Saving Energy with Just In Time Instruction Delivery

Tejas Karkhanis
Univ. of Wisconsin - Madison
1415 Engineering Drive
Madison, WI 53706
1+608-265-3826
karkhani@ece.wisc.edu

James E Smith
Univ. of Wisconsin - Madison
1415 Engineering Drive
Madison, WI 53706
1+608-265-5737
jes@ece.wisc.edu

Pradip Bose
IBM T J Watson Research Center
PO Box 218
Yorktown Heights, NY 10598
1+914-945-3478
pbose@us.ibm.com

ABSTRACT

Just-In-Time instruction delivery is a general method for saving energy in a microprocessor by dynamically limiting the number of in-flight instructions. The goal is to save energy by 1) fetching valid instructions no sooner than necessary, avoiding cycles stalled in the pipeline -- especially the issue queue, and 2) reducing the number of fetches and subsequent processing of mis-speculated instructions. A simple algorithm monitors performance and adjusts the maximum number of in-flight instructions at fairly long intervals, 100K instructions in this study. The proposed JIT instruction delivery scheme provides the combined benefits of more targeted schemes proposed previously. With only a 3% performance degradation, energy savings in the fetch, decode pipe, and issue queue are 10%, 12%, and 40%, respectively.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles – *adaptable architectures, pipeline processors.*

General Terms

Performance, Design

Keywords

Low-power, adaptive processor, instruction delivery

1. INTRODUCTION

Instruction delivery – fetch, decode, renaming, dispatch, and issue – account for a significant proportion of energy consumed in a superscalar microprocessor. For example, instruction delivery in the Alpha 21264[9] accounts for 25.5% of the total energy. Instruction delivery energy consumption is higher than necessary, however, because of the performance-driven design philosophy that is typically followed. In particular, a conventional superscalar processor attempts to maximize the number of "in-flight" instructions at all times. Following a branch misprediction, it begins fetching at full speed and continues until the next branch misprediction flushes the pipeline *or* until the issue queue (or re-order buffer) fills, the decode pipeline backs up, and instruction fetching begins to stall.

This philosophy often wastes energy because 1) useful instructions are fetched earlier than needed, then spend many cycles stalled in the decode pipeline and/or sitting in the issue queue waiting for operands, and 2) when a branch misprediction occurs, all the speculative instructions following the mispredicted branch in the issue queue and decode pipeline are flushed [7].

1.1 Just-In-Time (JIT) Instruction Delivery

We propose a simple, unified scheme for saving energy in the entire instruction delivery subsystem. This scheme monitors and dynamically adjusts the maximum number of in-flight instructions in the processor. The maximum number is determined by monitoring processor performance and is adjusted to the lowest number that does not reduce performance significantly. When the maximum number of in-flight instructions is reached, instruction fetching is inhibited. Often this occurs well before all pipeline stages and issue window slots are full. In effect, instructions are fetched *just-in-time* so performance is relatively unchanged, but fewer instruction delivery resources consume energy with stalled and/or flushed instructions. Overall, the resulting scheme works better than other previously proposed, more targeted approaches.

1.2 Prior Approaches

Several studies have focused on reducing energy in the instruction delivery portion of the microprocessor. Pipeline Gating [7] attempts to reduce flushed (mis-speculated) instructions by inhibiting instruction fetching when the number of *low-confidence* branch predictions exceeds a certain level. In a realistically modeled superscalar pipeline, our approach performs better at reducing energy due to flushed instructions.

Buyuktosunoglu, et al. [2] and Folegnani and Gonzalez [1] attempt to reduce energy by resizing the issue queue. The objective to reduce the number of instructions stalled in the issue queue, a high energy consumer. Again, our JIT scheme does better in terms of reduction of energy consumed by the issue queue.

In [5] Banasadi, et al. attempt to save energy by gating the decode pipeline when the number of instructions to be decoded is fewer than the decode width. They do so by delaying the execution of instructions and managing the pipeline at single instruction

ISLPED'02, August 12-14, 2002, Monterey, California, USA.

granularity. We perform no direct comparison with this method; we assume instructions flow up the pipeline in coarser granularity groups as would be done in a real microprocessor (see section 4.1).

