Exploring Practical Benefits of Asymmetric Multicore Processors

Jon Hourd, Chaofei Fan, Jiasi Zeng, Qiang(Scott) Zhang
Micah J Best, Alexandra Fedorova, Craig Mustard
{jlh4, cfa18, jza48, qsz, mbest, fedorova, cam14}@sfu.ca
Simon Fraser University
Vancouver Canada

***Abstract*—Asymmetric multicore processors (AMP) are built of cores that expose the same ISA but differ in performance, complexity, and power consumption. A typical AMP might consist of a plenty of slow, small and simple cores and a handful of fast, large and complex cores. AMPs have been proposed as a more energy efficient alternative to symmetric multicore processors. They are particularly interesting in their potential to mitigate Amdahl's law for parallel program with sequential phases. While a parallel phase of the code runs on plentiful slow cores enjoying low energy per instruction, the sequential phase can run on the fast core, enjoying high single-thread performance of that core. As a result, performance per unit of energy is maximized. In this paper we evaluate the effects of accelerating sequential phases of parallel applications on an AMP. Using a synthetic workload generator and an efficient asymmetry-aware user-level scheduler, we explore how the workload's properties determine the speedup that the workload will experience on an AMP system. Such an evaluation has been performed before only analytically; experimental studies have been limited to a small number of workloads. Our study is the first to experimentally explore benefits on AMP systems for a wide range of workloads.**

I. Introduction

Asymmetric multicore processors consist of several cores exposing a single ISA but varying in performance [1], [4], [5], [6], [10], [11]. AMP systems are envisioned to be built of many simple slow cores and a few fast and powerful cores. Faster cores are more expensive in terms of power and chip area than slow cores, but at the same time they can offer better performance to sequential workloads that cannot take advantage of many slow cores. AMP systems have been proposed as a more energy-efficient alternative to symmetric multicore processors (SMP) for workloads with mixed parallelism. Workloads that consist of both sequential and parallel code can benefit from AMPs. Parallel code can be assigned to run on plentiful slow cores, enjoying low energy per instruction, while sequential code can be assigned to run on fast cores, using more energy per instruction but enjoying much better performance than if they were assigned to slow cores.

In fact, recent work from Intel demonstrated performance gains of up to 50% on AMPs relative to SMPs that used the same amount of power [1]. Recent work by Hill and Marty [3] concluded that AMPs can offer performance significantly better than SMPs for applications whose sequential region is as small as 5%. Unfortunately, prior work evaluating the potential of AMP processors focused either on a small set of applications [1] or performed a purely analytical evaluation [3]. The question of how performance improvements derived from AMP architectures are determined by the properties of the workloads in real experimental conditions has not been fully addressed. Our work addresses this question.

We have created a synthetic workload generator that produces workloads with varying degrees of parallelism and varying patterns and durations of sequential phases. We also developed a user-level scheduler inside Cascade that is aware of the underlying system's asymmetry and the parallel-to-sequential phase changes in the application. The scheduler assigns the sequential phases to the fast core while letting the parallel phases run on slow cores. As an experimental platform we use a 16-core AMD Opteron system where the cores can be configured to run at varying speeds using Dynamic Voltage and Frequency Scaling (DVFS).

While theoretical analysis of AMP systems indicated their promising potential, these benefits may not necessarily translate to real workloads due to the overhead of thread migrations. A thread must be migrated from the slow to the fast core when the workload enters a sequential phase. The migration overhead has two components: the overhead of rescheduling the thread on a new core and the overhead associated with the loss of cache state accumulated on the core where the thread ran before the migration. In our experiments we attempt

to capture both effects. We use the actual user-level scheduler that migrates the application's thread to the fast core upon detecting a sequential phase, and we vary the frequency of parallel/sequential phase changes to gauge the effect of migration frequency on performance. We use workloads with various memory working set sizes and access patterns to capture the effects on caching. Although the caching effect has not been evaluated comprehensively (this is a goal for future work), our chosen workloads were constructed to resemble the properties of real applications. For the workloads used in our experiments, our results indicate that AMP systems deliver the expected theoretical potential, with the exception of workloads that exhibit very frequent switches between sequential and parallel phases.

