An Adiabatic Framework for a Low Energy μ-Architecture & Compiler

PRAMOD RAMARAO, AKHILESH TYAGI
Computer Architecture Lab,
Department of Electrical & Computer Engineering,
Iowa State University, Ames, IA.
{pramod, tyagi}@iastate.edu

Abstract

Adiabatic process in thermodynamics transfers energy across zero temperature difference. The adiabatic CMOS design style attempts to switch a transistor to transfer energy across its source and drain while the voltage difference is zero. We define an adiabatic microarchitecture that pushes instructions across zero IPC gradient. The IPC gradient can be zero across time: for the same stage IPC over time does not vary, or across space: adjacent pipeline stages have zero variance. The reason to consider adiabatic microarchitectures is that the energy for a given computation can be shown to be minimum for an adiabatic microarchitecture. An adiabatic compiler, really a backend, is defined to be a compiler to support an adiabatic microarchitecture achieve its goals. The minimal support provided by an adiabatic compiler includes a static estimation of program ILP. We add new passes to the MachSUIF compiler, to flag instruction groups that can potentially walk through a superscalar pipeline as a group. Hence, these instruction groups offer a fairly robust model of superscalar microarchitecture ILP. A compile time scheduling analysis can also generate instruction slack values. The slack indicates the program region within which an instruction can be scheduled. We also present a dispatch stage dynamic scheduling algorithm that utilizes the compiler annotated slacks to reschedule instructions with the explicit objective of minimizing the dispatch stage IPC variance. In other words, the proposed dispatch stage is adiabatic. Preliminary experimental results demonstrate an average reduction of 4.16% in IPC variance over SPEC2000 benchmarks with the adiabatic compiler and microarchitecture. The preliminary evaluation also shows the average processor dispatch stage energy reduction of 3.9% over the same SPEC2000 benchmarks. We expect to add similar IPC smoothening control knobs at instruction fetch and issue stages as well in the future, which should result in a more significant energy reduction.

1. Introduction

Energy consumption and power dissipation are considered to be major limitations for both high performance and portable processor design. The International Technology Roadmap for Semiconductors (ITRS) expects the processor power needs to scale over next 10 years to outstrip the package dissipation limits by a factor of 25x [8]. Only radical changes in microarchitecture and programming paradigms can help meet this tremendous gap. Several research thrusts have addressed the processor energy and power reduction. Many techniques have been proposed to handle energy and power issues at various abstraction levels. At circuit level, these include voltage scaling [18], clock gating [4] and multi-threshold CMOS [15]. Some of the microarchitecture level techniques are pipeline gating [12], branch prediction confidence [17], segmented pipeline components [11], [10], selective cache way shutdown [19] and fetch throttle [22]. Compiler level optimizations to reduce energy have also been considered [9].

The energy reduction technique identified in this paper is more of a global optimality criterion for the program execution. In a nutshell, this criterion states that the IPC (instructions per cycle) of a computation should not vary over time. A more practical statement might be that the IPC should vary as little as possible. The intuitive reasoning behind this is as follows. Whenever a system admits high variance in its workload, it is forced to design its components for the peak needs. Invariably, this ends up creating system components that punish the average workloads by allocating peak resources. This scenario arises in a microarchitecture in the following way. A microarchitecture

is responsible for supporting all the inter-instruction communication. A microarchitecture designed to support instruction level parallelism (ILP) of k will incorporate all of the interconnection needed to sustain the inter-instruction communication for a group of up to k instructions. Even when only $k' < k$ instructions go through the microarchitecture, they are forced to communicate through a structure designed to support a much higher number of instructions. Hence, the capacitance switched to support k' instructions ends up being proportional to the peak capacitance switched to support k instructions. Herein lies the crux of the problem.

