Something Old and Something New: P-states can Borrow Microarchitecture Techniques Too

Yasuko Eckert*[†] Srilatha Manne* Michael J. Schulte* David A. Wood[†]*

*AMD Research
Advanced Micro Devices, Inc.
{yasuko.watanabe, srilatha.manne, michael.schulte} @amd.com

[†]Department of Computer Sciences
University of Wisconsin – Madison
david@cs.wisc.edu

ABSTRACT
The limited utility of voltage scaling in nano-scale technologies has led high-performance processors to rely increasingly on frequency scaling for power management. However, frequency scaling provides only a linear dynamic power reduction.

In this paper, we make a case for dynamically disabling performance optimizations, leveraging previously proposed low-power techniques, for more efficient power-performance trade-offs. By carefully selecting which optimizations to turn off, our lowest P-state consumes less than half the power achieved by frequency scaling, on average, for comparable performance. For all workloads, our approach performs as well or better than DVFS, demonstrating the effectiveness of our approach.

Categories and Subject Descriptors
B.8.2 [**Performance Analysis and Design Aids**]

General Terms
Design. Performance.

Keywords
Dynamic voltage and frequency scaling, frequency scaling, P-states, power gating, dynamic microarchitectural power saving.

1. INTRODUCTION
Modern processors support performance states, or P-states, for dynamic power-performance management. The dominant P-state mechanism is dynamic voltage and frequency scaling (DVFS) because of its cubic relationship between (dynamic) power and performance: for a 3% reduction in power, DVFS reduces performance by about 1%. Unfortunately, the utility of DVFS has been decreasing due to transistor scaling limitations [6]. Although various circuit-level techniques have been proposed to address this problem, such as ultra-low voltage operation [5][6], many incur performance, leakage power, and/or reliability penalties.

Dynamic frequency scaling is an established alternative to DVFS, especially for high-performance commercial microprocessors [11]. Frequency scaling simply reduces the operating frequency and leaves the supply voltage unchanged. Although simple in design, frequency scaling provides only a linear reduction in dynamic power and has no direct impact on the increasingly important static power. Traditionally, frequency scaling was used to modestly extend the power-performance curve and *only after* DVFS had scaled the supply voltage to its minimum (Vmin), as the left side of Figure 1 depicts. However, future technology nodes increasingly will rely on frequency scaling as the gap between maximum (Vmax) and minimum supply voltages shrink (the right side of Figure 1). Replacing DVFS with frequency scaling will significantly flatten the achievable power-performance curve, especially at lower frequencies. Figure 2 demonstrates that a 36% power saving through frequency scaling comes at a cost of a 44% performance loss on average. Furthermore, memory-intensive workloads (e.g., *libquantum*) are power- and performance-insensitive to core frequency scaling because the majority of the baseline chip power comes from the static-power-dominated L2 and L3 caches and frequency scaling only has indirect impact on static power.

This paper proposes new mechanisms to implement P-states that achieve a power-performance curve closer to DVFS than frequency scaling. Our mechanisms leverage previously proposed low-power techniques that result in a performance loss. Previously, these techniques might have been dismissed due to the effectiveness of DVFS; however, they become viable alternatives to frequency scaling. We focus on techniques that selectively disable or constrain microarchitectural performance optimizations, trading performance for power reductions. By coordinating the

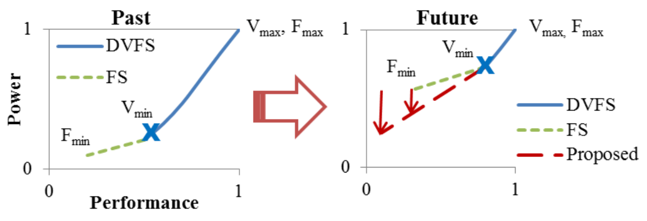

Figure 1. Conceptual example (lower-right is better).

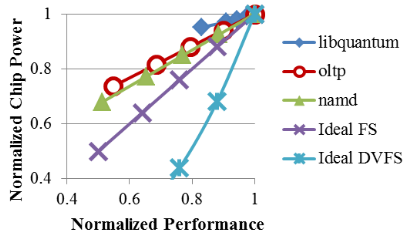

Figure 2. Core frequency scaling (FS).

techniques rather than using them in the isolated fashion of previous implementations, we can gain much more power savings.

