In Proceedings of the 34th International Symposium on Microarchitecture, December, 2001

Reducing Power with Dynamic Critical Path Information

John S. Seng Eric S. Tune Dean M. Tullsen

Dept. of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{jseng,etune,tullsen}@cs.ucsd.edu

Abstract

Recent research has shown that dynamic information regarding instruction criticality can be used to increase microprocessor performance. Critical path information can also be used in processors to achieve a better balance of power and performance. This paper uses the output of a dynamic critical path predictor to decrease the power consumption of key portions of the processor without incurring a corresponding decrease in performance. The optimizations include effective use of functional units with different power and latency characteristics and decreased issue logic power.

1. Introduction

Power consumption is an increasingly limiting factor in the design process of microprocessors. Higher clock rates and increased device count directly result in more dynamic power consumption. While the increased clock rates are partially offset by reductions in V_{dd}, trends still point toward increasing dynamic power consumption. Trends in process technology also indicate that static power consumption is increasing, both in absolute terms, and in proportion to dynamic power consumption. This paper presents microarchitectural techniques to reduce both dynamic and static power consumption. Performance losses are minimized by applying these techniques intelligently to individual instructions based on information about their *criticality*. This results in better ratios of performance to power consumption than could be achieved without the use of this information. The particular applications of this technique that we present in this paper reduce power in the functional units and in the instruction issue window.

Recent research has demonstrated critical instructions can be identified during execution by the use of a critical path predictor [9, 21]. A critical path predictor predicts instructions which are believed to be on the critical path of execution — that is, those instructions whose execution latency constrain the overall speed of the program.

This paper demonstrates that critical path information can be used in conjunction with power-conscious microprocessor structures to increase the execution efficiency (the performance/power ratio) of a microprocessor. Dynamic critical path prediction allows the processor to identify instructions which require aggressive handling in order to maintain a high overall program execution rate, and which instructions (the non-critical instructions) do not need such aggressive handling. We can send the latter instructions through resources which are optimized for power rather than latency with little impact on performance. Thus, sections of the processor can be designed with a circuit style that utilizes less dynamic power and with higher threshold voltage transistors, resulting in considerable power savings.

In this research we demonstrate two such power optimizations that exploit information provided by a critical path predictor. For this research we use the critical path predictor described by Tune, et al [21].

Typically, integer functional units perform most operations in a single cycle. In order to perform these operations within one cycle, the circuitry of the functional unit may be designed with wider transistors, with a faster (higher power) circuit style, and/or with a reduced threshold voltage to allow high switching speed. The cost in power can be significant. If we can remove the single-cycle constraint on the design of some functional units, considerable power savings can be attained. To avoid a corresponding reduction in performance, only non-critical instructions should use a slower functional unit. Additionally, many functional units, such as adders, admit to many different designs. When the cycle time constraint is relaxed, it may be possible to use a design with fewer transistors, which also results in less static and dynamic power consumption. We show that the negative performance impact of replacing most of the functional units with slower versions can be minimized by utilizing a critical path predictor, while still providing significant functional unit power reductions.

The combination of the increasing processor-memory gap and increasing processor issue width dictates large instruction queues (or large numbers of reservations stations) to reveal more instruction-level parallelism (ILP). The power consumed by an issue unit can be reduced by reducing the size of the queue, by reducing the fraction of the queue over which the scheduler must search for issuable instructions, and by the number of functional units served by the queue. Our designs do all these things. In order to offset the reduced performance from a constrained instruction window, instructions are handled differently based on their criticality.

Power is not only a significant concern in mobile environments, where whole-chip power dissipation is the dominant metric. It is also a concern in high-performance processors, where power density (the power consumed per given area) becomes a limiting design factor. The optimizations in this paper will reduce whole-chip power, but our primary focus is on hot-spot elimination. We demonstrate techniques which increase the performance/power ratio of two functional blocks likely to exceed power density limits — the execution units and instruction issue mechanism.

This paper is organized as follows. Section 2 provides some background information on the critical path of a program. Section 3 discusses related studies. Section 4 describes the simulation methodology and tools. Section 5 presents power optimizations enabled by the use of a critical path predictor. Section 6 concludes.

2. Background

Modern, wide issue, out-of-order processors are not constrained to execute instructions in program order. The execution time is constrained by the data-dependences between instructions, and by constraints induced by processor-specific attributes. Not all instructions impact the execution time of the program. Those that do constitute the *critical path of execution*, and are typically only a fraction of all instructions. A critical path prediction which indicates each instruction's likely criticality helps us in two ways, for the purposes of this research. We can retain the high performance of the processor by ensuring critical instructions use only fast resources. We can decrease power consumption by routing non-critical instructions through power-optimized circuits which may have longer latency.

Previous work described Critical Path Prediction for dynamically identifying instructions likely to be on the critical path, allowing various processor optimizations to take advantage of this information. Tune, et al. [21] introduced the concept, observing dynamic heuristic events in the pipeline which are shown to be highly correlated with instruction criticality. A critical path buffer then predicts future criticality based on past behavior. Fields, et al., [9] proposed a predictor which passes tokens to more explicitly track dependence chains. In this research, we utilize a variant of the QOLD predictor from [21] primarily because of its simplicity. It adds almost no complexity to the core of the processor, which is important in a power-conservative design. However, the techniques shown in this research would adapt easily to a different critical path predictor.

