Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architectures

Jonathan A. Winter
Google Inc.
Mountain View, CA
jawinter@google.com

David H. Albonesi
Computer Systems Laboratory
Cornell University, Ithaca, NY
albonesi@csl.cornell.edu

Christine A. Shoemaker
CEE, Applied Math, & ORIE
Cornell University, Ithaca, NY
cas12@cornell.edu

ABSTRACT

Future many-core microprocessors are likely to be heterogeneous, by design or due to variability and defects. The latter type of heterogeneity is especially challenging due to its unpredictability. To minimize the performance and power impact of these hardware imperfections, the runtime thread scheduler and global power manager must be nimble enough to handle such random heterogeneity. With hundreds of cores expected on a single die in the future, these algorithms must provide high power-performance efficiency, yet remain scalable with low runtime overhead.

This paper presents a range of scheduling and power management algorithms and performs a detailed evaluation of their effectiveness and scalability on heterogeneous many-core architectures with up to 256 cores. We also conduct a limit study on the potential benefits of coordinating scheduling and power management and demonstrate that coordination yields little benefit. We highlight the scalability limitations of previously proposed thread scheduling algorithms that were designed for small-scale chip multiprocessors and propose a Hierarchical Hungarian Scheduling Algorithm that dramatically reduces the scheduling overhead without loss of accuracy. Finally, we show that the high computational requirements of prior global power management algorithms based on linear programming make them infeasible for many-core chips, and that an algorithm that we call Steepest Drop achieves orders of magnitude lower execution time without sacrificing power-performance efficiency.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.4 [**Performance of Systems**] – *design studies, fault tolerance, modeling techniques*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems – *sequencing and scheduling*.

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability, Experimentation, Theory.

Keywords

Many-Core Architectures, Heterogeneous Chip Multiprocessors, Thread Scheduling, Global Power Management, Process Variations, Hard Errors, Scalability, Computational Complexity.

1. INTRODUCTION

As the semiconductor industry continues to deliver exponentially increasing transistor density over time, tens and eventually hundreds of cores are expected on a single chip [14]. These "many-core" processors will likely be heterogeneous, either by design or due to variability and intrinsic and extrinsic defects [4]. The latter form of heterogeneity is particularly challenging, as it means that chips with homogeneously designed cores may not only differ in their core-to-core characteristics (frequency, leakage, and hardware functionality) out of the manufacturing line, but aging defects may cause them to further change over the lifetime of the product. The ability to adjust per-core frequencies to account for variability and to deconfigure portions of the core pipeline in the face of defects [1,6,24,27,28,30] will allow these chips to remain operational. However, the challenge lies in keeping these randomly heterogeneous processors efficient in terms of maintaining acceptable levels of performance in the eyes of the user and staying within the power budget.

Figure 1 illustrates an example of an eight core processor where different pipeline components have suffered from faults and have been deconfigured (represented by '**X**'s). In addition, due to variations, each core may have different frequency and leakage characteristics. Runtime schedulers that ignore this heterogeneity may incur large performance losses [34], and similarly oblivious global power managers may yield unacceptably high power dissipation [12]. Thus, while these chips may remain operational, they may no longer function at a level that is acceptable to the user.

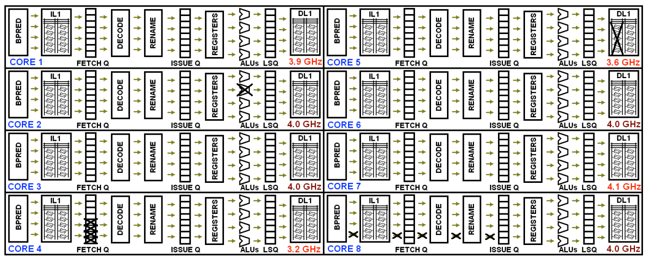

Figure 1: An illustrative example of an eight-core randomly heterogeneous microprocessor.

Recent work has begun to address the challenge of mitigating power-performance efficiency losses in the face of hard errors and variability in small-scale chip multiprocessors (CMPs). For example, Bower et al. [5] and our previous work [34] motivate the need for intelligent thread scheduling policies for randomly heterogeneous processors. Both efforts demonstrate that scheduling algorithms that are oblivious to heterogeneity may yield unacceptably high performance losses. The latter effort also proposes new scheduling algorithms that largely mitigate the power-performance impact of hard errors and variability in small-scale CMPs. Teodorescu and Torrellas [32] is the only work to our knowledge to consider both scheduling and power management in CMPs suffering from process variations. They conduct a design exploration, propose a number of schedulers to satisfy different objectives, and develop a linear programming solution for power management. Herbert and Marculescu [12] also study global power management (but not scheduling) for variability-affected CMPs and develop dynamic voltage and frequency scaling (DVFS) algorithms that are aware of both the frequency and the leakage heterogeneity.

While prior research has developed scheduling and power management algorithms that provide good power-performance efficiency for small-scale multi-core processors, the proposed techniques may not be effective for many-core processors with tens to hundreds of cores. Furthermore, many previously proposed runtime algorithms employ brute-force approaches [15], require sampling numerous configurations [34], or execute computationally intensive algorithms such as linear programming [32]. For multi-core processors with only a few cores, the sampling requirements and decision overhead for running these scheduling and power management algorithms may be reasonable, especially if they are only employed at a coarse interval granularity. However, the move to many-core architectures brings the scalability of these prior algorithms into question, and calls for a detailed investigation of their sampling and computational requirements.

This paper provides a detailed analysis of both thread scheduling and global power management for future many-core architectures. In particular, this paper makes the following key contributions:

- The performance, power, sampling requirements, and runtime overhead of a wide range of scheduling and power management algorithms are studied on heterogeneous many-core architectures with up to 256 processing cores.
- The computational complexities of thread scheduling and global power management techniques are formally analyzed.
- An experimental assessment is conducted to determine if coordination is needed between many-core scheduling and power management algorithms.
- Highly scalable scheduling and power management algorithms that achieve close to optimal performance are developed for future many-core processors.

The rest of this paper proceeds as follows. The next section discusses the problem of runtime management for randomly heterogeneous many-core architectures. Section 3 discusses a range of scheduling and power management algorithms, and proposes more scalable approaches suitable for many-core systems. Section 4 describes the evaluation methodology and Section 5 presents the experimental results. Finally, related work is described in Section 6 and conclusions presented in Section 7.

