Underclocked Software Prefetching: More Cores, Less Energy

Power consumption is a concern for helper-thread prefetching that uses extra cores to speed up the single-thread execution, because power consumption increases with each additional core. This article analyzes the impact of using power-saving techniques in the context of intercore prefetching (ICP), and shows that dynamic frequency scaling coupled with ICP is a more energy-efficient way to improve the single-thread execution.

Md Kamruzzaman
Steven Swanson
Dean M. Tullsen
University of California, San Diego

Multicore architectures are ubiquitous today, but many applications remain serial for several reasons, including the prevalence of nonparallelized legacy codes, the difficulty of parallel programming, and the inherently serial nature of many programs. For parallel codes, we can apply multiple cores to the computation and improve performance. In many scenarios, however, it is even more important that we can apply more cores and reduce energy or improve energy efficiency (for example, using energy-delay products).

This same opportunity exists for inherently serial code. Using helper-thread techniques on a multicore processor, we can decrease not only runtime for serial code but consumed energy as well. Furthermore, we demonstrate a helper-thread runtime system, which is uniquely constructed to exploit higher core counts for decreased energy and to manage the complex energy/performance/core count tradeoffs. In fact, nontraditional parallelism (such as helper thread-based parallelism, where parallel speedup is achieved without dividing up the original computation) creates new opportunities for energy optimization. Most parallel code creates homogeneous threads, leaving little opportunity for per-thread energy optimization. Helper-thread parallelism is inherently heterogeneous, meaning each thread can be optimized separately (for example, running the main thread at a high frequency and helper threads at a low frequency). That heterogeneity exists across two dimensions. First, the main thread is computationally intensive while the helper thread is memory intensive, meaning their tolerance to frequency scaling is likely to differ. Second, they are typically imbalanced, letting us exploit slack by scaling a fast helper thread's frequency.

This article examines cache-prefetching helper threads. Traditionally, prefetching helper threads have either had to execute on the same hardware-multithreaded core (letting them prefetch into the Level-1 [L1] cache)[1,2] or execute on a separate core but only target a distant shared cache.[3] Thus, we would have had to choose between performance or the ability to control the power of the prefetcher for a study like this. Intercore prefetching (ICP) enables the helper threads to execute on distinct cores, yet still target the L1 cache, on existing architectures.[4] It does so by migrating the main thread to a core for which the helper has already prefetched the data, simultaneously migrating the helper thread to another core to preload new data into that cache. In this way, the main thread is constantly executing in a core for which the current data working set is already present in the data cache, regardless of the working set's size. Because helper threads execute in different cores, their frequency and, in the future, voltage can be decoupled from that of the main thread.

The combination of intercore prefetching and per-core frequency control creates several unique opportunities. First, with the helper thread in a separate core, we can manage the power of the main thread and the helper threads separately, exploiting the natural heterogeneity. Second, this lets us minimize the incremental power consumed by the helper threads, while preserving ICP's full performance gain, resulting in dramatic reductions in energy. Third, ICP easily handles slow helper threads by simply using more cores—thus, we can exploit the complete spectrum of core count versus per-core power to find the optimal operating point.

This work shows that ICP itself can decrease energy over traditional execution, by a factor of two. Additionally, by controlling the helper threads' frequency, we can match the performance of the best ICP result, while further cutting the energy by as much as 47 percent. However, we also show that this potential is not realized on current systems, because ICP is sensitive to the (currently very high) latency of changing frequency. However, future systems with lower latency will be able to exploit this technology.

ICP with frequency scaling

We apply frequency scaling to maximize ICP's energy efficiency using *ICP-dynamic*, a technique that uses per-core dynamic frequency scaling, potentially allowing different frequencies for the main thread and the helper threads. The optimal use of dynamic power control depends heavily on the system's underlying capabilities and properties, so we need to understand processor power and frequency management before analyzing frequency scaling's impact on applications.

Processor power management

Most modern processors support multiple performance states (P-states) to manage power consumption. Each P-state is characterized by an operating voltage and frequency. A lower P-state means the processor operates at a lower voltage and frequency. Power consumption follows the equation $P \propto V^2 f$ where P, V, and f stand for power, voltage, and frequency, respectively. So, there is a considerable increase in power consumption when the processor moves into a higher P-state.