2. QUANTIFYING ENERGY ACTIVITY

We consider instruction delivery to be composed of three major parts: 1) instruction cache access, 2) instruction decoding, renaming, and dispatching into the issue queue, and 3) the issuing from the issue queue. For brevity, we refer to the entire decode/rename/dispatch portion simply as "instruction decode".

We focus on five types of instruction delivery activities. Energy is directly related to these activities.

- ***I fetch:*** *an instruction cache access.*
- ***Decode pipe active:*** *a valid instruction is in the decode pipeline (decode, rename, dispatch), is being processed, and is moving to the next pipeline stage on the following cycle.*
- ***Decode pipe stall:*** *a valid instruction is in the decode pipeline, but is being held (stalled) this cycle; it does not move to the next pipeline stage the following cycle.*
- ***Issue queue active:*** *a valid instruction is in the issue queue and is issuing for execution this cycle.*
- ***Issue queue stall:*** *a valid instruction is in the issue queue, but is not issuing this cycle, for example because its operands are not available or a required resource is not available.*

We then divide each of the above five activities into two groups – one for instructions that eventually commit (*Used*) and the other for mis-speculated instructions (*Flushed*). For example, **I**ssue-queue **S**talled **U**sed is the activity for the instructions that stall in the issue queue and eventually commit.

This breakdown of energy-consuming activity allows for a form of clock gating where active instructions may consume more energy than stalled instructions, and where valid instructions may consume more energy than invalid ones (i.e. empty pipeline slots). For example, consider the logic shown in Figure 1. Here, a typical pipeline latch is shown, as might appear in the decode pipeline. An input multiplexor (typically built into the latch) is used to "recirculate" latched pipeline values when the hold signal is active. In addition, the valid bit from the preceding stage is used to gate the latch itself; if there is no valid data being fed into the latch, then the latch is not clocked. In this system, a certain amount of energy is consumed if an instruction moves up the pipeline (the hold signal is inactive) and is latched into the next stage. A different (lower) amount is consumed if the hold signal is active, the multiplexor feeds the same data back into the latch and the latched is clocked, but the logic following the latch does not see any of its inputs change. Finally, a different (still lower) amount of energy is consumed if the valid signal is off, and the latch is not clocked at all. Similarly, in the issue queue, a particular issue queue slot may consume different amounts of energy depending on whether or not it holds an active instruction and whether or not the instruction actually issues.

The activities given above can be used to compute overalldynamic energy consumption, given the amount of energy per activity. For most of this paper, we focus on the activity counts,

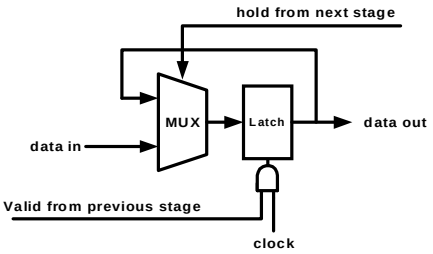

Figure 1: A pipeline latch. A valid bit from the previous stage is used to gate the clock signal. A hold signal from the succeeding stage is used to switch the multiplexcr and recirculate data being stalled.

rather than energy numbers, to reduce the dependence of results on specific circuit and logic design styles[1]. In the penultimate section, however, we give energy estimates for a particular state of the art circuit/logic design technology.

3. IMPLEMENTING JIT INSTRUCTION DELIVERY

The method we propose is illustrated in Figure 2. The total number of *in-flight* instructions is kept in register *instruction count.* For each instruction fetched, *instruction count* is incremented; for each instruction that is either committed or squashed, it is decremented. There is also an adjustable *MAXcount* , and instruction fetching is inhibited whenever *MAXcount* is exceeded by *instruction count*.

To dynamically adjust the *MAXcount* value, we use an algorithm that is very similar to one previously proposed for finding optimal cache sizes in [8]. This algorithm is implemented either in hardware or low-level software, and "tunes" for the least value of *MAXcount* such that performance is not reduced by some threshold amount, e.g. 2%. For brevity, we only summarize the algorithm.