2. RUNTIME MANAGEMENT OF RANDOMLY HETEROGENEOUS MANY-CORE ARCHITECTURES

As the number of cores grows to hundreds on a single die, ensuring power-performance efficiency becomes a complex optimization problem for the runtime manager. This challenge is further exacerbated by the random heterogeneity created by manufacturing faults, wear-out, and process variations. Two runtime managers are chiefly responsible for controlling the operation of the applications running on a many-core processor: the thread scheduler and the global power manager (GPM). Both the scheduler and power manager operate over a quantum of time which consists of two phases, a short sampling period and a longer steady-state period. During the sampling period, the performance and power statistics of the applications and heterogeneous cores are assessed by running different scheduling assignments (for the scheduler) or power settings (for the power manager) over smaller intervals of time. The manager then employs an algorithm to use these interval statistics to make a decision – a scheduling assignment or DVFS settings – at the end of the sampling period. This decision is maintained for the steady-state period until the next quantum. Figure 2 describes this process, assuming a 100ms quantum for thread scheduling and a 10ms quantum for power management, in line with prior work [12,15,32,34].

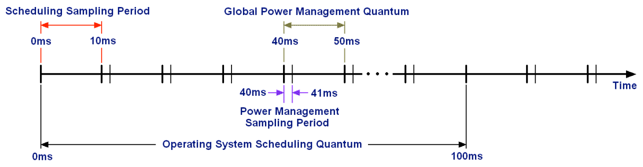
Scheduling Sampling Period
0ms
10ms
Global Power Management Quantum
40ms
50ms
Time
40ms
41ms
Power Management Sampling Period
Operating System Scheduling Quantum
0ms
100ms

Figure 2: The operation of the thread scheduler and GPM.

2.1 Scalability Issues

For the application scheduler and GPM to operate effectively, the performance and power statistics taken during the sampling period must be reflective of the true application behavior on the processor cores. This requires the manager to have sufficient time to take enough samples, each of reasonable length, to prevent thread migration effects, thermal time constants, and other effects of moving applications and changing power settings from dominating the statistics. Furthermore, the runtime of the algorithm used to make the decision must be short relative to the quantum. Otherwise, the steady-state period will be consumed by the algorithm's execution and little time will be left to run in the selected scheduling assignment or designated power settings. This paper investigates how these dual issues of sufficient sampling time and algorithm runtime are impacted by scaling to hundreds of cores on a chip.

Regarding algorithm execution time, a fundamental method for assessing algorithm scalability is to derive its computational complexity. We analyze the computational complexity of each scheduling and power management algorithm and then provide experimental results corroborating these findings. Traditionally, polynomial time algorithms were considered sufficiently scalable. However, when the runtime manager is tasked with making decisions tens or hundreds of times per second for architectures with hundreds of cores, we show that even O(n^3) and O(n^4) algorithms scale poorly, where *n* is the number of cores.

In order to provide intuition for the importance of algorithm complexity, Figure 3 shows a comparison of the growth in runtime of algorithms of different complexity as the number of cores on the chip is increased. In this abstraction, we assume that the unit for measuring complexity is the number of processor cycles required to compute the solution to the scheduling or power management problem. It can be seen from this graph that algorithms with factorial (O(n!)) or exponential ($O(p^n)$) complexity rapidly become extremely time-consuming to run even for a processor with sixteen cores, making them poor candidates for future many-core architectures. Likewise, an $O(n^4)$ algorithm takes over one billion cycles (250ms on a 4GHz processor) at 256 cores, making it impossible to run at millisecond granularities. Even an $O(n^3)$ algorithm requires tens of millions of cycles to execute, bringing into question its feasibility.

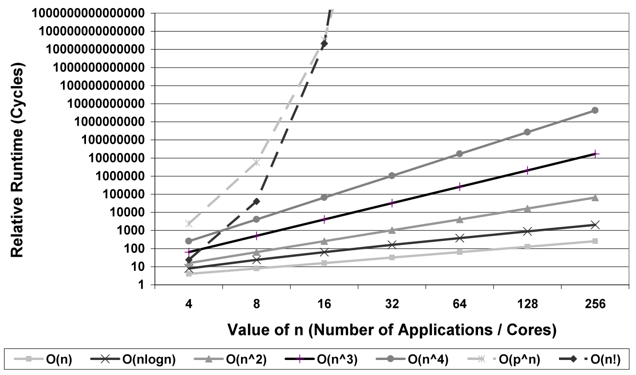

Figure 3: The growth in the runtime of various algorithm complexity classes.

2.2 Coordinating Thread Scheduling and Global Power Management

Future randomly heterogeneous many-core processors will require intelligent scheduling and power management algorithms that are aware of hardware degradations in order to mitigate their performance loss. A key question in terms of scalability is whether a lack of coordination between the two algorithms significantly degrades performance or whether they can produce good results working independently. If no coordination is necessary, then the overhead of runtime management is greatly reduced because scheduling and power management can be optimized separately, thereby avoiding the exploration of the combined search space.

We hypothesize that scheduling and power management can in fact be performed independently with little loss in efficiency and validate this claim in Section 5.1. The key intuition regarding the lack of interference comes from understanding how scheduling and power management affect application performance. In randomly heterogeneous many-core architectures, the runtime of thread *i* on core *j* can be considered a function of four components described in the following equation:

$$\text{Runtime}(i,j) = \text{IPC}(i,j) \times \text{Base\_Freq} \times \text{Variation\_Freq\_Scale\_Fac}(j) \times \text{DVFS\_Freq\_Scale\_Fac}(i,j)$$

The first component is instructions per cycle (IPC) which is a function of the application's instruction-level parallelism (ILP) and its memory access patterns, as well as the degree to which hard errors in the core affect the application's performance on that core. The base frequency of each core is the same since the processor was designed as a homogeneous architecture. The third component is a scaling factor that results from the impact of process variations on the frequency due to reduced transistor switching speeds. Together these three components dictate the inherent performance capability of an application on a core. The fourth component takes into account DVFS, which allows the core to operate at a range of frequencies below the core's maximum inherent frequency established by the architecture and impacted by variability. While changing frequency has some impact on IPC due to off-chip memory access and other asynchronous activity, for the most part, DVFS affects application runtime by altering core frequency rather than influencing IPC.

If the DVFS levels for all the cores on the chip were held constant, the application scheduler would optimize for the inherent performance capability of the applications on the cores. The resulting performance values would be modulated by the global power manager seeking to meet a power target by adjusting voltages and frequencies without impacting the benefit of the scheduling assignment. Thus, scheduling and power management tackle different elements of the application/core performance equation. In order to fairly assess different scheduling options, our application schedulers always sample applications at the same voltage levels to make DVFS independent decisions. We set all cores to the middle DVFS level to avoid exceeding the power budget during the scheduler's sampling period.

3. THREAD SCHEDULING AND GLOBAL POWER MANAGEMENT ALGORITHMS

3.1 Overview

The tasks of determining the best assignment of applications to cores and determining the optimal voltage/frequency settings in a many-core processor are essentially large-scale optimization problems. These optimization challenges can be approached from a number of perspectives. In this paper, we make an effort to be comprehensive and present a variety of algorithms for both problems that cover the basic styles of optimization. First, we discuss brute force approaches that find the optimal solution but have major scalability limitations. Next, we examine greedy approaches designed to be simple and fast to provide high levels of scalability. We then develop heuristic techniques based on well-known methods in combinatorial optimization. We also study variants of linear programming, a classical and effective approach for solving a wide range of optimization problems. Finally, we consider hierarchical algorithms designed to cut down the complexity of managing many-core processors, and hence reduce the number of samples and the execution time. In addition to evaluating the effectiveness of the solutions computed by the algorithms, we also assess their sampling requirements and computational complexity. Both of these components must scale efficiently to ensure the algorithms' feasibility in future many-core architectures.