Our research draws insight from the performance optimization rule established by the Intel Pentium M: a 1% performance improvement should come with no more than a 3% power increase; otherwise, DVFS could do better [8]. If we use this rule *in reverse*, disabling optimizations that meet this 3:1 power-to-performance ratio, we can effectively extend the cubic DVFS power-performance curve. Although some optimizations may have much less than a 3:1 ratio, others may exceed the ratio for a given workload, allowing our techniques to perform better than DVFS.

For the initial evaluation of this concept, this paper selects, without in-depth analysis, two sets of previously introduced intuitive techniques to implement a pair of microarchitectural P-states: front-end and L2 power reduction. The former power-gates front-end structures designed for a worst-case scenario, including the entire checkpointing hardware and some of the fetch buffer and physical register file. In addition, we employ simplified speculation control [14] that selectively stalls the front-end with varying levels of aggressiveness.

The L2 P-states, on the other hand, exploit the fact that not all workloads require the optimized L2 cache access latency or the full cache capacity. We employ drowsy cache technology [7] and power-gate parts of the cache [17]. When combining these front-end and L2 P-states, we show that the lowest P-state consumes less than half the power of frequency scaling, on average, for comparable performance. The power savings possible with our P-states are bounded, on the low end, by the static power of the power-gated components. The upper bound, on the other hand, depends on the dynamic power of those components as well as the impact those components have on the rest of the system.

In this work, we assume these P-states are initiated as soon as we go below P0, which is the highest-performance state. However, our proposed P-states can also be applied in combination with DVFS or after reaching the Vmin. Compared to traditional DVFS-based P-states, our mechanisms have a longer transition delay (~87 µs [18]) due to power-gating overheads. We assume that we stay at a P-state long enough to compensate for the delay.

The contributions of this paper are:

- We make a case for using microarchitectural techniques to implement P-states. We selectively turn off power-dominant performance optimizations for more efficient scaling than frequency scaling. This approach can complement frequency scaling and be used beyond Vmin. To the best of our knowledge, this is the first paper to propose using microarchitectural mechanisms to implement P-states.
- We re-examine old (i.e., previously proposed) low-power techniques in a new context and apply them without complex policies or logic to obtain significant power savings. Our results demonstrate that these simplified techniques outperform frequency scaling for all workloads, and provide better power scaling than DVFS in most cases.

2. FRONT-END P-STATES

We first explore opportunities in the front-end for trading instruction-level parallelism (ILP) and/or memory-level parallelism (MLP) for power. The front-end is an attractive target: it is organized for quick retrieval of instructions after a pipeline flush, and it is generally underutilized once the scheduler is full. In addition, the rate of instruction flow to the back-end sets an upper bound on achievable ILP and MLP.

Hence, our aim is to reduce wasteful power from wrong-path instructions and lessen or turn off the capabilities of power-hungry front-end structures—even at the cost of performance. We propose a new synthesis of four old ideas (checkpoint removal, speculation control, and fetch buffer and register size reductions) to provide more efficient power-performance scaling than frequency scaling.

Checkpoint removal. Many high-performance out-of-order (OoO) processors checkpoint architectural state to recover from branch mispredictions. Although checkpoints facilitate fast misprediction recovery at detection [16], they result in modest performance improvement for highly-accurate branch predictors [19]. When allocating checkpoints to all branches, only 0.002-8.0% of all checkpoints are used to recover from mispredictions and the corresponding branches eventually commit (results not shown).

Moshovos proposed reducing checkpoints by predicting which branches are likely to require checkpoints and/or using OoO checkpoint release [16]. While both of these techniques still recover from mispredictions as soon as detected, we propose a novel technique that selects between recovery at detection and recovery at commit depending on P-states. During P0, we use the conventional checkpoint-based recovery at detection. To increase power efficiency during lower-performance P-states (i.e., P1 to Pn), we disable all checkpointing hardware and recover at commit by flushing any structures with corrupted states. Compared to the checkpoint-based recovery at detection, recovery at commit lengthens misprediction penalties by the cycles between detection and commit (unless it takes longer to re-fill the window), which can be significant. We mitigate the penalties by flushing wrong-path instructions in the OoO execution engine on a misprediction detection and re-fetching the correct-path instructions. We also stall the rename stage, which contains corrupted states, until the recovery action is taken.

