New Findings on Using Queue Occupancy to Integrate Runtime Power-Saving Techniques Across the Pipeline

UNIV. OF VIRGINIA DEPT. OF COMPUTER SCIENCE TECH. REPORT CS-2003-15

Yingmin Li, Kevin Skadron, Mircea R. Stan
Depts. of Computer Science and
Electrical and Computer Engineering
University of Virginia
Charlottesville, VA 22904
{yl3e,skadron}@cs.virginia.edu, mircea@virginia.edu

July 2003

Abstract

This paper provides new insights on how to integrate power-saving techniques by using queue occupancies to dynamically match the power-saving modes of various pipeline stages with the current instruction throughput. (This paper focuses on fetch, decode, integer execution, and data cache.) Architects have proposed many runtime power-saving techniques, most of which reduce power dissipation in a single microarchitectural unit. But very little work has been done to integrate these disparate techniques to ensure that they cooperate rather than interfering with each other.

We use both queuing theory and experimental results to justify the use of superscalar decoupling queues to guide dynamic control of power settings. This permits integrated power control for multiple units across the pipeline, with minimal negative interaction, by matching the throughput of each stage and the application's current instruction-level parallelism. Our findings verify but also improve upon those in previous work by Semerraro et al. In particular, our approach is robust in jumping out of the bad power modes configuration incurring radical performance degradation, and our approach allows the fetch stage (a significant source of power dissipation) to realize power savings, something that prior integrated, queue-based techniques have not been able to accomplish.

1 Introduction

In recent years, techniques for runtime power savings have been an active area of research. A great deal of work has focused on developing power-saving mechanisms for specific functional units or execution domains. For example, [1, 2, 11, 14] introduced several power saving mechanisms for cache, [4] and [10] introduced power saving mechanisms for queue structures in microprocessor, and so forth. While all these work make important contributions to power saving, it is also important to integrate techniques across the pipeline so that each operates at a level that is globally rather than locally optimal.

To realize power savings typically requires sacrificing latency and/or throughput. However, high-performance processors are typically designed for peak throughput, so for more typical levels of instruction-level parallelism (ILP), significant power savings can be achieved with minimal performance loss. This in turn entails having different units in the pipeline operate at different speeds or power-saving levels. But if each unit adapts autonomously, without considering the consequences of its behavior on other units' adaptation, a radical degradation in performance can occur. Slowdowns caused by one unit can make other units believe they can reduce their latency/throughput, resulting in a disastrous positive-feedback loop.

Semeraro et al. [12] recently described an algorithm for *integrated* adaption across multiple execution domains in a multiple-clock domain processor. Their algorithm uses feedback control of occupancy in decoupling queues to identify differences in the required operating speed of each execution domain, and to guide each execution domain's frequency and voltage to the minimum that still maintains satisfactory performance.

This paper expands on that work in several ways. First, we use queuing theory to briefly clarify why an approach based on queue occupancy is effective. Second, we show that this approach does suffer substantially from noisy program behavior, but can still attain attractive energy savings and energy-delay improvements. Third, we show that the attack-decay algorithm in [12] can't quickly jump out of the power modes configuration which will lead to very low global performance and show how our version of this algorithm avoids that problem. Finally, we show how to achieve energy savings in the processor's front end, something that [12] was not able to achieve.

2 Related Work

Some other work has explored how to simultaneously control power modes for more than one unit in the processor. Iyer [7] introduced a mechanism to optimize processor energy-delay product at runtime by sampling the performance and energy data for each possible combination of the processor resources configurations and identifying the best configuration. While this mechanism will work when the number of possible configurations is small, it is impractical to traverse all combinations when this number is large. And this number can easily be very large, because it is the product of the number of the power modes of each unit. Dynamic search algorithms like ours and [12], on the other hand, avoid this problem.

Ponomarev et al. [10] propose a way to tune the size of the queue structures in the processor for energy savings. They believe that queue occupancy provides an accurate measure of the queue size required for the pipeline to maintain full speed for the running program. Based on this observation, they propose to sample the queue occupancy and average the sampled values, then resize the queue occupancy to this average value in the next interval. Because queues in the processor are responsible for a considerable portion of the whole power consumption, this yields good energy saving but with some performance loss. In our experiment, we find that this queue resizing mechanism, combined with the power modes control for the front-end, can increase the energy savings beyond that attained by our version of the feedback control of per-domain voltage and frequency scaling.

Dropsho et al. [5] integrate power modes control for caches and queue structures. They use local information for effective control of multiple independent adaptive caches and adaptive scheduling queues. They did not explore how to integrate control for units other than queues and caches. Because they only use information local to each unit for controlling, their mechanism pays no attention to the speed differences among units in the running time. based on the occupancy of the decoupling queues, Our control framework utilizes such speed differences information to get better performance power consumption tradeoff.

Karkhanis et al. [8] use just-in-time instruction delivery to save energy for the fetch engine, the decoding part and the issue queue by dynamically limiting the number of in-flight instructions. The optimal number of in-flight instructions is determined by profiling at the beginning of each interval according to the the global IPC.

In [13], Seng et al. use the output of a dynamic critical path predictor to assign non-critical instructions to lower-power functional units and issue mechanisms.

Huang et al. in [6] proposed a dynamic processor power modes adaption algorithms based on program code position. Their ideas exploit the fact that one segment of program code is quite possible to be used multiple times in the running time. If you can try out the best configuration for the first run of the this piece of code, it's very reasonable to apply this configuration for the following runs of this code segment.

Finally, in the paper that is most directly related to our work here, Semeraro et al. [12] control the voltage and frequency of each domain in a globally-asynchronous/locally-synchronous or multiple-clock-domain processor by tracking occupancy of the issue queue *before* each domain. However, it

only touches the back-end units power modes regulation problem and pay no attention to the front end power saving possibility, which recent work [9] has shown to be a significant portion of overall power consumption in the processor. In more recent work [4], Buyuktosunoglu et al. explored the front end units power saving regulation based on the issue queue occupancy and application parallelism characteristics. They also found that coupled with queue resizing techniques proposed in [4], their techniques will lead to more power savings and better power-delay results.

3 Framework for Power Modes Control

3.1 Terminology, Definition and Basic Assumption

In this paper, we assume that domain-independent frequency and voltage scaling, similar to that used by [12], is available so that we can change the frequency of each domain independently to get different power modes, and attain the cubic power savings that result from scaling frequency and voltage jointly. We assume there are five such domains, they are fetching engine domain, load/store domain, decoding domain, integer execution domain and floating execution domain as shown in Figure 1. We call the fetching engine domain and decoding domain as front end, the integer execution domain, the floating execution domain and the load/store domain as back end. From Figure 1, we can see there are four decoupling queues in the architecture we simulated. They are the instruction window which is used to buffer the instruction from the instruction cache. This is a FIFO queue. Another three queues are integer issue queue, floating point issue queue and load/store queue. In this order these three queues are each before the integer ALU, floating point ALU and data cache. The decoded integer instructions, floating point instructions and memory access instructions will be put in these three queues separately and wait to be selected for execution. Our control for each unit is based on the occupancy of the queue sitting before or behind the unit. From the arrow in the Figure 1, we can see the control for fetching engine domain is based on the queue occupancy of the instruction window, other such dependency relations can be found in the figure easily. For each domain, we assume there are thirty two frequency scaling steps available. We believe that our algorithm can be generalized to a variety of power-saving modes beyond multiple-clock-domain DVS, but that is beyond the scope of this paper.