The rest of this paper is organized as follows: Section 2 introduces the synthetic workload generator. Section 3 discusses theoretical analysis. Section 4 describes the experiment setup. Section 5 presents the experiment results.

II. The Synthetic Workload Generator

To generate the workloads for our study, we used the Cascade parallel programming framework [2]. Cascade is a new parallel programming framework for complex systems. With Cascade, the programmer explicitly structures her C++ program as a collection of independent units of computation, or tasks. Cascade allows users to create graphs of computation tasks that are then scheduled and executed on a CPU by the Cascade runtime system. Figure 1 depicts a structure typical of the Cascade program we created for our experiments. The boxes represent the tasks (computational kernels), arrows represent dependencies. For instance, arrows going from tasks B, C, and D to task E indicate that task E may not run until tasks B, C, and D have completed. We use the graph structure depicted in Figure 1 to generate the workloads for our study. In particular, we focus on two aspects of the program: the structure of the graph and the type of computation performed by the tasks. All graphs start with a single task (A) to simulate a sequential phase. Once A finishes, several tasks start simultaneously (B, C and D) to simulate a parallel phase. B, C and D perform the same work so that they start and end at roughly the same time. Once B, C and D finish, the next sequential phase (E) is executed. The last phase of all graphs is a sequential phase (I).

While the structures of our generated graphs are similar to the graph shown in Figure 1, they vary as follows:

1. The number of sequential phases can be varied according to the desired phase change frequency. The number of parallel phases is one fewer than the number of sequential phases.

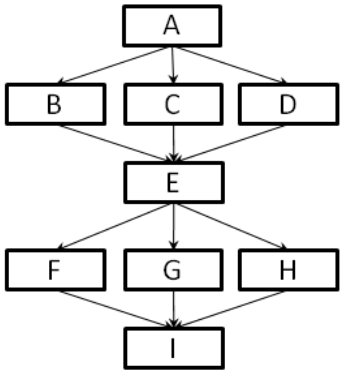

Fig. 1. Task Graph

2. The number of parallel tasks in each parallel phase can also be varied. For our purpose all parallel phases have the same number of parallel tasks.

3. The total computational workload of the entire graph can be precisely specified.

4. We can also specify the percentage of code executed in sequential phases.

Once a percentage of code executed by sequential phases is specified, the corresponding amount of the total workload is distributed equally to each sequential task so that the execution time for each sequential phase is roughly the same. The same method is applied to parallel phases so that all parallel computational tasks (e.g., B, C, D, F, G, and H in Figure 1) have roughly the same execution time.

In our initial experiments, all computational tasks execute an identical C++ function that consists of four algorithms, each taking roughly the same time to complete: (1) Ic, a CPU-intensive integer based pseudo-LZW algorithm; (2) Is, a CPU-intensive integer based memory array shuffle algorithm; (3) Fm, a floating point Mandelbrot fractal generating algorithm (also CPU-intensive); (4) Fr, a memory-bound floating point matrix row reduction algorithm.

III. Theoretical Analysis

Amdahl's Law states that the speedup is the original exexution time divided by the enhanced execution time. Following the method used by Hill and Marty [3], we use Amdahl's Law to obtain a formula to predict a program's performance speedup when its serial and parallel portions and processor performance are known:

$$ExecutionTime = \frac{f}{perf(s)} + \frac{(1-f)}{perf(p) \times x}$$

$$Speedup = \frac{ExecutionTime(Original)}{ExecutionTime(Enhanced)}$$

where f is the percent of code in sequential phases, $perf(s)$ is the performance of serial core with frequency s, $perf(p)$ is the performance of parallel cores with frequency p, x is the number of cores used in parallel phase. $perf(x)$ is a function that predicts the performance of a core with frequency x. For simplicity, we assume that it is proportional to the frequency. This formula assumes that parallel portions are entirely parallelizable and that there is no switching overhead. Both of these assumptions are to simplify the model and not necessarily expected to hold in a practice.

Using this formula, we generate the expected speedup of parallel applications on three systems: (1) SMP_16: a symmetric multicore system with 16 cores, (2) SMP_4: a symmetric multicore system with four cores, where each core runs at 2 times the frequency of each core in SMP_16, and (3) AMP_13: an asymmetric multicore system consisting of one "fast" core (of the speed similar to cores on the AMP_4 system) and 12 "slow" cores (of the speed similar to cores on the SMP_16 system).