Speculation control. To reduce wasteful power from wrong-path instructions during lower-performance P-states, we also apply a simplified version of speculation control that gates the rename stage when the amount of speculation in a window exceeds a certain threshold [14]. Prior proposals measure speculation by the number of unresolved low-confidence branches. We instead use the total number of unresolved branches regardless of their confidence as a proxy. This coarse approximation penalizes workloads with very low squashes per thousand instructions (SPKI) and phases with low SPKI; however, we obviate the need for a confidence estimator, thus reducing complexity and power consumption. The rest of the speculation-control mechanisms remain the same. The smaller the threshold, the more aggressively we suppress wrong-path instructions, with a greater risk of performance loss.

Fetch buffer resizing. A complementary method for regulating speculation is to lessen fetching aggressiveness by shrinking the fetch buffer [21]. The fetch buffer size is usually larger than the fetch width to mask I-cache miss latencies. However, a larger buffer size generally increases the probability of fetching wrong-path instructions. Even for workloads with small misprediction rates, the full fetching capability is not necessary most of the time

Table 1. Baseline configuration

Component	Configuration
Branch prediction	TAGE [19], 1K-entry tagged tables, 5 history bits, 4-way 256-entry BTB, 16-entry RAS
Pipeline width	4-wide fetch, rename, dispatch, issue, and commit
Front-end	7 stages, 16-entry fetch buffer, 16 renamer checkpoints, 128 physical registers
Execution	32-entry unified OoO IQ, 2IALU, 2 FPALU, and 2 AGEN, 128-entry window
Dis-ambiguation	NoSQ [35], 1024-entry predictor, 1024-entry double-buffered SSBF
L1-I cache	32KB, 4-way, 64B line, 2 cycles, per-core private, next-line prefetching
L1-D cache	32KB, 4-way, 64B line, 2 cycles, per-core private, write-invalidate, inclusive, 2 ports
L2 cache	1MB, 8-way, 2 banks, 64B line, 12 cycles, write back, inclusive, per-core private, high-performance device type
L3 cache	8MB, 16-way, 4 banks, 24 cycles, shared, low-standby-power device type
Main memory	2 QPI-like links (up to 64GB/s), 300 cycles
Coherence	MOESI-based directory protocol
Technology	3GHz, 0.9Vdd, 32-nm process

Table 2. Front-end and L2 P-state mechanisms

Front-end	A	B	C	D	L2	E	F	G	H
Base	*	16	16	128	Base	N	N	8	12
P1-FE	8	0	4	64	P1-L2	Y			13
P2-FE	4				P2-L2		Y		14
P3-FE	3				P3-L2			4	
P4-FE	2				P4-L2			2	
P5-FE	1				P5-L2			1	

A = Max in-flight unresolved branches. **B** = Checkpoint count. **C** = Fetch buffer size. **D** = Physical registers. **E** = Drowsy L2 data. **F** = Drowsy L2 tags. **G** = L2 associativity. **H** = L2 Access cycles. * = Unconstrained.

because the fetch is typically designed for infrequent window re-fills. Thus, in lower-performance P-states, we power-gate a portion of the fetch buffer, which in turn reduces accesses to the I-cache and instruction translation look-aside buffer.

Register file resizing. The preceding two techniques reduce in-flight instructions, creating opportunities to proportionally resize other structures as well. Although a more balanced approach is desirable, this initial work focuses on a major source of front-end power: physical registers. Dynamic physical register file resizing is not an easy task due to potentially scattered valid register mappings. Thus, when a lower P-state is requested, we stop allocating registers that will be power-gated (half of the registers in our experiments) and enter the P-state once those registers do not contain valid mappings. This operation occurs only once, and the power-gated portion remains fixed throughout the P-state.

3. L2 CACHE P-STATES

The second P-state implementation addresses static power in large caches. Static power imposes an upper limit on power savings achievable by frequency scaling, and large caches exacerbate the issue. As transistors become leakier in future technology nodes [7], it becomes more important to scale down static power along with dynamic power for large power savings.

This section examines existing circuit techniques for static power management and employs those techniques during lower P-states. Hence, we use them without any complex policies to fully enjoy the power-saving benefits of the techniques.