First, the algorithm uses a small number of counters to moni-

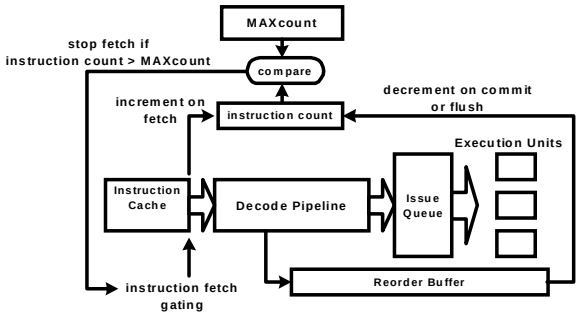

Figure 2: Pipeline with control logic to dynamically limit the number of in-flight instructions.

[1] This is not unlike giving microarchitecture performance in terms of instructions per cycle (IPC) rather than instructions per second, which would require estimation of the exact cycle time.

tor performance characteristics. There is a processor cycle counter and a committed instruction counter, incremented for each committing instruction. By reading and clearing these counters at fixed intervals (e.g. 100K instructions) overall performance (instructions per cycle) can be determined. Also, there is a counter for the number of branch instructions executed during an interval; this counter is used to detect the occurrence of program phase changes [8].

To perform dynamic tuning, *MAXcount* is set to the maximum possible number of in-flight instructions (e.g. 64) and the performance is recorded after one 100K interval. In the following interval, *MAXcount* is set to the minimum in-flight instructions (e.g. 8). In subsequent intervals *MAXcount* is incremented by eight until the performance for an interval is within a *threshold* value (e.g. 2%) of the performance for the maximum *MAXcount;* this process is called a *tuning cycle*. *MAXcount* is kept at this "optimal" value until either the performance (IPC) or the number of dynamic branches changes by more than some "noise" margin; in practice this is often hundreds of 100K instruction intervals. If a tuning cycle results in no change in the optimal *MAXcount*, then the IPC and branch noise levels are increased to prevent unnecessary tunings. The tuning algorithm itself has an overhead of at most a few tens of cycles every 100K instructions and has minimal performance impact [8].

4. EVALUATION METHODOLOGY

This section presents the methodology used for evaluating the performance of the proposed JIT instruction delivery scheme. First, the simulation model is described, then the spectrum of simulated schemes and the workload. Finally, metrics used to evaluate effectiveness of various methods are defined.

4.1 Simulation Model

To evaluate performance, we used a modified version of the SimpleScalar simulator [4]. Modifications are intended to model the instruction delivery system in more detail and with greater accuracy than is done in baseline SimpleScalar. In particular, the issue queue is modeled as a separate structure from the re-order buffer. The issue queue issues out-of-order at most four instructions every cycle. Also, each stage in the instruction decode pipeline is modeled and counters are provided to count the number of instructions in each stage, whether useful or later-to-be-flushed. Finally, the instructions in the pipeline are segmented at granularity equal to the pipeline width. That is, the fetched groups of instructions that enter the pipeline together can only move from one pipeline level to the next as a unit. This is in contrast to the SimpleScalar method of modeling the entire pipeline as one large queue with single-instruction granularity. In effect, the single instruction granularity would require a complex switching network connecting all the instruction slots within and between successive pipe stages..

Our simulation model also differs in other ways from Pipeline Gating (PG) [7] and the Adaptive Issue Queue (AIQ) [2]. In [7], the authors use four stages before the issue stage; we assume a more realistic and slightly deeper pipeline of five stages. Thus the branch misprediction penalty as well as the penalty for incorrect confidence estimation increases. In [2], the authors use fetch/decode width of 16 instructions and issue width of 8 instructions. We use more conservative fetch/decode/issue widths of four because performance benefits significantly diminish when going beyond four, and we believe four is a more realistic number if power efficiency is a major design consideration.

Table 1 summarizes the processor parameters used in all simulations.