The system configurations were constructed to have roughly the same power budget. The power requirements of a processing unit are generally accepted to be a function of the frequency of operation [1]. For a doubling of clock speed, a corresponding quadrupling in power consumption is expected [3]. Thus, a processor running at frequency x will consume four times less power than the processor running at frequency $2x$. Therefore, one core running at speed $2x$ is power-equivalent to four cores running at speed x. As such, the three systems shown above will consume roughly the same power.

Figure 2 shows that using our execution time formula, we determine that the AMP system will outperform the SMP_4 system for all but completely sequential programs and it will outperform the SMP_16 system for programs with sequential region greater than 4%.

The results presented in Figure 2 are theoretical and they mimic those reported earlier by Hill and Marty [3]. In the following sections we present the experimental results to evaluate how close they are to these theoretical predictions.

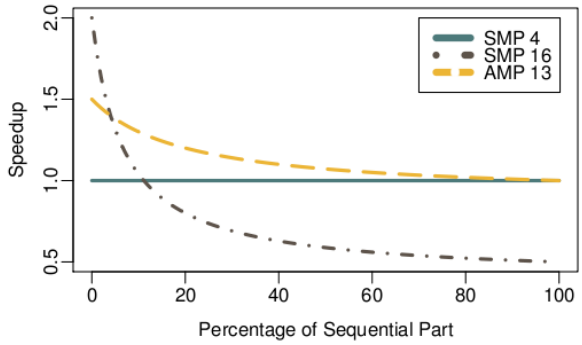

Fig. 2. Theoretical Speedup Normalized Baseline SMP_4

IV. Experimental Setup

A. Experiment Platform

We used a Dell-Poweredge-R905 as our experimental system. The machine has 4 chips (AMD Opteron 8356 Barcelona) with 4 cores per chip. Each core has a private 256KB L2 cache and 2MB L3 victim cache that is shared among cores on the same chip. Our system is equipped with 64GB of 667MHz DDR, and it runs Linux 2.6.25 kernel with the Gentoo distribution.

This system supports DVFS for frequency scaling on a per core basis. The available frequency of AMD Opteron 8356 is from 1.15GHz to 2.3GHz. By varying the core frequency and turning off unused cores, we created three configurations with the same power budget as shown in Table 1.

	Number of Cores	Frequencies
SMP_4	4	4×2.3GHz
SMP_16	16	16×1.15GHz
AMP_13	13	1×2.3GHz + 12×1.15GHz

TABLE I
Experimental configuration

Our user-level scheduler assigns tasks (recall Figure 1) to threads at runtime. Upon initialization, the scheduler creates as many threads as there are cores and binds each thread to a core. When the task graph begins to run, tasks are assigned to threads. On symmetric configurations, scheduling is purely demand-driven: a newly available task is assigned to any free thread. On an AMP configuration, one thread is bound to the fast core and is called the fast thread; other threads are bound to slow cores and are called slow threads. When there is only one runnable task, Cascade assigns it to the fast thread. When there are multiple runnable tasks, they are assigned to slow threads. Although this scheduling policy does not utilize the fast core during the parallel phase, it is a reasonable approximation of a realistic AMP-aware scheduler. Figure 3 demonstrates one example of workload assignment during runtime: each thread is assigned to one core; sequential parts are always executed on thread 0, which is a fast thread,

while parallel parts are executed in parallel on other slow threads.

B. Workloads

We varied several parameters in our graph generator to generate a task graph that could capture major characterizations of real applications.

Iterations: This parameter represents the number of computational tasks of the whole graph, in other words, the execution time of the program. By setting $iterations = 1$, there will be 10^7 computational tasks, each consisting of four C++ algorithms.

Phase_change: This parameter defines how many sequential and parallel phases there are in the graph representing the computation. A graph always starts and ends with a sequential phase. By setting $phase\_change = 2$, there will be two sequential phases and one parallel phase.

Parallel_width: This parameter defines how many parallel tasks are there in the parallel phase. By setting $parallel\_width = 4$, there will be four parallel tasks in the parallel phase.

