaining portion is due to more clustered accesses.

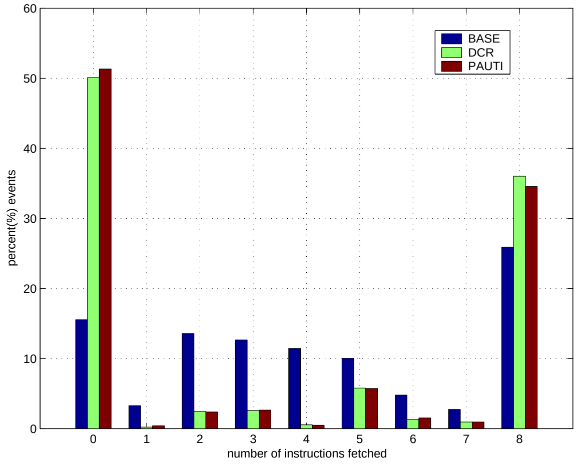


**Figure 7. Breakdown of number of instructions accessed for each fetch operation (averaged across all benchmarks)**

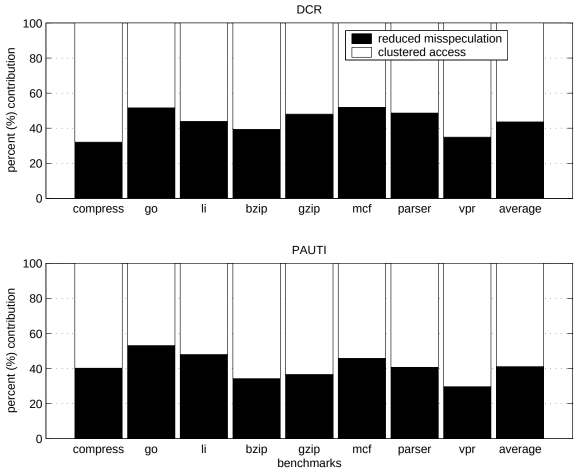


**Figure 8. Contribution of reduced mis-speculation and clustered accesses to overall Icache energy savings for each benchmark and overall average for DCR and PAUTI**

In comparing the two fetch-gating approaches (Figure 5), our issue-centric approach to fetch gating yields significant benefits. PAUTI achieves a 20% greater reduction in energy-delay, and a 44% greater reduction in issue queue energy, than DCR. The energy-delay gain is largely due to the 23% greater overall energy savings achieved with PAUTI. Much of the savings with DCR is attributable to li, which experiences a CPI improvement over the baseline due to the aforementioned branch prediction effects. PAUTI provides much more consistent results, outperforming DCR in energy-delay on most of the benchmarks.

Figure 9 shows how DCR, and to a greater degree PAUTI, reduces the utilization of the issue queue (in terms of four ranges of occupied entries) versus the baseline. For instance, overall, over 50% of the time the baseline is near full (25-32 entries) while this occurs roughly 33% of the time for DCR and 20% of the time for PAUTI. For compress, go, gzip, and parser PAUTI outperforms DCR (Figure 5(a)) yet uses fewer entries.

We now investigate the combination of fetch gating and dynamic adaptation. As is shown in Figure 5, the additional fetch stalls introduced with dynamic adaptation slightly increases the performance degradation with the combined approach. However, the combined approach achieves a significant reduction in issue queue energy yet with a slightly better overall energy-delay than fetch-gating alone. To provide more insight on the interaction of these approaches, Figure 10 shows the time spent in each of the four configurations for the adaptive issue queue scheme ADQI with and without the use of fetch gating for each of the benchmarks. With the high utilization of the baseline scheme (Figure 9), the adaptive scheme is only able to downsize to 24 entries in most cases, and in many cases, not downsize at all. The reduction in issue queue utilization enabled by fetch gating allows for more aggressive downsizing, and much greater energy savings. For parser for instance roughly 70% the queue is at its full size using ADQI alone. With DCR, this is reduced to less than 20%, and in addition, the queue can be downsized to 16 entries almost 20% of the time. With PAUTI, the full queue is almost never used. Figure 6 shows the magnitude of the issue queue energy savings with the combined approach as compared to either issue queue adaptation or fetch gating alone. The issue queue energy savings achieved with the combined approach is greater than the combined savings of each technique in isolation. For example, ADQI achieves a 31% reduction in issue queue energy, while the reduction with PAUTI is 14%. Combining these approaches results in a 54% reduction in issue queue energy, a 20% greater savings than expected from the results of each individual technique. This is due to the complementary aspects of the two approaches as discussed above and in Section 1. Thus, the combination of issue-centric fetch gating and dynamic issue queue adaptation not only provides good overall energy-delay savings, but significantly alleviates the issue queue power density problem.

## 6 Related Work

Prior related work can be divided in two groups: in the area of fetch-gating and in dynamically adaptable issue queue designs.