The QOLD predictor marks instructions which repeatedly reach the bottom of the instruction scheduling window, indicating that they have dependences which are satisfied later than any instruction which precedes them in the instruction stream. The predictor utilizes a table of saturating counters, like a branch predictor, to predict future criticality based on past behavior.

By using a dynamic predictor rather than a static predictor [20], the technique uses no instruction space, adapts to the actual execution rather than depending on profiling with other input, and adapts to changes in the dynamic behavior of the program. The last point is important, because the dynamic behavior does change as critical-path based optimizations are applied, and the predictor recognizes those changes and constantly adjusts the critical path predictions accordingly.

When optimizing for power, we can utilize critical path information by sending as many instructions as possible through low power structures, while reserving fast (high power dissipation) structures for those instructions along the critical path. If we can apply power reduction techniques and still protect the critical path, we can significantly increase the ratio of performance to power for those structures in the processor. In order for this to be effective, we must have multiple paths through the processor, some of which are designed for high performance, others optimized for low power. In modern processors, nearly all paths are optimized for high performance, creating a heavy dependence on fast-switching, leaky transistors. This research demonstrates techniques that make it possible to significantly reduce the use of power hungry transistors. We specifically model two such techniques in this paper, reduced-speed functional units and specialized instruction queues, but we believe that several other opportunities exist.

3. Related Work

The previous section described the critical path predictors proposed by Tune, et al. [21] and Fields, et al., [9]. This research uses a predictor proposed by [21], but the actual predictor used is less important than the fact that the prediction can be used to improve the power efficiency of the processor.

Pyreddy and Tyson [15] present research on how processor performance is impacted when dual speed pipelines are used for instruction execution. That paper also uses criticality heuristics to mark and send instructions through execution paths with varying latencies. The heuristics used in that work are based on those in [21], but utilize a profile based method for marking instructions, whereas this work utilizes a run-time mechanism.

Casmira and Grunwald [8] present a different measure of instruction criticality (or non-criticality). They define *slack* as the number of cycles than an instruction can wait before being issued and becoming critical. That paper lists possible uses for power reduction but does not provide any further study.

Srinivasan *et al.* [17] developed heuristics for finding critical load instructions. They proposed hardware for finding those critical load instructions. Therefore, we would expect that their predictor would not perform well in conjunction with the power optimizations described in this paper, compared to a predictor which considers all instructions.

Bahar *et al.* [1] measured the criticality of instruction cache misses based on the number of instructions in the out-of-order portion of the pipeline. On an instruction cache miss, the fills were placed in different parts of the memory hierarchy depending on whether they were critical. They observed an improvement in performance and a reduction in energy.

Attempts to minimize functional unit power via micro-architectural techniques were described in [3]. The research in that paper exploits the fact that the sizes of operands are often less than 64 bits, the size of the available functional units.

Several architecture-level power models have been developed recently for use in architecture performance/power research; these include: Wattch[5], SimplePower [22], and the Cai-Lim model [7, 16]. The work presented in this paper is based parts on the power model described in [5].

An additional work which presents research based on per instruction biases to reduce power consumption is [10]. In that work, biases are placed against more incorrectly speculated instructions to limit the overall amount of speculation in the processor. This work presents research which places biases on a per-instruction basis based on the criticality of an instruction instead of its speculation history.

Some other micro-architectural studies have targeted power consumption and thermal limitations but at the full-chip level. Dynamic thermal management [4, 11, 16] management techniques recognize when the entire chip (or possibly particular regions) have exceeded allowable thermal limit, and slow down the entire processor until power consumption is reduced. Our techniques are intended to target (at design-time) particular units with high power density.

4. Methodology

Simulations for this research were performed with the SMTSIM simulator [19], used exclusively in single-thread mode. In that mode it provides an accurate model of an out-of-order processor executing the Compaq Alpha instruction set architecture. Most of the SPEC 2000 integer benchmarks were used to evaluate the designs. All of the techniques modeled in this paper could also be applied to the floating point functional units and floating point applications, but for this research we chose to demonstrate them on the integer execution units. All simulations execute 300 million committed instructions. The benchmarks are fast forwarded (emulated but not simulated) a sufficient distance to bypass initialization and startup code before measured simulation begins. The benchmarks used, their inputs, and the number of instructions fast forwarded is shown in Table 1. In all cases, the inputs were taken from among the reference inputs for those benchmarks.

Benchmark	input	Fast forward (millions)
art	c756hel.in	2000
crafty	crafty.in	1000
eon	kajiya	100
gcc	200.i	10
gzip	input.program	50
parser	ref.in	300
perlbmk	perfect.pl	2000
twolf	ref	2500
vortex	lendian1.raw	2000
vpr	route	1000

Table 1. The benchmarks used in this study, including inputs and fast-forward distances used to bypass initialization.

For the instruction queue and functional unit power simulations, the power model used is the Wattch power model from [5]. The model is modified to include leakage current as described later in this section and is incorporated into the SMTSIM simulator. This combination enables the tracking of power consumption by functional block on a cycle-by-cycle basis, including power used by wrong-path execution following mispredicted branches.

For the initial simulations regarding the functional unit power, the following power estimation technique is utilized. The total power consumption is separated into 2 components: $P_{dynamic}$ and P_{static}.

