

# Power and Energy Reduction Via Pipeline Balancing

R. Iris Bahar\*  
Brown University  
Division of Engineering  
iris\_bahar@brown.edu

Srilatha Manne  
Compaq Computer Corporation  
VSSAD/Alpha Development Group  
srilatha.manne@compaq.com

## Abstract

Minimizing power dissipation is an important design requirement for both portable and non-portable systems. In this work, we propose an architectural solution to the power problem that retains performance while reducing power. The technique, known as Pipeline Balancing (PLB), dynamically tunes the resources of a general purpose processor to the needs of the program by monitoring performance within each program. We analyze metrics for triggering PLB, and detail instruction queue design and energy savings based on an extension of the Alpha 21264 processor. Using a detailed simulator, we present component and full chip power and energy savings for single and multi-threaded execution. Results show an issue queue and execution unit power reduction of up to 23% and 13%, respectively, with an average performance loss of 1% to 2%.

## 1. Introduction

The primary goal of state-of-the-art general purpose processor design is performance. Given the large number of targeted applications, the processor is designed to achieve the best performance on the greatest number of applications. Therefore, most general purpose processors contain resources that are appropriate to a subset of programs but not all programs. Furthermore, enough variation exists within the execution of each application such that the full architectural features of the processor are required only for a portion of the program's execution. In this work, our goal is to determine the changing needs of each program and tune processor resources to the program with the aim of reducing power dissipation.

Previous work also addresses methods to reduce energy consumption by tuning the microprocessor for program needs. Albonesi suggested balancing clock rates with architectural features in [1], and modifying cache associativity for the individual needs of a program in [2]. Manne [14] showed that energy can be reduced without impacting performance by restricting instruction fetch when the machine was likely executing wrong-path instructions.

Wilcox notes in [18] that out-of-order issue logic and execution dissipated a significant portion of the power on the Alpha 21264

\*This work was completed while on sabbatical at Compaq Computer Corporation and was supported in part by an NSF-CAREER grant number MIP-9734247.

processor. Our work, henceforth referred to as *Pipeline Balancing*, or *PLB*, targets these high power components by adjusting the pipeline issue and execution capabilities to the varying needs of the program. Unlike [14], we address both integer and floating point programs, as well as multi-threaded execution. Furthermore, *PLB* does not require complex implementation of clocks to save power as required by [1].

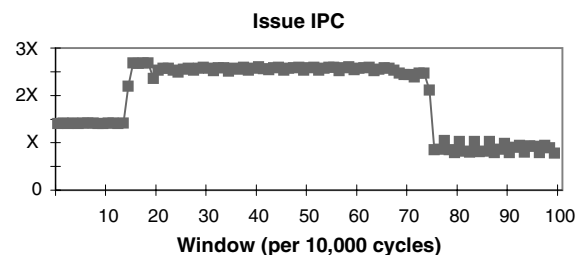


Figure 1. Issue IPC variation for the *psi* benchmark.

Wall showed that the amount of instruction level parallelism (ILP) within a single application varies by up to a factor of three [17]. Figure 1 shows an example of the issue rate over time for the SpecFP95 benchmark *psi* running on an 8-way superscalar, out-of-order processor. Each plot point represents the average number of instructions *issued* per cycle (issue IPC) over a window of 10000 simulated cycles. The issue IPC varies by a factor of three over a million cycles of execution. The variation may result from either the inherent program behavior or the behavior of the program within the constraints of the hardware (e.g., the data size of the program relative to the cache size). Regardless, these issue variations can be exploited to reduce power dissipation. In the case where *psi* does not require the full issue capabilities of the processor, a lower issue width machine would have been appropriate, while the high issue rate might require a 6 or 8 wide machine.

Ghiasi *et al.* also exploits IPC variations to reduce energy consumption in processors [10]. In her work, the operating system (OS) indicates the IPC requirement of each program, and the processor is adjusted to meet OS requirements. The base microarchitecture is an out-of-order, 4 or 8 way superscalar machine that can be modified to either gate the pipeline as proposed in [14], or change from an out-of-order to an in-order machine. The determi-

nation of when to switch modes is based on comparing the commit IPC of the processor to the optimal IPC requested by the operating system. Ghiasi did not extrapolate on the hardware required to change from in-order to out-of-order execution. One way to implement this feature is to have two issuing structures on the processor, one being out-of-order and the other being in-order. The out-of-order structure is clock gated when the processor switches to in-order issue and vice-versa. Switching queues requires the out-of-order queue to be drained of all instructions before switching to the in-order queue. This can take a non-trivial amount of time, depending on the number of instructions already in the out-of-order queue. Ghiasi measures performance as the overall frame rate of the MPEG benchmarks she uses. In the rate controlled workloads, there is slack time available at the end of many frames. She can slip slightly on each individual frame and not pay any penalty on the overall frame rate because of the slack.

As with Ghiasi's work, *PLB* also exploits IPC variations to reduce energy consumption. However, we work purely at the microarchitectural level without any OS guidance, and we take advantage of both performance variations between different programs and fine grained variations within a given program. Furthermore, we only modify the issue width of the machine and do not modify the scheduling algorithm from out-of-order to in-order issue. The fine grained switching required in *PLB* and the type of benchmarks we use (Spec95) make it difficult to absorb the performance cost of transitioning from out-of-order to in-order issue.

The *PLB* algorithm is quite simple. First, we monitor the issue needs of the program. When the program does not require the full issue capabilities of the processor, we reduce the processor issue width and enter the *low-power* mode. When the program switches behavior again, we return the processor to normal operation. Figure 2 shows the pipeline of the simulated superscalar processor. It contains 8 execution pipelines that execute a combination of integer, floating point and memory operations. When executing normally, all 8 pipelines are in use. When in low-power mode, some or all the shaded regions of the figure are disabled, resulting in power savings. Details about the monitoring scheme, pipeline design, and power savings from disabling the pipelines are provided in latter sections.

Implementation of *PLB* has some additional requirements. The monitoring scheme for *PLB* must be simple since the solution cannot be more power hungry than the problem. Also, *PLB* cannot sacrifice performance to reduce energy. First, the market demands high performance from state-of-the-art processors. Second, energy is a function of the total execution time of the program, and reducing power dissipation by increasing execution time does not always result in overall energy savings [4].

The rest of the paper discusses the implementation details and results of the *PLB* technique. Section 2 details the metrics and analysis techniques used in *PLB*. Section 3 discusses the implementation of the issue queue, and analyzes the potential power savings with *PLB*. Results are presented in Section 4, and Section 5 concludes the paper.

## 2. Implementation

We used the ASIM [7] simulation infrastructure to model a cycle level, execution driven simulator that generates performance

and activity numbers. We modeled our baseline processor on the Alpha 21264 [6] and next generation processors cited in literature [5] [8]. The model is an 8-way issue, out-of-order machine with a 128 entry instruction queue, capable of simultaneously executing multiple threads from different processes. Similar to the 21264, we have separate instruction issue and commit queues. We doubled the functional unit resources of the 21264, and allow up to 8 integer, 4 floating point, and 4 memory operations to execute per cycle. The 21264 processor has a seven stage pipeline and currently operates at frequencies around 900MHz. However, given the increase in issue width from 4 to 8 and the need for operating frequencies of 1.5GHz and higher in the next processor generation, we modeled the pipeline length to be comparable to the 20 cycle, super-pipelined, Pentium 4 processor from Intel [11]. Although the simulation infrastructure we chose limits the study to a specific architecture implementation, the benefit is a higher level of accuracy in the final performance and power numbers.

### 2.1. Disabling resources

The architecture of the issue logic, shown in Figure 2, determines what resources get disabled when in low-power mode. Algorithmically speaking, *PLB* could reduce the issue rate from 8 down to a single instruction issued per cycle. However, our specific implementation of the machine requires that to retain a full set of functional operations, i.e., to perform all the operations required by the ISA, we need to have at least one cluster of functional units active. Therefore, an issue rate of 4 per cycle is the lower bound for restricting issue width. A 4-way issue machine is also the simplest modification to make since it is easy to disable one cluster of functional units as shown in Figure 2.