Sequential_percentage: This parameter defines the portion of code that is sequential. By setting $sequential\_percentage = 50$, 50% of the graph will be executed in sequential phases and 50% will be in the remaining parallel phases.

Setting $iterations = 10$, $phase\_change = 4$, $parallel\_width = 3$, $sequential\_percentage = 20$ will produce the same graph as in Figure 1. Each sequential task will have $\frac{10\times10^7\times20\%}{3}$ algorithmic iterations, while each parallel task will have $\frac{10\times10^7\times80\%}{2\times3}$ algorithmic iterations.

For each experimental configuration, we configure the graph such that the parallel width is equal to the number of cores available in the parallel phase, which corresponds to the way users often configure the threading level in their applications.

V. Experimental Results

In the first experiment we set the number of iterations to 100 and the *phase_changes* parameter to 5. Figure 4 shows the speedup for workloads with sequential percentage ranging from 0%~100% (with 5% increment) on SMP_16 and AMP_13 relative to SMP_4. Comparing these results to the theoretical results in Figure 2 we see that the actual experimental results closely follow the theoretical results with all data on average within 1% range of the analytically derived values. When the workload is purely parallel, SMP_16 outperforms SMP_4 by a factor of 2 approximately, as seen in the theoretical graph. With the increase of sequential code fraction, the fast core in SMP_4 begins to show its power: SMP_4 outperforms SMP_16 beyond the sequential fraction of 15%. Most importantly, AMP_13 almost always outperforms SMP_4 and SMP_16. This is simply because the single fast core speeds up the sequential phases while the remaining slow cores are able to efficiently execute the parallel phases. Only when the sequential code fraction is below 5% does SMP_16 outperform AMP_13 since SMP_16 is better able to utilize a large number of cores for highly parallel workloads.

To experiment with shorter tasks (and thus more frequent phase changes), we reduced the number of total iterations by setting $iterations = 10$ and left the number of phase changes set to five. In this case, the pattern of task graph is the same as in the previous test and the only difference is the length of each task (1/10 of that in previous task graph). The results shown in Figure 5 demonstrate that when the tasks are shorter, the effect of the overhead comes into play. The speedup of AMP_13 is on average 3.5% within the range of theoretical results, and the speedup for SMP_16 is on average within 1.9% of theoretical results.

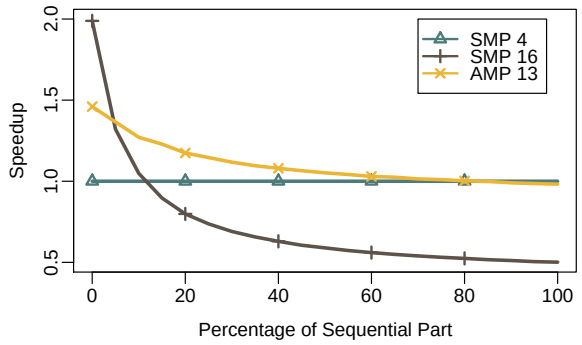

Fig. 4. Speedup. ($iterations = 100$, $phase\_change = 5$)

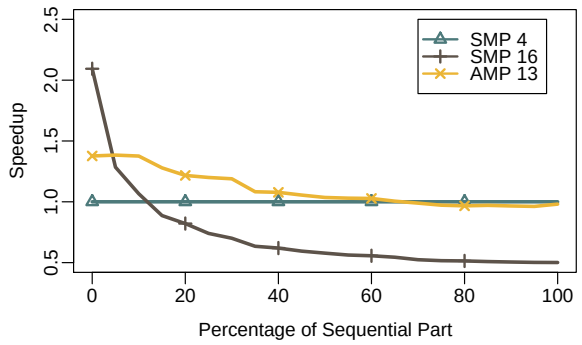

Fig. 5. Speedup. ($iterations = 10$, $phase\_change = 5$)