For gradual power-performance scaling, we enable five P-states (P1 through P5) in the L2 cache. P1 puts all the L2 data arrays into a drowsy mode to cut the L2 static power. Drowsy mode is an effective technique that supplies just enough voltage (drowsy voltage) to preserve the state of the memory cells and switches to a higher active voltage to safely read out the data [7]. By leveraging drowsy mode at the cache line granularity, significant static power reduction is possible. Accessing a drowsy line, however, takes an extra cycle to wake up, increasing the L2 access latency to 13 cycles in our implementation (Section 4). Given the goal of this work—providing a lower power-performance curve than frequency scaling—we are more interested in aggressive power reduction, and hence we simply apply drowsy mode uniformly to the cache.

P2 also puts the L2 tag arrays into drowsy mode, lengthening the access latency by another cycle (i.e., 14 cycles).

The remaining three P-states (P3 to P5) gradually trade off the L2 cache capacity for power by power-gating some of the associative ways, which is permanent during each level. Again, we apply this technique uniformly to all L2 sets for simplicity and maximum power savings, rather than optimizing for performance [17]. When entering each of the three P-states, any dirty lines in the ways that will be power-gated must be written back to the L3. Without attempts to minimize the associated power and latency cost [17], this work simply writes back dirty lines before gating. As expected, Section 5 shows that the performance impact of the reduced associativity is workload-dependent, depending on the workload's working set size, data locality, and re-use distance. However, compared to P1 and P2, power-gating a portion of the L2 completely eliminates the static power of the disabled portion *and* reduces the dynamic energy of L2 tag accesses.

4. METHODOLOGY

Our target machine is an 8-core CMP. Each node consists of a core, private L1 and L2 caches, and one bank of shared L3. We simulate the design with a full-system cycle-accurate simulator using Virtutech Simics [13], GEMS's Ruby [15], and in-house timing-first processor models. Because frequency scaling is often locally applied [11], we run single-threaded workloads from the SPEC CPU 2006 benchmark suite [9] and the Wisconsin commercial workloads [1] on a single core while power-gating the other cores for the initial investigation of our work. The workloads were compiled for the 64-bit SPARC ISA with the Sun Studio 11 compiler, and we simulate each workload for 100 million instructions on an unmodified SPARCv9 operating system after fast-forwarding the initialization phase and warming up architectural state for two million instructions. For architectural-level chip power approximations, we augmented our simulators with Wattch [3] and CACTI 5 [20]. Our power model assumes aggressive clock gating of logic structures not in use with no reactivation delay.

We use a 4-way OoO superscalar processor as our baseline core design. Table 1 details the configuration of the core and the

Table 3. Workload characterization

Workload	L3 MPKI	BPKI	SPKI	L2 Sensi-tivity	Workload	L3 MPKI	BPKI	SPKI	L2 Sensi-tivity	Workload	L3 MPKI	BPKI	SPKI	L2 Sensi-tivity
libquantum	36.1	251	.003	1.0	astar	2.9	129	23.8	2.1	h264ref	0.5	6.7	2.8	12.0
mcf	36.0	205	6.7	1.3	jbb	2.0	172	11.7	3.8	bzip2	0.4	166	2.4	7.9
bwaves	28.1	3	0.2	1.2	soplex	1.6	200	6.0	8.0	gobmk	0.4	140	19.0	40.0
milc	23.8	14	.009	1.0	xalancbmk	1.4	212	4.5	1.7	sjeng	0.3	137	18.4	14.9
lbm	22.9	12	0.1	1.5	oltp	1.2	155	20.1	8.1	tonto	0.3	74	8.2	13.4
Gems	15.1	6	0.1	2.9	gromacs	1.1	12	0.9	3.7	wrf	0.2	120	2.3	14.8
zeus	8.8	153	15.6	4.2	hmmer	0.9	6	0.2	1.4	namd	0.2	53	2.4	22.0
apache	8.1	154	20.7	4.0	omnetpp	0.9	231	7.9	12.4	povray	0.2	101	3.7	57.0
sphinx3	7.9	56	10.3	1.7	gcc	0.8	185	13.0	9.3	gamess	0.1	53	2.7	89.2
perlbench	5.8	176	5.2	2.3	leslie3d	0.7	116	13.2	19.6	HMean	1.7	17	0.1	13.8
cactus	4.2	1	0.1	2.2	calculix	0.6	45	1.4	5.5					
zeusmp	3.4	16	0.1	5.9	dealII	0.5	43	5.7	5.5					