We implement *PLB* with two possible issue widths when operating in low-power mode: 4-wide issue and 6-wide issue. We save more power when in 4-wide issue, but we also do not enter this mode very often, if at all, in some programs. On the other hand, we have less power savings in the 6-wide issue mode, but enter this mode more frequently and across many programs. There is a further complication to the 6-wide mode that involves which functional units to disable. In 6-wide mode, we disable two sets of functional units. In our simulated machine, we split the functional units between the clusters so that each cluster executes a maximum of 4 instructions per cycle consisting of up to 2 floating point operations, 4 integer operations, and 2 memory operations. All four issue slots can handle integer operations, while two of the slots handle memory operations and the remaining two handle floating point operations. If we disable half the issue slots, we must be careful which operations we remove or we end up with a machine that is unbalanced for many applications. We could either disable the 2 floating point units, the 2 memory units, or one of each. We decided to disable the 2 floating point units in 6-wide mode and leave the memory units intact because memory bandwidth is critical for both floating point and integer performance. To remedy problems we face in some benchmarks due to reduced floating point bandwidth, we added a secondary release mechanism as discussed below.

### 2.2. Triggering *PLB*

*PLB* uses enabling and disabling conditions to determine when

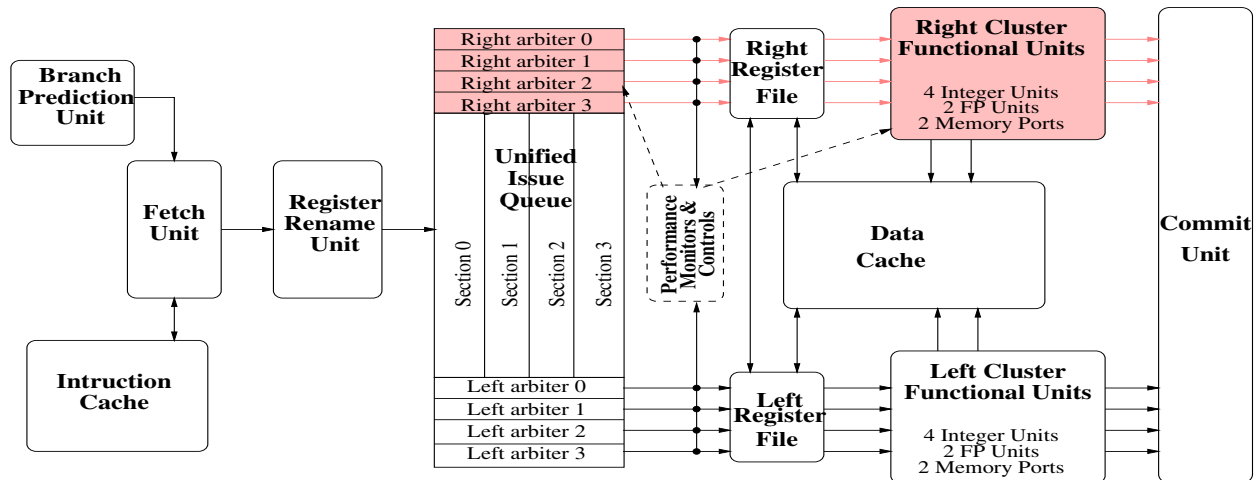


Figure 2. Pipeline organization of 8-wide issue processor.

to enter and exit low-power modes. The conditions are a function of primary and secondary thresholds that were determined through experimentation and analysis. Figure 3 shows the state machine for transitioning between normal issue and low-power modes. The enabling and disabling conditions for 4-wide and 6-wide are  $EC_{4w}$ ,  $DC_{4w}$ ,  $EC_{6w}$ , and  $DC_{6w}$ , respectively. Note that we can transition from normal issue to either of the low-power modes. However, when we disable either the 4-wide or 6-wide mode, we transition immediately to normal mode in order to minimize potential performance loss.

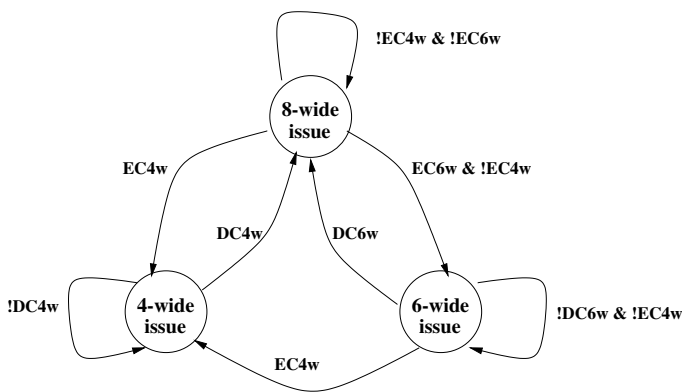


Figure 3. State machine for enabling/disabling PLB. Transition conditions are based on our chosen threshold values.

**Sampling window** The basic premise of the PLB algorithm is that past program behavior indicates future program needs. Therefore, based on past program characteristics such as issue IPC ( $I_{IPC}$ ), we predict the future resource needs of the program. To smooth out short bursts of activity, we measure these values over a fixed sampling window. In addition, to reduce spurious transi-

tions between low-power and normal modes, we require that the pipeline stay in the current mode for at least one sampling window before switching modes. Hence, the sampling window has to be large enough to handle short bursts of high or low issue IPC without unnecessarily enabling or disabling PLB, but small enough such that we do not lose performance by staying in low-power mode too long after the program changes behavior.

To guide us in our window size selection, we ran experiments that varied window size from 64 to 512 cycles. We found that for several integer benchmarks such as *gcc* and *xlisp*, performance suffered more with increased window size. The long window interval did not provide the opportunity to switch issue width in a timely fashion when the pipeline state changed due to branch mispredictions. This is critical since correct path instructions fetched after a branch misprediction generally have high ILP, requiring the full issuing capabilities of the machine. Since energy is a function of both power and performance, this negatively impacted our main objective of energy reduction. On the other hand, with a small window we were too sensitive to small program variations in benchmarks such as *gcc* and *applu* and were not able to exploit low power modes as well. Based on these findings we found a window size of 256 cycles to be a good compromise.

**Primary and secondary triggers** We use performance triggers to determine when to enter and exit the low-power modes. The enabling and disabling conditions are a function of these triggers and their associated thresholds. The primary trigger, and the one that achieves most of our energy savings, is issue IPC ( $I_{IPC}$ ). We use issue IPC instead of commit IPC because the pipeline modifications we focus on are in the issue logic, and commit IPC is not necessarily indicative of the issue needs of the program. Wrong-path and out-of-order execution, for example, can perturb the issue state of the pipeline while the commit state shows no discernible change.

We also use two secondary triggers, *floating point issue IPC* ( $FP_{IPC}$ ) and *mode history*, to improve performance in some of the benchmarks. In 6-wide mode, we reduce the floating point

bandwidth from 4 to 2 instructions per cycle, resulting in an unbalanced machine. Programs such as *tomcatv*, however, require a balanced machine and the full floating point bandwidth for much of its execution. These programs show a relatively low  $I_{IPC}$  when in 6-wide mode because their performance is limited by the floating point bandwidth. Therefore, monitoring both  $I_{IPC}$  and  $FP_{IPC}$  helps reduce the performance loss in these floating point programs.

The other secondary trigger, mode history, reduces spurious transitions from one issue mode to another. If the program behavior varies from low ILP to high ILP frequently, we lose performance because we spend a large portion of the program execution time in low-power mode waiting to transition to normal issue mode. To alleviate this problem, we use a technique common to branch prediction. We do not transition to 4-wide issue unless we see two consecutive sampling windows that satisfy the other triggering conditions. The mode history bit is incremented every time we meet the necessary conditions, and reset to zero once these conditions are no longer satisfied. We could have used a similar mechanism for enabling 6-wide mode, but chose not to because the performance penalty for spuriously entering 6-wide mode is not as great as the performance penalty for entering 4-wide mode.