$P_{dynamic}$ represents the dynamic power portion of the total power consumed by the logic. This is the power used in charging and discharging input and output capacitances and is proportional to the switching activity of the circuit.

In order to model the effect of using 2 different circuit speed implementations for fast (standard speed) and slow functional units, the fast functional unit is assigned a baseline dynamic power consumption value and the reduced speed power consumption is scaled down by a ratio. In the base case, the ratio of dynamic power consumption for a slow functional unit to a fast unit is 0.8. This ratio exists because of the reduced timing requirements placed on the slower functional unit circuitry. Because of the reduced amount of computation required per cycle, slower circuit styles may be used [2] which consume less power. Higher threshold transistors can also reduce the dynamic power consumed[23]. In addition, transistor sizing may also be used to reduce the dynamic component of power consumption. There are many factors that would go into determining the actual ratio of dynamic power used by the two designs,

Parameter	Value
Fetch bandwidth	8 instructions per cycle
Functional Units	3 FP, 6 Int (4 load/store)
Instruction Queues	64-entry FP, 64-entry Int
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache (on-chip)	1 MB, 4-way, 64-byte lines
Latency (to CPU)	L2 18 cycles, Memory 80 cycles
Pipeline depth	8 stages
Min branch penalty	6 cycles
Branch predictor	4K gshare
Instruction Latency	Based on Alpha 21164

Table 2. The processor configuration.

including the aggressiveness of the original design; thus, we will investigate the impact over a range of reasonable ratios. No single ratio would be appropriate for all readers. However, we feel the default ratio of 0.8 used for later results represents a conservative estimate.

P_{static} is the static power component of total power consumption. For the fast functional units, lower threshold voltages are typically used resulting in higher leakage current. As a baseline value, static power consumption for the fast functional units is modeled as 10% of of the dynamic power consumption (the power consumed when the unit is active). This value is for a .1 micron feature size process [18].

For the slow functional unit, we assume that static power consumption is reduced by a factor of two, to 5%. This value is conservative since sub-threshold leakage current scales exponentially with decreasing threshold voltage [6]; thus, a slight sacrifice in switching speed can greatly reduce sub-threshold leakage current.

Details of the simulated processor model are given in Table 2. The processor model simulated is that of an 8-fetch 8-stage out-of-order superscalar microprocessor with 6 integer functional units. The instruction and floating-point queues contain 64 entries each, except when specified otherwise. The simulations model a processor with full instruction and data caches, along with an on-chip secondary cache.

The critical path predictor is a 64K-entry table of saturating counters updated according to the QOLD heuristic, as described in [21]. This technique proved to be effective in performance-based optimizations, and also outperformed the other heuristics for our power-based optimizations. The 64K-entry table would not necessarily be the best choice for a power-conscious design, but allows us to initially compare these techniques ignoring the issues of contention. However, Section 5.3 shows that much smaller tables give nearly identical performance. The default counter threshold at which instructions are marked as critical is set to 5 with an instruction marked as critical incrementing the counter by one, and non-critical instructions decrementing by one. Thus the counters can be as small as 3 bits.

The primary metric utilized in this paper is execution efficiency for particular microprocessor structures. This metric is equal to the performance (in instructions per cycle) divided by the power consumed by that structure. This does not necessarily correspond to chip-level performance/energy ratios which are important in a battery-conservation environment. However, it does reflect the usefulness of a technique in improving the performance/power density ratio for a functional block which is a potential hot spot. This metric is appropriate assuming the power density of the targeted structure is a constraint on the total design (and is similar in spirit to a cache performance study that assumes the cache sets the cycle time of the processor [12]). In that scenario, the optimization which has the best performance to component power ratio, and reduces power density to acceptable levels, would represent the best design.

5. Optimization

This section examines two optimizations which exploit information available from a critical path predictor. First, it directs the use of asymmetric functional units which differ in speed and power consumption. Second, it simplifies instruction scheduling by splitting the critical and non-critical instructions into different issue queues and simplifying the critical queue, or possibly both queues.

5.1. Low Power Functional Units

Functional units are likely to be designed with fast, power hungry devices so that they meet cycle time requirements. If we remove the requirement that all functional units need to be fast, we can reduce our reliance on high-power transistors. This section describes how critical path information can be used to determine where to send instructions when functional units of different power consumption (and also latency) are present. Non-critical instructions should be able to tolerate a longer execution latency without a corresponding slowdown in the execution of the overall program. This optimization exploits the tradeoff that is available to circuit designers - that of power consumption versus execution speed. In these experiments, we assume the processor has both *standard speed* functional units and *reduced speed* functional units, which we will refer to as fast and slow functional units for simplicity. The fast functional units are assumed to execute most integer operations in a single cycle, while the slow functional units perform operations in two cycles. We assume that the slow units are still pipelined, so that we can sustain the same issue rate.