3.2 Thread Scheduling Algorithms

Heterogeneous many-core processors present a distinctly complex scheduling problem. Asymmetry resulting from variations, manufacturing defects, and wear-out is particularly challenging as it cannot be anticipated at runtime, and it manifests itself in myriad ways. The possible number of distinctly degraded cores increases exponentially with the number of failure modes. Consequently, scheduling algorithms must be robust and broadly applicable. A further complication for scheduling is that there is no simple effective *a priori* way to model the power-performance tradeoffs of running a given application on a particular core. Unlike power management where it can be assumed that performance is linearly related to voltage and power is cubically related (see Section 3.3), there is no clear-cut method for estimating the interaction of core heterogeneity and application behavior. Consequently, our approach is to conduct online sampling of the applications on the degraded cores [32,34].

In the following paragraphs, we describe the scheduling algorithms studied in this paper, including the rationale for each approach, the nature of the sampling required, and an analysis of their computational complexity. A summary of the thread scheduling algorithms can be found in Table 1.

Table 1: A summary of the thread scheduling algorithms.

Scheduling Algorithm	Computational Complexity	# of Sampling Intervals
Brute Force	O(n·n!)	n
Greedy Algorithm (VarF&AppIPC)	O(n·logn)	n
Local Search (n/2 swaps)	O(n^2)	n
Hungarian Algorithm (Linear Programming)	O(n^3)	n
Sequential Hierarchical Hungarian Algorithm	O(n)	32
Parallel Hierarchical Hungarian Algorithm	O(1)	32

Brute Force: The simplest method for determining the best assignment of threads to cores is to try every possibility and pick the best one. However, this technique suffers from two critical drawbacks when cores can differ due to random heterogeneity. On a chip with *n* cores running *n* applications, there are n! ways of assigning applications to cores. This necessitates taking an infeasible number of samples as the number of cores increases even beyond four cores. If the scheduler assumes that the interactions between threads running on different cores are minimal and ignores them, the algorithm can reduce the sampling to trying every benchmark on every core once, for n^2 samples. Since all applications can be sampled on one of the cores during each sampling interval, collecting these n^2 samples requires *n* sampling periods. This approach is analogous to the sample-one dynamic scheduling heuristic from Kumar et al. [17], but for random rather than designed heterogeneity. While this heuristic greatly reduces the number of samples, the scheduler must still compute the sum of performances of each application/core pair (O(n) computation) for each of the n! assignments, leading to an infeasible O(n·n!) runtime algorithm. Due to the impractical runtime of this algorithm, it is not considered further.

Greedy Algorithm: On the other end of the spectrum from brute force are greedy approaches. Greedy algorithms are popular due to their simple implementation and low runtime complexity. However, they are most effective when solving problems with straightforward solution spaces, such as convex optimization, since greedy solvers typically find local optima. For this study, we adapt the VarF&AppIPC scheduling algorithm from Teodorescu and Torrellas [32], which has been shown to be very effective when combined with global power management on multi-core processors that suffer from process variations (but without manufacturing defects and wear-out faults). This algorithm ranks the applications by average IPC and ranks the cores by inherent frequency (before applying power management) and matches applications and cores by rank in an effort to assign high ILP threads to high frequency cores and memory-bound threads to low frequency cores [32]. Since our cores are heterogeneous, we developed a modified version of VarF&AppIPC. Our approach samples the IPC of each thread on every core and averages the results to obtain an IPC value that can be fairly compared between benchmarks. This requires n^2 samples as in the sample-one technique. The complexity of the Greedy Algorithm is O(n·logn) because the rate determining step sorts the applications by IPC to determine their rank. (Sorting the cores by frequency can be done offline, since the impact of process variations on frequency can be determined at manufacturing time and the degradation due to wear-out happens over months of use.) Consequently, the Greedy Algorithm executes far faster than Brute Force.

Local Search: For our combinatorial optimization algorithm, we implement the Local Search Algorithm from our previous work [34]. However, in our present work, we optimize for maximum overall throughput rather than energy-delay-squared (ED^2). Local Search is an archetype for iterative optimization approaches and the basis for many more advanced approaches. The algorithm starts with a random assignment of applications to cores and proceeds by selecting another schedule in the neighborhood of the current one and accepting this new solution if it is better than the previous one. In this Local Search Algorithm, the neighborhood is defined as a scheduling assignment that can be derived from the current one by pair-wise swapping of the applications on the cores [34]. In our implementation, we swap all threads such that for an *n* core processor, there are *n*/2 pairs of threads. While Local Search algorithms are greedy by nature, the improvement introduced by [34] whereby a solution can be partially accepted offsets much of this limitation. Partially accepting a solution involves retaining any pair-wise swaps that locally improve the performance of the two benchmarks involved and rejecting those swaps that do not, rather than accepting a solution only in full. During each iteration of the algorithm, Local Search selects an assignment among the neighbors of the current best solution and then samples the applications on their assigned cores to determine the performance of this schedule. In our implementation, we run *n* iterations with *n* cores, which means that O(n^2) samples are again required. Furthermore, each iteration does O(n) amount of computation, leading to an overall complexity of O(n^2) for the algorithm.

Hungarian Algorithm: Linear programming is a highly general solution method for solving any kind of optimization problem. The key requirement is finding a scheme for converting the constraints and optimization objective of a problem into linear equations or inequalities. While generalized linear programming solvers can be the most effective approach for finding a good solution, certain linear programming (LP) problems can be solved more efficiently by exploiting the special structure of the given problem.

By simplifying the scheduling problem and assuming that the interactions between two sequential applications running on different cores is negligible, thread scheduling can be modeled as the classic Assignment Problem from operations research [34]. While general linear programming tools can be used to solve the Assignment Problem, the special structure of this problem where solutions are a one-to-one mapping of applications to cores, allows for the application of the Hungarian Algorithm [8]. One clear advantage of the Hungarian Algorithm over the Simplex Method (the most widely used general LP method) is that the Hungarian Algorithm has a bounded worst-case runtime of $O(n^3)$ for a processor with *n* cores [8]. On the other hand, the Simplex Method has an exponential runtime in the worst case and a polynomial-time average case complexity that is highly dependent on the nature of the objective function and constraints [11]. The Hungarian Algorithm must sample each application on each core, requiring *n* sampling intervals, to create a matrix of the benefit of assigning each application to each core before running the actual algorithm.