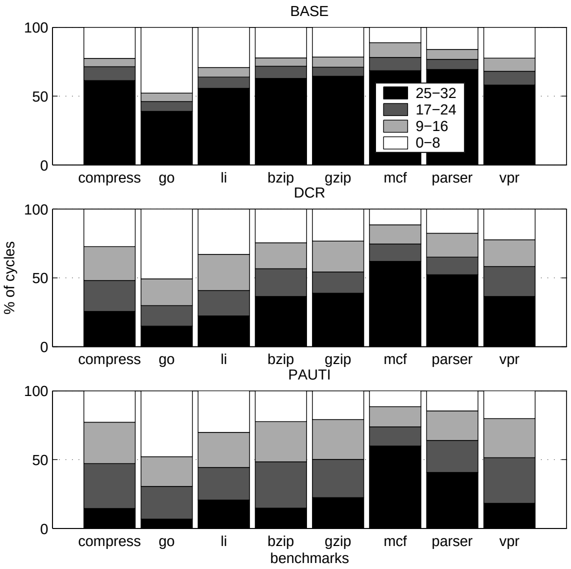


**Figure 9. Percent of cycles that the issue queue has 0-8, 9-16, 17-24, and 25-32 entries occupied for the baseline, with DCR, and with PAUTI**

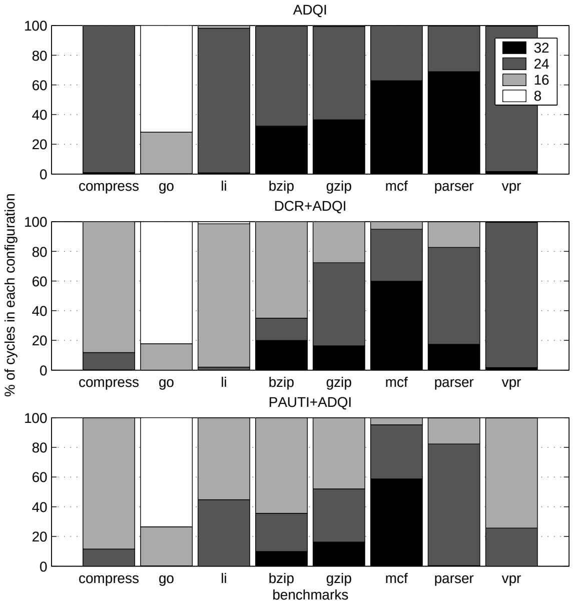


**Figure 10. Percent of cycles in each configuration of the dynamic adaptive issue queue with different schemes**

In [17] the authors save the wasted energy used for fetching, decoding, issuing, and executing later-to-be-flushed instructions. They estimate the confidence of every branch prediction when that branch is fetched [15]. When a certain number of low confidence branches enter the processor the fetch unit is gated-off. Fetching resumes when the number of low-confidence branches within the processor falls below the gate-off threshold. Their scheme is limited to short pipelines. Increasing the pipeline depth increases the penalty for incorrect confidence estimation. In particular, if a branch is estimated as a low-confidence branch but it is predicted correctly, the execution is starved of useful instructions. This limitation was pointed out by authors in [17]. Furthermore, the confidence estimator itself consumes energy. We focus on the out-of-order behavior of applications by observing key instruction issue characteristics and fetch gating accordingly.

In [16] the authors dynamically change the number of inflight instructions. An instruction count register keeps track of the number of in-flight instructions, and is incremented when an instruction is fetched and decremented when an instruction is committed. Another register, MAXcount, sets the limit on the allowable in-flight instructions. Whenever the instruction count exceeds MAXcount instruction fetching is stopped. The algorithm searches for the "optimal" number of in-flight instructions and changes the value of MAXcount at intervals of 100K committed instructions.

In [22] the authors develop a compiler-driven static IPC estimation scheme that is based on dependence testing in compiler backend. This estimation is used to drive fine-grained fetch-throttling energy saving heuristics. However, dynamic factors such as cache misses and branch mispredictions can dilute the efficiency of these static IPC-estimation-based heuristics.

The PowerPC G3 and G4 microprocessors include a Thermal Assist Unit (TAU) to provide dynamic thermal management. In these systems, the TAU invokes a form of instruction throttling, namely instruction flow reduction to lower the temperature, based on a programmable temperature threshold [20].

Dynamic adaptation of the issue queue size to match application demands is proposed in [1, 2] in order to increase performance and reduce power dissipation. However, it was assumed that the best issue queue size for a given application was known a priori; no attempt was made to adapt within an individual application, and the circuit-level design issues associated with an adaptive issue queue were not addressed in detail. In [11] the issue queue is designed to be a circular queue structure with head and tail pointers, and its effective size is also dynamically adapted to fit the ILP content of the workload during different periods of execution. However, a detailed circuit-level design and simulation of the proposed implementations are not reported. Without such an analysis, it is difficult to assess whether the cycle time overhead of the extra circuitry required for dynamic adaptation, or its power dissipation overhead, are overridden by the power dissipation benefits of these approaches.