Each fetched instruction receives a 1-bit critical-path prediction, which is carried with the instruction as long as it might be needed – in this case, up until issue. The use of asymmetric functional units can significantly complicate instruction scheduling (and use more power) if the assignment

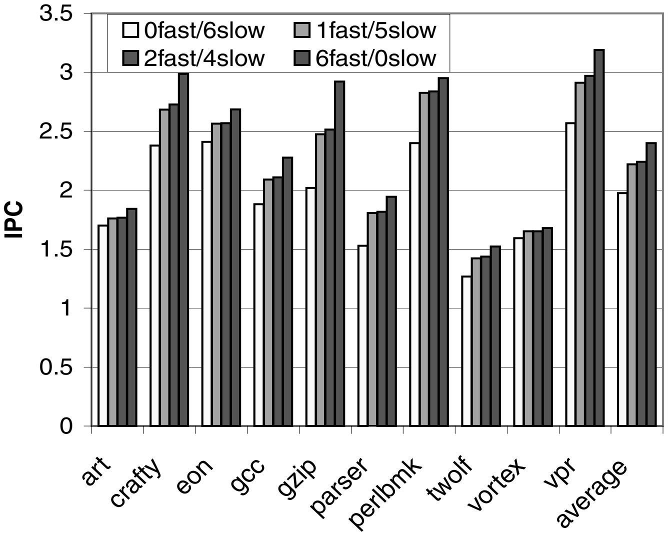

Figure 1. Performance when varying the number of fast and slow functional units.

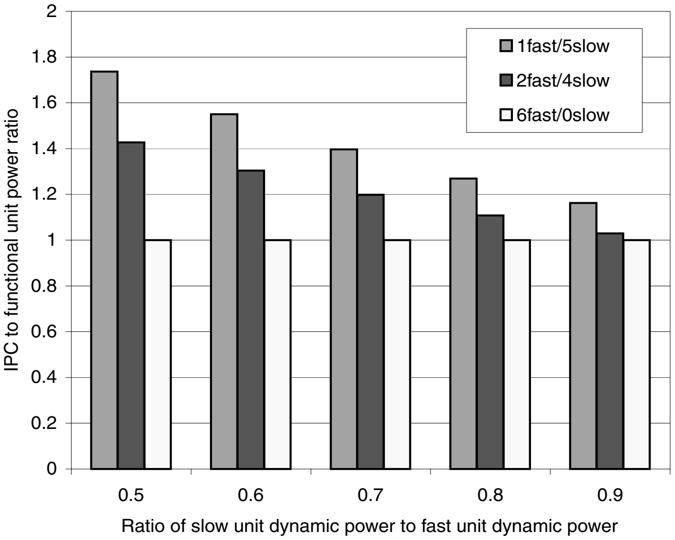

Figure 2. The ratio of IPC to functional unit power for various FU configurations and various assumptions about the power dissipation of the slow units.

to units is done flexibly at scheduling time. This is because the two scheduling decisions, that of the fast units and that of the slow units, become dependent on each other. Instead, we take an approach used by the Alpha 21264 [13] with their clustered functional units, and "slot" an instruction for a functional unit type before dispatch into the instruction queue. Thus critical path instructions are slotted to the fast functional units, and non-critical instructions are slotted to the slow functional units. Once slotted, an instruction must issue to the functional unit type it is assigned to.

Figure 1 shows the performance with various combinations of fast and slow integer functional units. In all cases the total number of units is 6. These results show the importance of using the critical path predictions. In most cases, over half the performance difference between the 0-fast results and the 6-fast results is achieved with just a single fast functional unit by slotting instructions based on the critical path predictions. In general, critical path prediction allows us to prevent heavy drops in performance, relative to the baseline of 6 fast functional units. Gzip has the biggest difference between the 1-fast FU result and the baseline at 18%. The average difference is 8.1%. If the functional units are a potential hot spot, we may be willing to take that loss if there is a correspondingly greater reduction in power.

Figure 2 confirms that this is true. It gives the ratio of performance (IPC) to power for the functional units, as an average over all benchmarks. Because the actual power consumed by the functional units will vary widely with how aggressively they are designed — how close they are to the chip's critical timing path, and how much high-speed circuitry is used — this figure plots the power/performance for five different ratios of fast-unit dynamic power to slow-unit dynamic power for an active unit. Even when there is very little difference in the dynamic power, the difference in static power still more than compensates for the lost performance due to the slow functional units. All further results will assume a ratio of 0.8.

One artifact of slotting instructions to functional units is that load imbalance (too many instructions being slotted to either the critical or non-critical units, while the others sit idle) can also reduce performance significantly. Thus, we add an explicit load-balance technique to ensure that the processor does not degrade too heavily from the static partitioning of the functional units. The load balancing mechanism utilizes a shift register, which holds the criticality prediction of the previous 60 instructions, allowing us to track a moving average of the number of predicted critical instructions. The mechanism attempts to maintain this value between two experimentally determined thresholds (which are approximately the ratio of fast functional units to total functional units). If an incoming instruction will cause the moving average to exceed one of the thresholds, its criticality prediction will be inverted. This mechanism improves performance by 7% on the average over the critical path predictor alone, which is reflected in the results shown here. The same critical path predictor may be conservative or liberal depending on the specific characteristics of an application, so some load-balancing mechanism is important in achieving consistently good results. We will utilize the 1-fast, 5-slow configuration in most of the remaining results in this paper.