An advantage of the Hungarian Algorithm over the above algorithms is that it finds the optimal solution to the simplified scheduling problem. Thus, provided that the assumption of negligible interference between applications holds and that the execution samples accurately reflect the applications' behaviors, the Hungarian Algorithm can be the most effective thread scheduler.

Hierarchical Hungarian Algorithm: The above four scheduling algorithms suffer from two main drawbacks. First, each algorithm requires *n* sampling intervals to provide the necessary performance evaluation of the different thread-core matchings. In future many-core processors, this will require hundreds of sampling intervals. These sampling intervals must be of reasonable length in order to ensure that they are reflective of the actual application behavior. In our experimental work, we found that sample lengths must be on the order of millions of cycles to amortize the impact of context switching and cache and branch predictor warm-up. However, running hundreds of million-cycle samples is impractical because most of the time between scheduling intervals would be consumed just with sampling. The second drawback is the time complexity of most of the above algorithms. Clearly brute force, with exponential runtime, is infeasible. Nonetheless, as our experimental results will demonstrate, even an $O(n^2)$ or $O(n^3)$ algorithm becomes too time consuming to perform at a desirable scheduling interval granularity. For these reasons, an alternative scheduling algorithm is required for future many-core chips.

As we will show, the Hungarian Algorithm is the highest performing of the above techniques (Section 5.2), and thus we chose to develop a Hierarchical Hungarian Algorithm that requires significantly fewer samples and a far shorter runtime than the previously proposed methods. This algorithm divides the cores of the CMP into groups of 32 cores (experimentally determined to be the best group size) and obtains a locally effective scheduling assignment within each group. Rather than sampling all threads on all cores, applications are only sampled on those cores in their group and thus only 32 samples are required for each application. Likewise, since the size of the groups is fixed, the Hungarian Algorithm needs to solve constant-sized problems regardless of core scaling. With *n* cores, the Hungarian Algorithm must be run separately on *n*/32 groups, meaning that the computational problem grows linearly (O(n)). A further improvement can be made by noting that the sampling and computation for each group is completely independent and thus can be conducted in parallel by employing one core in each group to execute the Hungarian Algorithm. This parallelized version runs in constant time, since it is only a function of the group size, making it extremely scalable.

3.3 Global Power Management Algorithms

In addition to developing scalable thread scheduling algorithms, we investigate global power management (GPM) for many-core architectures suffering from manufacturing defects, lifetime wear-out, and process variations. As per prior work [12,15,19,25,32], the objective is to maximize throughput under a chip-wide power budget. In this study, we focus on dynamic voltage and frequency scaling (DVFS), the most widely implemented GPM approach. Like [12,15,19,25,32], we assume that each core has independent frequency and voltage control. Future architectures may employ DVFS at coarser granularities, grouping multiple cores into single voltage domains. With coarser-grain domains, the scheduler, in addition to matching threads to cores, might also consider the affinity of threads for domains for which voltage would be similarly scaled. Likewise, the global power manager would need to consider balancing the needs of one thread versus another while scaling voltage and frequency domains. To allow a more direct comparison between our study and prior work, we focus on per-core DVFS and leave a study of the voltage scaling granularity to future work.

We consider a DVFS mechanism that scales frequency linearly with voltage (as per prior work) and has seven discrete voltage levels spaced out evenly between 0.7V and 1.0V (the nominal voltage). The corresponding frequency range is dependent on the impact of process variations on a given core (as described below) but would vary from 2.8 GHz to 4.0 GHz (the nominal frequency) on a core unaffected by variations.

$\text{power} \propto \text{frequency} \times \text{voltage}^2$	(1)
$\text{frequency} \propto \text{voltage}$	(2)
$(1)\ \&\ (2) \rightarrow \text{power} \propto \text{voltage}^3$	(3)
$\text{throughput} \propto \text{frequency}$	(4)
$(2)\ \&\ (4) \rightarrow \text{throughput} \propto \text{voltage}$	(5)

Figure 4: GPM power-performance modeling assumptions.

One distinct difference between GPM and scheduling is that the impact of changing the voltage and frequency of a core on application performance and power dissipation can be estimated effectively by knowing the power-performance characteristics of the application on the core at the current DVFS level. We employ the model of Isci et al. [15] described in Figure 4, and assume that within the narrow range of DVFS levels, performance is linearly proportional to voltage and power is a cubic function of voltage. Consequently, GPM algorithms using this model need only one sample per application/core pair.

As with the scheduling algorithms, we examine five approaches to power management: a brute force method, a greedy algorithm, a heuristic combinatorial optimization scheme, linear programming, and a hierarchical approach. In the following paragraphs, we discuss these algorithms, their sampling requirements, and their computational complexity. A summary of the algorithms is given in Table 2.

Brute Force: Isci et al. propose the MaxBIPS algorithm [15], which uses the model discussed above to calculate the performance and power dissipation achieved for each combination of power settings available on the chip. Assuming that DVFS can be set to *p* discrete levels (for Isci et al., *p* = 3 and for our work *p* = 7), there are p^n possible power settings for a CMP with *n* cores. For each power setting option, the calculated performance and power of each core must be summed to determine the chip throughput and power, requiring O(n) time. While MaxBIPS is very effective at calculating a good DVFS assignment with a single sample per application/core pair taken at the middle voltage level, clearly even for *p* = 3, the $O(n \cdot p^n)$ computation cost is prohibitive for many-core processors. Consequently, we do not consider MaxBIPS further in this paper.

Table 2: A summary of the GPM algorithms.

Global Power Management Algorithm	**Computational Complexity**	**# of Sampling Intervals**
Brute Force (MaxBIPS)	$O(n \cdot p^n)$	1
Greedy Algorithm	$O(n \cdot \log n)$	1
Steepest Drop	$O(p \cdot n \cdot \log n)$	1
LinOpt (Linear Programming)	$O(n^4)$ (average case)	3

Greedy Algorithm: We develop a simple greedy approach to power management which leverages a key intuition about global power management for maximum throughput. Essentially, performance is maximized by shifting power to applications that can individually generate the highest throughput. To achieve this, the Greedy Algorithm gives as much power as possible to application/core combinations with the greatest inherent performance capability. As with MaxBIPS, a single sample is taken for each scheduler-assigned application/core pair at the same voltage setting (the middle level) and the throughput is calculated. The throughput is a function of both the application IPC on the assigned degraded core and the core's operating frequency due to variations.

Using the samples and the voltage/performance/power model, the Greedy Algorithm estimates the power consumed by each core while running at the lowest DVFS setting. By subtracting the minimum power consumed by each core from the total power budget, the algorithm then determines how much extra power is available to assign to high throughput application/core combinations. The pairs are then ranked by throughput, and starting with the highest ranked pair, cores are greedily set to the highest voltage/frequency setting proceeding down the ranking until, according to the voltage/performance/power model, the power budget is reached. If there is some leftover power that was insufficient to allow the final core to be set to the highest setting, that core is set to the highest setting that still meets the budget. The rest of the lower ranked cores are then left at the lowest DVFS setting. Since the most complex step of the Greedy Algorithm is ranking the application/core pairs by throughput, the algorithm's complexity is $O(n \cdot \log n)$.