**Enabling and disabling trigger thresholds** Threshold values are constrained by the issue limits of the machine. For instance, we should not enable 4-wide mode if  $I_{IPC}$  is greater than 4 since we know we would never be able to maintain this performance in this mode. On the other hand, determining when to return to normal issue mode is not so straight forward. When in 4-wide or 6-wide issue, we can never accurately determine if we should return to normal issue mode because we never exceed the restricted issue width of our current low-power mode of operation. Therefore, the thresholds chosen for disabling low-power mode are a function of the maximum issue width possible in the current mode. Noting these constraints, we chose the actual threshold values based on empirical data obtained from experiments with varied thresholds.

The values shown in Table 1 produce the best power savings with the least performance loss, on average, across all benchmarks. The table describes the enabling and disabling conditions for the selected mode as a function of primary and secondary triggers and their chosen thresholds. For example, we enable 4-wide mode if  $I_{IPC}$  is less than 3.0 and  $FP_{IPC}$  is less than 1.4, and return to normal operation from 4-wide mode if  $I_{IPC}$  exceeds 3.2 or  $FP_{IPC}$  exceeds 1.6. Note that Table 1 shows different enabling and disabling threshold values for each trigger. We found that this added flexibility allows us to better tune *PLB* for optimal performance and power. A mechanism which dynamically sets threshold values may lead to improved results; however, this is beyond the scope of this work.

### 2.3. Monitoring *PLB* triggers

For *PLB* to be worthwhile, the performance monitoring scheme required to dynamically enable/disable low-power mode must be simple and energy efficient. The hardware for *PLB* consists of a counter and two comparators for each IPC trigger monitored (issue IPC, floating-point IPC), and a 2 bit counter for mode history. We also need registers to hold information indicating current mode

Trigger	Threshold Values
$EC_{4w}$	$(I_{IPC} < 3.0)$ AND $(FP_{IPC} < 1.4)$ AND (mode history of 2 Consecutive Windows)
$DC_{4w}$	$(I_{IPC} > 3.2)$ OR $(FP_{IPC} > 1.6)$
$EC_{6w}$	$(I_{IPC} < 4.5)$ AND $(FP_{IPC} < 1.4)$
$DC_{6w}$	$(I_{IPC} > 5.0)$ OR $(FP_{IPC} > 1.6)$

**Table 1.** Enabling and disabling conditions for different low-power modes. EC refers to enabling conditions, and DC refers to disabling conditions.

of operation for the pipeline. At the beginning of each sampling window, the issue width counters are reset. Each cycle we increment the counters by the number of instructions of the specified type issued during that cycle. At the end of the sampling period we compare each counter value to the preset enabling and disabling threshold values. If the machine is currently operating in normal, 8-wide mode, then we use the enabling comparators' results to determine if we should enable 6-wide or 4-wide mode<sup>1</sup>. Likewise, if we are already in one of the restricted issue modes and the counter values are greater than one or more of the appropriate disabling threshold(s), we disable low-power mode. These counters and comparators add a negligible amount of power to the base processor.

### 2.4. Measuring the accuracy of *PLB*

To measure the validity of this assumption that past behavior is a good indicator of future behavior, we borrowed a diagnostic mechanism used to measure the usefulness of various confidence estimation techniques for branch prediction [13].

		Predicted Mode	
		$P_L$	$P_N$
Resulting Mode	$R_L$	LL	NL
	$R_N$	LN	NN

**Table 2.** Quadrant table for analyzing the accuracy of the *PLB* prediction mechanism.

Table 2 shows the quadrant table similar to the one used in [13] to describe the conditional probabilities which occur between the estimated correctness of a branch prediction and the resolution of that branch. In this table,  $P_N$  and  $P_L$  refer to *predicted normal mode* and *predicted low-power mode* while  $R_N$  and  $R_L$  refer to *resulting in normal mode* and *resulting in low-power mode*, respectively. Note that low-power mode refers to both 4-wide and 6-wide modes. For *PLB*, the quadrant table shows the intersection between the prediction of low-power mode or normal mode of operation at the end of the previous sampling window, and the

<sup>1</sup>Note that the actual value of the enabling and disabling thresholds is not the average instructions issued per cycle, but rather the total number of instructions that would have to be issued within the sampling window in order to obtain the desired IPC threshold. This scheme avoids having to divide the counter value by the window size, and is similar to that proposed by Ghiassi [10].

resulting resource usage in the current sampling window based on this prediction. For example, if at the end of the previous sampling window we predict 4-wide mode, then we switch to 4-wide mode for the current sampling window. At the end of the current sampling window, we measure the  $I_{IPC}$  and  $FP_{IPC}$ . If these values do not trigger the disabling of 4-wide mode for the next sampling window, then our prediction was correct. That is, we predicted low-power mode and the result was low-power mode based on the selected enabling and disabling conditions. On the other hand, if these values measured in the current window trigger a disabling of 4-wide mode, then our prediction was incorrect; we predicted low-power mode and the result was normal mode.

Ideally we want high values in quadrants **LL** and **NN**. A high value in quadrant **NN** is good for performance, and a high value in quadrant **LL** is good for power. Quadrant **LN** indicates situations which are bad for performance and quadrant **NL** is bad for power. The best case scenario has  $LN = NL = 0$ . However, this can never happen unless  $NN = 100$ , given the algorithm for determining when to switch issue modes. We always base our decision on switching modes after we detect at least one window of cycles where we should have been in a different mode. Therefore, if we ever switch modes, we will always have entries in quadrant **LN** and **NL**. Furthermore, the number of entries in quadrant **LN** and **NL** will either be equal or will differ by one. Mode history was introduced to address these frequent transitions from one mode to another; by using mode history, we reduce the number of mode transitions and hence the number of entries in quadrants **LN** and **NL**.

Unlike branch prediction, there is no easy method for determining the correct resulting mode because the characteristics of the resulting mode in *PLB* are a function of the prediction. If we predict low-power mode for the current sampling window, then the issue rate in the current window never exceeds the maximum possible (either 4-wide or 6-wide) for the predicted low-power mode. One way to more accurately determine whether the prediction is correct is to run the simulator twice over the same window, once with the low-power mode set, and once with no issue width restrictions. If the two runs over the same window produce similar  $I_{IPC}$  and  $FP_{IPC}$  results, then the prediction is correct. Otherwise, the prediction is incorrect. Then the machine state is set to the state of the execution with the restricted issue width, and the process is repeated again for the next window. This procedure is time consuming and difficult to implement, and still produces subjective results based on what we consider to be similar  $I_{IPC}$  and  $FP_{IPC}$  results for the two executions over the same window.

We decided to use the enabling and disabling conditions to determine the “correct” result for *PLB* and weigh these subjective quadrant table values against the resulting power and performance numbers for each program. The quadrant table is only a good indicator of the accuracy of the prediction assuming we select appropriate enabling and disabling conditions. The final arbiters of the goodness of our methodology, metrics, and algorithm are the performance and power numbers resulting from *PLB*.

### 3. Power Calculations

*PLB* saves energy by modifying the activity rate in two primary components: issue queue and execution logic. There may be

an activity reduction in other components of the processor, such as the register file, but this only occurs if the total number of instructions issued is lower when using *PLB*. A reduction in instructions issued occurs if *PLB* reduces the number of wrong-path instructions; however, since *PLB* does not directly address the topic of wrong-path execution, reduction in wrong-path instructions cannot be guaranteed for every program.