L3 MPKI = L3 misses per thousand instructions. **BPKI** = Branches per thousand instructions. **SPKI** = Branch-misprediction-caused squashes per thousand instructions. **L2 sensitivity** = Increase in L2 miss rate from an 8-way L2 to a direct-mapped L2. Shaded cells indicate workloads presented in graphs.

memory hierarchy. Our frequency-scaling model is based on IBM's POWER7 [11], which implements per-core frequency scaling. We therefore assume that core frequency scaling affects only the associated L1 and L2; the frequencies of the L3 and the main memory remain the same. Furthermore, we idealize asynchronous domain crossing as having no overhead, resulting in an optimistic frequency-scaling model. Although POWER7 allows a fine-grained frequency step of 25 MHz, we simulate only the nominal and minimum frequencies and three intermediate frequencies while fixing the voltage, and use linear interpolation to estimate the remaining ones. Thus, the simulated normalized core frequency points are 1.0, 0.88, 0.76, 0.64, and 0.50.

We derive DVFS curves by assuming a 22% operating voltage range based on Intel Pentium M. We factor in temperature fluctuations when estimating the corresponding static power.

Table 2 summarizes our proposed microarchitectural P-states, using the notation of <P-state>-<mechanism type>. The left half of the table is for front-end P-states, which selectively stall renaming with varying degrees of aggressiveness. All the P-states, except P0, power-gate the entire checkpointing hardware, three quarters of the baseline fetch buffer, and half of the baseline physical register file and free list. The right half of Table 2 summarizes the L2 P-states that gradually trade the L2 cache performance for power. We model a drowsy cache based on the work by Flautner *et al*. [7].

5. EVALUATION

This section compares the power-performance of our microarchitectural P-states to both frequency scaling and DVFS. Our goal is to do better than frequency scaling's 1:1 power-to-performance ratio and approximate the 3:1 ratio of DVFS.

Table 3 lists the crucial characteristics of all workloads. As discussed earlier, frequency scaling is less effective in reducing power usage of memory-intensive workloads. Memory intensity, in this case, is measured by L3 misses per thousand instructions (MPKI). The front-end P-states exploit branches per thousand instructions (BPKI) and branch-misprediction-caused squashes per thousand instructions (SPKI) to gradually reduce the aggressiveness of the front-end. The performance sensitivity of the L2 P-states depends on the workload's L3 MPKI and L2 miss-rate sensitivity to smaller associativity (L2 Sensitivity). Due to space constraints, we show only the results of best, worst, and typical workloads for each P-state implementation, as well as the harmonic mean of all workloads.

Front-end P-states. Figure 3 presents power and performance normalized to the baseline for best (*gobmk*), typical (*povray*), and worst (*bwaves*) workloads as well as the harmonic mean of all SPEC and the commercial workloads. The data points labeled *Freq Scaling* are simulated results using the configurations in Section 4, and *P*-FE* represents *P1-FE* through *P5-FE* in Table 2. *Analytical DVFS* represents the derived DVFS curve for each workload. Although the degree and the scalability vary, all of our configurations yield lower power-performance curves than frequency scaling, indicating a more efficient power-performance trade-off. Furthermore, some of the *P*-FE* data points of the best and typical workloads even lay on top of the DVFS curves. We verified that the effectiveness of our configurations holds true for the rest of the workloads, and summarize the results using harmonic mean.

These favorable results demonstrate that the previously proposed techniques we have redeployed become viable alternatives to frequency scaling, even though the individual techniques by themselves may not have worked well under peak performance restrictions. Coordinating the techniques reduces the capability of structures designed for worst-case performance and decreases wasteful energy. In contrast, frequency scaling uniformly slows execution regardless of whether instructions are speculated or on the correct path.