We [21] have argued that optimal VLSI computations neither overdesign nor underdesign the wire lengths for a given algorithm. This is only achievable when each wire and gate is kept busy for each unit of time. This leads to optimal packing of logic gates resulting in minimum possible average wire length for the required communication topology. The overall energy is minimized in this fashion. This observation is stated as a *Principle of Least Computational Action* [21] motivated by a similar principle of least action in physics. The computational action can be defined as the product of overall energy and time. From various perspectives, a computation that minimizes the computational action is an optimal one – not necessarily the fastest one. In this paper, we project this principle into the domain of superscalar microarchitectures. The resulting optimality criterion appears to be that the IPC variance delivered by the microarchitecture should be minimized. A program that is able to deliver uniform IPC across all computation cycles for a given microarchitecture consumes the same amount of energy each cycle. That is one extreme operating point which would rarely be achievable for most interesting programs. The natural, base IPC of each program varies over different parts of the program as different algorithms are activated, or within the same algorithm, a different computational phase is initiated. The objective of this work is to explore the possibility of maintaining the uniform IPC within the program epochs where a natural uniform IPC level exists. We also strive to quantify the benefits of such a strategy.

The methodology explored in this paper uses a compiler to estimate the natural program IPC in small granularity chunks, typically a basic block. However, the compiler does not reschedule instructions to generate uniform IPC packets. The assumption is that the IPC model of the compiler is not fully accurate given the traditional run-time uncertainties resulting from not knowing the program input. Hence, the instruction rescheduling is still delegated to the dispatch stage to exploit the benefits of a dynamic scheduling scheme. The compiler generated information, however, ensures that the cycle time is not stretched beyond the existing critical paths in the dispatch stage. Uniform IPC would be of little help if the microarchitecture is not designed to deliver IPC level l, for $1 \leq l \leq k$, with the energy needs of an l-wide microarchitecture. The energy consumption, $e(k)$, of a superscalar microarchitecture to support peak issue width of k has been shown to be superlinear: $e(k) = ck^3$ by Palacharla [16], $e(k) = ck^a$ for $a > 2$ by Zyuban et al. [23]. We call a microarchitecture *decouplable* if it can deliver IPC l with energy equivalent to $e(l)$ for $1 \leq l \leq k$ where k is the peak issue width supported by the microarchitecture. Zyuban et al. [23] have considered several microarchitectural components for a decoupled design: ROB, issue queue. Pipeline balancing [3] is also an effort in this direction for FU allocation. This still leaves several microarchitectural components whose designs are tightly coupled (in the sense that switched capacitance cannot be reduced below the peak for a lower delivered IPC). We are investigating decoupled designs for these components.

The term *adiabatic* in thermodynamics refers to a process where no heat is gained or lost. Adiabatic switching [2] was developed with the same intuitive objective. In adiabatic CMOS, the charge is transferred across as low a voltage difference as possible – zero asymptotically. We view the temporal IPC gradient (IPC differences at the same stage over two consecutive cycles) or the spatial IPC gradient (IPC differences between two adjacent pipeline stages) to serve the same role as voltage or pressure gradient at the algorithmic and microarchitectural levels. Hence, an adiabatic algorithm or a microarchitectural orchestration of it should exhibit as low an IPC gradient as possible for all the instruction flows. This reasoning leads us to call the proposed microarchitecture *adiabatic*.

The paper is organized as follows. We specialize the principle of least computation action to superscalar microarchitectures in Section 2. The compiler scheduling and annotations are described in Section 3. The dynamic dispatch stage scheduling to even the IPC variance is described in Section 4. The concept of decouplable, segmented microarchitecture components is introduced in Section 5. We present our implementation framework in Section 6. We conclude the paper in Section 7.

2. Least Computational Action for Microarchitecture *or* Adiabatic Microarchitecture is Optimal

In this section, we specialize the principle of least action [21] to a superscalar microarchitecture. The princi-