In [8] the authors re-size the issue window based on its utilization. The objective is to reduce the number of instructions needlessly waiting in the issue queue. The issue queue is monitored every cycle to measure the utilization. If the utilization is below a certain threshold the queue is sized down for the next interval. If the utilization is above the size-up threshold then it is sized up for the next interval. Also, if the performance after sizing down the queue is lower than a factor of the performance before resizing it then it is sized up again. The re-sizing is done at 8K cycle intervals. Similarly, Marculescu proposes a mechanism to dynamically adapt the fetch and execution bandwidth based on profiling at the basic-block level [18], while Ponomarev et al. propose dynamic allocation of multiple datapath resources, including the issue queue, for low-power [19].

None of these prior approaches combine fetch gating with re-sizing of the issue queue. We have shown that combining an implementable, issue-centric fetch gating algorithm with dynamic issue queue adaptation yields much larger improvements than intuitively obvious.

## 7 Conclusions

The criticality of the issue queue in bridging front and back end flow and in enabling out-of-order issue calls for new fetch gating schemes that take this important resource into account. In this paper, we propose PAUTI, a fetch gating scheme that attempts to match the size of the instruction window resident in the issue queue to application ILP characteristics, while keeping the utilization of the queue high enough to avoid back-end starvation. In comparing this issue-centric scheme with a common flow-rate-matching approach, we determine that it achieves 20% better overall energy-delay and 44% additional issue queue energy savings.

We further identify the Icache as the greatest source of energy savings with fetch gating. We determine that although the avoidance of fetching mis-speculated instructions produces a significant portion of this savings, a greater source is the more efficient usage of the full Icache bandwidth due to increased fetch buffer slots.

Finally, we explore the combination of issue queue adaptation and fetch gating. We determine that the complementary aspects of these approaches yields a 20% greater reduction in issue queue energy than the sum of the individual savings from each scheme. The result of this combination is good overall energy-delay savings with a significant alleviation of the issue queue hot-spot problem.

## References

[1] D. H. Albonesi. Dynamic IPC/Clock Rate Optimization. In *Proceedings of the 25th International Sympo-*

*sium on Computer Architecture*, pages 282–292, June 1998.

[2] D. H. Albonesi. The Inherent Energy Efficiency of Complexity-Adaptive Processors. In *Proceedings of ISCA Workshop on Power-Driven Microarchitecture*, June 1998.

[3] A. Baniasadi and A. Moshovos. Instruction Flow-Based Front End Throttling for Power-Aware High-Performance Processors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 2001.

[4] P. Bose, D. M. Brooks, A. Buyuktosunoglu, P. W. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, S. E. Schuster, J. E. Smith, V. Srinivasan, V. Zyuban, D. H. Albonesi, and S. Dwarkadas. Early-Stage Definition of LPX: A Low Power Issue-Execute Processor Prototype. In *Proceedings of HPCA Workshop on Power-Aware Computer Systems*, February 2002.

[5] D. M. Brooks and M. Martonosi. Dynamic Thermal Management for High Performance Microprocessors. In *Proceedings of 7th International Symposium on High-Performance Computer Architecture*, January 2001.

[6] D. M. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimization. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[7] D. Burger and T. Austin. The SimpleScalar toolset, version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, 1997.

[8] A. Buyuktosunoglu, D. H. Albonesi, S. E. Schuster, D. M. Brooks, P. Bose, and P. W. Cook. *Power Aware Computing*, chapter Power Efficient Issue Queue Design, pages 37–60. Kluwer Academic Publishers, 2002.

[9] A. Buyuktosunoglu, S. E. Schuster, D. M. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In *Proceedings of 11th Great Lakes Symposium on VLSI*, pages 73–78, March 2001.

[10] C. J. Anderson et al. Physical Design of a Fourth-Generation POWER Ghz microprocessor. In *IEEE International Solid-State Circuits Conference- Digest of Technical Papers*, pages 232–233, February 2001.

[11] D. Folegnani and A. Gonzalez. Energy-Effective Issue Logic. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 230–239, June 2001.

[12] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. *Intel Technology Journal*, March 2001.

[13] T. Halfhill. Transmeta Breaks x86 Low Power Barrier. *Microprocessor Report*, 14(2):1–19, February 2000.

[14] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays. In *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[15] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of 29th International Symposium on Microarchitecture*, pages 142–152, December 1996.

[16] T. Karkhanis, P. Bose, and J. E. Smith. Saving Energy with Just in Time Instruction Delivery. In *Proceedings of the International Symposium on Low Power Electronics and Design*, August 2002.

[17] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 132–141, June/July 1998.

[18] D. Marculescu. Profile-Driven Code Execution for Low Power Dissipation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 253–255, July 2000.

[19] D. Ponomarev, G. Kucuk, and K. Ghose. Dynamic Allocation of Datapath Resources for Low Power. In *Proceedings of 34th International Symposium on Microarchitecture*, pages 90–102, December 2001.

[20] H. Sanchez. Thermal Management System for High Performance PowerPC Microprocessors. In *Digest of Technical Papers IEEE COMPCON*, 1997.

[21] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal Research and Development*, 46(1):5–27, January 2002.

[22] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-Fetch: Compiler-Enabled Power-Aware Fetch Throttling. *IEEE Computer Architecture Letters*, 1, April 2002.