In the case of multicore processors, different systems allow different levels of heterogeneity—the same voltage and frequency on each core, the same voltage but a different frequency, or different voltage and frequency settings per core. Systems like Intel Nehalem require the same voltage and frequency for all active cores, whereas Advanced Micro Devices (AMD) systems (starting from generation 0x10) allow cores to run in a different frequency but with the same voltage. AMD calls this *independent dynamic core* technology. There are no general-purpose processors yet that allow different voltage and frequency settings per core. Therefore, we use per-core dynamic frequency scaling in this study—future systems that allow per-core dynamic voltage and frequency scaling (DVFS) will see greater improvements in energy efficiency than what we measure and project in these results.

The AMD Phenom processor supports five P-states, including one turbo state. Table 1 shows the different P-states for the AMD Phenom system that we use. P0, the turbo state, is only available when at least half of the total cores are inactive. AMD Phenom allows different P-states for different cores. In this case, all cores will operate at the highest P-state's voltage, but will run at different frequencies. So, if core-0 is in P2,

Table 1. Operating voltage and frequency for different P-states in the Advanced Micro Devices (AMD) Phenom system used in our experiments.

P-state	Voltage (V)	Frequency (MHz)
P0	1.25	3,100
P1	0.90	2,600
P2	0.85	2,000
P3	0.76	1,400
P4	0.76	800

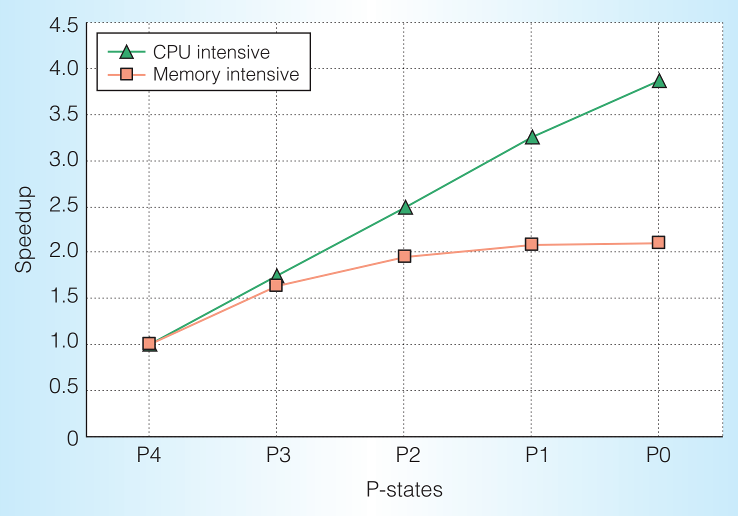

Figure 1. Speedup for CPU-intensive and memory-intensive kernels in different P-states. The memory-intensive kernel is less sensitive to the change of frequency.

and core-1 is in P4, both cores will operate at 0.85V, but at 2,000- and 800-MHz frequency, respectively. This consumes less power than running both cores in P2, but uses more power than using different voltages.

Impact of frequency scaling

Applications show different sensitivity to changing P-states (that is, frequencies). Figure 1 shows the impact on performance using different P-states for two different kernels—one that is CPU intensive, and one that is memory intensive. As expected, the memory-intensive kernel is less sensitive to the change of frequency. If we change the frequency from 800 MHz to 3,100 MHz (an increase of 3.9×), performance only improves by 2.1×. This happens for two reasons. First, the memory subsystem runs in a different (and slower) frequency domain than the cores, so changing the core frequency does not impact the memory performance. Second, the CPU stalls frequently while executing memory-intensive kernels, and reducing the frequency decreases the latency of CPU stalls (measured in processor cycles). For CPU-intensive kernels, performance is highly proportional to the frequency.

Decreasing frequency makes an application run longer, but at lower power, so the impact on energy is unclear. Additionally, two competing factors make the problem even more complex. First, higher frequency requires higher voltage, so there can be a cubic increase in the power consumption. This favors lower frequency. On the other hand, other system components, including the memory hierarchy, consume a large amount of fixed power, irrespective of whether the processor is executing, and this favors executing as quickly as possible. Current systems are not energy proportional and the latter effect tends to dominate. Therefore, for CPU-intensive applications, the most performance-efficient execution is usually also the most energy efficient. However, the scenario changes for the memory-intensive applications. Prior works address these issues[5-7] and show the effectiveness of DVFS for energy efficiency.

These insights impact our results in several ways. First, ICP transforms the main thread from memory intensive to CPU intensive. Second, the helper threads themselves are memory intensive; however, because our helper threads exhibit much higher memory-level parallelism than typical code, their forward progress scales with frequency somewhat more than typical memory-intensive code.