ple of least computational action states that a VLSI algorithm that has the least computational action, the product of energy and time, is the most optimal one for that problem. The proof of a similar theorem specialized for a superscalar microarchitecture is considerably simplified. It has been shown previously [23], [16] that the energy consumption of a superscalar microarchitecture scales superlinearly as a function of n, the dispatch or issue width. More specifically, the switched capacitance per instruction in the wakeup logic of issue stage is at least quadratic in issue width [16] leading to this assertion. This has significant ramifications on how the instructions of a program should be distributed over time within the processor. Let us consider the execution of a program P with dynamic instruction count IC. Let IC_t be the number of instructions executed in cycle t. If the execution time of the program P is T for a specific superscalar microarchitecture, then $\sum_{t=0}^{T-1} IC_t = IC$. Let $e(k)$ be the energy function to capture the energy consumption in the given superscalar microarchitecture as a function of the number of instructions issued, k for $1 \leq k \leq IW$ with IW being the designed issue width of the processor. Note that Palacharla demonstrates this energy function to have the profile $e(k) = ck^3$ for some constant c. Similarly, Zyuban et al. [23] argue that $e(k) = ck^a$ for $a > 2$. The following theorem constitutes the heart of the paper.

Theorem 1 (Adiabatic computation theorem) *The energy of the execution of a program P with instruction count IC taking time T on a superscalar microarchitecture with a super-linear energy function e is minimized if the number of instructions issued each cycle is uniformly IC/T.*

Proof: Let IC_t be the number of instructions issued at time $0 \leq t \leq T-1$. It follows then that $IC = \sum_{t=0}^{T-1} IC_t$. The energy of this computation is $E_P = \sum_{t=0}^{T-1} e(IC_t)$. The sum of such super-linear factors under a linear constraint of the form $IC = \sum_{t=0}^{T-1} IC_t$ is minimized when each factor has the same value. That occurs when the IPC is uniform across all the T cycles, *i.e.*, $\forall t : \ IC_t = \frac{IC}{T}$. This is true since the derivative of this sum with respect to any IC_i with a super-linear function $e(IC_i) = c * IC_i^{1+\epsilon}$ has the form $c * (1+\epsilon) * IC_i^{\epsilon}$ for $\epsilon > 0$. This says that the first derivative is a monotonically increasing function of the IPC IC_i, or it evaluates to a higher value for a higher IPC. Any positive departure from the mean, IC/T, in the kth cycle IPC ($IC_k = ([IC/T] + \Delta)$) will be compensated for by a negative departure in one or more terms. For the simplicity of the proof, assume that the k'th cycle's IPC $IC_{k'} = ([IC/T] - \Delta)$ is reduced to compensate for the kth cycle's higher than average IPC. Now consider $e(IC_k) + e(IC_{k'})$ which is approximated by Taylor's series expansion w.r.t. the first derivative only as $e([IC/T] + \Delta) + e([IC/T] - \Delta) = 2 * e(IC/T) + \Delta * \frac{\delta e}{\delta IC_k}\left[\frac{IC}{T} + \Delta\right] - \Delta * \frac{\delta e}{\delta IC_{k'}}\left[\frac{IC}{T} - \Delta\right]$. Note that $\frac{\delta e}{\delta IC_k} = \frac{\delta e}{\delta IC_{k'}}$. Moreover, since the first derivative is monotonically increasing, it evaluates to a higher value at a higher point, *i.e.*, $\frac{\delta e}{\delta IC_k}\left[\frac{IC}{T} + \Delta\right] > \frac{\delta e}{\delta IC_{k'}}\left[\frac{IC}{T} - \Delta\right]$. Also note that for a uniform IPC, $e(IC_k) + e(IC_{k'}) = 2 * e(IC/T)$. Hence, the excess energy due to non-uniformity $\Delta * \left(\frac{\delta e}{\delta IC_k}\left[\frac{IC}{T} + \Delta\right] - \frac{\delta e}{\delta IC_{k'}}\left[\frac{IC}{T} - \Delta\right]\right) > 0$. This proves the theorem.

■

3. Static Compiler Models for IPC

We model the superscalar microarchitecture IPC at two places: dispatch stage and issue stage. The objective is to have control knobs both at dispatch stage and at issue stage attempting to even out the variance in IPC. We describe the compiler models for both dispatch and issue stage IPC.