### 3.1. Execution unit design

Given that the total number of instructions executed does not change with *PLB*, power in the execution units is reduced by gating the clock. In the Alpha 21264, clocks are a significant portion of the power of the chip. The local clocks alone account for approximately 20% of the power of the integer execution units and issue logic [18]. Therefore, clock gating the execution units measurably reduces the overall energy consumption. The Alpha 21264 uses clock gating to reduce power dissipation in the floating point (FP) units by only driving the clock for the functions required that cycle [12]. This saves power in some parts of the floating point datapath, but not the entire floating point execution unit. With *PLB* we save additional power when in low-power mode because we can safely disable portions of an entire functional unit cluster which includes not just the datapath but all associated control logic and clocks.

An argument can be made that every execution unit should be gated on a cycle by cycle basis by default, instead of doing so only when the machine enters a low-power mode. Since we know at issue time whether an instruction will be sent to a particular execution unit, we can gate or ungate the clocks to that execution unit every cycle. There are, however, some implementation issues with this technique.

First, the Alpha 21264 processor uses global and local clock patches to distribute clocks with a minimal amount of skew [18]. In future processors, it may be necessary to use more than two levels of hierarchy to distribute a clean clock signal. If the granularity of gating is one execution unit, then we can only gate the clocks to the execution unit in question, and not the next higher level of the clock patch. With *PLB*, we know in advance that we are either gating half of the right cluster or the entire right cluster. Hence we have the opportunity to gate more of the clock circuitry for greater power savings. Second, the clock enable wires and logic have a higher switching activity with cycle by cycle gating than with the *PLB* technique where we only gate or ungate at the end of every sampling window. Given that the clock patches we are gating are large, the power dissipated for enabling and disabling the clock patch is also large. Hence, gating on a coarser time scale reduces the amount of dynamic power dissipated by the clock gating circuitry. For the reasons stated, we believe that *PLB* allows for a more general clock gating scheme within this architectural environment.

### 3.2. Issue queue design

Quantifying issue queue (IQ) energy savings is a more complex problem because the power savings is a function of the IQ architecture. The complexity of designing and implementing any IQ is related to the problem of picking  $N$  data ready instructions out

of  $M$  entries in the IQ. Therefore, IQ issue logic consists of two primary components: the queue dependency logic (scoreboard), holding relevant information such as the readiness of instruction operands and the dependency between instructions, and the structures known as *arbiters* that pick which instruction to issue out of the queue.

We modeled our processor's IQ hardware on the design of the Alpha 21264 issue logic [6] [9]. The Alpha 21264 contains two IQs, an integer queue of 20 entries and a floating point queue of 16 entries. The integer IQ issues up to 4 data ready instructions out of a queue of 20 instructions. In the nomenclature given above,  $N = 4$  and  $M = 20$ . To simplify the integer IQ further, the 21264 splits it into two clusters, with each cluster picking and executing up to 2 out of 20 instructions. A cluster contains 2 arbiters, one copy of the integer register file, and 2 functional unit pipelines. Instructions are assigned to a specific integer cluster based on their position in the chunk of 4 instructions fetched into the integer IQ, and the type of instruction. At most, 2 arbiters within the same cluster can service an instruction.

We use a 128 entry IQ in our simulations of a 8-way issue processor. Unlike the 21264, our IQ is an integrated integer/floating point queue. Therefore, we need to select 8 instructions out of 128 entries every cycle. Figure 4 shows a 128 entry IQ implementation based on the 21264 design. We use this implementation to determine our power savings. To simplify the figure, we only show the arbiters, the execution units and the instruction scoreboard. Similar to the 21264, we slot instructions into left and right clusters, but each cluster now issues 4 instructions instead of 2. We also split the scoreboard into 4 sections of 32 instructions each. One of the critical paths in queue design is scanning the available entries in the queue to determine the readiness of an instruction. The 21264 split the IQs between floating point and integer to minimize the number of instructions scanned every cycle. Therefore, it is unlikely that we could design a queue which contains more than 4 times as many instructions as the 21264 integer IQ, and still be able to issue instructions in one cycle. By segmenting the scoreboard into 4 sections of 32 instructions each, we reduce the problem of picking 8 out of 128 to a more manageable problem of picking 8 out of 32.

Each 32 entry scoreboard section has 8 arbiters. Each arbiter per section issues to a set of associated functional units. There could be 4 instructions, one from each scoreboard section, competing for the same functional unit. Hence, some further control logic is required to arbitrate between instructions competing for the same execution resources. Figure 4 also shows the wiring and muxes necessary to decide which instruction wins the bid for the functional unit. When there is contention between instructions in different sections, the oldest instruction wins.

In the 6-wide low-power mode, we disable one half of the right cluster of arbiters for all four sections of the queue. This limits the number of instructions issued per cycle to 6, but the queue still picks from a total of 128 instructions. The pickers issuing to the floating point units are disabled, reducing the floating point issue bandwidth from 4 to 2 but retaining a memory bandwidth of 4 instructions per cycle. For the 4-wide low-power mode, we disable the entire right cluster of arbiters. Figure 4 highlights the disabled issue queue components. For both modes, instructions are reassigned to the non-disabled arbiters and clusters when

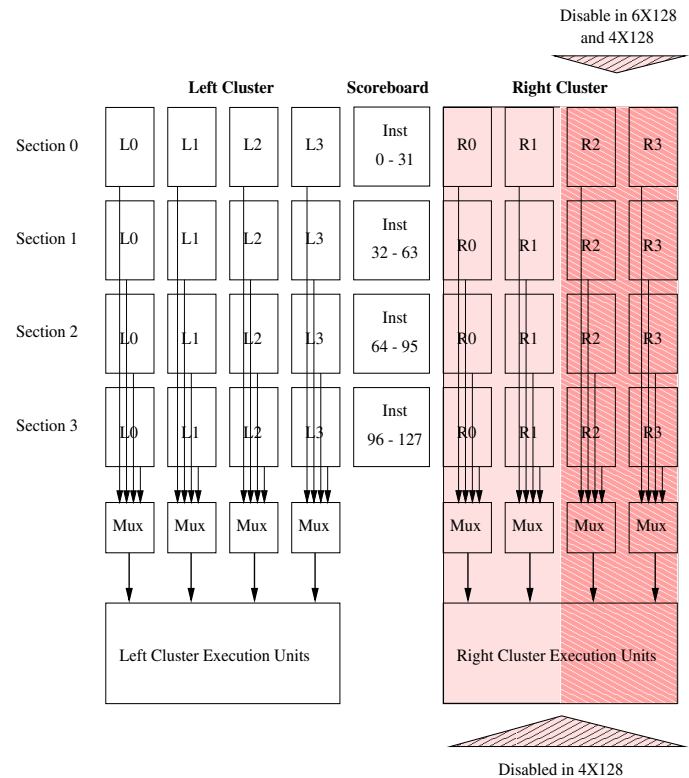


Figure 4. Issue queue hardware implementation.

entering low-power mode. Similarly, the machine begins assigning instructions to the right cluster when it exits low-power mode. Neither of these operations requires additional logic or time since each instruction already has a set of 8 arbiter enable signals indicating its arbiter assignment, and these signals can redirect arbiter assignment as necessary.

### 3.3. Power savings with *PLB*

To estimate the total energy savings of our processor when operating in low-power mode, we extrapolated from available 21264 power estimates [18] [3]. Wilcox showed that the 8-way issue Alpha 21464 is expected to dissipate between 125-150 Watts, with 46% of the power dissipation resulting from issue logic, and 22% from execution units [18]. Issue logic includes register files, register mapping, and the IQ. Execution unit power includes both floating point and integer functional units. Of the 46% associated with issue logic power, we assume that half of that power results from the IQ based on the relative power numbers shown for the 21264 processor. Therefore, the IQ and execution units account for 23% and 22% of the total chip power, respectively.

**Execution unit energy savings** In the execution logic, we save energy by disabling the clocks in unused portions. Wilcox noted that clocks comprised approximately 40% of total chip power for both the Alpha 21164 and the Alpha 21264. Assuming that this ratio is constant for all components of the chip, 40% of the execution unit power results from the clocks as well. In 6-wide

mode, we disable one quarter of the execution units, which is 10% ( $1/4 \times 40\%$ ) of total execution unit power. In 4-wide mode, we disable one half of the execution units, resulting in a maximum savings of 20% overall execution unit power. The actual power savings is a function of how much time we spend in each mode.