Intercore prefetching

ICP uses helper threads to prefetch data into private caches, paving the way for the main thread that performs the program's actual computation.[4] Execution proceeds by chunks, where each chunk is a set of loop iterations. In the simplest case of one main thread and one helper thread, the main thread executes chunk-1 in core-0, and the helper thread prefetches data for chunk-2 in core-1. Once they complete their chunks, they swap cores, and the main thread finds the data it needs to execute chunk-2 already in cache. In the meantime, the helper thread starts prefetching chunk-3 in core-0, and so on. This process decouples high-latency cache misses from the main computation and lets them proceed in parallel.

There are two key parameters while applying ICP—the number of prefetch threads and the chunk size. Prefetch threads are independent and can prefetch different chunks in parallel to reduce the average prefetching time. By adjusting the number of prefetch threads, ICP can ensure that the prefetching is done before the main thread starts processing. On the other hand, chunk size (a chunk's approximate memory footprint) determines the level of the cache hierarchy that the prefetching will target and how often the main thread will need to synchronize with a helper thread to swap cores. Smaller chunks fit in the L1 cache and make prefetching the most effective, but they also maximize synchronization and migration overhead.

ICP and frequency scaling

ICP provides an opportunity to apply per-core dynamic frequency scaling to make single-thread execution more power and energy efficient.

ICP decouples the memory-intensive prefetching stream from the execution stream. The prefetching stream is less sensitive to core frequency and can execute in parallel. Thus, we can use multiple prefetch threads to decrease the average prefetching time. Conversely, the execution stream is more sensitive to core frequency and is strictly sequential. Thus, the only way to accelerate the main thread (after ICP is already applied) is to use a higher frequency.

ICP-dynamic can use different frequencies for the prefetching and execution streams. However, because the main thread and helper threads swap cores at frequent intervals, we can't use a fixed frequency for each core. Instead, we must change the frequency dynamically when a core switches from prefetching to execution or vice versa. This makes ICP-dynamic highly sensitive to the latency of changing frequency, which on current systems is very high.

We can explain the impact of ICP-dynamic (ignoring for now the latency of frequency changes) using the following equation:

$$T = \max(e, (e + p)/n) \qquad (1)$$

Here, T is the average time to process a chunk using ICP, e is the time to execute (after data is already prefetched) a chunk by the main thread at some frequency, p is the time required to prefetch data for a chunk by the helper thread at some frequency, and n is the total number of cores used—for example, in the case of one helper thread, n is 2. This equation holds whether the main thread and helper threads use the same or different frequency.

Equation 1 demonstrates several things. ICP's theoretical maximum speedup is b/e, where b is the time to execute a chunk without ICP. When e is larger than p, we need only one helper thread to get the best speedup. In this case, the helper thread waits $e - p$ to synchronize with the main thread. We can lower the helper thread's operating frequency while $e \geq p$. This will reduce power consumption without sacrificing any performance benefit. When p is larger than e, a more complicated case arises. Lowering either stream's frequency will hurt performance in this case. However, we can reduce the prefetching stream's frequency if we increase the number of prefetch threads. A lower frequency increases p, but increasing n easily offsets that because prefetching is less sensitive to frequency changes. Meanwhile, using a lower frequency reduces per-core power consumption.

ICP-static is an alternative option. In this case, core frequencies remain fixed over time but different cores can operate at different frequencies. Threads in this case alternate in different frequencies. ICP-static is not typically a great option compared to ICP-dynamic or *ICP-same* (all cores at the same frequency), but it could be used to keep the power consumption under some budget.

Methodology

Here, we describe our experimental methodology—the application kernels that we target and the systems that we use to evaluate ICP-dynamic and ICP-static.

Table 2. Memory hierarchy information, migration cost, and the latency for frequency and voltage scaling for the AMD Phenom processor.

CPU components	Parameters
CPU model	AMD Phenom II X6 1035T
Number of cores	Single socket, six cores
Level-1 (L1) cache	64-Kbyte private cache, three cycles
Level-2 (L2) cache	512-Kbyte private cache, 15 cycles
Level-3 (L3) cache	6-Mbyte shared cache, 40 cycles
Memory	8 Gbytes, 120 cycles
Latency to frequency change	4 to 9 μs
Latency to voltage and frequency change	14 to 70 μs
Idle-state power consumption	94.5 W

Evaluation systems

We use a six-core AMD Phenom desktop system running Ubuntu Linux 2.6. Table 2 gives more information about the system. From Table 2, it takes more time to change voltage than just frequency. The time also scales with the difference between the P-states. For example, when a core switches from P0 (1.25 V, 3,100 MHz) to P4 (0.76 V, 800 MHz), it takes around 70 μs. However, switching to P2 (0.85 V, 2,000 MHz) from P0 takes around 14 μs. The core is unavailable during the P-state switching time.