Steepest Drop: We call our heuristic optimization algorithm Steepest Drop, which is a directed Local Search method. Rather than randomly select a configuration in the neighborhood of the current best known configuration as in Local Search scheduling, Steepest Drop exploits the known correlation between voltage, performance, and power to direct the search. We modify the algorithm from Meng et al. [19] that was designed to address the large search space resulting from applying multiple power optimizations simultaneously. In our work, we only use DVFS, but because of the large scale of our many-core architecture, the optimization problem is sufficiently challenging with DVFS alone. Again, only one sample at the middle DVFS level is needed to calibrate the voltage/performance/power model.

The algorithm starts by assuming each core is set to the highest power setting. Then using the analytical model, if the power is estimated to be over the chip-wide budget, the algorithm selects the application/core pair that would provide the biggest ratio of power reduction to performance loss if the voltage was dropped one step. This new configuration's power dissipation is estimated and, if the power is still over budget, the steepest drop is again calculated from the new configuration. This process is repeated until the power budget is met. To optimize the runtime of Steepest Drop, our version uses a max-heap data structure for storing the ranking of the power reduction to performance loss ratio for each application/core pair. In the worst case, the Steepest Drop algorithm would have dropped the voltage/frequency settings from the highest values all the way to the lowest for each core. This would involve *n* x *p* iterations for *n* cores and *p* power levels. By using the efficient heap data structure, our approach only takes O(logn) time to access the steepest drop and update the data structure during each iteration, for a total complexity of $O(p \cdot n \cdot \log n)$.

LinOpt: Teodorescu and Torrellas [32] propose using linear optimization to solve the global power management problem in a multi-core chip afflicted with process variations. Their algorithm, LinOpt, involves three steps. First, the power management task is formulated as a linear programming problem. Then, the formulation is run through a linear programming solver that implements the Simplex Method. Linear programming requires continuous-valued variables, and thus the linear solver can return voltage settings that lie between the discrete DVFS levels. Thus, the third step conservatively drops any voltage values to the next lowest DVFS setting. As in all the other algorithms, performance is modeled as linearly dependent on voltage. However, the cubic relationship between voltage and power cannot be captured in a linear program. Instead, a linear approximation is found that minimizes the error with the true relationship as determined by three samples taken at the lowest, middle, and highest DVFS setting [32]. We implement this linear approximation using linear least squares fitting (LLSF), which can be computed in O(1) time. Since this must be done for each application/core combination, the total time is O(n).

As mentioned above, the Simplex Method has exponential worst case complexity and polynomial time average case complexity and thus dominates LinOpt's runtime. Experimental and stochastic analysis [11] have concluded that average case runtime estimates for linear programming are $O(n^4)$ when considering problems where the number of constraints is of the same order as the number of variables, such as in our case. In our results, we evaluate whether this high-order polynomial runtime becomes a problem for many-core processors.

Hierarchical GPM Algorithms: A logical direction to pursue would be to design hierarchical algorithms for global power management to increase scalability in an analogous manner to the Hierarchical Hungarian Algorithm. For instance, a Hierarchical LinOpt Algorithm would divide the chip into groups that are given a fraction of the chip-wide power budget and then solve each resulting sub-problem by applying linear programming.

However, our results in Section 5.3 show that Steepest Drop is very effective at finding a power-performance efficient DVFS setting and is also highly scalable from a computational complexity perspective, obviating the need for pursuing a hierarchical approach.

4. EVALUATION METHODOLOGY

4.1 Simulation Infrastructure

To model many-core multiprocessors, we take an approach similar to [9,20] of building a hierarchical framework where cycle-accurate simulations of individual cores are combined by a top-level chip-wide simulator to model an entire many-core processor. Our framework is illustrated in Figure 5.

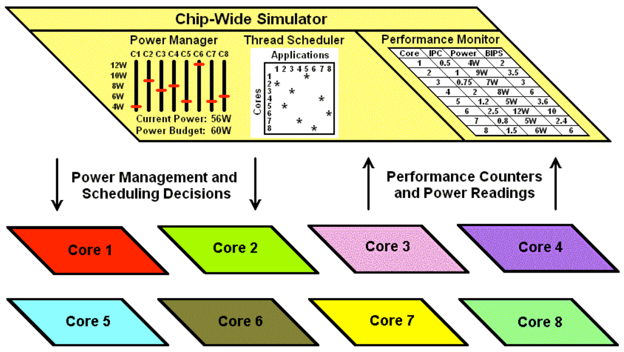

Figure 5: Hierarchical and parallel many-core simulation framework.

The lower level of the hierarchy consists of microarchitectural simulations of each core using an improved version of the SESC simulator [22]. We augmented SESC's power and thermal modeling with Cacti 4.0 [31], an improved version of Wattch [7], the block model of Hotspot 3.0 [29], and an improved version of HotLeakage [35]. Our baseline core, unaffected by process variations and defects, is a single-threaded, four-way superscalar, out-of-order processor. Table 3 lists the main architectural parameters.

The top-level chip-wide simulator performs two roles. First, it serves as a tool for managing the many-core simulations and is responsible for combining the performance, power, and thermal statistics from the SESC simulations of each core into the complete statistics for the whole processor. Second, the chip-wide simulator implements the application scheduler and global power manager. In this role, it directs the sampling and steady phases of the runtime managers, it executes the actual algorithms, and it manages the execution of the single-core simulations by migrating threads among the cores and changing DVFS settings as specified by the algorithms.

4.2 Simulating Heterogeneous Many-Core Processors

We evaluate many-core architectures with 4-256 cores, for which we consider three types of degradation that cause core-to-core heterogeneity. We model the disabling of all or part of a processor component such as an ALU or a set of queue entries as a consequence of a manufacturing defect or wear-out. We assume a processor core can have at most two faults of this type. Cores also have variable frequencies that are randomly assigned to be in the range of 60% to 110% of the nominal frequency of 4GHz, similar to prior work [13,32,34]. Finally, sections of the core can suffer from increased leakage causing higher than normal static power. Table 4 presents a list of the core degradations possible for each type of degradation. A heterogeneous n-core processor is generated by randomly picking *n* cores each affected by some or all of these three types of degradations.

Table 3: Core microarchitectural parameters.