3.1. Dispatch Stage IPC Model

The dispatch stage in a superscalar microarchitecture is designed to dispatch all the available instructions within the dispatch window. Even the instructions that are dependent can be dispatched simultaneously through renaming of the dependence operand. The IPC limiting events at dispatch stage are as follows. All the instruction fetch inefficiencies get reflected in reduced number of instructions in the dispatch window, and hence limit the dispatch IPC as well. These factors include I-cache block fragmentation whenever the target of a branch is in the middle of a cache block. The branch misprediction is another major cause of instruction fetch inefficiency limiting the dispatch IPC. Other causes of dispatch IPC variation include structural hazards on reservation stations and reorder buffer (ROB).

It is easiest to model the branch misprediction at compile time among these factors. One simple way to model the dispatch IPC independently from the microarchitecture design parameters would be to measure the number of instructions between two control instructions. These instructions can be considered to be dispatchable simultaneously. This model can be refined further if the target description for the compiler can include some of the microarchitecture parameters such as branch prediction accuracy bp, cache block size B, instruction fetch width IFW. These will allow the dispatch IPC to reflect cache block fragmentation effect and branch misprediction effect. This still leaves

out reservation station and ROB structural hazards based IPC loss. We believe that a fairly detailed microarchitecture modeling would have to be incorporated into the compiler to account for structural hazard induced IPC loss. In the first phase of this work, we have chosen to omit this factor from the dispatch stage IPC model. We have implemented the branch-to-branch instruction count as the dispatch IPC model currently. Future extensions will bring in cache block size and branch misprediction rates.

3.2. Issue Stage IPC Model

We build on the static *IPC* prediction model developed by [22]. This model is a compiler based static prediction scheme which gives an aggregate measure (average over thousands of cycles) of the commit stage ILP. It is a dependence driven model in which the instruction stream is analyzed only for true data dependences (Read-after-Write). It is assumed that named dependences are eliminated by the hardware through renaming as superscalar processors have sufficient resources for renaming. The rename space is assumed to be large enough not to affect the IPC. Note that the IPC loss from issue stage to commit stage is very low (as has been observed in several papers [6]). A group of instructions without a true dependence ought to be able to issue simultaneously (assuming their producer instructions all belong to the same preceding issue group(s), or are far ahead enough not to cause delays at issue stage). This is why we believe that a true dependence driven IPC model captures IPC at the issue stage well.

The instruction stream is divided into annotation blocks wherein each block contains instructions that can be typically issued together. True dependences serve as the boundaries of these blocks. The model is a fairly accurate prediction of the actual *IPC*.

In order to allow for the dynamic scheduling of instructions, we use the notion of *slack* defined by Fields et al. [7]. They define *local slack* as the number of execution cycles between a producer and its consumers. The producer can be delayed by the local slack available so that there is no impact on the execution of the consumers. They also define a variant called *Global Slack* of an instruction, which is the sum of its local slack and the minimum global slack of its consumers. They show that approximately 20% of the instructions have local slack greater than five cycles. On the other hand, they observe that 40% of the instructions have global slack of more than 50 cycles.

In addition to considering local slack to delay instructions at dispatch, we also consider slack as the number of cycles that an instruction i can be pushed forward in an instruction stream. Considering the slacks of an instruction in both directions (delay as well as push forward) provides the dispatch stage scheduling algorithm more flexibility to choose instructions to issue together in order to establish a uniform IPC rate.

We estimate the *local slack* of each instruction in the instruction stream at the compiler-level. Dependency analysis is performed to predict the IPC at the compiler-level. The dependency analysis is performed on both register and memory accesses using a monotone data flow analysis [13] Register dependences are established using the reaching uses algorithm [1]. For memory accesses, we follow the model of [22] and perform alias analysis by instruction inspection. This analysis distinguishes between memory accesses to different regions. Accesses which use the base-displacement modes are also considered. Accesses with the same base register but different displacements are identified as having no true data dependence between themselves.

The slack values for the instructions are also computed in an analogous manner. Once the true dependences are established between instructions through the data flow analysis, the slack value is then the distance between the producer and the consumer. The instructions are then annotated with these values.