5.2. In-order Issue of Critical Instructions

Given that the processor is assigning instructions to a functional unit type prior to issue, it is not necessary that

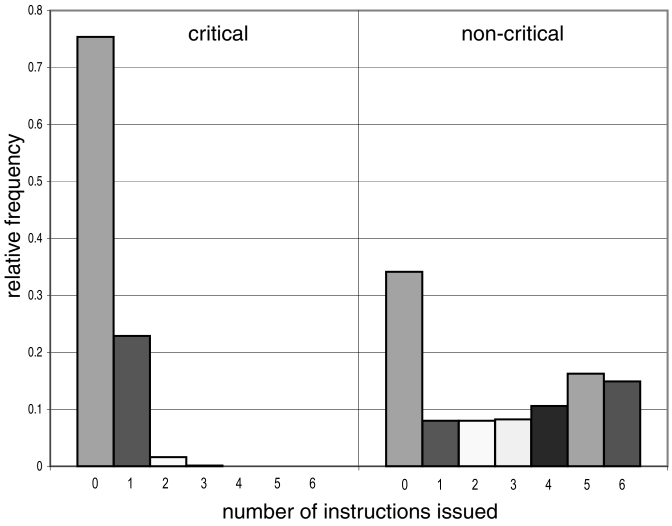

Figure 3. The average number of instructions issued per cycle for an out-of-order instruction queue.

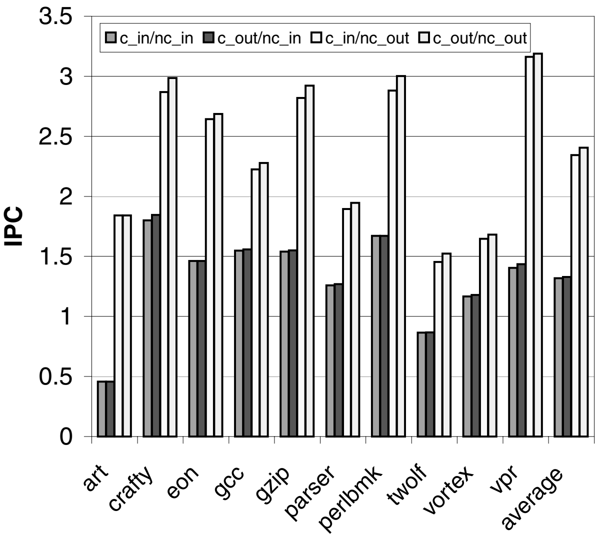

Figure 4. Sensitivity of critical and non-critical instructions to out-of-order vs. in-order issue. c_in and c_out represent when critical instructions are issued in-order and out-of-order, respectively. nc_in and nc_out denote how non-critical instructions are issued.

they even share the same scheduling resource. There is an immediate advantage to dividing up the scheduling. The implementation complexity for scheduling increases with the number of entries [14], so multiple smaller queues can be more efficient than one large window. But the advantages go beyond that. Once we split the critical and non-critical instructions, we find that their scheduling needs are quite different.

Figure 3 is a histogram of the number of each type of instruction that get scheduled each cycle, assuming an out-of-order scheduler with symmetric functional units. This figure demonstrates that for these applications the critical path is typically a very serial chain of dependent instructions, and there are rarely even two instructions that are available to schedule in parallel. This is further confirmed by Figure 4. This figure presents the results of performance simulation assuming two partitioned instruction queues (critical and non-critical instruction queues), assuming either could be in-order issue or out-of-order issue, showing all possible permutations. It shows that the critical instructions are almost completely insensitive to the difference between in-order issue and out-of-order issue, while the other instructions are extremely sensitive.

This is not because the instructions in the non-critical queue are more important, but rather because placing a serial constraint on the non-critical instructions creates long critical path sequences where previously there were none. Looked at another way, it is out-of-order scheduling that makes most of those instructions non-critical in the first place, and removing that optimization returns many of them to the critical path.

Splitting the critical and non-critical instructions provides three opportunities for power reduction, two of which are explored here. First, splitting the one queue into smaller queues reduces the complexity of selection. Second, we can further simplify the critical queue by allowing it to issue in-order. Third, we could also reduce the speed of the out-of-order queue, giving it two cycles to make a (pipelined) decision, rather than one. In the worst-case, this would add a cycle to the latencies seen by instructions in that queue. This last optimization is not explored in this paper, but the performance effect would be similar to that shown in Figure 1.

An in-order queue utilizes less power overall because of the relaxed issue requirement. In fact, when coupled with a single fast functional unit, the scheduler need only visit a single instruction to make a scheduling decision.

There is an additional benefit to splitting the queues and functional units in this manner. Since the fast functional units are bound to the in-order queue and the slow functional units are bound to the out-of-order queue, the higher power structures (out-of-order queue and fast functional units) are not placed together. This will reduce power density problems even further when both functional blocks are problematic.