We compile all codes using GCC version 4.5.2 with the -O3 optimization level and keep hardware prefetching enabled for all experiments. We do all power measurements using a ‘‘Watts up? .Net’’ power meter (http://www.wattsupmeters.com), which measures the system’s power at the wall and reports power consumption in 1-second intervals with ±1.5 percent accuracy.

We read and write the model-specific registers using privileged instructions *rdmsr* and *wrmsr* to change a core’s P-state.

Application kernels

We extract kernels from 11 applications from several different benchmark suites—*bt*, *cg*, *mg*, *lu*, and *sp* from NASA Advanced Supercomputing (NAS); *equake* from Spec2000; *lbm*, *libquantum*, *mcf*, and *milc* from Spec2006; and *svm-rfe* from Minebench. All these kernels are key processing loops of the applications and dominate their performance. We exclude benchmarks from these suites that are not written in C/C++ or are not memory intensive. Using kernels instead of the full application lets us keep the power consumption steady for more accurate measurement.

Results

We evaluate ICP frequency scaling in several steps. First, we measure the system’s baseline power consumption behavior. Then we measure the performance improvements and power consumption of our kernels using ICP-same. Third, we apply ICP-dynamic and ICP-static and measure the performance, power, and energy. Finally, we simulate the effect of lowering the latency of frequency or voltage scaling.

Here, we use the term *active power* to represent the power that we get by subtracting idle power from the power measured at the wall. So, if the system uses 120 watts of power, the active power is $120 - 94.5 = 25.5$ W. Similarly, we can compute active energy.

Power consumption for different P-states

We measure the power consumption for different P-state combinations to understand how it varies when the P-state changes. For this experiment, we use a simple loop doing arithmetic operations and no memory accesses. Table 3 shows the power consumption for different combinations. The second column represents the power when one core is active, and columns 3 through 7 represent the additional power required when we activate one more core at a particular P-state. So, when two cores run at P0, the power consumption is $125.6 + 13.8 = 139.4$ W.

Table 3 shows three things. There is a large jump in active power consumption when we increase the voltage (active power is 2.1 W at P4 and 31.1 W at P0). Second, as expected, the incremental power of adding a core depends on the first core’s P-state, because the first core (the one at higher P-state in this table) determines the voltage. Third, it takes more power to use a high P-state core rather than multiple low P-state cores. For example, two cores at P0 will consume 139.4 W, but one core in P0 and three

Table 3. Power consumption for different P-state combinations.

		Additional core power (W)				
Core-0 P-state	**Power consumption (W)**	**P0**	**P1**	**P2**	**P3**	**P4**
P0	125.6	13.8	N/A	7.5	6.1	3.3
P1	106.8	N/A	8.4	6.6	4.7	2.9
P2	102.8	N/A	N/A	6.0	4.4	2.6
P3	98.1	N/A	N/A	N/A	3.9	2.3
P4	96.6	N/A	N/A	N/A	N/A	2.2

cores in P4 will consume 125.6 + 3.3 × 3 = 135.5 W. This last characteristic is an opportunity for ICP because we can often replace one fast prefetch thread with multiple slower ones with no loss in performance.

ICP without frequency heterogeneity

ICP provides significant performance improvements for memory-intensive kernels. Figure 2 shows the speedup averaged over all kernels using traditional ICP (ICP-same) and without using ICP (No-ICP) at different P-states. We use a chunk size of 128 Kbytes and apply ICP with both one and two helper threads.

From Figure 2, ICP's effectiveness increases with higher P-states (higher frequency). The processor-memory gap accounts for this. Cache misses are more expensive in P0 than in P4, so eliminating those profits more. We also see that ICP continues to improve performance linearly with frequency, whereas without ICP, performance tapers off at a higher frequency. This is expected—without ICP, most of these applications are memory bound; with ICP, they become CPU bound.