Another model we propose for the issue stage IPC is derived from the dataflow graph of a basic block. The DFG is levelled through breadth-first search (BFS) starting at the root node. All the nodes at the same level represent independent instructions. These instructions (within the same level) can be issued at the same time and form a single producer group as well as consumer group with respect to issue stage operand garnering. This model leaves out the dependences coming in from the preceding blocks. We assume their effect to be minimal for issue stage modeling. The compiler then needs to annotate instructions in the same level of the BFS by a level number, which will identify them as a producer and/or consumer group to the dispatch and issue stages.

4. IPC Smoothening Dispatch & Issue Stages

4.1. Dispatch Stage

The dispatch stage is responsible for reducing the variance of IPC of the instructions moving from the dispatch stage to issue stage. Each fetched instruction has an associated slack amount annotated by the compiler. These slack values allow the dispatch stage to smoothen the IPC variance without having to stretch its critical path. Figure 1 shows the basic dispatch stage schema.

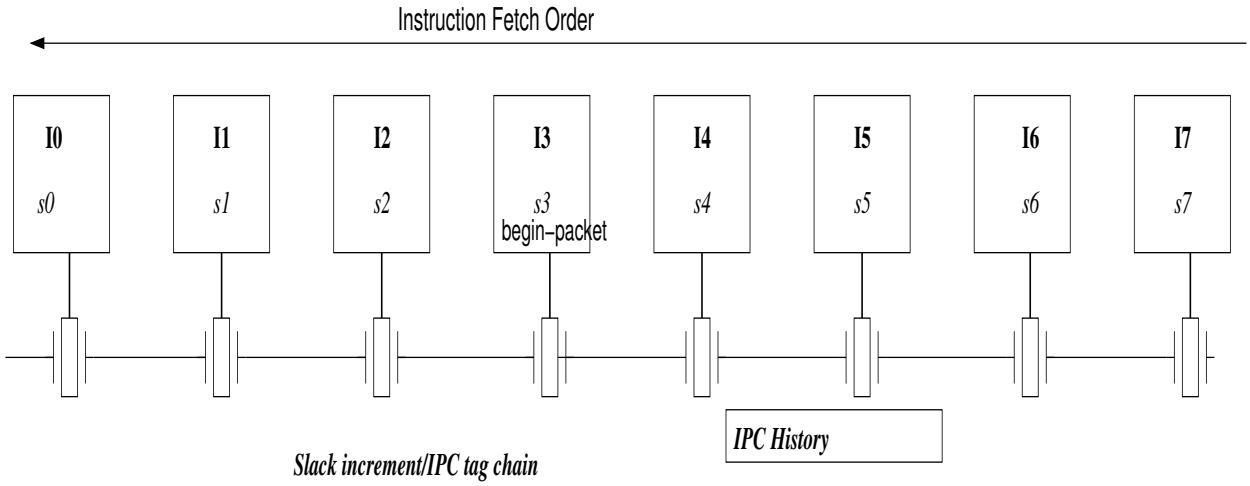

Figure 1. Dispatch Stage Schema.

The dispatch stage maintains a history of dispatch IPC for one or more cycles, IPC_{last}. Currently we assume a history of one cycle. We use a *Percolation Algorithm* to percolate instructions from one dispatch packet to another. Based on the history of the IPC, instructions can be delayed at the dispatch stage thereby allowing them to propagate to following packets. If the current dispatch packet size is more than the IPC history, one or more instructions are selected to propagate in order to reduce IPC variation. The decision of selecting instructions to propagate could be based on their slack values, which would allow multiple instructions to be delayed at dispatch. We follow the simple model of delaying the last instruction of every dispatch packet. The Percolation algorithm is as shown below.

```
for (every cycle of dispatch stage)
 {
   if (oldIPC < size of dispatch packet)
     {
     Initialize count =
            (size of dispatch packet - 1);
     Repeat {
      while (reservation station
       available/ reorder buffer not full/
        fetch queue not empty)
       {
        Dispatch instruction at head of
                            fetch queue;
        count = count - 1;
       }
      }
      Until (count reaches zero);
      Place the delayed instruction at head of
        fetch queue to be dispatched in the
        next cycle;
    }
 }
Store oldIPC = IPC of this cycle;
```

Advancing an instruction based on its slack is another option to provide another degree of freedom to the dispatch stage. This would be useful if the number of dispatchable instructions is at least two fewer than IPC_{last}. The causes for lack of dispatchable instructions are: *(1)* too few instructions in the instruction window, *(2)* structural hazards such as ROB or reservation station non-availability. Solution to the first cause requires fetch enhancements. The second cause can only be alleviated by altering the ROB size and/or number of reservation stations dynamically. In this case, some instructions might be able to move ahead to increase the current cycle IPC.