In this paper we will investigate the property of two kinds of curves. This first kind of curve is the global performance versus the working speed of a specific unit curve with the frequency of all other units fixed, in the following paper we will call it as performance curve for that specific unit. The second kind of curve is the queue occupancy versus the working speed of a specific unit curve with the frequency of all other units fixed, from now on, we will call it as occupancy curve for that queue and unit. For these two kinds of curves, when we say a curve saturates with value x, we mean when the variable value is larger than x the absolute value of the derivative of that curve will be near zero. The minimal value of such x is called the saturation point for this curve. We will also investigate the property of the function P(v1,v2,v3,v4,v5) while v1-5 represent the working frequency of each unit in the processor and the function value is the global performance. We will call this function as P function in this paper.

Our dynamic control algorithm is based on our static analysis results. Static analysis is done as following. We divide program simulation into intervals, each interval is 10K instructions long. We run the simulation thirty two passes with the working frequency of one unit varying from one to thirty two and the other units' frequency fixed at their maximum. By recording the global performance and queue occupancy data for each running pass and each interval in the file, we can have the performance and queue occupancy curves for each unit and for each 10k instructions interval by reading the data in these files.

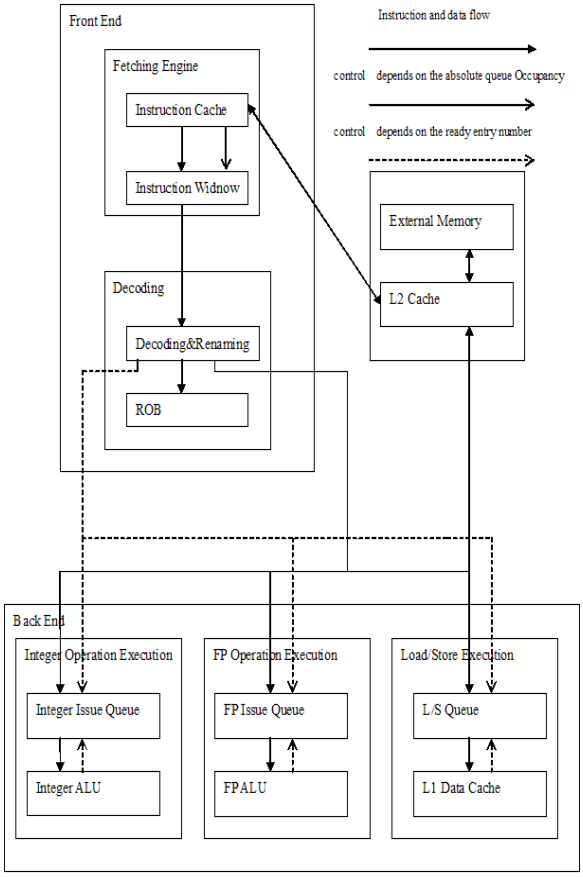

Figure 1: Domains in the superscalar processor

3.2 Performance and Occupancy Saturation as a Function of Unit Operating Speed

In this subsection, we will introduce our findings from static analysis which will be the basis of our dynamic control algorithm. Basically there are two findings. First, our static analysis indicates both the performance and the queue curve has saturation phenomenon in many cases. For any specific unit, the saturation point for its queue curve and performance curve normally coincide very well. Second we find the P function doesn't have local maximum point which is not globally maximal.

We can easily see from Figure 2 that the performance and queue curves for instruction cache and alu all have saturation phenomenon. Actually our experiments show it's common for all the units across the pipeline. We can also find from this figure that the saturation points of the performance and queue curve almost coincide for both alu and instruction cache.

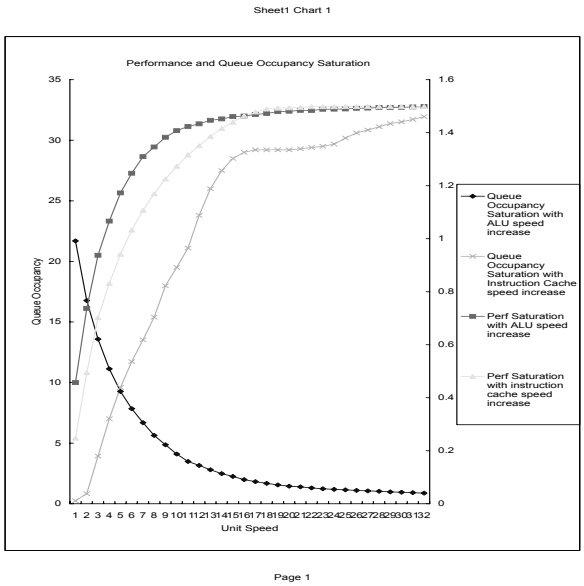

Figure 2: Performance and Queue Occupancy Saturation

It's not hard to understand why this saturation phenomenon exists. Let's consider a simple finite size queue with capacity N. It's easy to calculate out that the committing rate of this queuing system will be $(1 - \frac{1}{\sum_{j=0}^{N} \frac{L^j}{N}}) *$ if we assume the incoming rate and service rate , both with poisson pattern. Therefore, when we fix the service rate and change the incoming rate, finally this committing rate function saturate. Intuitively, this is due to the finite queue size. When the incoming rate is too high, the queue will hit the top of the queue too often so that the increase of the incoming rate will not convert to the final committing rate gain. On the other hand, when we fix the incoming rate and change the service rate, if the service rate is so large that the queue is empty quite often, then increasing the service rate will also not lead to quite a lot final committing rate gain.

However, the pipeline of the real processor is much more complex than this simple queue. First of all, in the real situation, the ILP(instruction level parallelism) can play an important role in the saturation. We can imagine in the situation when the ILP is very low, even if the fetching engine is working efficiently, the performance curve for the instruction cache will still reach its saturation region quite early. ILP is determined by many different factors, from the inherent program properties to the specific processor design choices like the issue logic (Normally the three issue queues as indicated in 1 are not simple FIFO queues).