Performance of ICP-dynamic and ICP-static

We compare ICP-same (all cores at the same frequency), ICP-static (cores at different, but unchanging, frequencies, whether running the main thread or helper thread), and ICP-dynamic (cores change frequencies when transitioning from the main thread to helper thread and back) on our AMD system. We compute the power measurements and performance for all possible P-state combinations for all of our kernels. We use 128-Kbyte chunks and vary the number of helper threads between one and two.

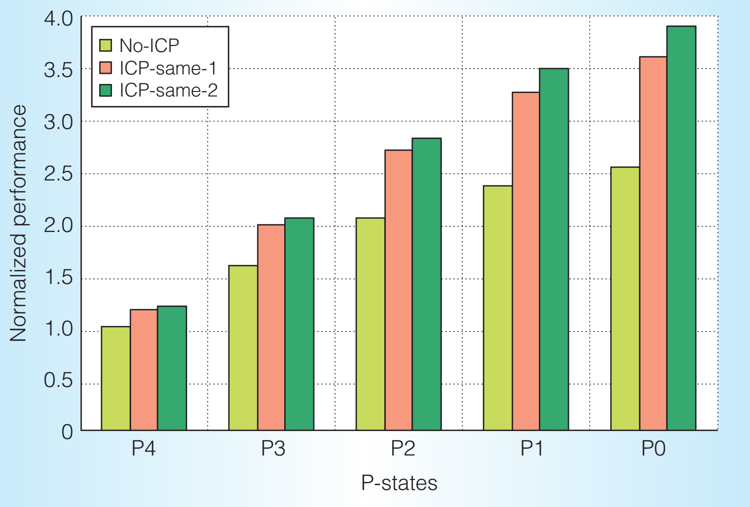

Figure 2. ICP-same speedup in different P-states. Here, 1 and 2 represent the number of helper threads.

In each case, we measure the power consumption, active power, energy, and active energy.

Because there are so many possible P-state combinations, we show a scatter plot of the possible combinations in Figure 3, plotting active energy and speedup. The Pareto-optimal points (connected by the line) are the interesting ones—those where no other single data point is both above and to the left. From this figure, we see that no ICP-static or ICP-dynamic points are Pareto-optimal, and ICP-same works best—six out of seven Pareto-optimal points are ICP-same. Looking more closely, some of the ICP-dynamic points are Pareto-optimal for individual kernels, but not in general across all kernels. This is a direct result of the high latency to switch frequency or voltage. As we mentioned earlier, ICP-dynamic performs

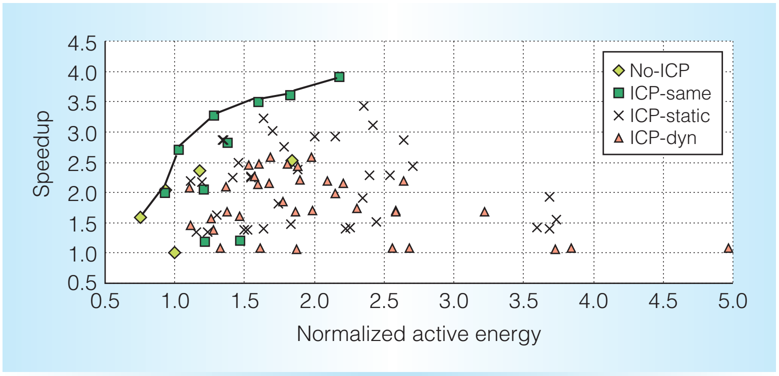

Figure 3. Scatter plot of normalized speedup and active energy consumption averaged over all kernels for different P-state combinations. The points on the line are Pareto-optimal.

two P-state changes per chunk and in some cases we see a nasty side-effect of the chip P-state policy—consider the case where the main thread is at P1 and the prefetch threads are at P3. In that case, the main thread determines the chip voltage. But, if the core running the main thread finishes early, it starts a prefetch thread, and the processor will seek to change the voltage because all threads are temporarily at P3, resulting in an expensive transition.

Increasing chunk size reduces the frequency of P-state changes. However, using a chunk size larger than 512 Kbytes (the size of the L2 cache) will drastically impede ICP performance. We run the above experiment with a 512-Kbyte chunk size and find that still none of the ICP-dynamic points are Pareto-optimal. The average switching interval in P0 becomes 100 μs compared to 24 μs with a 128-Kbyte chunk, and the performance drop is around 15 percent. On the other hand, the P-state change latency varies from 4 to 70 μs—the same order of magnitude as the chunk execution time in some cases. We expect a drop in the future voltage and frequency change latency, which will change these results significantly.