The dispatch stage is responsible for reducing the variance of IPC of the instructions moving from the dispatch stage to issue stage.

4.2. Issue Stage

The issue stage provides the second major juncture in the microarchitecture to control the IPC variance. The instructions that were in-order at the dispatch stage appear in an out-of-order sequence at issue stage. The instructions that were identified to be in a single issue IPC group by the compiler were in adjacent instruction slots in the dispatch window. However, at issue stage, they are dispersed. The compiler tags the lead instruction of a simultaneously

*issu*able group of instructions by `begin-packet` annotation (as shown in Figure 1).

The dispatch stage needs to assign identical tags to all the instructions between two `begin-packet` tags. This tag space is orthogonal to the rename space. The logic chain at the bottom of Figure 1 helps with an efficient assignment of this tag. The transmission gate driven by the `begin-packet` signal of each instruction is cut off for the instruction packet leader instructions (the transmission gate is driven by $\overline{\texttt{beginpacket}}$ signal). All the instructions in the same issuable packet are connected by a single segment of the increment/decrement or IPC-tag control signal.

The compiler estimated issue IPC value is attached to all of these instructions. The wakeup logic of issue stage generates a set of issuable instructions. The select logic selects from these instructions through arbitration. Now the select logic will also take into consideration IPC smoothening. A history of last few issue-IPC values can be maintained. Assume that the history is a single value `issue-IPC-last` in order to simplify the discussion. If all the awakened instructions are from the same issuable group (with the same IPC-tag), the select logic will attempt to issue as many instructions as `issue-IPC-last`± 1. If the size of the issuable group (also annotated with the instructions) exceeds this number (`issue-IPC-last`± 1), the issue of some instructions will be deferred. We have not yet implemented the issue IPC control knob.

5. Segmented Microarchitecture Components

In order to derive energy benefits from the uniform IPC at dispatch and issue stages, the microarchitecture components have to be designed in a certain way. Consider the energy equations in Theorem 1. The energy is lower for an adiabatic IPC distribution since the microarchitecture energy is super-linear in the number of processed instructions. For a given microarchitecture component C, let $e_C(n)$ signify the energy consumption in processing n instructions. The traditional way of designing the microarchitecture is to exercise the entire component C designed for a peak instruction width IW even if $k < IW$ instructions are processed. In other words, energy consumed by a microarchitecture component C is always $e_C(IW)$ for any $0 \leq k \leq IW$ processed instructions. In order to benefit from the reduced variance, C needs to consume energy $e_C(k)$ to process $k < IW$ instructions. Ghose et al. [11], [10] have developed segmented versions of several microarchitecture components such as ROB which have this property. For a lower IPC, only one segment is activated. Another segment kicks in when the IPC exceeds certain threshold, and yet another segment is activated when IPC exceeds some other threshold and so on. We assume that the underlying microarchitecture is designed with such segmented components to provide the desired energy profile.

Issue	4-way Out-of-order
Fetch Queue Size	4
Branch Prediction	2K entry bimodal
Int. Functional Units	4 ALUs, 1 Mult./Div.
FP Functional Units	4 ALUs, 1 Mult./Div.
L1 D- and I-cache	Each: 16Kb, 4-way
Combined L2 cache	256Kb, 4-way

Table 1. Baseline Parameters

6. Experimental Methodology

We implemented the static *IPC* prediction model developed by [22] and the static slack estimation using the MachineSUIF [20] compiler framework. New compiler passes were developed which are applied just before the Alpha assembly code generation pass is invoked. The passes also implement the data flow and slack estimation techniques outlined previously. The passes are applied before the assembly code is generated so that no other compiler optimization pass is invoked later. The estimated IPC and slack values are placed as annotations to the instruction. The annotations are preserved in the final binary generated.