The effectiveness of the front-end P-states, however, depends largely on the workloads' BPKI and SPKI due to the inherent dependency of branch characteristics. The higher the BPKI, the more unresolved branches exist in an instruction window, creating more opportunities for stalling the rename stage. Similarly, the higher the SPKI, the greater the reduction in wrong-path instructions, which in turn results in less wasted energy. Hence, high-BPKI and high-SPKI workloads (e.g., *gobmk*) exhibit the most power savings for the amount of performance loss and also scale well with different stalling aggressiveness. The converse—

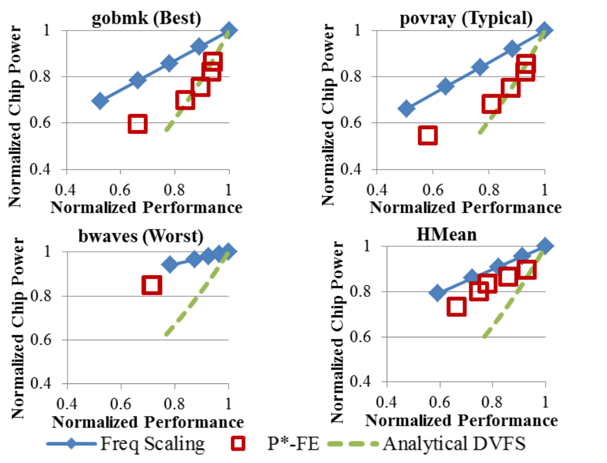

Figure 3. Power-performance of front-end P-states normalized to the baseline (lower-right is better).

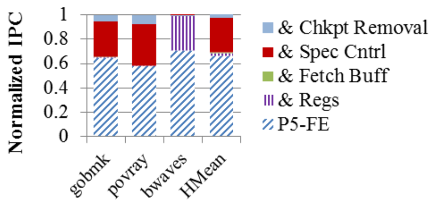

Figure 4. IPC impacts of the applied techniques with P5-FE.

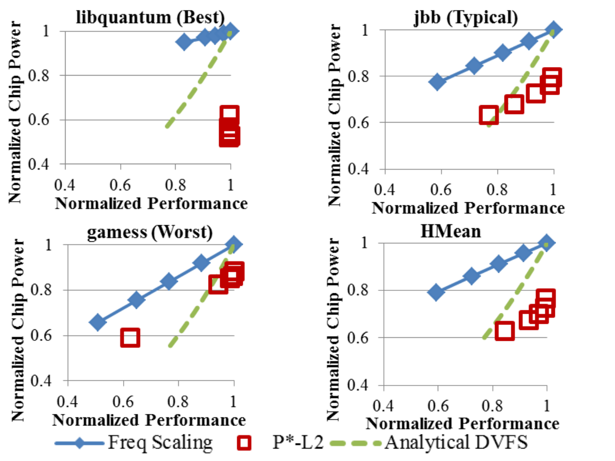

Figure 5. Power-performance of L2 P-states normalized to the baseline (lower-right is better).

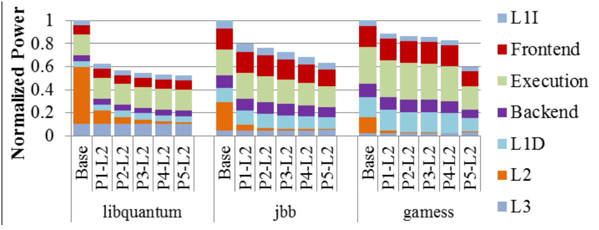

Figure 6. Power breakdown normalized to the baseline.

that low BPKI and low SPKI exhibit less significant power savings—unfortunately also applies, a fact exemplified by *bwaves*. The low BPKI makes this workload category insensitive to front-end stalling; instead it suffers from fewer physical registers, resulting in one power-performance point for all P-states. However, only six out of 33 workloads fit in this category. Most workloads have enough BPKI for our technique to be effective, as represented by the typical workload *povray*.

To understand the performance impact of each front-end technique, we applied one technique at a time to the baseline and measured the resulting IPC degradations normalized to the baseline (Figure 4). In other words, each stack from top to bottom represents the following in cumulative form: IPCs when adding commit-time misprediction recovery (*& Chkpt Removal*); our most aggressive speculation control of allowing only one in-flight unresolved branch (*& Spec Cntrl*); fetch buffer resizing (*& Fetch Buff*); and register file resizing (*& Regs*). The height of the bottom *P5-1* stack represents the IPC of all techniques combined, normalized to the baseline.