Sensitivity to P-state change latency

ICP-dynamic does not perform to its potential because of our experimental system's high P-state change latency. Recent research shows the potential for nanosecond-level voltage/frequency switching.[8] The authors report a voltage transition between 4 and 1.4 V within 20 ns. Also, systems such as IBM Power7 can gradually change core clock frequency while the core is fully operational—it has essentially zero P-state change latency.[9]

We built a simulation model that analyzes the impact of varying P-state change latency. We developed the model using the timing and power consumption data collected from our AMD Phenom system. The model requires six values for each P-state—time required to execute a chunk without ICP, time to prefetch data for a chunk, time to execute a chunk when the data is already in the cache, and power requirements in the previous three cases. We fed this matrix to our model, and it computed for each core the amount of time the core prefetches, executes, and spins for each P-state combination. For ICP-dynamic, the model also considered the latency to change P-states. The model needed two additional power consumption values—when the core is spinning, and when it is switching to a new P-state. The first was easy to measure. For the second, we took the average power consumption of the participating P-states. The model works for both ICP-dynamic and ICP-static, and handles any number of helper threads.

For a particular P-state combination, our model computes each participating core's power consumption and a chunk's average processing time (T in Equation 1), which we again use to compute the performance gain, energy, or energy-delay product (EDP). We denote a P-state combination by Pxy, meaning that the main thread runs at Px, while the helper thread runs at Py. So, P00 refers to ICP-same at P0, while P03 indicates ICP-dynamic with main thread at P0 and helper threads at P3.

Figure 4 shows the scatter plot similar to Figure 3 (128-Kbyte chunk, one to two helper threads), assuming there is no latency to change P-states. ICP-dynamic is Pareto-optimal with respect to all other techniques except P33 and P00. P33 takes the least energy of all ICP combinations, but the performance of that point is quite low. P00 provides a little more speedup at the expense of much larger energy. If we consider other metrics, such as EDP or active EDP, ICP-dynamic dominates those two points as well.

At the top of Figure 4, we see that ICP-dynamic comes within 1 percent of the maximum (P00) speedup (5.28×), using the P01 combination, which consumes 26 percent less active energy. Alternatively, we can get within 2 percent with P02 while consuming 38 percent less active energy.

We can underclock the helper threads further while using more cores to attain the same speedup as ICP-same. In our experiments, P3 seems to be the best choice for helper threads. Figure 5 shows the impact of varying the number of helper threads for the five ICP-same and three ICP-dynamic choices—P03, P13, and P23. We use up to five helper threads (six cores). The graph shows the advantage of using underclocked prefetchers. P00 reaches the theoretical maximum speedup by ICP using five cores. P03 eventually catches up to the speedup provided by P00 using six cores, but consumes 47 percent less active energy. Similarly, P13 consumes 32 percent less active energy and still provides the same speedup as P11. ICP-dynamic is effective in saving energy with no sacrifice in performance even over the most aggressive ICP scheme.

So far we have shown results with either full frequency switch latency or no latency. Figure 6 shows the change in speedup for P03 and P13 when the latency varies from 0 to 12 μs. We present the results for both a 128- and 512-Kbyte chunk size to better understand the sensitivity. Note that, given this graph's granularity, the most accurate estimate of expected future switch times (in a few nanoseconds[8]) is actually the 0 point. The vertical line shows the upper limit of the latency, beyond which the ICP-dynamic point would no longer be Pareto-optimal. For 128-Kbyte chunks, this happens at around 2 μs of latency. Use of the 512-Kbyte chunk tolerates much higher latency. In Figure 6b, P13 stays Pareto-optimal until 11 μs. However, the performance drop is significant for 512 Kbytes. If we consider both the 128- and 512-Kbyte set of points in our scatter plot, none of the 512-Kbyte points become Pareto-optimal

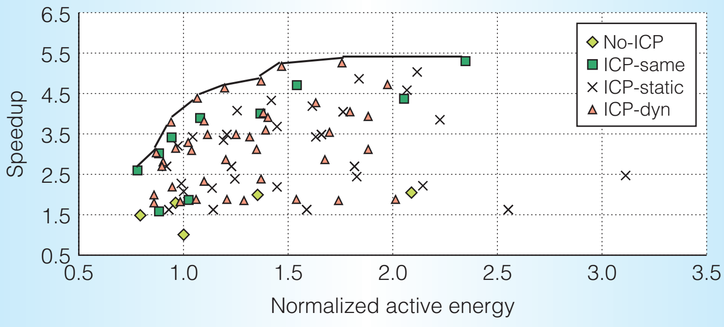

Figure 4. Scatter plot of normalized speedup and active energy consumption assuming zero latency to change P-states. Pareto-optimal points are shown on the line.