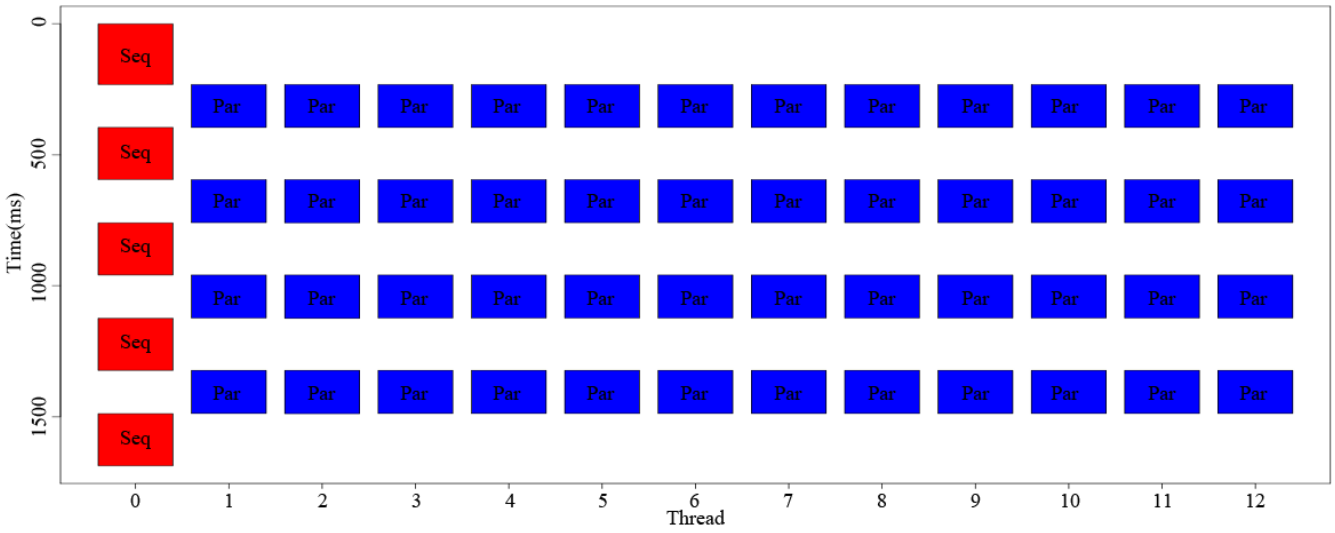

Fig. 3. Scheduling on AMP. (*iterations* = 10, *phase_change* = 8)

To investigate the performance under very frequent phase changes, we increased the number of phase changes to 15 and kept number of iterations equal to ten. In this experiment, each parallel task takes roughly 3 milliseconds when the width of the graph is 12, and each sequential task takes roughly 30 milliseconds. Therefore, the average frequency of phase changes is about 16 milliseconds. Figure 6 shows that the speedup for this set of workloads is by no means similar to the theoretical results. SMP_4 outperforms both SMP_16 and AMP_13 for all workloads.

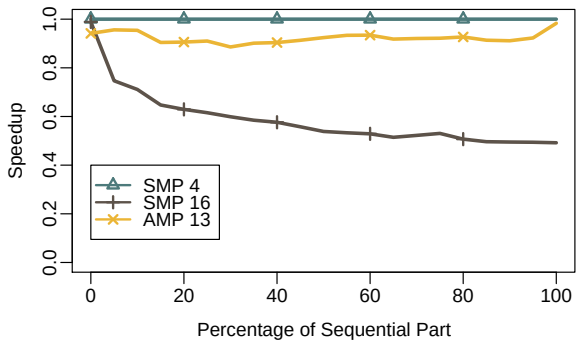

Fig. 6. Speedup. (*iterations* = 10, *phase_change* = 15)

To further investigate the effect of phase changes, we measured the slowdown for each configuration when phase change increased from five to fifteen while keeping the number of iterations equal to ten (Figure 7). SMP_16 and AMP_13 suffered more performance degradation than SMP_4 and the slowdown appeared to decrease as sequential percentage increased. This indicates that scheduling overhead was the reason behind poor performance. When switching between parallel and sequential phases, there is scheduling overhead associated with updating the scheduler's internal queues, handling interprocessor interrupts as well as migrating the thread's state architectural state to the fast core. Since the synthetic workloads on SMP_16 and AMP_13 have a greater parallel width than SMP_4, the overhead of task assignment was larger and this caused a greater slowdown. As the sequential code fraction increases, the size of each sequential task becomes larger, and so the overhead of scheduling is relatively small. In prior work we evaluated the efficiency of the Cascade scheduler [2] and found that it was rather efficient, so we conjecture that the overhead is not due to the implementation of the scheduler, but is inherent to any system that would be required to switch threads at such a high frequency.