Table 1: Processor Configuration

ROB size	64 entries
Issue Queue Size	32 entries
LSQ size	32 entries
IF, ID, IS, IC Width	4 instructions/cycle
Branch Predictor	gshare: 4K entries, 10 bit GHR
Return Address Stack	64 entries
Branch Target Buffer	1K entry, 4 way
Functional Units	4 Int. ALUs, 1 Int. MULT/DIV 4 FP ALUs, 1 FP MULT/DIV
L1 I and D Caches	1K sets, 2-way, 32 byte block size
L2 Unified Cache	2K sets, 4-way, 64 byte block size
Pipeline Depth	5 stages before the issue stage

4.2 Simulated Schemes

4.2.1 Establishing upper and lower bounds

To establish the envelope in which we are working we determine upper (oracle) and lower (baseline) bounds for instruction delivery activities. For the *baseline* simulation model, no activity saving mechanisms are used. The *oracle* model gives the same performance as the baseline and uses oracle knowledge to save energy. In particular, branch mispredictions occur, but the *oracle* model stops fetching until a mispredicted branch is resolved; no mis-speculated instructions are fetched. Furthermore, in the *oracle* scheme instruction fetching for *all* committed instructions is deferred as long as possible such that 1) the instruction issue time is not delayed for any instruction and 2) in-order instruction fetching of cache-line granularity is maintained.

4.2.2 JIT I-fetch

The proposed JIT instruction delivery scheme is simulated with performance tuning *thresholds* of 2%, 5%, and 10%. ***JIT{X%}*** will be used to identify the scheme where the performance tuning threshold is ***X%.***

4.2.3 Pipeline Gating

PG is simulated as proposed in [7]. A 128 entry JRS confidence estimator [10] is used. A branch is classified as high confidence when the counter accessed in the confidence table exceeds 12. At most three low confidence branches are allowed in the processor at any given time.

4.2.4 Adaptive Issue Queue

The AIQ scheme proposed in [2] is simulated. The issue queue is 32 entries. It is re-sized up/down in chunks of eight entries. Every 8000 cycles the utilization of the queue is sampled. If the utilization exceeds a *threshold* it is sized up – for example, if the current size is 8 and the utilization is 7 then the queue is sized up to 16 for the following interval. If the utilization is below a size down threshold then the queue is sized down. There are different size up/down thresholds for different number of active entries. Also, if the performance *after* sizing down the queue is worse than the performance *before* by a factor (0.9 in this case)

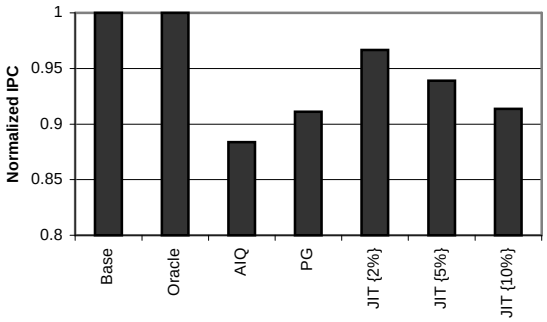

Figure 4: Normalized IPC

then the queue is sized up, again. The issue queue is non-collapsing, thus holes might be present in the queue if instructions issue from the middle of the queue. We assume the "holes" will be clock gated to save energy.

4.3 Benchmarks

We simulated SPEC 2000 INT benchmarks compiled with base optimization level (-arch ev6 –non_shared –fast). The test inputs were used and, all benchmarks were fast forwarded 100 million instructions and then simulated for 200 million committed instructions except *mcf*. Benchmark *mcf* completes execution at 159 million committed instructions after fast-forwarding 100 million instructions.

4.4 Performance Metrics

To evaluate energy savings, we collect the activity counts (refer to section 2) for the three parts of the instruction delivery subsystem. Both mis-speculated (flushed) and committed (used) instructions are included and are considered as separate activities. For each type of activity the average activity is computed by taking the arithmetic mean of that activity over all benchmarks. Then, counts for each of the three parts (fetch, decode, and issue queue) are normalized between 0 and 1, so that each is some fraction of the overall activity for the part being considered. Next, for all other (non-baseline) schemes the average activity of each part of instruction delivery is normalized with respect to the *baseline* total average activity.