More comprehensively, we did the following experiment to verify that the saturation point of performance curve and occupancy curve almost coincide. Based on our static analysis data in the

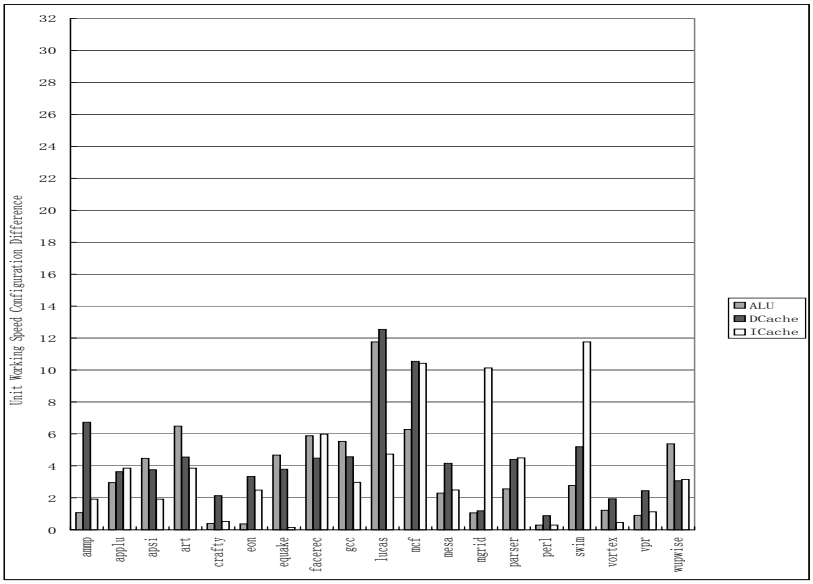

Figure 3: The control point difference of performance based static analysis and queue occupancy based static analysis

files, we can compare the difference between the saturation points of performance curve and queue curve for each interval. Averaging the results for all intervals of each benchmark, we get the results in Figure 3. We can see the absolute value of speed configuration difference is less than three for all ALU, Icache and Dcache for half of all the benchmarks. Other differences are normally less than six. With these two sets of configuration, the absolute value of global IPC differences as indicated by Figure 4 are all less than 0.06, normally around 0.02, which is negligible.

In addition to this queue saturation observation, we also make the following interesting finding on the P function which guarantees our final integration of the controls on the single unit will not lead to radical performance degradation due to positive feedback. This will be clear in the next section. We find the P function doesn't have any local maximum point that's not global maximal. Actually if there's no such points, when we set the initial working speed of each unit to the minimal, the following algorithm will terminate with the frequency configuration of the units under which the global performance is near the maximum of the P function value and each unit is working at its saturation point for its performance curve or at its maximal working speed. Why can't the performance with the termination configuration be far from global maximal? Because if we are in such a configuration and because there's no local maximal point that is not global maximal, at least one of the derivatives of the P function will not be near zero, which means this algorithm can't terminate with such a configuration. Otherwise, if there's local maximal point that's not global maximal, this algorithm may terminate with such a configuration.

```
Let's suppose the Initial frequency
of each unit is stored in an
array c[N]={c1,c2,...,cN}

do
  c_old = c
  for (i=1;i<=N;i++){
```

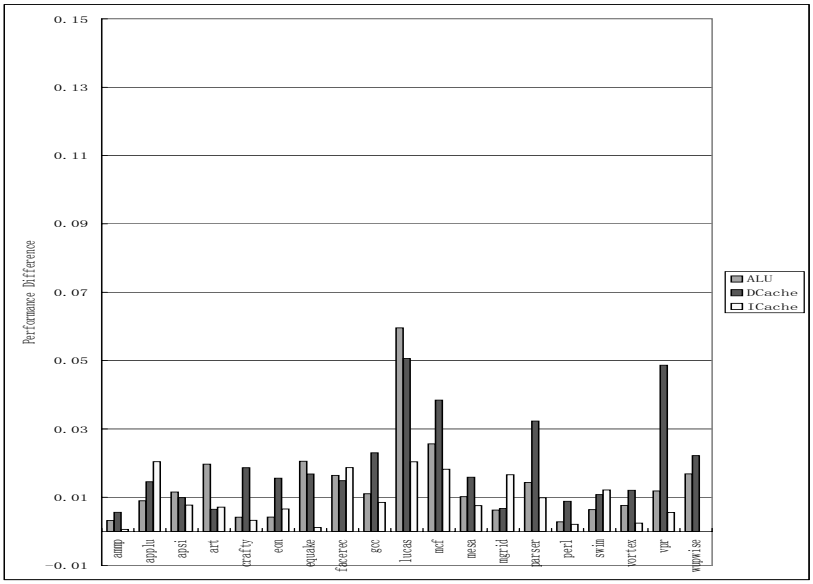

Figure 4: The performance difference of performance based static analysis and queue occupancy based static analysis

```
    Find the saturation point
    for unit i under current
    configuration, let's suppose
    the new configuration point for
    i is ci'

    c[i]=ci'
  }
until(c_old == c)
```

We can try to apply this algorithm on the two scenarios in Figure 3.2 3.2 and Figure 3.2 3.2. Let's suppose we are trapped in (2,2) configuration combination from the beginning. The first 2 represents the speed for instruction cache while the second 2 represents the speed for ALU. So in the first scenario, following the above algorithm, we will experience the following control path to reach a steady control point: $(2,2) -> (2,2) -> (2,4) -> (4,4) -> (4,8) -> (8,8) -> (8,16) -> (16,16) -> (16,26) -> (22,26) -> (22,32) -> (22,32)$ For the second scenario, we will experience the following control path to reach a steady control point: $(2,2) -> (4,2) -> (4,12) -> (4,12)$

In the first case, at the steady optimal control point, the ALU is working at its full speed and the instruction cache is running at less than its full speed. Even when the instruction cache is not running at its maximum speed, it's working at its saturation point while the ALU is not at its saturation working point although it's already working at its full speed. In the second case, both are working at less than their full speed and both are working in the their saturation regions. This means there's another factor that's the bottleneck of the processor other than these two units. In this case, the bottleneck is the memory speed because in this scenario, processor is experiencing high cache miss ratio. In other cases, the inherent program ILP can also be a performance limiting factor that will make the second scenario appear.