Figure 5 confirms that there is little performance lost by using smaller, partitioned queues, examining two different partitionings of the original 64-entry queue, and assuming the asymmetric execution units of the previous section. The performance difference in IPC between the partitioned

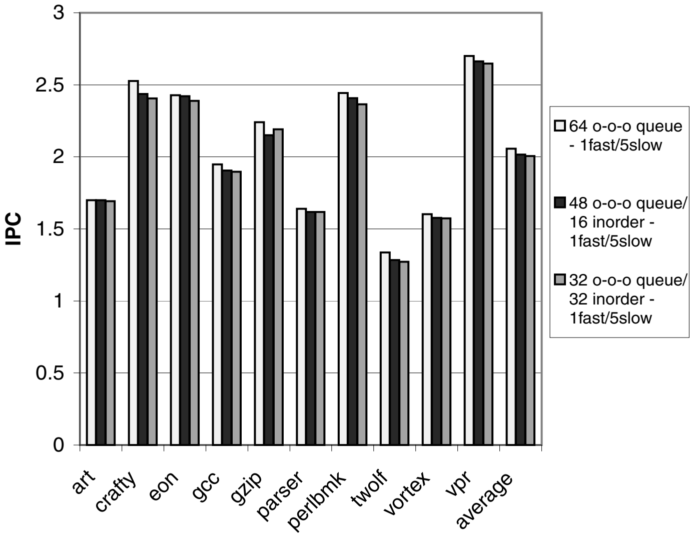

Figure 5. The effect on performance when varying the queue size. The designs modeled have either a single, conventional instruction queue, or a split instruction queue with an in-order critical-instruction queue, and a out-of-order non-critical instruction queue.

queues (with the in-order critical queue) and a single complex queue is much less than the effect seen due to the slow functional units. For example, the difference between two 32-entry queues and a 64-entry queue is only 2.5%.

Figures 6, 7, and 8 show the difference in execution efficiency for various combinations of queue sizes, queue organizations, and functional unit speeds on functional unit efficiency, queue efficiency, and combined efficiency, respectively. Each of the results have been normalized to the results for the 64 entry out-of-order queue simulations. Figure 6 shows the functional unit impact. While it is not surprising that queue organization has little impact on execution unit power, the advantage of having mostly slow functional units is clear across a variety of organizations. In the cases where the size of the queue does effect the functional unit power ratio, this occurs because of the effect of instruction queue size and organization on speculation. When the instruction queue size is reduced and some of the instructions are issued in-order, the total number of wrong path instructions that are executed is reduced as well. This reduces the average power consumed by the functional units.

The effect on the queues is seen in Figure 7 where we see a clear advantage to the partitioned queues. The best ratio in this graph is provided by a 32-entry non-critical queue and a 32-entry critical queue, with an average increase of over 33% in performance to power consumption. The 32-entry critical queue does not fill often, but the cost of the large queue is low since the issue mechanism is in-order, and it prevents pipeline conflicts which might occur when a smaller critical-instruction queue would become full.

The combined effect on the functional units and the queues is seen in Figure 8. The best combined efficiency comes from the combination of the best functional unit configuration and the best queue configuration. In this case, a 32-entry in-order queue with five slow functional units and a 32-entry out-of-order queue with one fast functional unit clearly excels, proving more than a 20% gain in efficiency over the baseline design.

The combination of split queues and liberal use of slow functional units can produce a significant decrease in power, more than proportional to the decrease in IPC, in the very core of the processor where power density is often of greatest concern.

5.3. Saving Energy With Critical Path Prediction

The focus of this paper is on power reduction of processor hot spots. However, these optimizations will also be useful in an energy-constrained (e.g., mobile) environment as they each will reduce whole-chip energy usage, as would other optimizations that might also use the critical-path predictor. In a battery-constrained environment, however, we would need to evaluate these optimizations somewhat differently.

For this paper, where the primary focus is power density in high-performance processors, we focus on the performance/power ratio for individual blocks, which is appropriate when the power density of a unit becomes a design bottleneck. For energy, we would need to look at whole-chip energy. In that case, the performance/energy gains would be less than the performance/power gains shown here. This is because the energy gains would be somewhat diluted by the energy of the rest of the processor not affected by the optimization.

Second, we would need to ensure that the sum total of the reductions enabled by the critical path predictions was not negated by the cost of the predictor itself. When focusing on power density constraints, it is only necessary that it not become a hot spot. Of course, the less power and energy it uses, even in high-performance applications, the better.

The critical path predictor need not be a high-power block. Like other memory-based structures, power density is not likely to be a problem because the fraction of devices activated each cycle is low.

We use a relatively large predictor for the results in this paper, but Figure 9 shows that it need not be large to provide high-quality predictions. This graph shows the average performance over the benchmarks we simulated when varying the critical path buffer size. A 1K predictor provides nearly equivalent performance to a 64K, but still significantly outperforms no predictor (for the size zero results, a random predictor was used).

Although the structure is read and updated multiple times per cycle, that also need not translate into complexity