For evaluating performance we use the number of committed instructions per cycle (IPC). Average IPC is calculated by taking the harmonic mean of the IPC over all benchmarks. Performance is then normalized with respect to the baseline.

5. RESULTS

Figure 3(a-c) has normalized activities averaged over all the benchmarks as described in the preceding section. Fig. 4 shows the normalized IPC averaged over all benchmarks.

First, consider instruction fetch activity in Figure 3(a). Because it has foreknowledge of branch misprediction, the oracle method wastes no energy fetching flushed instructions. PG, based on branch prediction confidence, reduces the activity for flushed instructions substantially, as is intended; it saves about half the wasted flush activity. However, PG reduces performance by 10%(Figure 4). The reason for the performance drop is that some branch predictions are assigned low confidence, yet are correct predictions. This occasionally causes the instruction decode pipe-

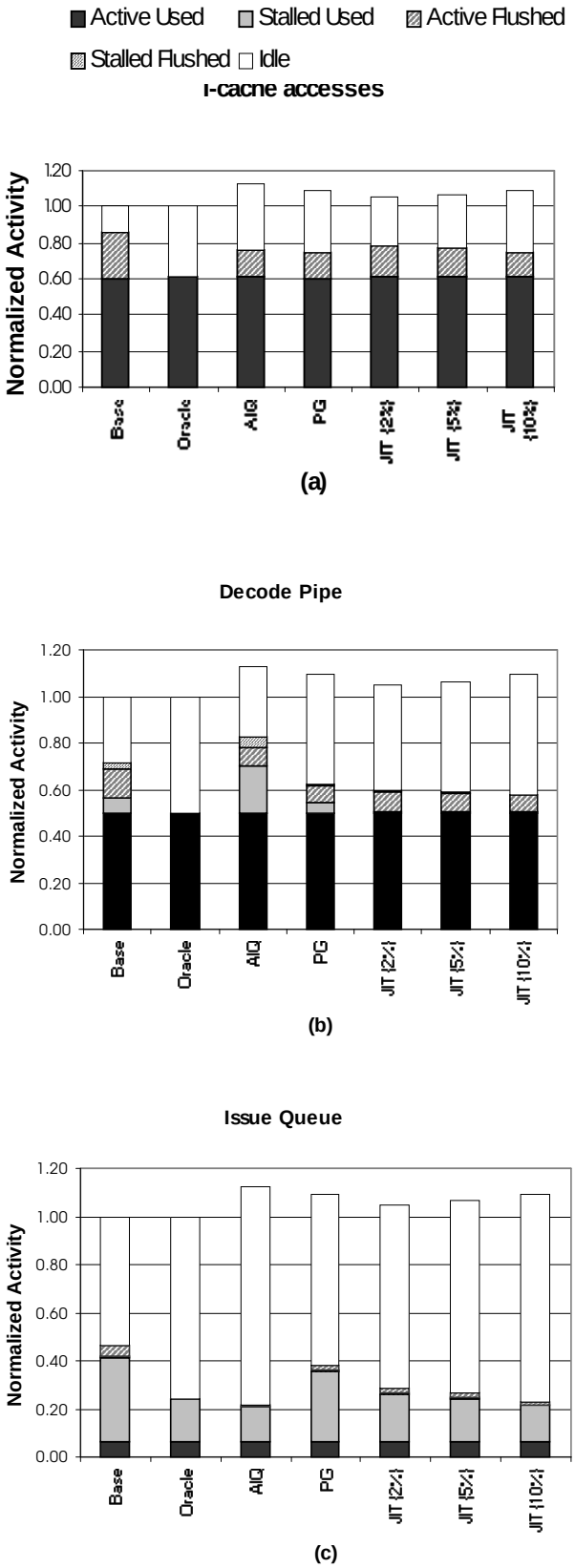

Figure 3: Normalized Activity

line to be needlessly starved of instructions. The performance degradation we observe is worse than that observed by the authors in [7] primarily due to the longer instruction pipeline (this tendency was pointed out in [7]).