In order to implement the dynamic dispatch scheduling algorithm, we use the SimpleScalar simulation toolset [5]. The dispatch stage is modified to control the issue of instructions from the dispatch window to the reservation stations based on the IPC and slack information annotated by the compiler. The system model is a typical out-of-order superscalar processor. Table 1 describes the baseline parameters used.

We studied the effect of the dispatch stage percolation algorithm. We performed experiments with programs from the SPEC2000 CPU benchmark suite. We fast forward the simulation by 200 million instructions. The next one million cycles are simulated and the resulting IPC values are noted. We examined the standard deviation of the new IPC values against those generated using a baseline model.

Figure 2 shows the IPC variation over these benchmarks. Using this algorithm, we obtained a reduction in the IPC variation for all the programs. The reduction ranges from 2.56% to 8.72% with the average around 4.16%. Using a more sophisticated dispatch algorithm, we expect to extract more reduction in the dispatch IPC.

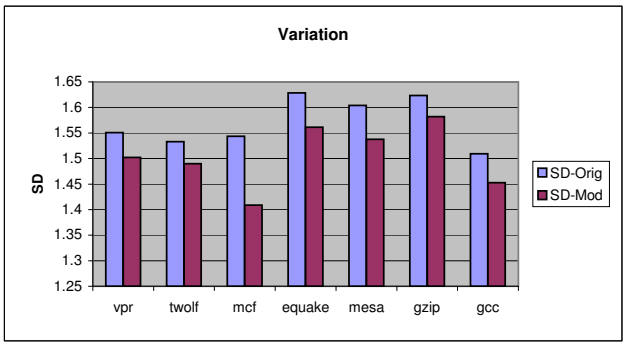

Figure 2. Dispatch Stage IPC Variance.

In order to evaluate the effect of this IPC reduction on the dispatch stage energy, we used the following empirical and analytical hybrid model. The energy of the dispatch stage (of rename map table instance) varies as n^2 for n instructions in the dispatch window. We calculate the sum $\sum_{t=0}^{N}(DW_t)^2$ where DW_t is the observed dispatch width at time t for both the original dispatch scheme and the IPC smoothening dispatch scheme.

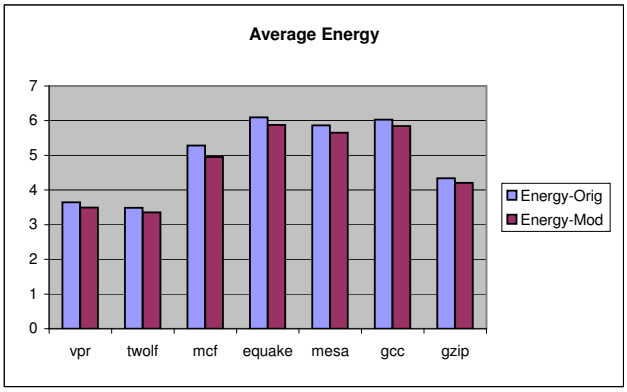

Figure 3. Dispatch Stage Quadratic Model Energy Comparison.

The results are shown in Figure 3. We obtained a decrease in the dispatch stage energy ranging from 3.05% to 6.31% with the average around 3.9%.