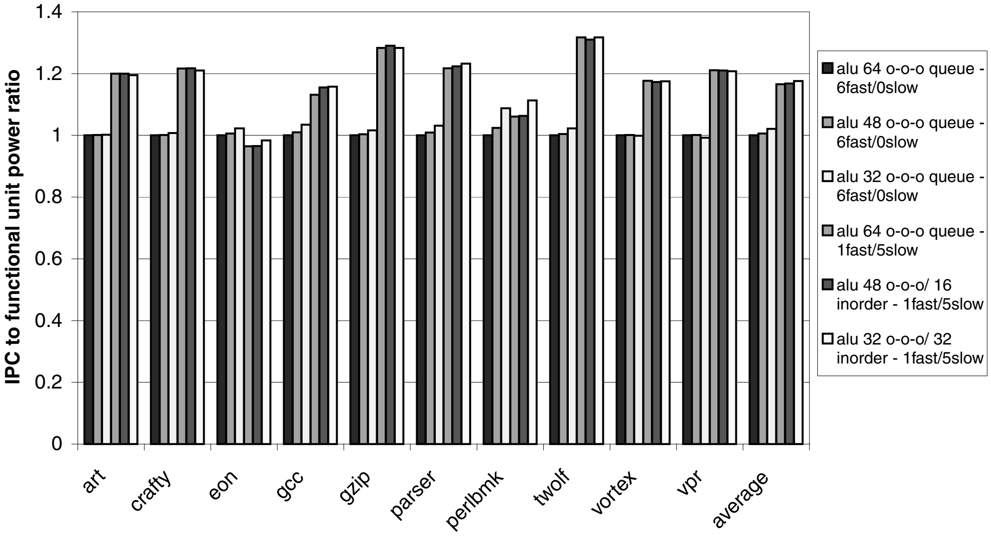

Figure 6. The ratio of IPC to functional unit power for various FU and queue configurations.

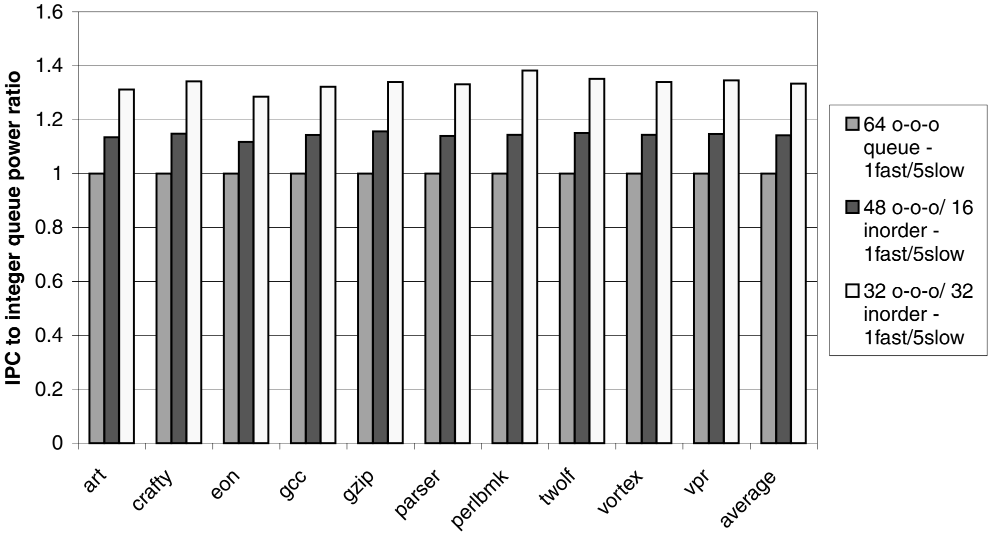

Figure 7. The ratio of IPC to queue power for various queue configurations.

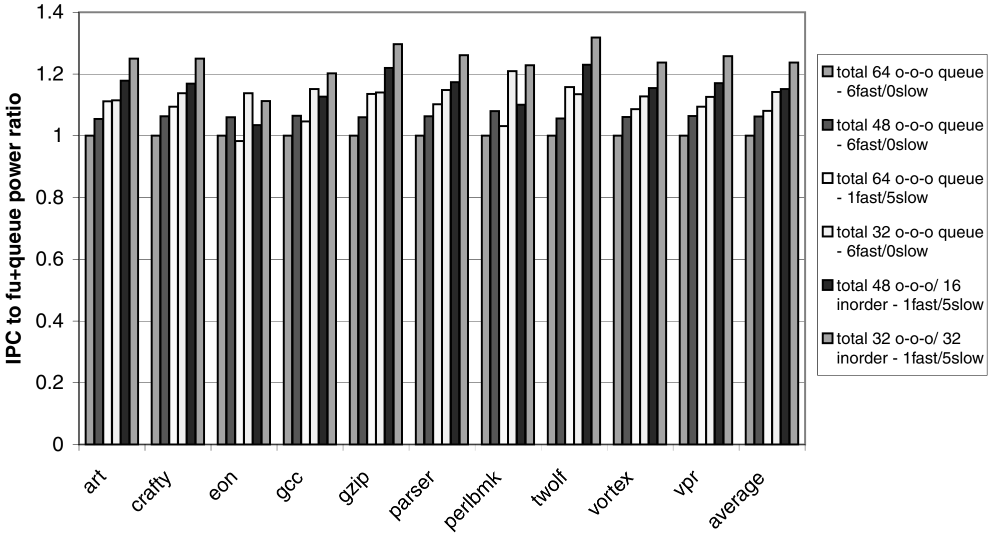

Figure 8. The ratio of IPC to functional unit plus queue power for various FU and queue configurations.