Front-End Parameters	
Branch Predictor	hybrid of gshare and bimodal with 4K entries in each
Branch Target Buffer	512 entries, 4-way associative.
Return Address Stack	64 entries, fully associative.
Front-End Width	4-way
Fetch Queue Size	32 entries
Re-Order Buffer	128 entries
Retire Width	4-way
Back-End Parameters	
Integer Issue Queue	48 entries, 4-way issue
Integer Register File	80 registers
Integer Execution Units	4 ALUs/address calculation units and 1 mult/div unit
FP Issue Queue	24 entries, 1-way issue
FP Register File	80 registers
FP Execution Units	1 adder and 1 mult/div unit
Memory Hierarchy	
L1 Instruction Cache	8KB, 2-way associative, 1 port, 1 cycle latency
Instruction TLB	32 entry, fully associative., 1 port
Load Queue	48 entries, 4 ports
Store Queue	24 entries, 4 ports
L1 Data Cache	8KB, 2-way associative, 2 ports, 1 cycle latency
Data TLB	32 entry, fully associative., 2 ports
L2 Cache	1MB, 8-way associative, 1 port, 10 cycle latency
Main Memory	1 port, 200 cycle latency

4.3 Workloads

We randomly generate workloads from among the 17 SPEC CPU 2000 benchmarks for each many-core processor configuration. We use three fast-forward points (one, two, and three billion instructions) for each benchmark to add further diversity to the workloads. In the main scalability study of Sections 5.2 and 5.3, each algorithm is run on four different randomly generated workloads and four different randomly degraded many-core configurations, resulting in 16 different test cases for each size many-core processor.

4.4 Assessing Algorithm Runtimes

In addition to the computational complexity results presented in Sections 3.2 and 3.3, we empirically assess the execution requirements of our algorithms. Each of the algorithms is implemented in C and compiled with full optimizations into a special MIPS binary that can be executed with the SESC simulator. These binaries are then run on SESC while modeling the microarchitecture of the cores in our many-core processor in order to accurately characterize the runtime of the algorithms.

Table 4: Possible forms of core degradation due to hard faults and variations.

Type of Degradation	List of Options
Degraded Component	none
	memory latency is doubled
	half the L2 cache
	half the L1 instruction cache
	a way of the L1 instruction cache is broken
	front-end bandwidth is reduced from 4-way to 3-way fetch/decode/rename
	half the integer issue queue
	integer issue bandwidth is reduced by one
	one or more integer ALUs are disabled
	half the rename registers are broken
	half the load queue
	half the store queue
	half the L1 data cache
	a way of the L1 data cache is broken
	half the re-order buffer
Frequency Degradation	60 – 110 % of the nominal, set at intervals of 2.5%
Increased Leakage	none
	2X nominal in L1 caches and TLBs
	2X nominal in front-end and ROB
	2X nominal in integer back-end
	2X nominal in floating point back-end
	2X nominal in load and store queues
	2X nominal across core (excluding L2)

5. RESULTS AND DISCUSSION

5.1 Coordinating Thread Scheduling and Global Power Management

Section 2.2 provides an analytical argument as to why scheduling and power management affect different components of the performance equation. In this section, we investigate the importance of coordinating thread scheduling and power management in future many-core architectures. We compare the performance and power dissipation of running independent schedulers and power managers against an oracle combined policy, which can compute the optimal scheduling assignment and power management settings without accounting for sampling or computational overheads. In every quantum, the oracle algorithm employs a brute force examination of all possible scheduling and power management combinations and then selects the application-to-core assignment and DVFS settings that provide the maximum possible performance while staying within the chip-wide power budget. For the uncoordinated algorithms, we run the Hungarian Scheduling Algorithm together with MaxBIPS, LinOpt, and Steepest Drop. If these independent approaches can achieve performance results very close to that of the oracle scheduler and power manager, then there is no point implementing a coordinated technique.

For our three uncoordinated algorithms, we always run the scheduler first and then the power manager in order to ensure that the chosen schedule does not lead to power overshoots. We verified experimentally that running power management first and then scheduling leads to less optimal power-performance efficiency and does indeed create over-budget scenarios. During the scheduler sampling period, the runtime manager sets the DVFS level of each core to the middle level, calculates the best schedule, and then employs the GPM during the scheduler's steady phase to find the best DVFS assignment.

The performance losses for the three combinations in comparison to the oracle are shown in Figure 6 for an eight core system with four different degraded configurations and four different workloads. The largest losses – at most 3% – occur when the uncoordinated algorithms slightly undershoot the power budget. These results validate our assertion that the algorithms can operate independently with near-optimal performance.

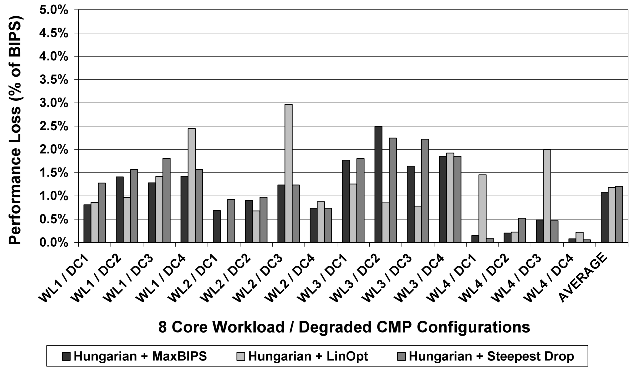

Figure 6: A comparison of uncoordinated scheduling and GPM algorithms relative to the oracle manager.

5.2 Thread Scheduling Algorithms

We now evaluate the scheduling algorithms described in Section 3.2 in terms of their performance relative to the Hungarian Scheduling Algorithm and their runtime overhead. Figure 7 shows the runtime overhead of each algorithm over a range of four to 256 core organizations expressed as a percentage of the scheduling quantum. We implement both a sequential (SQ) version of the Hierarchical Hungarian Algorithm, where a centralized scheduler makes the assignments for all the groups, and a parallel (PA) version where the scheduling task is partitioned among the groups of cores, leaving each group responsible for its own local schedule. For both hierarchical algorithms, we experimented with groups of size 8, 16, and 32 and found that groups of 32 provided the best balance of algorithm runtime and performance. For a small number of cores, the overhead for all scheduling algorithms is minor, but grows rapidly for the less-scalable Hungarian and Local Search algorithms, which have O(n^3) and O(n^2) complexity to over 16% and 9% of the scheduling quanta. On the other hand, the Greedy Algorithm and Sequential Hierarchical Hungarian Algorithm have low overheads even for 256 cores, but only the Parallel Hierarchical Hungarian Algorithm remains scalable beyond 256 cores.

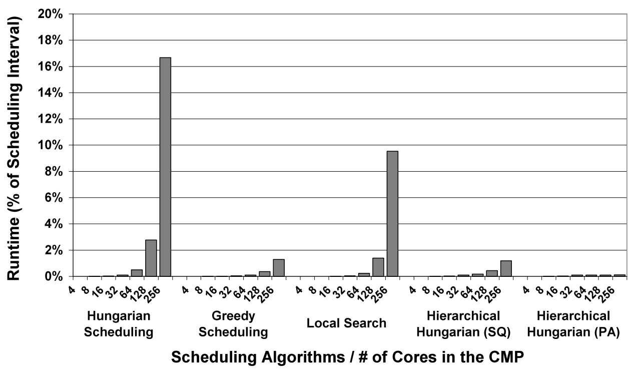

Figure 7: Scheduling algorithm runtimes as a percentage of the scheduling quantum.