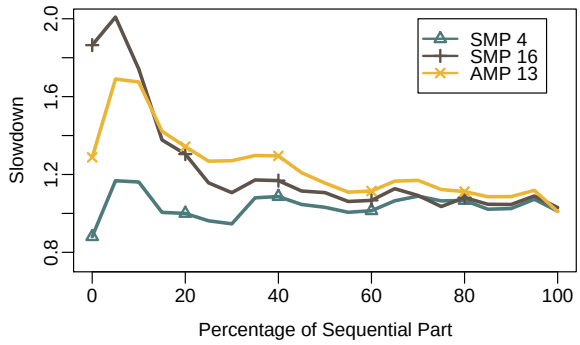

Fig. 7. Slowdown (*phase_change* = 15)

VI. Conclusions and Future Work

In this paper we have evaluated the practical potential of AMP processors by analyzing how the performance benefits delivered by these systems are determined by the properties of the workload. We create synthetic workloads to simulate real applications and use DVFS technique to model AMP processors on conventional multicore processors. Our results demonstrate that AMP systems can deliver their theoretically predicted performance potential unless the changes between parallel and sequential phases are extremely frequent.

As part of future work we would like to further investigate the overhead behind thread migrations, perhaps deriving an analytical model for this overhead based on the architectural parameters of the system and the properties of the workload. The effects of migration on cache performance in the context of AMP systems must also be investigated further.

Our synthetic workloads aim at simulating parallel behavior of applications with a fine granularity. But assumptions about the synthetic workloads, i.e., computing-bound with consistent pattern, may not be a good reflection of real applications. More diversified workloads with various parallel width and percentage should be tested more systematically. To improve the reliability of our synthetic workload generator, further investigation on the behavior of real applications will also be needed.

Scheduling is another future area for investigation. Since we didn't fully utilize fast cores, migrating parallel tasks to fast cores when they are idle may achieve significantly better performance in parallel phases. To further optimize the performance of parallel phase, more sophisticated scheduling algorithms [11] may be introduced. While several schedulers for AMP systems have proposed in prior work [5], [7], [8], they have primarily addressed the ability of these systems to address instruction-level parallelism in the workload. Only one work addressed the design of an asymmetry-aware operating system scheduler that caters to the changes in parallel/sequential phases of the applications [9]. It would be interesting to validate our results with that scheduler, and to evaluate the difference in the overhead resulting from the user-level and kernel-level implementations.

References

[1] M. Annavaram, E. Grochowski, J. Shen. Mitigating Amdahl's Law Through EPI Throttling, ISCA 2005

[2] M. J Best, A. Fedorova, R. Dickie et al. Searching for Concurrent Patterns in Video Games: Practical Lessons in Achieving Parallelism in a Video Game Engine, submitted to EuroSys 2009

[3] M. Hill and M. Marty. Amdahl's Law in the Multicore Era. IEEE Computer, July 2008

[4] R. Kumar et al. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. MICRO, 2003

[5] R. Kumar et al. Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance. ISCA, 2004

[6] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. An Asymmetric Multi-core Architecture for Accelerating Critical Sections, in ASPLOS, 2009

[7] Daniel Shelepov, Juan Carlos Saez, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, Viren Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems, in Operating Systems Review, vol. 43, issue 2, (Special Issue on the Interaction among the OS, Compilers, and Multicore Processors), pp. 66-75, April 2009

[8] M. Becchi and P. Crowley. Dynamic Thread Assignment on-Heterogeneous Multiprocessor Architectures. In Proceedings of the 3rd Conference on Computing Frontiers, 2006

[9] Juan Carlos Saez, Alexandra Fedorova, Manuel Prieto, Hugo Vegas. Unleashing the Potential of Asymmetric Multicore Processors Through Operating System Support, submitted to PACT 2009

[10] Engin Ipek, Meyrem Krman, Nevin Krman, and Jose F. Martinez. Core Fusion: Accommodating Software Diversity in Chip Multiprocessors, in 34th annual international symposium on Computer architecture

[11] Jian Li and Jose F. Martinez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors, in High-Performance Computer Architecture, 2006