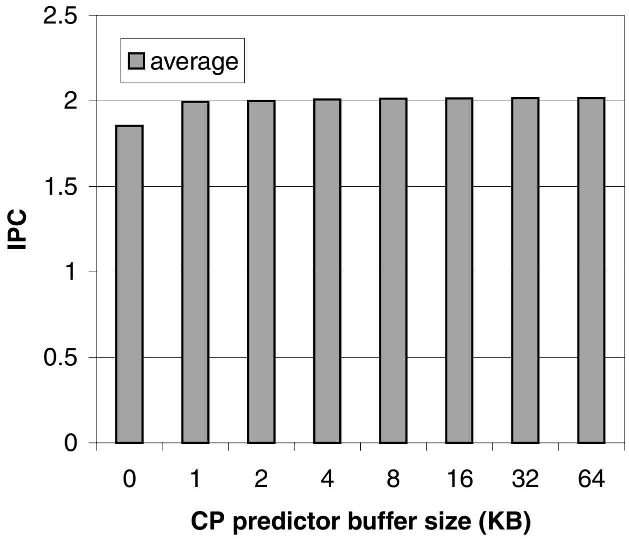

Figure 9. The effect of critical path predictor buffer size on the average IPC of the benchmarks tested. The processor model includes a 48-entry out-of-order queue, a 16-entry in-order queue, 1 fast and 5 slow functional units.

or power. Like the branch target buffer, the predictor is accessed by a fetch block at a time (both at fetch, and possibly at commit). Thus, like a BTB, we can interleave the structure so that it can still be single-ported (e.g., for read) and indexed by a single address.

The more places the processor makes use of critical path prediction, the better it amortizes the power or energy used by the predictions. This paper only describes two such optimizations, but those do not exhaust the possibilities.

One other area we did explore for using critical-path information to attack energy is the use of criticality-controlled speculation. This attempted to reduce total speculation by limiting speculation based on criteria such as the criticality of individual instructions, the criticality of branches, on how much of the critical path was in the processor, etc. Although we achieved some small gains, that work brought the following conclusions — critical path instructions are in general poor candidates for execution speculation (as demonstrated by the fact that out-of-order execution is not necessarily required for critical instructions), and criticality is less useful than speculation/confidence measures in controlling speculation.

6. Conclusions

In modern processors, designers pack as much functionality as possible into each pipeline stage within the allowable cycle time constraints. Performance is optimized when there is not one critical timing path, but rather a large number of nearly equal critical timing paths. This design methodology, however, produces designs with very high reliance on high-power, high-speed devices and circuits.

This paper presents techniques which can be used to decrease reliance on high-power circuits. This is done by differentiating, on a per instruction basis, between those that need high performance and those that do not. This presents the opportunity to reserve high-speed circuitry for the instructions which effect performance, while using power-optimized circuitry for those that do not.

This work relies on recent work on critical path prediction to dynamically identify those instructions whose execution latency has the greatest impact on overall program execution speed. Two specific techniques are modeled that exploit this information to eliminate high-power logic. The first replaces most of the functional units of a superscalar processor with slower units, giving the critical-path instructions exclusive access to the fast unit or units. This results in significant gains in in the ratio of performance to power density. This is true over a range of assumptions about the power that could be saved with the slow devices.

The second optimization shows that separate scheduling queues for the critical and non-critical instructions allows lower overall complexity, including much simpler issue complexity for the critical-instruction queue. The impact on performance of this change is minimal, and the power savings is significant.

7. Acknowledgments

Early conversations with C.J. Newburn and George Cai of Intel influenced the direction of this research. We would like to thank the anonymous reviewers for their useful comments. This work was funded by NSF CAREER grant No. MIP-9701708, grants from Compaq Computer Corporation and the Semiconductor Research Corporation, and a Charles Lee Powell faculty fellowship.

References

[1] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED-98)*, pages 64–69, New York, Aug. 10–12 1998. ACM Press.

[2] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, (4), July 1999.

[3] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999.

[4] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan. 2001.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[6] J. Butts and G. Sohi. A static power model for architects. In *33rd International Symposium on Microarchitecture*, Dec 2000.

[7] G. Cai and C. Lim. Architectural level power/performance optimization and dynamic power estimation. In *Proceedings of Cool Chips Tutorial at 32nd International Symposium on Microarchitecture*, Nov. 1999.

[8] J. Casmira and D. Grunwald. Dynamic instruction scheduling slack. In *2000 KoolChips workshop*, Dec. 2000.

[9] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[10] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[11] W. Huang, J. Renau, S. Yoo, and J. Torrellas. A framework for dynamic energy-efficiency and temperature management. In *33rd International Symposium on Microarchitecture*, Dec 2000.

[12] N. Jouppi. Cache write policies and performance. In *20th International Symposium on Computer Architecture*, May 1993.

[13] R. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, Dec. 1998.

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

[15] R. Pyreddy and G. Tyson. Evaluating design tradeoffs in dual speed pipelines. In *2001 Workshop on Complexity-Effective Design*, June 2001.

[16] J. Seng, D. Tullsen, and G. Cai. Power-sensitive multithreaded architecture. In *International Conference on Computer Design 2000*, Sept. 2000.

[17] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson. Locality vs. criticality. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[18] S. Thompson, P. Packan, and M. Bohr. MOS scaling: Transistor challenges for the 21st century. In *Intel Technology Journal*, 1998.

[19] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[20] D. Tullsen and B. Calder. Computing along the critical path. Technical report, University of California, San Diego, Oct. 1998.

[21] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Feb. 2001.

[22] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[23] L. Wei, Z. Chen, M. Johnson, and K. Roy. Design and optimization of low voltage high performance dual threshold cmos circuits. In *Design Automation Conference*, June 1998.