AIQ saves some I-fetch activity, although not as much as PG. Reducing the issue queue ultimately reduces fetch activity whenever the queue and pipeline become full. The JIT method saves as much activity as PG, when their performance levels are the same (this occurs with JIT using a threshold of 10%, noted as JIT{10%}). With JIT{2%}, the overall performance loss is only 3%, and 9% of the instruction fetch activity is saved.

Now consider the decode pipeline activity (including decode, rename, dispatch) shown in Figure 3(b). Here AIQ has more activity than any of the other methods, including the baseline. These are primarily stall cycles because a shortened issue queue causes instructions to more readily back up into the rest of the pipeline. PG and the JIT methods provide similar activity savings, but the JIT method has slightly less activity when the IPC performance is the same (i.e for JIT{10%}).

In saving issue queue activity (Figure 3(c)) AIQ performs quite well, as expected. In fact, it has fewer instructions stalled in the queue than the oracle scheme. But the undesirable effect is that its performance degradation is 12%. Note that this loss is significantly more than reported in [2] where the degradation is 4%. As mentioned above – our simulation model has fetch/decode/issue width of four instructions whereas the authors in [2] have an issue width of eight instructions. Another contributor to the performance difference is more accurate modeling of the individual pipeline stages.

PG shows relatively little (6%) savings in stalled useful instructions, and reduces the active flushed instructions by about half compared with the baseline. The JIT (10%) method provides activity savings as good as the AIQ method, and performs better.

To summarize, with equivalent (or less) performance degradation, the JIT scheme performs as well as PG in reducing activity due to flushed instructions, and simultaneously it performs as well as AIQ at saving wasted activity in the issue queue. Additionally, it reduces more activity in the instruction decode pipeline than either of these schemes. Although JIT saves activity for accessing the data cache and in the execution units we do not include these savings here.

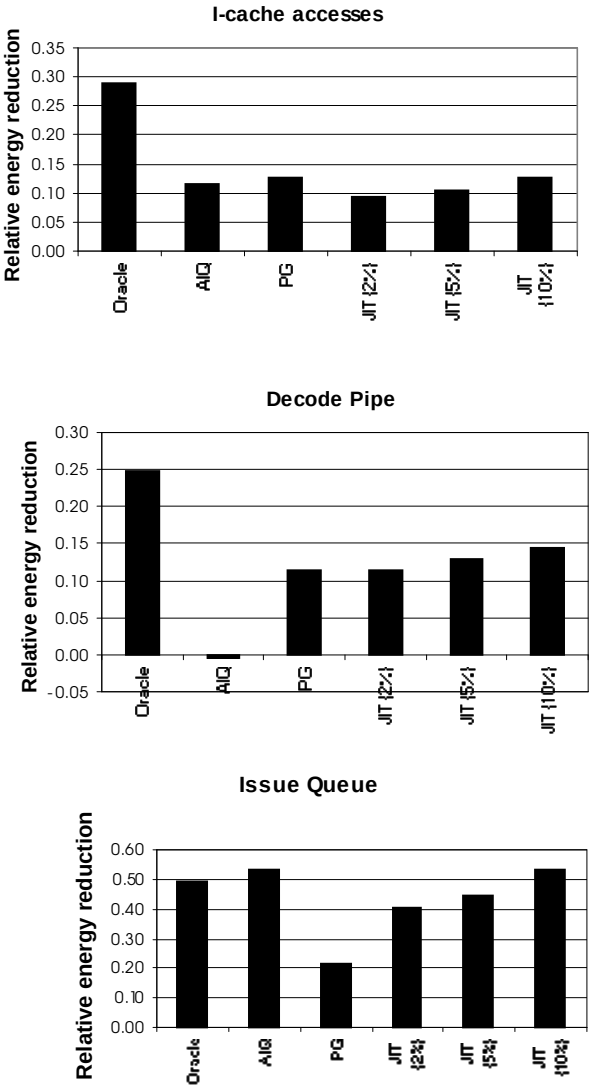

Figure 5: Relative Energy Reduction in a High Performance Processor.