In terms of total chip savings, the clock power numbers calculated above are weighted by the contribution of the execution unit power to the total chip power. The execution unit power accounts for 22% of total chip power, so the full chip energy savings in the 6-wide and 4-wide mode is  $22\% \times 10\% = 2.2\%$  and  $22\% \times 20\% = 4.4\%$ , respectively.

**Issue queue energy savings** In the issue queue (IQ), we save energy both by disabling portions of the clock and by disabling some amount of the queue logic. When in low-power mode, we pick either 6 out of 128 or 4 out of 128 instructions to issue per cycle, depending on the low-power mode. When an arbiter cluster is disabled, we reduce the activity on the arbiter enable signals, the bid logic and signals, and the selection logic and signals. Each cluster of arbiters accounts for approximately 35% of the IQ power [3]. The remaining 30% is distributed between the scoreboard, instruction storage array and miscellaneous logic. This results in a maximum savings of 17.5% of the IQ power in the 6-wide scheme, where we disable one-half of one arbiter cluster, and a 35% IQ power savings in the 4-wide scheme where we disable an entire arbiter cluster. Again, the actual power reduction is a function of the amount of time spent in each mode. For total chip savings, we weighted the IQ energy savings with the IQ contribution to total chip power. Given that the IQ contribution is 23% of total chip power, the total chip energy savings from 4-wide and 6-wide low-power modes is  $23\% \times 17.5\% = 4.0\%$ , and  $23\% \times 35\% = 8.0\%$ , respectively.

**Summary of energy savings** Depending on the IQ scheme enabled, we can estimate the total chip energy saved while operating in low-power mode by simply adding together the appropriate numbers calculated above. For instance, if we always use the 4X128 scheme, we reduce total chip energy by  $4.4\% + 8.0\% = 12.4\%$ . Likewise, the 6X128 scheme produces a savings of  $2.2\% + 4.0\% = 6.2\%$ . Table 3 summarizes the potential energy savings calculated above for each component and configuration scheme. All values are rounded to the nearest whole number, and they represent the maximum power savings possible for the given configuration. The total power reduction for each program is a function of the numbers given in Table 3 and the percent of execution time the program spends in each low-power mode.

## 4. Experimental Results

We used the ASIM infrastructure with the processor described in Section 2 and Spec95 benchmarks to analyze performance and component activity with the *PLB* technique. Some of the Spec95 benchmarks are combined to produce 2-threaded runs. The 2-threaded benchmarks are combinations of 2 integer benchmarks, 2 floating point benchmarks, and one floating point and one integer benchmark. All results are reported relative to the base processor described in Section 2. For single threaded runs, we skip from

Configuration	Component	%Savings
4-wide	Execution Unit	20%
	Issue Queue	35%
	Total Chip	12%
6-wide	Execution Units	10%
	Issue Queue	17%
	Total Chip	6%

**Table 3.** Summary of localized and full chip power savings with low-power modes. Numbers shown are the maximum realizable power savings for each mode.

250 million to 2 billion instructions, warm up the simulator for 1 million instructions, and simulate each benchmark anywhere from 100 to 200 million instructions. The multi-thread benchmarks are simulated for 100 million instructions total for both threads, and each program within the multi-threaded run skips the same number of instructions as it would have in a single threaded run. All benchmarks are compiled with the official Compaq Alpha Spec95 flags.

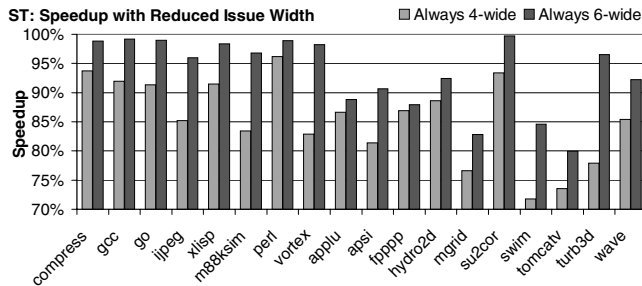
### 4.1. Reduced issue width results

*PLB* saves energy by reducing processor resources when the program does not require them. To determine the worst case performance loss, we analyzed the performance of the machine when it is *always* operating in 4-wide or 6-wide issue mode. Figure 5 shows the relative performance loss for a processor that always issues a maximum of 4 or 6 instructions per cycle instead of a maximum of 8 instructions<sup>2</sup>. Speedup numbers above 100% indicate performance improvement relative to the base case, while numbers below 100% indicate a performance loss. The rest of the paper represents performance in the same manner.

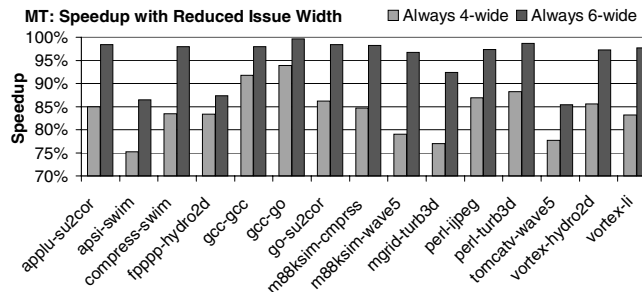
Results show a relative performance loss ranging from less than 1% for 6-wide issue *su2cor* to nearly 30% for 4-wide issue *swim*. The data indicate that the simulated machine is architecturally balanced with reference to issue width; otherwise, reducing issue width would not have resulted in such a dramatic performance loss. In general, integer benchmarks suffer less performance loss than floating point benchmarks. Integer benchmarks do not have the ILP (either due to data dependencies or due to branch misprediction) to take advantage of an 8-wide issue processor; a 6-wide issue processor is adequate for capturing most of the performance requirements. Also, many multi-threaded benchmarks, especially those that contain at least one integer benchmark, show minimal loss with 6-wide issue. The 4-wide issue processor, however, measurably degrades performance across all benchmarks, indicating that even programs with low IPC (such as *compress*) sometimes require a wider machine. Although a 4-wide issue processor

<sup>2</sup>The relative performance of multi-threaded benchmarks is measured according to the methodology presented in [16]. The performance of each program in the multi-threaded run is measured against the program's single threaded performance. The resulting speedups from each thread are summed and the sums of the base case and the 4-wide or 6-wide executions are compared for the final speedup numbers.

is detrimental to performance, *PLB* results presented later in this section show that many programs can safely enter 4-wide issue mode for a portion of the program's execution without paying a performance penalty.



(a) Spec95-ST

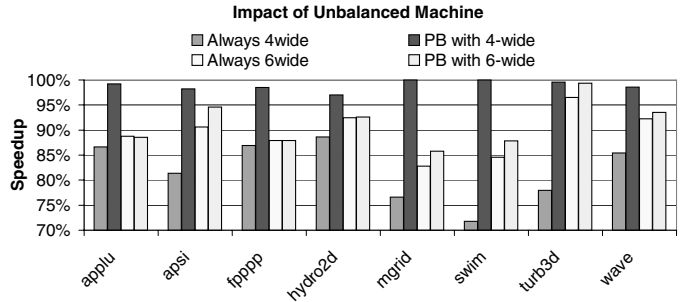


(b) Spec95-MT

**Figure 5.** Speedup for single and multi-threaded benchmarks with reduced issue width of 6 and 4. Note that the Y-axis begins at 70%.

In the next set of experiments we ran *PLB* with a single low-power mode of either 4-wide or 6-wide instead of the multi-mode formulation discussed in Section 2 in order to analyze the impact *PLB* has on performance. The enabling and disabling conditions for these experiments are based solely on  $I_{IPC}$ . In *PLB* with 6-wide issue, we either stay in normal mode or enter a low-power mode and reduce the issue width to 6. Similarly, in 4-wide issue, we reduce issue width to 4 when in low-power mode.