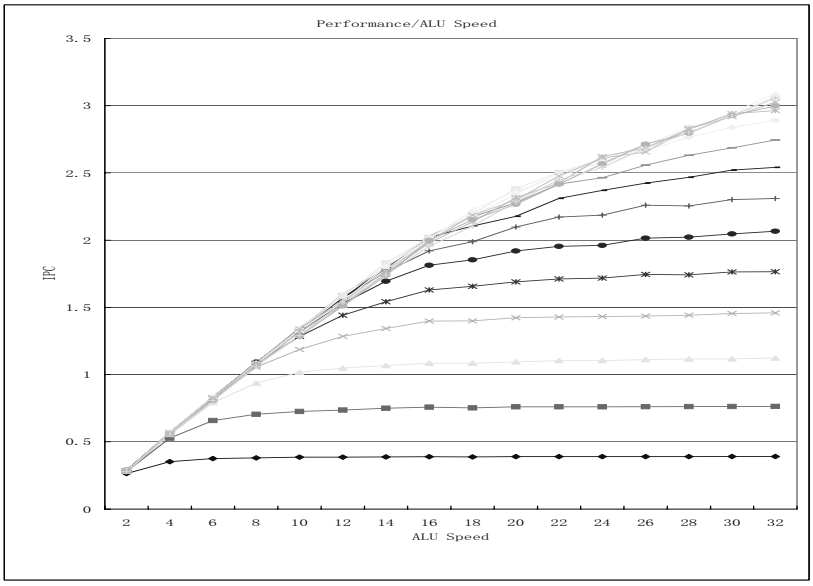

Figure 5: Why there's no radical degradation(scenario 1)

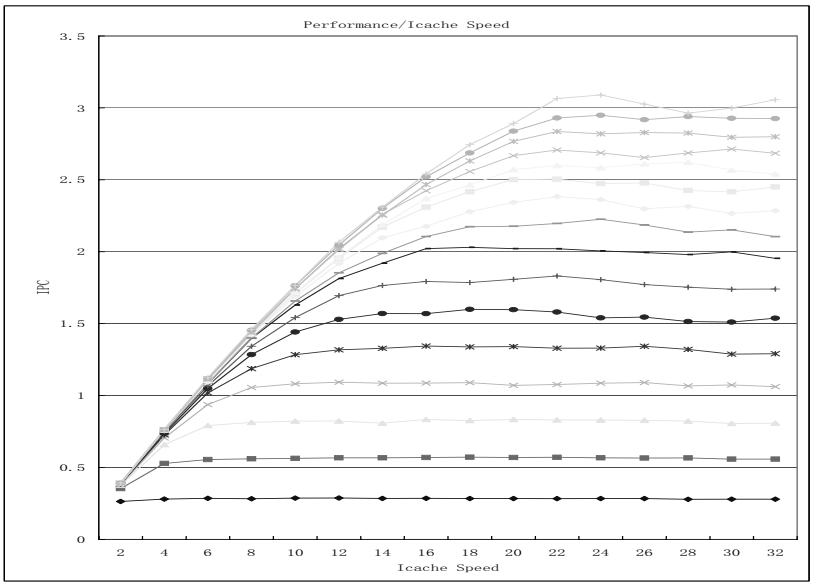

Figure 6: Why there's no radical degradation(scenario 1)

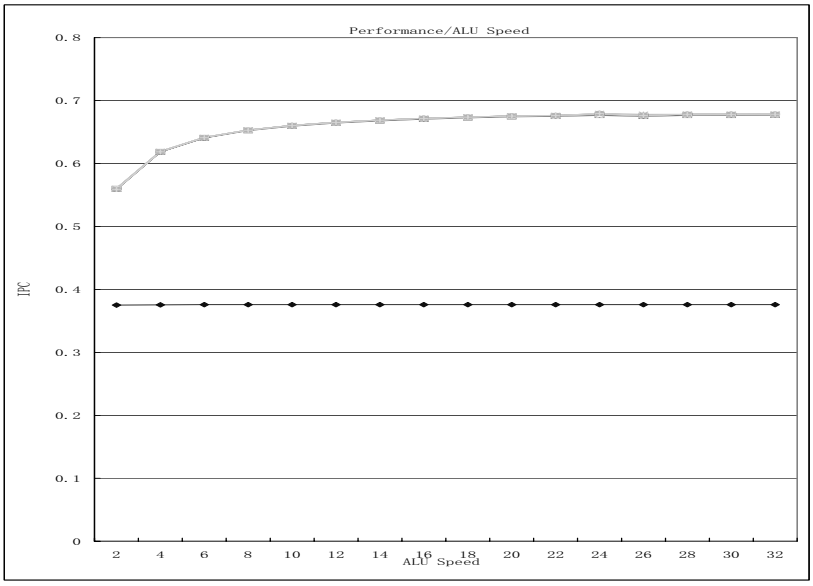

Figure 7: Why there's no radical degradation(scenario 2)

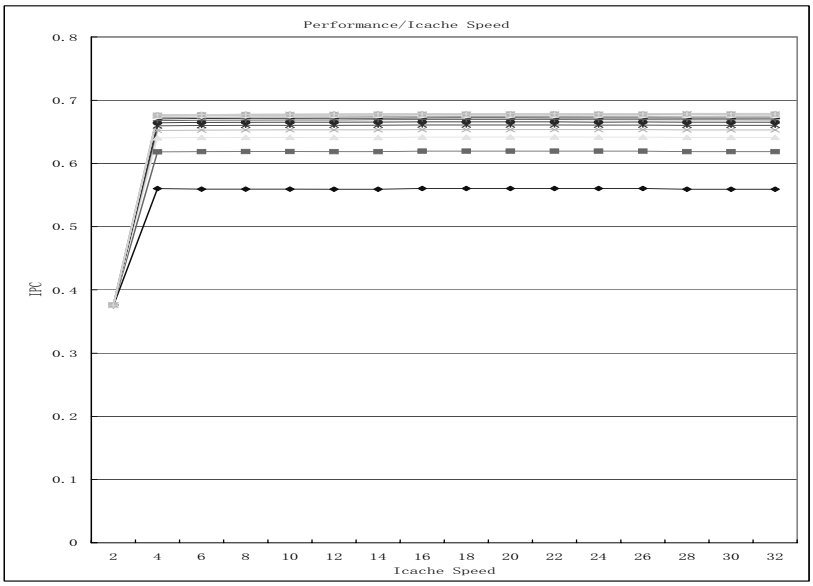

Figure 8: Why there's no radical degradation(scenario 2)

The observation in this subsection is the basis for our dynamic control framework. First, if we only want to dynamically tune the speed of a single unit and keep other units at their full frequency, we should let this unit working at the saturation point of its performance curve. If the speed of this unit is slower than the saturation point speed, the processor performance will be hurt too much; on the other hand, we waste energy unnecessarily. Second, in stead of deciding the saturation point based on IPC, we prefer the queue occupancy as the criteria. Its validity is guaranteed by the coincidence of the saturation points of the performance curve and queue occupancy curve. We will introduce the necessity of this choice in the next subsection. Finally, we point out all local maximal points in the P function are all global maximal points. This conclusion is crucial to guarantee there will be no unwanted positive feedback happening in the dynamic scheme.

3.3 The Framework

In this subsection, we will first introduce our dynamical control algorithm on the power modes control of a single unit and then we will introduce the integration scheme which will integrate these separate controls without negative interferences among these controls.

Our mechanism is based on monitoring the absolute value of the queue occupancy changing rate when we dynamically scale the frequency of the unit. If the changing rate is too large, it means this unit is working outside the saturation region and the whole processor is suffering dramatic performance degradation, on the other side, if the rate is very small, we are in the saturation region, this means we can reduce the speed of that unit to reach the saturation point.