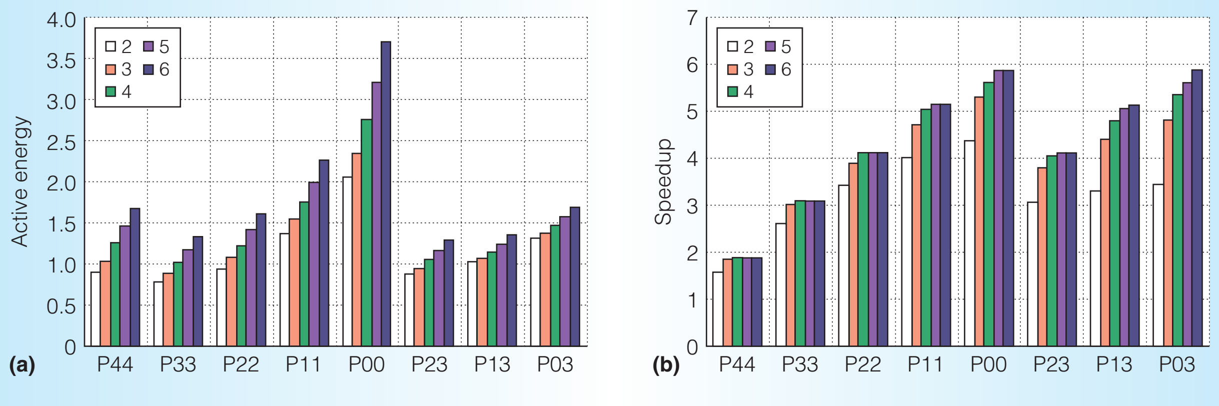

Figure 5. The impact of adding more cores for different P-state combinations: active energy (a) and speedup (b). Pxy means that the main thread is in state Px and the helper threads all operate in state Py.

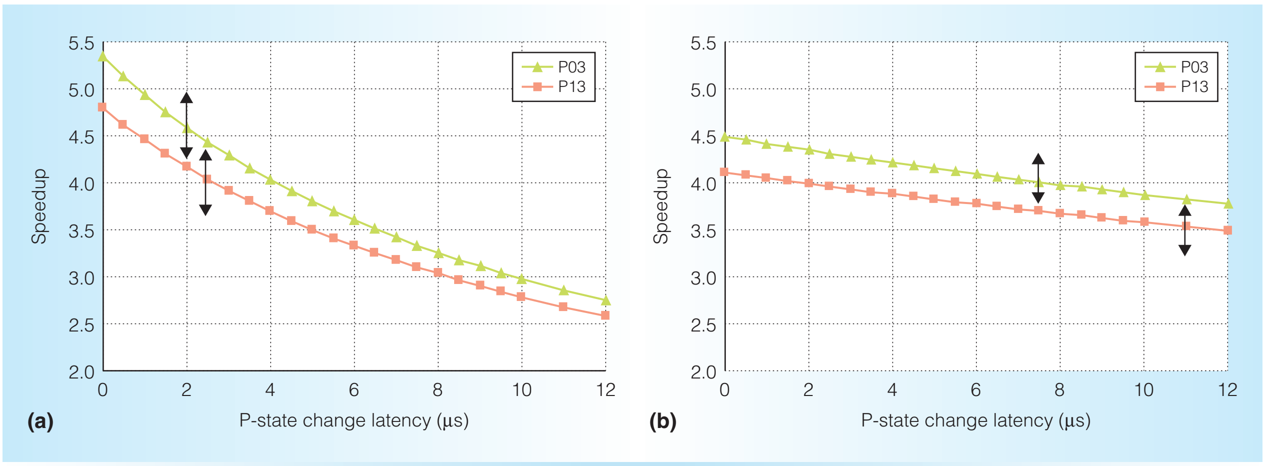

Figure 6. Impact of varying the latency of P-state change on P03 and P13: 128-Kbyte chunk (a) and 512-Kbyte chunk (b). ICP-dynamic gives Pareto-optimal points as long as the latency is lower than that marked by the vertical lines. Larger chunks are less sensitive to the frequency change latency.

for any P-state change latency. So, the additional performance provided by the smaller chunks is critical.