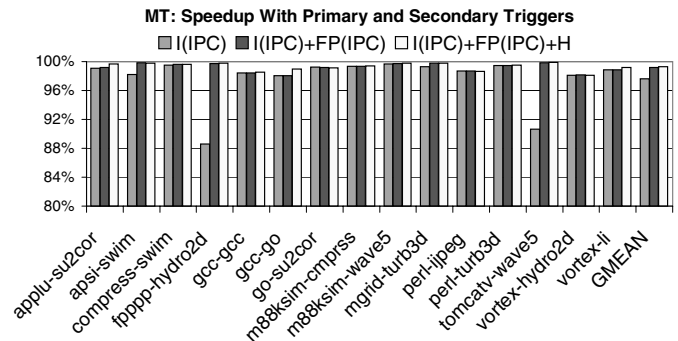
The data in Figure 6 shows that *PLB* with  $I_{IPC}$  alone effectively reduces issue width without jeopardizing performance for the 4-wide configuration, but not for the 6-wide configuration. For the sake of brevity, we only show a few single threaded, floating point benchmarks. Given that always issuing 4-wide is worse for performance than always issuing 6-wide, we expected *PLB* to more effectively address the performance loss in 6-wide mode than in 4-wide mode. However, as pointed out in earlier discussions, the 6-wide mode results in an unbalanced machine by reducing the floating point bandwidth. The data in Figure 6 shows that the problem with *PLB* using 6-wide mode is not with the *PLB* technique, but with  $I_{IPC}$ . The  $I_{IPC}$  metric is more indicative of the program's resource needs for a balanced, 4-wide machine than for the unbalanced, 6-wide machine.



**Figure 6.** Results for some floating point benchmarks with 4 or 6-wide issue with and without *PLB*. Note that the Y-axis begins at 70%.

## 4.2. Impact of primary and secondary triggers

Figures 7 and 8 show the impact of primary and secondary triggers on performance when using the multi-mode *PLB* technique. In multi-mode *PLB*, we enter either 6-wide issue or 4-wide issue as described in Section 2. Note that the y-axis begins at 80% speedup.



**Figure 8.** Performance impact of primary and secondary triggers on multi-threaded benchmarks. Note the Y-axis begins at 80%.

The difference between the first and second bars is the addition of the secondary trigger  $FP_{IPC}$ . Integer benchmarks are not affected by  $FP_{IPC}$  because they have very few, if any, floating point operations. However, floating point benchmarks benefit significantly because of the unbalanced nature of the 6-wide issue mode. Without this secondary trigger, benchmarks such as *applu*, *fp3ppp*, *mgrid*, *swim* and *tomcatv* would stay in low-power mode erroneously because of a low  $I_{IPC}$  resulting from the floating point bandwidth reduction. The geometric mean of speedup for floating point benchmarks improves from 91% to 99% with the addition of the  $FP_{IPC}$  trigger.

The third bar shows the impact of mode history. Mode history primarily helps the performance of integer benchmarks since they are more susceptible to widely varying issue rates due to branch

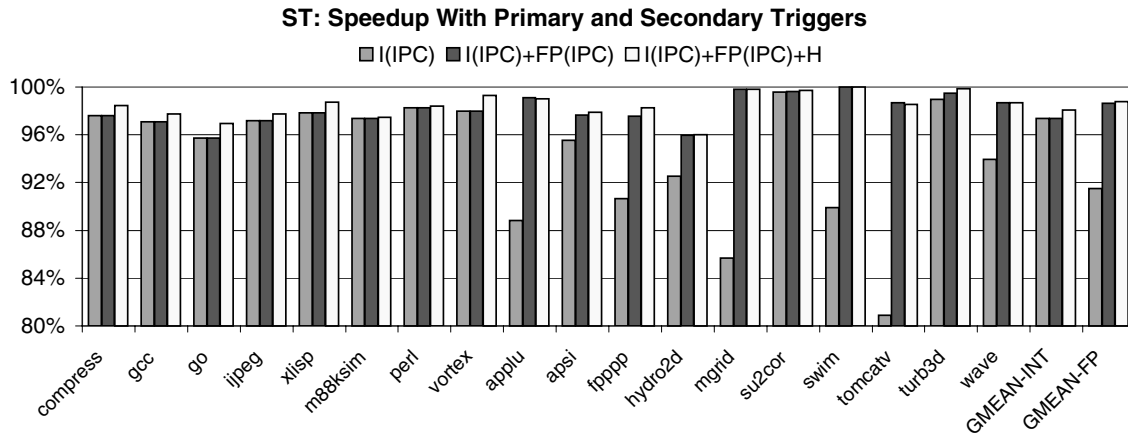


Figure 7. Performance impact of primary and secondary triggers on single threaded benchmarks. Note the Y-axis begins at 80%.

mispredictions. Branch mispredictions result in low  $I_{IPC}$  for a portion of the sampling window because of the time required to flush wrong-path operations and fetch correct path instructions. However, once correct path instructions begin moving through the pipeline, they will require the full issue capabilities of the processor. Mode history reduces spurious transitions to 4-wide mode by requiring at least two consecutive windows which satisfy the other 4-wide enabling triggers.

Since the goal of  $PLB$  is power reduction and not performance improvement, at best we retain the performance of the base case. With the primary and secondary triggers, we reduce performance loss to approximately 2% and 1%, on average, for integer and floating point benchmarks, and less than 1% for multi-threaded benchmarks. The only exception is *hydro2d*, which shows good issue IPC variation and should be an ideal candidate for  $PLB$ . However, even with secondary triggers, performance loss is still at 4%. Further analysis showed that the performance loss for *hydro2d* is related to the clustering of long latency, non-pipelined operations (floating point divides and square roots).

In the simulated architecture, a non-pipelined operation can overlap execution with a pipelined operation and vice-versa, but two non-pipelined operations cannot overlap with each other in the same floating point functional unit. Hence, programs with clustered, non-pipelined operations result in low  $I_{IPC}$  and  $FP_{IPC}$ . Because  $I_{IPC}$  and  $FP_{IPC}$  no longer indicate the true floating point execution needs of the program, the processor enters and stays in low-power mode erroneously, resulting in a performance loss. Numerous benchmarks execute non-pipelined instructions. For example, *apsi* has the same relative number of non-pipelined instructions as *hydro2d*; however, these operations are spread over many sampling windows in *apsi*, resulting in a low density of non-pipelined instructions per sampling window, while *hydro2d* has many non-pipelined instructions spread over fewer sampling windows, resulting in a clustering of non-pipelined operations. Of the benchmarks simulated, *hydro2d* is the only one which showed a measurable performance loss due to this phenomenon. This problem may be alleviated by disabling low-power mode when the

number of non-pipelined operations exceeds a given threshold in the current sampling window.

### 4.3. Accuracy of $PLB$

The quadrant table proposed in Section 2 analyzes the fundamental assumption that past behavior is indicative of future needs. To determine if this assumption is valid and to see the benefits of mode history, we plotted the quadrant table values for all single threaded benchmarks in Figure 9. These results are for multi-mode  $PLB$  with and without mode history but with all other triggers ( $I_{IPC}$  and  $FP_{IPC}$ ). The first bar shows the results without mode history, and the second bar shows results with mode history. The values in each quadrant are normalized to the sum of all quadrants for each benchmark. A high value in the LL quadrant is best for power, and a high value in NN quadrant is best for performance. Ideally, we want very few instances in the LN and NL categories.

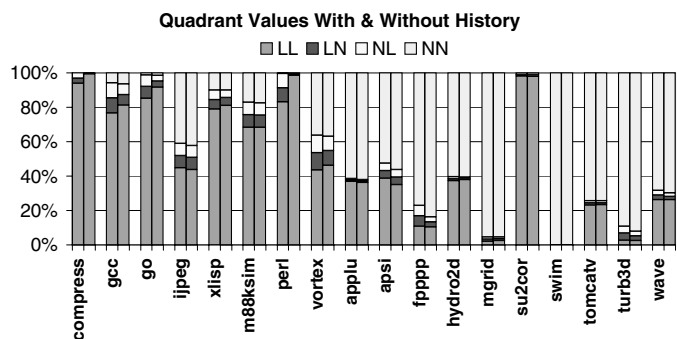


Figure 9. Quadrant table values for single threaded benchmarks. The first bar shows results without mode history, and second bar shows results with mode history.