The following is a typical algorithm for the control over the power modes of a single back end unit. This algorithm will be applied at the beginning of each interval. The Direction variable is used to track the control action of this algorithm taken at beginning of the previous control interval. HOLD means there's no action in the previous activation of this control algorithm because the unit is working at it's full frequency, DOWN means last time the action is decreasing the unit frequency by one step, UP means last time the action is increasing the unit frequency by one step. Combined with this information, we can decide what kind of action we'd like to carry out for this interval based on the difference between the average queue occupancy of the past interval and the interval before the past interval(CurrentQoccupancy and PrevQoccupancy). The threshold we actually used is: integer issue queue: 0.08,load/store queue: 0.25,floating point issue queue: 0.04, instruction window: 0.03.

3.3.1 Control Algorithm for One Unit

```
if (Direction == HOLD &&
CurrentQoccupancy - PrevQoccupancy
> Threshold)
     UnitSpeed -= SpeedStep;
     Direction = DOWN;
else if ((PrevQoccupancy -
CurrentQoccupancy > Threshold &&
Direction == UP) ||
(CurrentQoccupancy -
PrevQoccuancy < Threshold &&
Direction == DOWN))
    if (UnitSpeed < MAXIMUMSPEED)
        UnitSpeed += SpeedStep;
        Direction = UP;
    else
        Direction = HOLD;
else
    if (UnitSpeed > MINIMUMSPEED)
```

```
            UnitSpeed -= SpeedStep;
            Direction = down;
        else
            UnitSpeed += SpeedStep;
            Direction = UP;

    PrevQocupancy = CurrentQoccupancy;
```

Actually, the way we used in this section to find out the saturation point for a unit is somewhat different from the way we used in static analysis. In the previous section, we are comparing the queue occupancy difference or performance difference for the same interval of instructions. For dynamic control ,we can never do that. We can only compare the difference for two adjacent intervals. However, as indicated by [8], the architectural property like IPC, queue occupancy will not change much from interval to interval when they are in the same program phase. That's why our dynamic control can find out the saturation point for any specific unit in one program phase.

On the other hand, even when we don't apply any control and for the same program phase, the queue occupancy and IPC will change slightly from interval to interval, we call such fluctuation noise. And we noticed the fluctuation is minimized when the interval duration is around 100k instructions. Because of this inherent noise problem, we should carefully select our control granularity(the unit frequency changing step) and the threshold. If the control granularity is too big, the deviation from the optimal point will be too large because our control always force the unit speed oscillating around the saturation point instead of being fixed at the saturation point. On the other hand, due to the existing of noise, we can't use too fine control granularity in our framework, or the noise will dominate the control. We will see in the next section, in the cache-decay mechanism put forward in [12], the granularity for decay is too small to be effective.

Now we introduce how to integrate the separate controls. When we are trying to integrate the controls, we must overcome the negative interferences among these controls. For example, the control for the instruction cache unit will rely on the occupancy information of the instruction window. However, changing the speed of ALU will also change the queue occupancy of the instruction window too.

To decoupling these kinds of interactions, we apply the control for only one of the instruction cache, decoding unit and the execution core(integer ALU, floating point ALU and data cache) each period. So each period only one part is under control and the other two parts remain their frequencies. This is done as the following algorithm indicates. The ControlPeriod variable is the cycle number in one interval and the MAINTAINTIMES variable is the interval numbers that are needed to apply control for a specific unit before the algorithm switch to control another unit. So each period has ControlPeriod * MAINTAINTIMES cycles. There are also unwanted interactions among the controls for data cache, integer ALU and floating point ALU, but as [12] indicated, this kind of interaction is not very strong, so we can turn on the controls for these three units simultaneously. We can see this integration actually works in the same way as the algorithm does in the previous subsection, so radical performance degradation will not happen, this is guaranteed by the property of P function we introduced in the previous subsection.

```
if (cycles < ControlPeriod * MAINTAINTIMES)
   Apply control for icache
else if (cycles <
   ControlPeroid * MAINTAINTIMES * 2)
   Apply control for decode
else if (cycles <
   ControlPeriod * MAINTAINTIMES * 3)
   Apply controls for ALU and Data Cache
```

Now it's clear why we base our control mechanism on queue occupancy in stead of IPC, if we want to rely the control on IPC, we must also apply the controls for these three units one by one

in stead of simultaneously because the control for one back end unit is coupled with the control for another back end unit by IPC tightly. We don't want the time between the two control phase for the same unit too long, otherwise the system will not be reacting quickly enough, especially when the program has short phase. However, because there are three queues before the back end, the queue occupancy based controls only have interaction through different issue queues before them, this kind of interaction is much weaker. Another reason to use queue information as a guide for control is that the queue information is a kind of local information compared with the global IPC.

4 Simulation Setup

Our work is based on an enhanced Wattch [3] simulator. Wattch is a cycle by cycle power estimator based on simplescalar performance simulator. As we know, original simplescalar simulator adopted a unified buffer to simulate all the function of issue queue, load/store queue and reorder buffer in modern superscalar processor. However, this is far from the real situation. We changed the simplescalar out-of-order simulator so that the enhanced simulator has separate integer issue queue, floating point issue queue, reorder buffer and load/store queue. The power model is also changed accordingly. Such change is necessary both for reflecting the real modern superscalar processor implementation and for our power modes control mechanism because our mechanism is based on those queues related information. We use the configuration in table 1 throughout our experiments, the benchmarks we used are from spec2000cpu benchmarks.

Table 1: Configuration

Parameter	Configuration
Fetch width	8
Decode Width	8
Integer instruction issue width	8
Floating point instruction issue width	4
load/store instruction issue width	4
ROB Size	128
Load/Store Queue Size	32
Integer Issue Queue Size	40
Floating point Issue Queue Size	32
Instruction Window Size	32
L1 I-Cache	32K, 2-way associativity
L1 D-Cache	32K, 2-way associativity
L1 Cache Latency	1 cycle
L2 Unified Cache	1M, 1-way associativity
L2 Cache Latency	12 cycles
Memory Latency	200 cycles
Inter-chunk Memory Latency	2 cycles
Integer-ALU	8+2MULT/DIV
FP-ALU	4+2MULT/DIV
BTB/Predictor	Hybrid
Global component	8K entries, 12-bit history
Bimodal predictor size	8K entries
Comb predictor size	8K entries
BTB	4K entries, 2-way associativity
Branch Misprediction Penalty	6 cycles

5 Results and Discussion

We measure the performance, energy and energy-delay data in five situations. The results are in Figure 11 12 13. In these figures, for each benchmark there are five groups of data. As shown in the figures, "Static" means we get the optimal working frequency for each unit from static analysis and then apply this frequency configuration to get the results. "Dynamic" is for our control mechanism. "Naive" is for the simple control integration mechanism, in such integration, we simply turn on the control for all units simultaneously, "attack-decay" is for the mechanism put forward in [12]. "all" indicates we are applying the control for all the units. "front end" means we only apply the control for the units in the front end. "back end" means we only apply the control for the units in the back end.