While both the Sequential and Parallel Hierarchical Hungarian Algorithms are identically partitioned, the Sequential Algorithm is slowed down by the need to process sample results for the whole processor, the sequential computation of the Hungarian Algorithm for each group, and the time to construct the chip-wide scheduling assignment. Together, the factors slow down the sequential approach on a 256 core machine by an order of magnitude relative to the parallelized version. Furthermore, the Parallel Hierarchical Hungarian Algorithm runs 150X faster than the standard Hungarian Scheduler.

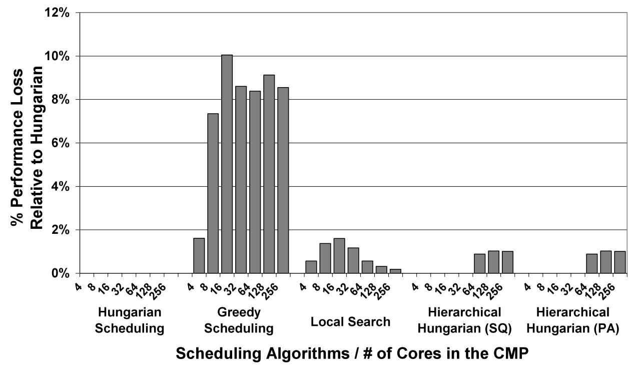

Figure 8: **Scheduling algorithm performance percentage loss relative to the Hungarian Algorithm.**

Figure 8 shows the performance loss of the different algorithms relative to the performance of the Hungarian Algorithm. The Greedy Algorithm experiences the largest loss by far, 8-10% for the larger organizations. This is not surprising considering that VarF&AppIPC's method of ranking applications purely by IPC does not fully address the complexity of scheduling for randomly heterogeneous many-core architectures. A given application may suffer significantly due to the particular degradations on one core and have very high IPC on another. The average of this thread's IPC across these cores can be misleading when trying to rank the thread as compute or memory bound. The performance of the Local Search Algorithm is quite good as the number of cores increases but it is more than offset by the higher runtime overhead associated with the extra search intervals. The Parallel Hierarchical Hungarian Algorithms offers the best combination of runtime overhead and performance of the scheduling assignment.

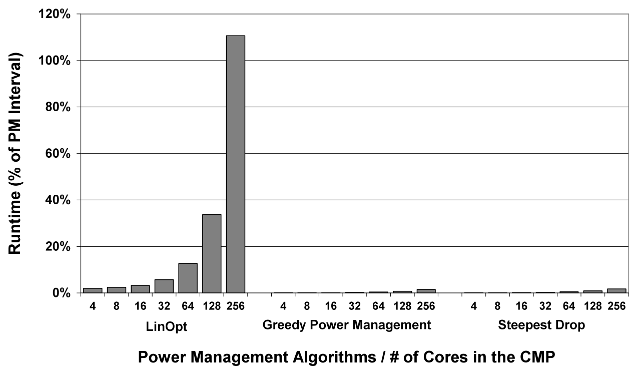

Figure 9: GPM algorithm runtimes as a percentage of the power management quantum.

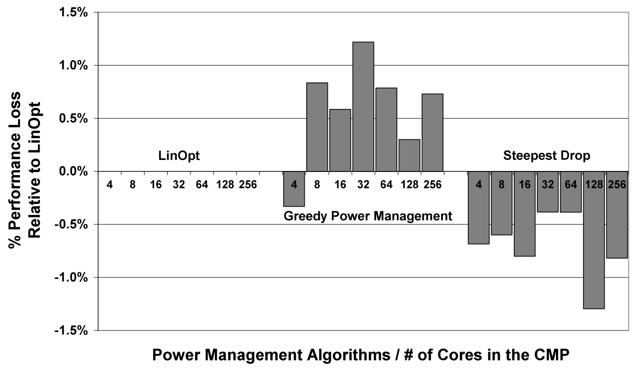

Figure 10: GPM algorithm performance percentage loss relative to the LinOpt algorithm.

5.3 Global Power Management Algorithms

The runtime overhead and performance relative to LinOpt for the power management algorithms (from Section 3.3) are shown in Figures 9 and 10, respectively. Due to its high-order polynomial average runtime, the overhead of LinOpt grows rapidly with the problem size (number of cores), and even exceeds the length of the power management quantum (10ms) for 256 cores. Due to the fact that they have almost identical complexity (O(n·logn) versus O(p·n·logn)), the Greedy Algorithm and Steepest Drop have about the same overheads, less than 2% for 256 cores, and run 75X and 62X faster (respectively) than LinOpt. Looking at the performance results shown in Figure 10, Steepest Drop outperforms the Greedy Algorithm and even slightly outperforms LinOpt in all cases. Overall, Steepest Drop has much more scalable runtime behavior than LinOpt and superior performance to the Greedy Power Manager. Given that LinOpt does not provide a performance benefit over Steepest Drop and linear programming is much more complex to implement in hardware or in the operating system, we see no reason to explore a hierarchical implementation of LinOpt.

In summary, our results demonstrate that thread scheduling based on a Hierarchical Hungarian Algorithm coupled in an uncoordinated fashion with a Steepest Drop global power manager provides the most scalable and effective solution to the challenge of maintaining high performance and power efficiency for future many-core processors that suffer from process variations and intrinsic and extrinsic defects.

6. RELATED WORK

In the introduction, we discussed prior work addressing power management and scheduling for randomly heterogeneous small-scale CMPs. In this section, we discuss remaining related research involving global power management in homogeneous multi-core processors as well as scheduling and power management in designed-heterogeneous CMPs.

Juang et al. [16] argue for coordinated formal control-theoretic methods to manage energy efficiency in multi-core systems. Isci et al. [15] introduce the problem of trying to maximize total throughput under a chip-wide power constraint by dynamically tuning DVFS to workload characteristics and develop the effective (but limited in scalability) brute force MaxBIPS algorithm. Sharkey et al. [25] extend this work by exploring algorithms based on both DVFS and fetch toggling, and by studying design tradeoffs including the granularity at which the GPM is called and local versus global management.

Sartori and Kumar [23] propose decentralized power management algorithms for homogeneous many-core architectures. They propose alternative approaches to DVFS such as setting cores to high and low power states at a coarse granularity and migrating benchmarks at a finer granularity to meet the power budget. In a similar vein, Rangan et al. [21] explore the use of scheduling on cores statically set to different voltage and frequency levels as an alternative power management approach to fine-grained DVFS. Wang et al. [33] propose a coordinated approach to global power and temperature management based on optimal control theory. They use multi-input-multi-output control strategies and model predictive control, which require matrix-matrix multiplication and either matrix inversion or factorization. Since these high-order polynomial-time matrix algorithms scale poorly, we do not consider them in our work.