As expected, disabling checkpoints has very small performance implications because of the low useful checkpoint rate in the baseline (Section 2). Even the high-SPKI workload, *gobmk,* has only 7% performance degradation from the longer commit-time misprediction penalties, and the fraction becomes negligible for low-SPKI workloads (*povray* and *bwaves*). Adding the most aggressive speculation control results in a more modest performance loss for high- to medium-BPKI workloads (*gobmk* and *povray*). In contrast, the low-BPKI *bwaves* allows few opportunities for the speculation control, thereby showing insensitivity to the technique.

Combining these two techniques with fetch buffer resizing has negligible performance impact for different reasons. On one hand, the speculation-control-sensitive workloads already have much fewer in-flight instructions than the baseline, and the reduced fetching capability does not significantly affect the overall performance. On the other hand, *bwaves* does not need a full-capability front-end to begin with because it has a mostly idle back-end due to the memory intensity as well as the infrequent misprediction-caused pipeline flushes. *bwaves* shows the most sensitivity (28%) to the reduced physical register count. The large IPC loss is also an indication that the baseline design is not skewed during P0.

Even though our approach targets only the core front-end, the rest of the pipeline and the caches also result in power reductions. *gobmk*, for example, lowers the power of the execution by 47%, back-end by 50%, and L1-D by 50% with *P5-FE* (results not shown) by scaling down the aggressiveness of the front-end.

L2 cache P-states. To evaluate the L2 P-states, we selected best (*libquantum*), typical (*jbb*), and worst (*gamess*) workloads. Figure 5 plots the power-performance scaling by our microarchitectural P-states (Table 2), frequency scaling, and the derived DVFS. With our mechanisms, all three workloads achieve much more power savings for the same performance than frequency scaling—and even more than DVFS in many cases—by addressing both dynamic and static power of the L2. We achieve these better power-performance trade-offs despite our simple, non-optimized use of drowsy mode and L2 way shrinking. The effectiveness depends on a workload's memory intensity and the L2 miss-rate sensitivity to smaller associativity. Memory-intensive workloads (e.g., *libquantum*) are the most significantly impacted because more than a third of the chip power comes from the static-power dominated L2 in the baseline. Hence, placing the L2 into drowsy mode reduces *libquantum*'s chip power by 43%. Although the rest of the workloads similarly benefit from drowsy mode, the much

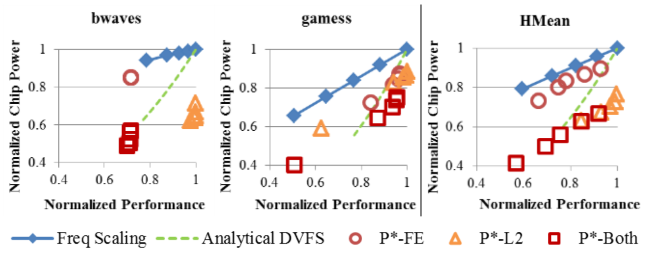

Figure 7. Power-performance of the combined P-states normalized to the baseline.

smaller L2 power fractions in the baseline cause the chip power reductions to be 24% and 14% for *jbb* and *gamess*, respectively.

P3-L2 through *P5-L2*, on the other hand, target both static and dynamic power of the L2 by power-gating some of the ways. Performance responds differently, depending on the L2 utilization and the miss-rate sensitivity to reduced associativity. Because the memory-intensive *libquantum* already has many misses to the memory, the workload has excellent power scaling with a 0.1% performance loss for up to 5% additional power savings. The compute-intensive *gamess* represents the opposite extreme. It has a small L2 power fraction, and its L2 miss rate remains largely insensitive as long as the L2 has at least two ways (*P4-L2*), resulting in the clustered four points in the upper-right region. However, *gamess* becomes the most sensitive workload once the associativity is reduced to one (*P5-L2*) and it experiences a significant decrease in performance. Yet our lowest P-state consumes 79% less power than frequency scaling for comparable performance. Most workloads (e.g., *jbb*) exhibit less L2 miss-rate sensitivity and yield more gradual power-performance trends, most of which outperform DVFS.