7. Conclusions & Future Work

We presented a concept – adiabatic microarchitecture, as an energy reduction technique. The adiabatic microarchitecture strives to maintain low IPC variance so that the instructions flow through as low an IPC gradient as possible. This kind of adiabatic flow is shown to have minimum switching energy for a microarchitecture with components whose energy needs grow super-linearly with the number of supported instructions (instruction width). Hence, the concept is applicable to commonly employed superscalar microarchitecture. We present a preliminary evaluation of the concept. The three key places where a control knob to smoothen the IPC variance can be placed are instruction fetch, dispatch and issue stages, since the largest IPC losses occur at these stages. This paper presents a greedy heuristic based compiler support and dynamic dispatch stage support for smoothening the IPC variation only at the dispatch stage. A preliminary scheme for issue stage IPC smoothening is also presented, but has not been implemented yet. We also anticipate adapting some of the recent work on fetch throttling [22] or instruction fetch deferral [14] for fetch IPC smoothening. The compiler uses a dispatch IPC model to mark instructions into dispatchable packets. It also explicitly annotates the dispatch IPC for each of these packets. The dispatch IPC control knob uses a percolation based instruction deferral mechanism to even the dispatch IPC. The dispatch IPC variance decreased by about 4.16% over seven SPEC2000 CPU benchmark programs. Assuming a quadratic energy model, the dispatch energy is predicted to decrease by 4% from this IPC smoothening. We expect the energy reductions to be more significant once the instruction fetch and issue stage IPC control knobs are in place.

References

[1] A. Aho, R. Sethi, and J. Ullmann. Compilers: Principles, techniques and tools. *Addison-Wesley Publishing Company*, 1985.

[2] W. Athas, L. Svensson, J. Koller, N. Tzartzanis, and E. Chou. A framework for practical low-power digital cmos systems using adiabatic-switching principles. *International Workshop on Low Power Design*, pages 189–194, 1994.

[3] R. Bahar and S. Manne. Power and energy reduction via pipeline balancing. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[4] L. Benini, G. D. Micheli, E. Macii, M. Poncino, and R. Scarsi. Symbolic synthesis of clock-gating logic for power optimization of synchronous controllers. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, pages 4(4):351–375, 1999.

[5] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. *Technical Report CS-TR-96-1308, University of Wisconsin, Madison*, 1996.

[6] T. Diep, C. Nelson, and J. Shen. Performance evaluation of the powerpc 620 microarchitecture. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 163–175, 1995.

[7] B. Fields, R. Bodik, and M. Hill. Slack: Maximizing performance under technological constraints. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[8] ITRS. International technology roadmap for semiconductors. *http://public.itrs.net*, 2001.

[9] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye. Influence of compiler optimizations on system power. *IEEE Transactions on VLSI Systems*, pages 801–804, 2000.

[10] G. Kucuk, K. Ghose, D. Ponomarev, and P. Kogge. Energy-efficient instruction dispatch buffer design for superscalar processors. *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2001.

[11] G. Kucuk, D. Ponomarev, and K. Ghose. Low-complexity reorder buffer architecture. *Proceedings of the 16th ACM International Conference on Supercomputing*, 2002.

[12] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[13] S. Muchnick. Advanced compiler design and implementation. *Morgan Kaufmann Publishers*, 1997.

[14] G. Muthler, D. Crowe, S. Patel, and S. Lumetta. Instruction fetch deferral using static slack. *Proceedings of the 35th International Symposium on Microarchitecture*, 2002.

[15] K. Nii, H. Makino, Y. Tujihashi, C. Morishima, Y. Hayakawa, H. Nunogami, T. Arakawa, and H. Hamano. A low power sram using auto-backgate-controlled mt-cmos. *IEEE/ACM International Symposium on Low Power Electronics and Design*, 1998.

[16] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.

[17] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, 2002.

[18] T. Pering, T. Burd, and R. Brodersen. Dynamic voltage scaling and the design of a low-power microprocessor system. *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 251–259, 2001.

[19] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. *Proceedings of the 34th International Symposium on Microarchitecture*, 2001.

[20] M. Smith. Extending suif for machine-dependent optimizations. *Proceedings of First SUIF Compiler Workshop, Stanford, CA*, pages 14–25, 1996.

[21] A. Tyagi. A principle of least computational action. *Proceedings of IEEE Workshop on Phyics and Computation*, pages 262–266, 1992.

[22] O. Unsal, I. Koren, C. Krishna, and C. Moritz. Cool-fetch: Compiler-enabled power-aware fetch throttling. *Computer Architecture Letters*, pages 6–10, 2002.

[23] V. Zyuban and P. Kogge. Inherently lower-power high-performance superscalar architectures. *IEEE Transactions on Computers*, pages 268–285, 2001.