Our model also lets us predict the effects of per-core voltage scaling. With the same P-states, zero latency switching, and the ability to vary voltage across cores, P03 can achieve the same speedup as P00, consuming 54 percent less active energy (frequency scaling alone achieves 47 percent).

We demonstrated techniques that enable the use of more processor cores to improve performance and energy efficiency, even on completely serial code. We combined a multicore-based helper-thread technique with explicit control of core frequency. The ability to manage the helper thread's power consumption enables the system to minimize the helper thread's incremental power without compromising the resulting performance gains. This results in heavy gains in both energy and EDP.

Furthermore, we showed that the long latency for frequency switching on current systems is a barrier to full realization of this technique, but future systems with faster switching will be able to benefit greatly. Compared to the state-of-the-art helper-thread prefetcher, we can match the same performance while consuming half the energy.

This work promotes several interesting research opportunities. We can apply techniques similar to ICP-dynamic for data-parallel and pipeline-parallel code. The challenge, in that case, is to figure out the right distribution of main thread and helper threads, and the extent to which we must underclock a particular thread.

Work is also required to find the best set of voltage and frequency points (other than what is available on current systems) to optimize these techniques (including ICP-dynamic) further. MICRO

Acknowledgments

We thank the anonymous reviewers for providing useful feedback. This work was funded in part by National Science Foundation grants CCF-1018356 and CCF-0643880 and by Semiconductor Research Corporation grant 2086.001.

References

1. J.D. Collins et al., ''Speculative Precomputation: Long-Range Prefetching of Delinquent Loads,'' *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM, 2001, pp. 14-25.
2. C.-K. Luk, ''Tolerating Memory Latency through Software-Controlled Pre-execution in Simultaneous Multithreading Processors,'' *Proc. 28th Ann. Int'l Symp. Computer Architecture* (ISCA 01), ACM, 2001, pp. 40-51.

3. J. Lu et al., ''Dynamic Helper Threaded Prefetching on the Sun Ultrasparc CMP Processor,'' *Proc. 38th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS, 2005, pp. 93-104.
4. M. Kamruzzaman, S. Swanson, and D.M. Tullsen, ''Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads,'' *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 11), ACM, 2011, pp. 393-404.
5. K. Choi, R. Soma, and M. Pedram, ''Dynamic Voltage and Frequency Scaling Based on Workload Decomposition,'' *Proc. Int'l Symp. Low Power Electronics and Design* (ISLPED 04), ACM, 2004, pp. 174-179.
6. C.-H. Hsu and U. Kremer, ''The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 03), ACM, 2003, pp. 38-48.
7. W. Kim et al., ''System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators,'' *Proc. IEEE 14th Int'l Symp. High Performance Computer Architecture* (HPCA 08), IEEE CS, 2008, pp. 123-134.
8. W. Kim, D. Brooks, and G.-Y. Wei, ''A Fully-Integrated 3-Level DC-DC Converter for Nanosecond-Scale DVFS,'' *IEEE J. Solid-State Circuits,* Jan. 2012, pp. 206-219.
9. M. Floyd et al., ''Introducing the Adaptive Energy Management Features of the Power7 Chip,'' *IEEE Micro,* Mar. 2011, pp. 60-75.

Md Kamruzzaman is a PhD candidate in the Computer Science and Engineering Department at the University of California, San Diego. His research interests include software optimization techniques for high-performance and energy-efficient computing in parallel architecture, locality-aware programming models, and runtime systems. Kamruzzaman has a BS in computer science and engineering from Bangladesh University of Engineering and Technology. He is a member of the ACM.

Steven Swanson is an associate professor in the Computer Science and Engineering Department at the University of California, San Diego, where he leads the Non-Volatile Systems Laboratory. His research interests include architectural, systems, and security issues surrounding nonvolatile, solid-state memories as well as dark silicon architectures and multithreading. Swanson has a PhD in computer science from the University of Washington. He is a member of the ACM.

Dean M. Tullsen is a professor in the Computer Science and Engineering Department at the University of California, San Diego. His research interests include architecture and compilers for multicore and multithreaded architectures. Tullsen has a PhD in computer science from the University of Washington. He is a fellow of IEEE and the ACM.

Direct questions and comments about this article to Md Kamruzzaman, Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Dr., La Jolla, CA 92093-0404; mkamruzz@cs.ucsd.edu.