6. ENERGY ESTIMATES

To give an idea of actual energy benefits of the proposed scheme, we evaluated the energy consumed for a state of the art microprocessor (POWER4™) [11]. The pipeline latches were taken from this high-end design environment. A 2-to-1 static mux was used to re-circulate the data when stalled. For the issue queue, wakeup logic is modeled by counting the energy in the comparators. For the selection logic, energy of one arbiter cell was calculated. Then the number of arbiter cells per arbiter was calculated based on the number of entries in the issue queue. We assume one arbiter per issue port – in our case four issue ports. Every entry in the issue queue has some comparators (for tag match).

Figure 5 gives the relative energy savings for the various schemes studied. The PG and JIT{10%} schemes save the most energy (13%), followed by a 12% savings for the AIQ. In the instruction decode pipe JIT{10%} saves 14% of the energy. PG saves 11%. AIQ increases the energy consumed in the instruction decode pipe by 0.5% because of more instruction back-ups as noted earlier. In the issue queue AIQ and JIT{10%} both reduce the energy by 53%. PG reduces 21% of the energy. AIQ and JIT{10%} reduce more energy than the Oracle but at a loss of 10% in IPC.

7. SUMMARY AND CONCLUSIONS

Energy reduction benefits come from avoiding fetch of misspeculated instructions and from avoiding stalls of useful instructions, especially in the issue queue. In effect, JIT instruction delivery combines the advantages of PG and AIQ methods.

Further, the implementation is simpler than either of the previously proposed schemes. In PG a branch confidence table is

added to the processor resulting in area and power overhead. As pipelines get deeper the penalty for incorrect confidence estimation will increase. AIQ has to continually monitor every stage of the issue queue to tune it. In contrast, the proposed JIT scheme uses only a few non-intrusive counters and control logic. The counters are very similar to the performance counters existing in current processors. Finally, with the JIT method, reconfigurations occur a much coarser granularity (100K instructions) than the other methods, allowing low level software or off-critical-path hardware to perform the dynamic adjustment of *MAXcount*.

8. ACKNOWLEDGMENTS

This work was supported by SRC contract 2000-HJ-782, NSF grant CCR-9900610, IBM and Intel.

9. REFERENCES

[1] Daniele Folegnani and Antonio González, "Reducing Power Consumption of the Issue Logic," *Proc. of the Workshop on Complexity-Effective Design held in conjunction with ISCA 2000*, June 10, 2000.

[2] Alper Buyuktosunoglu, et. al, "A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors," *Proc. of the on Great lakes Sym. on VLSI*, pp. 73-78, 2001.

[3] A. Klaiber, "The Technology Behind Crusoe Processors," Transmeta Technical Brief, http://www.transmeta.com/dev, Jan. 2000.

[4] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," University of Wisconsin-Madison Computer Sciences Department Technical Report #1342, June 1997.

[5] A. Baniasadi, and A. Moshovos, "Instruction flow-based front end throttling for Power-Aware High-Performance Processors," *Proc. of International Symposium on Low Power Electronic Devices*, Aug. 2001.

[6] Personal Communication with Alper Buyuktosunoglu, Nov. 2001.

[7] S. Manne, A. Klauser, and D. Grunwald, "Pipeline Gating: Speculation Control for Energy Reduction," International Symposium on Computer Architecture, Barcelona, Spain, June 1998.

[8] R. Balasubramonian, et. al, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *33rd International Symposium on Microarchitecture*, pp. 245-257, December 2000.

[9] M. Gowan, L. Biro, and D. Jackson, "Power Considerations in the Design of the ALPHA 21264 Microprocessor," *Design Automation Conference*, pp. 726-731, 1998.

[10] E. Jacobsen, et. al, "Assigning Confidence to Conditional Branch Prediction," International Symposium on Microarchitecture, pp. 142-152, Dec. 1996

[11] Anderson et al., "Physical design of a fourth-generation POWER GHz microprocessor," ISSCC 2001 Digest of Tech. Papers, Feb. 2001, p. 232.