Some prior work addresses designed-heterogeneous CMPs with different issue widths and pipeline complexities. Kumar et al. [17] focus on multiprogrammed performance and develop algorithms to schedule applications on cores that best match their execution requirements. However, since only two types of cores are used, the solution space is small and thus a simple sampling scheme achieves good assignments. Becchi and Crowley [3] extend that work to use performance driven heuristics for scheduling. A number of research papers look at a restricted form of heterogeneity, where cores run at different frequencies, allowing their experimental evaluation to be conducted using real hardware. Balakrishnan et al. [2] study the impact of frequency asymmetry on multi-threaded commercial workloads. Others [10,18,26] develop scheduling algorithms for chip multiprocessors with this kind of restricted heterogeneity. In comparison to these earlier research efforts, we develop methods for many-core processors with significantly more forms of heterogeneity and scaling to far more cores.

7. CONCLUSIONS

In the future, microprocessors containing hundreds of cores will need to tolerate manufacturing defects, wear-out failures, and extreme process variations. The resulting heterogeneity of these systems requires intelligent, yet highly scalable, runtime scheduling and power management algorithms. In this paper, we perform a detailed analysis of the effectiveness and scalability of a range of algorithms for many-core systems of up to 256 cores. First, we show that there is no need to coordinate scheduling and global power management, which greatly reduces the search space for runtime power-performance optimization. We develop the Parallel Hierarchical Hungarian Algorithm for thread scheduling and demonstrate that it is up to 150X faster than the Hungarian Algorithm while providing only 1% less throughput. We also demonstrate that the Steepest Drop global power management algorithm has 75X less runtime overhead (for 256 cores) than the LinOpt algorithm and similar performance. Our results show that it is essential to consider runtime overhead and scalability when designing scheduling and power management algorithms for future many-core processors.

ACKNOWLEDGMENTS

This research was supported by NSF grants CCF-0811729 and CNS-0708788 and equipment grants from Intel.

REFERENCES

[1] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA)*, June 2007, pp. 470-481.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 506-517.

[3] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings the of ACM International Conference on Computing Frontiers (CF)*, 2006, pp. 29-39.

[4] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. In *IEEE Micro*, Nov./Dec. 2005, 25(6):10-16.

[5] F.A. Bower, D. J. Sorin, and L.P. Cox. The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling. In *IEEE Micro*, May/June 2008, 28(3):17-25.

[6] F.A. Bower, P.G. Shealy, S. Orev, and D.J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN)*, June 2004, pp. 51-60.

[7] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000, pp. 83-94.

[8] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. Published by the Society of Industrial and Applied Mathematics, Philadelphia, PA, 2009, pp. 73-87.

[9] J. Chen, M. Annavaram, and M. Dubois. SlackSim: A Platform for Parallel Simulations of CMPs on CMPs. In the *Workshop on Design, Analysis, and Simulation of Chip Multiprocessors (dasCMP)*, Nov. 2008.

[10] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for Heterogeneous Processors in Server Systems. In *Proceedings*

of the ACM International Conference on Computing Frontiers (CF), May 2005, pp. 199-210.

[11] I. Griva, S.G. Nash, and A. Sofer. *Linear and Nonlinear Optimization*. Published by the Society of Industrial and Applied Mathematics, Philadelphia, PA, 2009, pp. 301-317.

[12] S. Herbert and D. Marculescu. Variation-Aware Dynamic Voltage/Frequency Scaling. In *Proceedings of the 15th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2009, pp. 301-312.

[13] E. Humenay, D. Tarjan, and K. Skadron. Impact of Process Variations on Multi-Core Performance Symmetry. In *Proceedings of Design, Automation and Test in Europe (DATE)*, April 2007, pp. 1653-1658.

[14] Intel Corporation. *From a Few Cores to Many: A Tera-scale Computing Research Overview*, Whitepaper, 2006.

[15] C. Isci, A. Buyuktosunoglu, C-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO)*, Dec. 2006, pp. 347-358.

[16] P. Juang, Q. Wu, L-S. Peh, M. Martonosi, and D. W. Clark. Coordinated, Distributed, Formal Energy Management of CMP Multiprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2005, pp. 127-130.

[17] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, June 2004, pp. 64-75.

[18] T. Li, D. Baumberger, D.A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC07)*, Nov 2007.

[19] K. Meng, R. Joseph, R.P. Dick, and L. Shang. Multi-Optimization Power Management for Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct 2008, pp. 177-186.

[20] M. Monchiero, J.-H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to Simulate 1000 Cores. In the *Workshop on Design, Analysis, and Simulation of Chip Multiprocessors (dasCMP)*, Nov. 2008.

[21] K.K. Rangan, G.-Y. Wei, and D. Brooks. Thread Motion: Fine-Grained Power Management for Multi-Core Systems. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 302-313.

[22] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. *SESC: Cycle Accurate Architectural Simulator.* http://sesc.sourceforge.net, 2005.

[23] J. Sartori and R. Kumar. Distributed Peak Power Management for Many-core Architectures. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, April 2009.

[24] E. Schuchman and T.N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 160-171.

[25] J. Sharkey, A. Buyuktosunoglu, and P. Bose. Evaluating Design Tradeoffs in On-Chip Power Management for CMPs. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2007, pp. 44-49.

[26] D. Shelepov, J.C.S. Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z.F. Huang, S. Blagodurov, and V. Kumar. HASS: A Scheduler for Heterogeneous Multicore Systems. In the *ACM SIGOPS Operating Systems Review*, April 2009, pp. 66-75.

[27] P. Shivakumar, S.W. Keckler, CR. Moore, and D. Burger. Exploiting Microarchitectural Redundancy for Defect Tolerance. In the *Proceedings of the International Conference on Computer Design (ICCD)*, Oct. 2003, pp. 481-488.

[28] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006, pp. 73-82.

[29] K. Skadron, M.R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, June. 2003, pp. 2-13.

[30] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, June 2005, pp. 520-531.

[31] D. Tarjan, S. Thoziyoor, and N.P. Jouppi. CACTI 4.0. *HP Laboratories Palo Alto Technical Report HPL-2006-86*, 2006.

[32] R. Teodorescu and J. Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, June 2008, pp. 363-374.

[33] Y. Wang, K. Ma, and X. Wang. Temperature-Constrained Power Control for Chip Multiprocessors with Online Model Estimation. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 314-324.

[34] J.A. Winter and D.H. Albonesi. Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures. In *Proceedings of the 38th International Conference on Dependable Systems and Networks*, June 2008, pp. 42-51.

[35] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. *University of Virginia, Department of Computer Science, Technical Report CS-2003-05*, March 2003.