The first observation from Figure 9 is that very little time is spent in the **LN** and **NL** quadrants. These values are a function of the number of times *PLB* switches from low-power to normal mode and vice-versa. This observation, combined with the performance and power data for *PLB*, shows that *PLB* works well and that past behavior is a good indicator of future program needs. The second observation is that mode history helps reduce the number of spurious transitions between low-power and normal mode. This is reflected in the smaller contributions quadrants **LN** and **NL** make in programs such as *compress*, *gcc*, *go*, *perl*, and *fpppp*.

#### 4.4. Component level power reduction

The data in Figures 7 and 8 shows that  $I_{IPC}$  combined with the two secondary triggers  $FP_{IPC}$  and mode history produces the best performance with *PLB*. Therefore, we use this configuration to determine per component and full chip power and energy reduction. Figure 10 shows the power reduction in the IQ and execution logic. The first and second bar in each cluster show the power saved in the execution logic and IQ, respectively. The bottom portion of each bar is the power saved when in 4-wide mode and the top portion is the power saved when in 6-wide mode. The numbers show an IQ power savings between 10% and 23%, and execution unit power savings between 5% and 12% for a number of benchmarks. We save more power in the IQ than in the execution logic because only clock power is impacted in the execution logic while the IQ saves power in the clocks and the instruction selection mechanism.

Integer benchmarks produce the best results for power savings. We found that there is no strict correlation between commit IPC and the percent of time each benchmark spends in 4-wide or 6-wide mode. For example, *compress* has a very low commit IPC, but is hardly ever in 4-wide mode. Although it has poor commit IPC, its issue IPC ( $I_{IPC}$ ) is generally above the 4-wide  $I_{IPC}$  enabling threshold value of 3.0. Hence *compress* spends nearly all its execution time in 6-wide mode without any performance loss, but rarely enters 4-wide mode. On the other hand, the expectation for *gcc* and *go* based on their commit IPC would be that they spend most of their time in 6-wide mode. However, results show that their power savings are garnered equally from 4-wide and 6-wide modes. Analysis of the time varying behavior of *gcc* and *go* shows a large  $I_{IPC}$  variation which puts both programs well below the 4-wide  $I_{IPC}$  threshold for a portion of the execution time. Other examples are *m88ksim* and *vortex*. The *m88ksim* benchmark has a higher commit IPC than *vortex*, yet spends some portion of its execution time in 4-wide mode while *vortex* hardly ever enters 4-wide mode. Again, this is because of the  $I_{IPC}$  variations seen in *m88ksim* which bring the  $I_{IPC}$  below the 4-wide  $I_{IPC}$  threshold.

*PLB* also works well for floating point programs, such as *apsi* and *applu*. As with some integer programs, these benchmarks have widely varying  $I_{IPC}$  behavior that makes them ideal candidates for *PLB*. The *su2cor* benchmark produces the best power savings of all floating point benchmarks by executing in 6-wide mode for nearly the entire program execution. Although *su2cor* has many floating point operations, the unbalanced 6-wide configuration does not affect this benchmark. The data in Figure 5a clearly shows that *su2cor* retains its performance even when running on a 6-wide machine all the time. This does not imply that

we should build a 6-wide machine since other benchmarks suffer significant performance losses with this configuration. However, it does imply that *PLB* is capable of detecting the resource requirements of *su2cor* by executing it always in the 6-wide, low-power mode.

*PLB* also reduces power in many multi-threaded benchmarks. This is unexpected for a couple of reasons. First, multi-threaded execution produces higher overall issue IPC; hence, *PLB* has less opportunity to reduce the issue resources of the processor. Second, Seng showed that multi-threaded benchmarks are inherently more energy efficient because they do not allow one program to dominate processor resources [15]. Figure 10 shows that there is still room for power reduction in multi-threaded programs which contain at least on integer benchmark such as *gcc-gcc*, *gcc-go*, *go-su2cor*, and *perl-turb3d*. The power savings most likely results from situations when one or more benchmarks has branch mispredicted, and the IQ is relatively empty of instructions for a period of time.

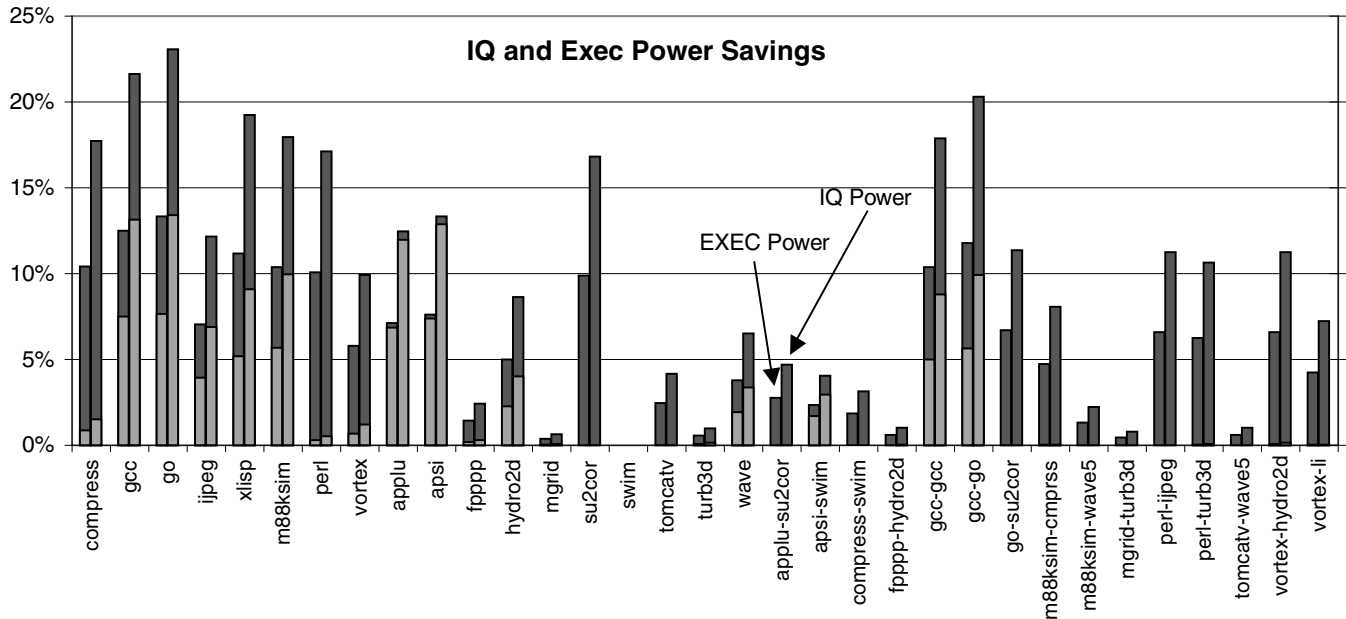
Finally, Figure 10 clearly shows why we chose to use a multi-mode *PLB* technique that has two low-power modes (6-wide and 4-wide) instead of only one. Although 4-wide mode produces the best power savings, many benchmarks cannot enter this mode without paying a performance penalty. However, with the introduction of the 6-wide low-power mode, we generate substantial power savings that would have otherwise not been achievable in programs such as *compress*, *perl*, *su2cor*, *gcc-go*, and *vortex-hydro2d*.

#### 4.5. Full chip power and energy reduction

Just as performance characteristics such as branch misprediction rate or cache miss rate must be looked at in the context of their impact on overall performance, so should localized power results be placed in the context of full chip power. The approximate upper bound for power savings with *PLB* is shown in Table 3. The best case scenario is to execute in 4-wide mode all the time, resulting in a full chip power savings of 12%.