From these experiment data, we can conclude that our integrated dynamic control is getting greater than 15% energy delay product improvement for most benchmarks. For some benchmarks like applu, because the inherent IPC is low in this benchmark, we can reduce both the back end and front end running speed radically to get great energy savings while losing very little performance. In such cases, the energy delay improvement can be as high as 70%. In most cases, the integrated way get better energy-delay product improvement than just turning control for the front end or for the back end and no radical performance degradation is observed in integration. As expected, the energy-delay improvement of the naive scheme can't parallel our dynamic scheme because sometimes the front end and back end interact radically so that the control points of this scheme deviate from the optimal point further.

Compare the static results with dynamic results in Figures 11 12 13, we find with similar energy savings, the dynamic way does suffer more performance loss than static results. Besides the noise, one major factor that's leading to this result is that we are always trying to oscillate around the optimal control point instead of just staying at that point. With relatively large control granularity, this kind of oscillating will lead to worse performance with the same energy savings compared with the static way.

Semeraro in [12] put forward a dynamic control mechanism for power modes regulation for the ALU and data cache. We implement their algorithm in our simulator. We find this algorithm suffers from noise problem in our simulation environment. They use a very fine unit frequency changing granularity, as we argued, this granularity should not be too small so that the control is overwhelmed by the noise. From the figure 10, we can see the attack times/decay times ratio is very high, normally around 10, so actually the attack is dominating the control. This means the decay is not working in most cases.

The Semeraro algorithm without decay still differs from our algorithm in some scenario. In our algorithm, if increasing unit speed leads to significant queue occupancy decrease, it means we are still walking on the steep part of the queue occupancy curve which indicates we should continue increasing the speed of that unit until the decreasing of queue occupancy is small. However, in this situation, the semararo cache/decay algorithm will just stop increasing the speed of that unit when that algorithm observes significant queue occupancy decrease. Actually if the algorithm is always working around its optimal point, this is not a serious problem because only one step speed change is enough to pull the unit back to its optimal working point. However, if the unit deviates from its optimal working point a lot (due to sudden program phase change or noise), semararo cache/decay algorithm lacks necessary robustness to go back optimal working point by continuing increasing the unit's speed. In Figure 5, we compare the control situation of our algorithm and the attack-decay algorithm for a typical running segment in which the attack-decay is trapped in poor unit working point. Due to static analysis, the optimal unit working speed for this segment should be fifteen, we intentionally set the working speed of that unit at two for our algorithm, we can see finally our algorithm successfully jump out of this non-optimal control region and get back to around the optimal point. The attack/decay keeps working around speed eight and can't get back to the optimal working point. We tried different threshold setting and get the same result, actually the incapability of continuous speed increasing is irrelevent of threshold setting. We can see from Figure 11 12 and 13 that the attack-decay does suffer great performance degradation which we believe is due to the

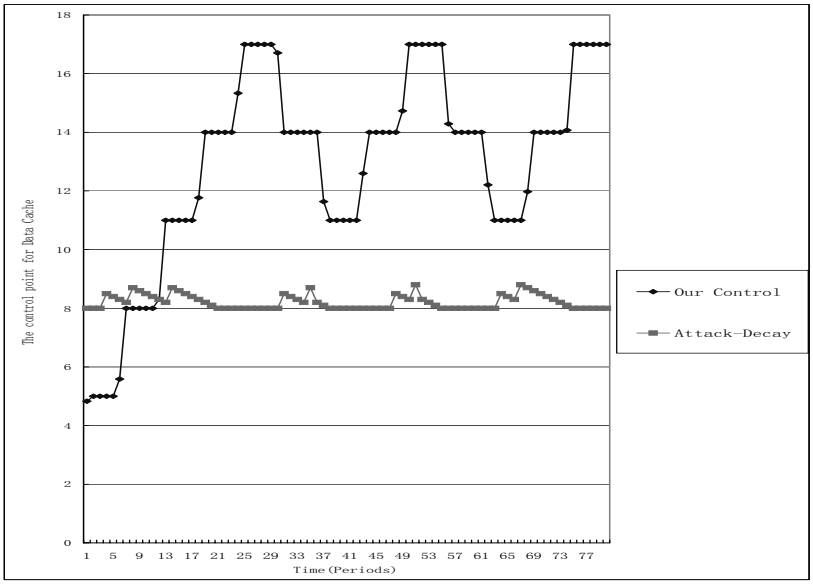

Figure 9: The Problem in the Attack-Decay Control

incapability of continuous units frequency elevation.

6 Conclusions and Future Work

In this paper, we explore the possibility to use local queue information for integrated power modes control across the whole pipeline. The static experiment shows the coincidence of the saturation points of the queue occupancy curve and the performance curve for a specific unit. This observation is the foundation of the dynamic control for the power modes of a single unit. The static experiment also shows that there's no local minimal point which is not global maximal on the P function curve (global performance/units frequency), this observation guarantees our integration of power modes control for single unit will not lead to radical performance degradation due to positive feedback. Our final simulation for the dynamic control results show our mechanism has better robustness than attack-decay mechanism when recovering from units frequencies configuration deviating from optimal control point too much which will lead to severe performance loss. Normally our dynamic mechanism can get greater than 15% energy-delay improvement. Integrating the control for the front end and the back end normally get better or the same energy-delay saving than just turning on the control for the front-end or the back end, which indicates our integration is effective.

Besides these positive results, we also get some negative results which don't coincide with previous research work very well. First, the inherent noise of the queue occupancy from one interval to another forces us to select a relatively larger control granularity. And our experiment also shows that this noise problem makes the attack-decay algorithm actually work with almost no decay action in the real situation. However, larger control granularity makes dynamic control deviate from the optimal