Combining front-end and L2 P-states. Because each of these two P-state implementations targets distinct parts of the processor, workloads that work well under one implementation do not always work as well under the other. Therefore, we applied the mechanisms together and plotted the resulting power-performance points in Figure 7. *P*-Both* represents the new combined P-state mechanisms, and the rest is the same as the data in Figure 3 and Figure 5. As expected, combining the mechanisms helps further optimize performance across all workloads. Our lowest P-state consumes less than half the power of frequency scaling, on average, for comparable performance. Even for *bwaves*, the worst-performing workload under the front-end P-state, implementing L2 P-states significantly decreases power with only a minor impact on performance, and implementing both front-end and L2 P-states reduces power even further, albeit with a larger impact on performance. For *gamess*, which is the worst-behaving benchmark when using only the L2 P-states, the combined lowest P-state uses only 61% as much power for comparable performance. In summary, all workloads perform as well or better than DVFS, making our P-state mechanisms a promising alternative to DVFS.

6. RELATED WORK

Previous microarchitectural designs also target low power [2] and/or a wider power range [10][12]. Most low-power proposals aim at improving power efficiency during peak performance, while we focus on efficient power and performance scale-down. Cebrian *et al*. use microarchitectural techniques to execute within a specific power budget [4], resulting in a complex control mechanism and ignoring performance impacts.

Our work incorporates many previously proposed techniques (Sections 2 and 3) but re-examines them in a different context: lowering the power scaling curve. Hence, we extract power savings those techniques offer for performance loss comparable to frequency scaling.

7. CONCLUSIONS

We proposed a new approach to implement P-states using previously proposed microarchitectural techniques. Our front-end and L2 P-states showed that we achieve more efficient power-performance scaling than frequency scaling across all workloads, and even exceed the power savings of DVFS in some cases.

8. ACKNOWLEDGMENTS

We thank John D. Davis, Mark D. Hill, and anonymous reviewers for their comments on the paper. This work was supported in part by the National Science Foundation (CNS-7020565, CNS-0916725, CCF-1017650, and CNS-1117280), Sandia/DOE (#MSN134324), and the University of Wisconsin. Wood has a significant financial interest in AMD.

9. REFERENCES

[1] Alameldeen et al. Variability in architectural simulation of multi-threaded workloads. In *Proc. of HPCA*, February 2003.
[2] Albonesi et al. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(2):49–58, 2003.
[3] Brooks et al. Wattch: A framework for architecture-level power analysis and optimization. *Proc. of 27th ISCA*, 2000.
[4] Cebrian et al. Efficient microarchitecture policies for accurately adapting to power constraints. In *Proc. of the Intl. Symp. on Parallel & Distributed Processing*, 2009.
[5] Chandrakasan et al. Technologies for ultradynamic voltage scaling. In *Proc. of the IEEE*, 2010.
[6] Dreslinski et al. Near-threshold computing: reclaiming Moore's Law through energy efficient integrated circuits. *Proc. of the IEEE*, 2010.
[7] Flautner et al. Drowsy caches: simple techniques for reducing leakage power. In *Proc. of the 29th ISCA*, 2002.
[8] Gochman et al. The Intel Pentium M processor: microarchitecture and performance. *Intel Tech. J.*, 2003.
[9] Henning. SPEC CPU2006 Benchmark Descriptions. *Computer Architecture News*, 2006.
[10] Ipek et al. Core fusion: accommodating software diversity in chip multiprocessors. In *Proc. of the 34th ISCA*, 2007.
[11] Kalla et al. Power7: IBM's next-generation server processor. *IEEE Micro*, 2010.
[12] Kumar et al. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *Proc. of the 36th Symp. on Microarchitecture*, 2003.
[13] Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 2002.
[14] Manne et al. Pipeline gating: speculation control for energy reduction. In *Proc. of the 25th ISCA*, 1998.
[15] Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS). *Comp. Arch. News*, 2005.
[16] Moshovos. Checkpointing alternatives for high performance, power-aware processors. In *Proc. of ISLPED* 2003.
[17] Naveh et al. Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*. 2006.
[18] Rogers et al. The Core-C6 (CC6) Sleep State of the AMD Bobcat x86 Microprocessor. In *Proc. of ISLPED*, 2012.
[19] Seznec et al. A case for (partially) Tagged Geometric history length branch prediction. *J. of Instr. Level Parallelism*, 2006.
[20] Shyamkumar et al. CACTI 5.1. Technical Report HPL-2008-20, Hewlett Packard Labs, 2008.
[21] Vandeputte et al. Offline phase analysis and optimization for multi-configuration processors. *Architectures, Modeling, and Simulation*, 2005.