Full chip power and energy calculations for *PLB* are presented in Figure 11. The power numbers are based on the local component savings and the impact each component has on the overall chip power. Overall chip energy is a function of the power dissipated in the execution of the program and the time required to execute the program. Therefore, the energy numbers are computed using the overall chip power savings and the increase in execution time due to the small performance degradation shown in Figures 7 and 8.

Unlike the substantial per component power savings seen in Figure 10, the overall chip power savings is at best 8% for *go*. However, when this value is placed in the context of a best case power savings of 12%, we achieve 66% of the maximum possible power reduction. Similarly, we achieve half or more of the maximum power savings for a number of benchmarks such as *compress*, *gcc*, *xlisp*, *su2cor*, and *gcc-go*. The energy numbers reflect the small performance loss resulting from *PLB*. All programs show smaller energy savings than power savings, but *PLB* still achieves a 5% full chip energy savings in a number of benchmarks. Two benchmarks, *fpppp* and *hydro2d*, show a small energy loss of 1%. Neither of these benchmarks reduced power dissipation enough to



**Figure 10.** Per component power savings. First bar shows the execution logic power savings, and the second bar shows the IQ power savings. The bottom portion of both bars is the power savings from entering 4-wide mode, and the top portion is the power savings from entering 6-wide mode.

compensate for their performance loss.

#### 4.6. Summary of power and energy savings

We chose to address power dissipation in two of the largest energy consumers in the processor according to the data presented in [18], and we achieved significant power savings in these components. Yet when these numbers are placed in the context of full chip power and energy, the overall savings is not as dramatic. Furthermore, as shown by the energy data, we cannot improve the power numbers by sacrificing performance because full chip energy is a function of both metrics. What the results show is that *PLB* as implemented here is one of many power saving techniques that need to be a part of processor design if we are to have a reasonable impact on overall processor power. In summary, it is best to take a holistic approach that combines *PLB* with other techniques such as those presented in [14] and [2] to produce significant full chip energy savings by attacking all components of the processor.

### 5. Conclusion

This paper introduced the *PLB* technique to reduce per component and full chip power without negatively impacting performance. *PLB* takes advantage of the inherent IPC variations within a program to adjust the pipeline issue rate to the dynamic needs of each program. Using the power data from the Alpha 21264 processor, we extrapolated a reasonable model for determining the per component and overall power and energy savings with *PLB*. Using the power model and a cycle level simulator of

a 8-way issue, out-of-order, multi-threaded processor, we showed that we can achieve up to 66% of the maximum power savings possible with the *PLB* technique.

This paper covers a limited number of uses and implementations of *PLB*. One possible expansion of this work is to explore the impact on power savings and performance when using dynamically adjustable trigger thresholds and sampling window size. Another variable to consider is IQ size. Initial experiments show that using *PLB* for reducing the IQ size along with the issue width results in additional power savings. Furthermore, the *PLB* technique is orthogonal to other power savings techniques such as that presented in [14]. Hence combining these techniques will likely produce better overall results. Finally, *PLB* can also be used for peak power management. By reducing the issue width when the processor gets too hot, we can trade some processor performance for reliable system behavior.

### Acknowledgments

The authors would like to acknowledge the following individuals for their contribution to this work: Joel Emer for his invaluable advice and encouragement throughout the process of researching and writing this paper, Matt Reilly for reviewing the basic idea, Chaun-Hua Chang and Vasu Arasanipalai for help in determining instruction queue power savings, and Steve Felix for information concerning implementation of clock gating. We would also like to thank the reviewers for their insightful comments which helped make this a better paper.

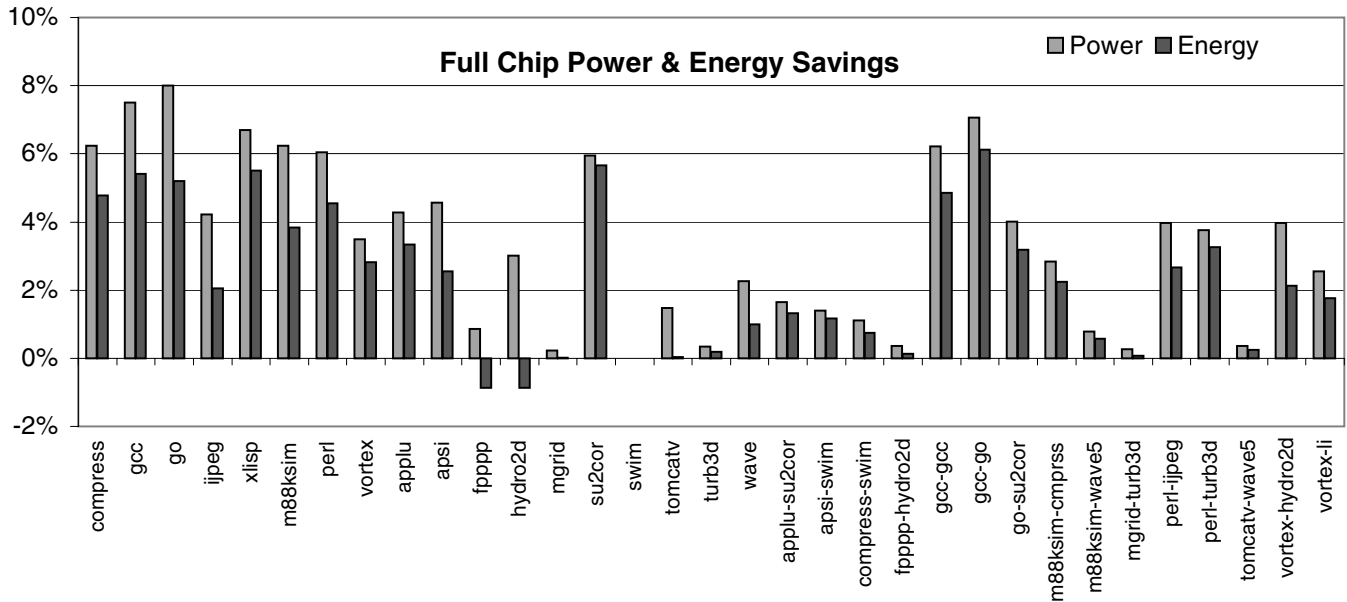


Figure 11. Full chip power and energy savings with *PLB*.

## References

- [1] D. Albonesi. Dynamic IPC/clock rate optimization. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.
- [2] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the International Symposium on Microarchitecture*, November 1999.
- [3] V. Arasanipalai. Private communication, Compaq Computer Corporation.
- [4] T. Burd. Processor design for portable systems. *Journal of VLSI Signal Processing*, August 1996.
- [5] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.
- [6] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.
- [7] Compaq Computer Corporation. *The ASIM Manual*, August 2000.
- [8] J. L. Cruz, A. Gonzalez, M. Valero, and N. Topham. Multiple-banked register file architectures. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.
- [9] J. A. Farrell and T. C. Fischer. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits*, May 1998.
- [10] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*, June 2000. Held in conjunction with the *International Symposium on Computer Architecture*.
- [11] P. N. Glaskowsky. Pentium 4 (partially) previewed. *Microprocessor Report*, August 2000.
- [12] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the alpha 21264 microprocessor. In *Proceedings of the Design Automation Conference (DAC)*, June 1998.
- [13] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.
- [14] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.
- [15] J. S. Seng, D. M. Tullsen, and G. Z. N. Cai. Power-sensitive multithreaded architecture. In *Proceedings of the International Conference on Computer Design*, October 2000.
- [16] A. Snively and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.
- [17] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 1991.
- [18] K. Wilcox and S. Manne. Alpha processors: A history of power issues and a look to the future. In *Cool-Chips Tutorial*, November 1999. Held in conjunction with the *International Symposium on Microarchitecture*.