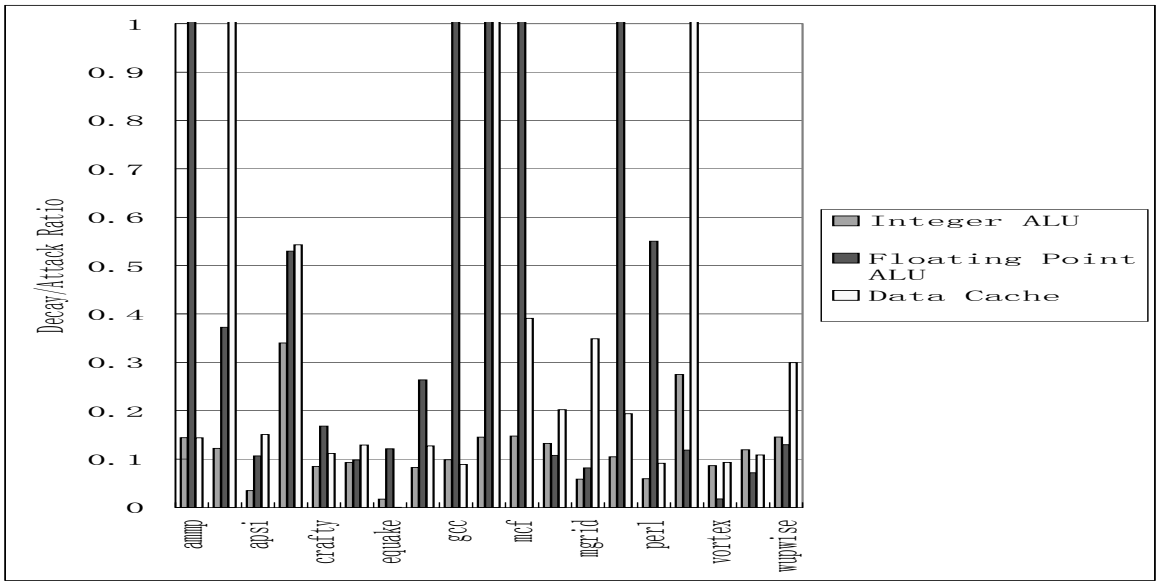

Figure 10: Attack/Decay Ratio

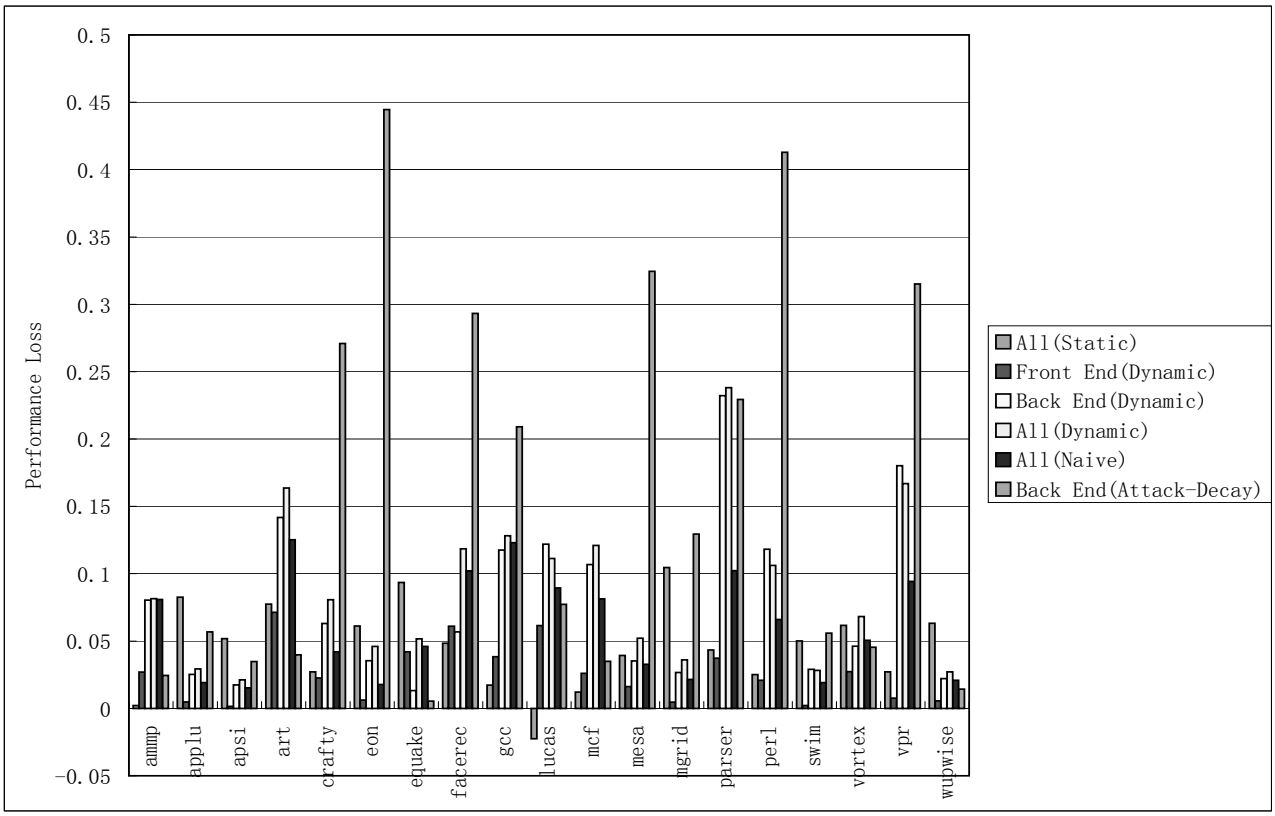

Figure 11: The Performance Loss

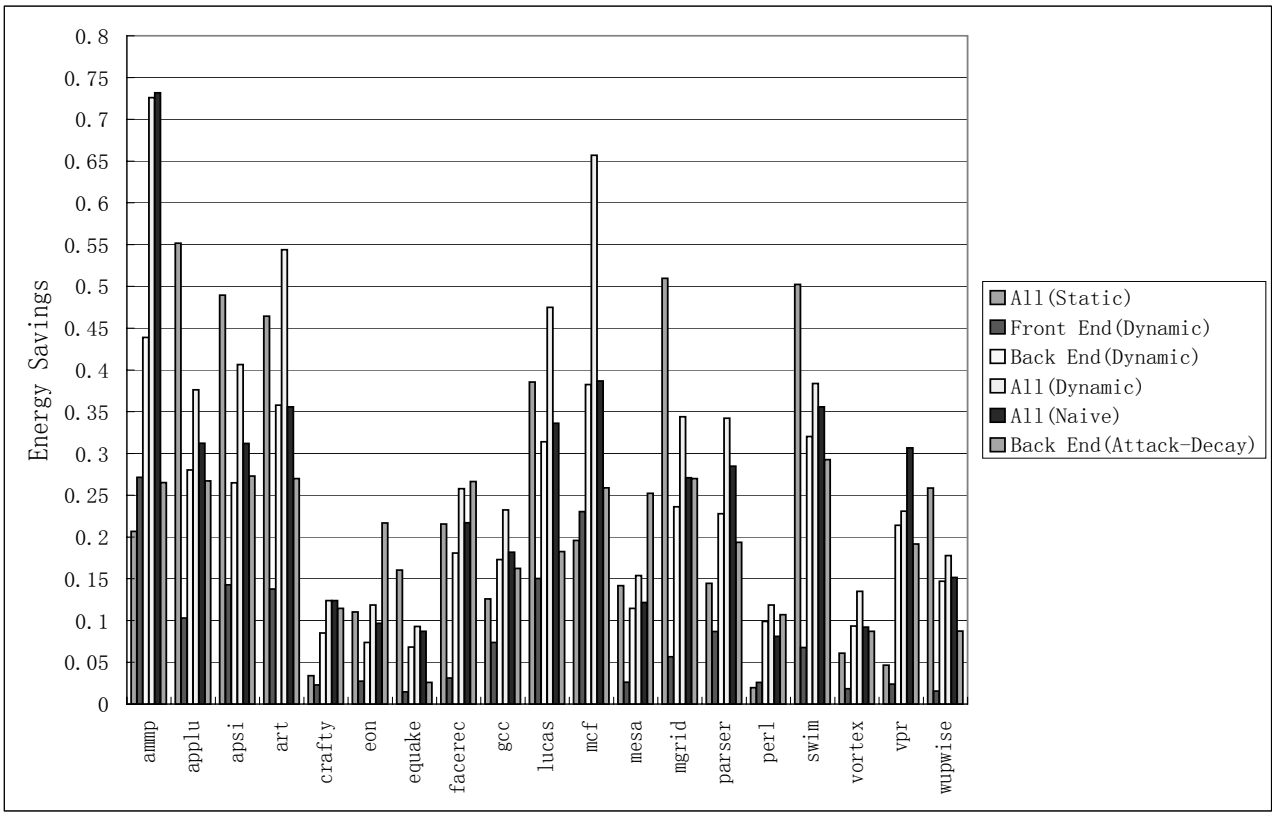

Figure 12: The Energy Saving

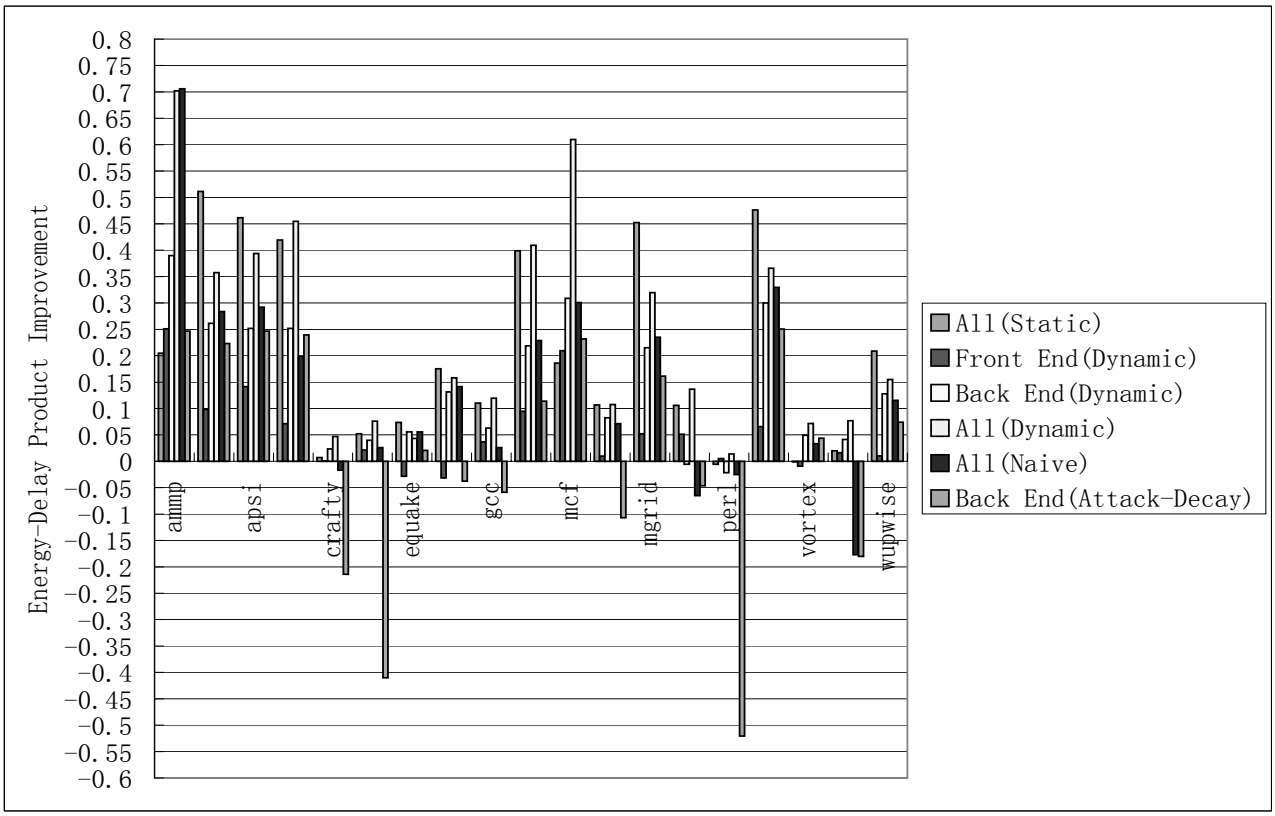

Figure 13: The Energy-Delay Product Improvement

control point further. Researchers interested in queue occupancy based power modes control should find an algorithm that can be immune from the noise affect and can keep as close to the optimal control point as possible. Second, our static results can't parallel the dynamic results from [12]. Especially for integer benchmarks, our static results show much less opportunity for getting energy savings by exploiting the differences of units working speed and the difference between units working speed and the available ILP. The static analysis method in [12] adopted "shaker" algorithm which took critical path information into consideration. Our static analysis is relatively simple.

Although we can't explain why we get some different results. The following factor can be the candidate of the origins of the differences. We had different simulation platform compared with previous works, for example, the attack-decay work was done on a GALS architecture simulator framework. We don't know if this difference can explain the incoincidence in the results.

There are several possible future work. In our current work, the only power modes of each unit we explored are frequency and voltage scaling. There are lots of other power saving techniques, like the dynamic cache resizing. How to dynamically apply these possible power modes besides frequency and voltage scaling is an open question.

Our work didn't take critical path information into consideration. [13] did some work on it, however, how to use critical information for power modes control across the whole processor is an untouched research problem.

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *32th International Symposium on Microarchitecture*, 1999.

[2] R. I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronics and Design*, 1998.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, 2000.

[4] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems, held at the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[5] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. L. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *11th International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[6] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *International symposium on computer architecture*, 2003.

[7] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proc. IEEE Design, Automation and Test in Europe Conf.*, 2001.

[8] T. Karkhanis. Saving energy with just in time instruction delivery. In *International symposium on Low power electronics and design*, 2002.

[9] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *8th International Symposium on High-Performance Computer Architecture*, 2002.

[10] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *34th International Symposium on Microarchitecture*, 2001.

[11] M. D. Powell, A. Agrawal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *34th International Symposium on Microarchitecture*, 2001.

[12] G. Semeraro, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *35th International Symposium on Microarchitecture*, 2002.

[13] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *34th International Symposium on Microarchitecture*, 2001.

[14] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanovic. Direct addressed caches for reduced power consumption. In *34th International Symposium on Microarchitecture*, 2001.