Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures

Jonathan A. Winter and David H. Albonesi
Computer Systems Laboratory, Cornell University
{winter, albonesi}@csl.cornell.edu

Abstract

In future large-scale multi-core microprocessors, hard errors and process variations will create dynamic heterogeneity, causing performance and power characteristics to differ among the cores in an unanticipated manner. Under this scenario, naïve assignments of applications to cores degraded by various faults and variations may result in large performance losses and power inefficiencies. We propose scheduling algorithms based on the Hungarian Algorithm and artificial intelligence (AI) search techniques that account for this future uncertainty in core characteristics. These thread assignment policies effectively match the capabilities of each degraded core with the requirements of the applications, achieving an ED^2 only 3.2% and 3.7% higher, respectively, than a baseline eight core chip multiprocessor with no degradation, compared to over 22% for a round robin policy.

1. Introduction

The microprocessor industry has transitioned to the strategy of incorporating additional processor cores on a die with each new process generation. While this chip multiprocessor (CMP) approach has the potential to provide supercomputer levels of processing performance on a single die, to bring this vision to fruition in the long term, architects must address a number of significant challenges, among them programmability, power consumption, and reliability.

In terms of the latter, while transient (soft) errors are the primary focus today, permanent (hard) errors and circuit variability are expected to become a major challenge in the future [4]. In this paper, we consider permanent faults and variability that are caused by imperfections in the manufacturing process and from wear-out over the lifetime of the chip. As transistors continue to shrink to microscopic dimensions, the manufacturing process becomes less dependable, resulting in more defective transistors and wires. Moreover, these components more easily wear out when subjected to the stress of high levels of activity, power, and temperature.

Ultimately, a major consequence of decreasing hardware reliability is that many cores on the die will provide performance/power efficiency levels below that to which they were designed. Some components will have faults, certain circuits will be leakier than normal, and some transistors on critical paths will be slower, thereby requiring reduced frequency for correct operation. Manufacturers will not have the option of shipping only fully functional chips as this will necessitate unaffordably low yields. Instead, in order to provide reasonable performance at acceptable cost, future CMPs will be designed to tolerate faults and variations and operate in a degraded state [29]. These degradations are largely random physical processes that occur during manufacturing and usage. Consequently, each core will be uniquely affected by manufacturing and wear-out defects. Thus, the resulting degraded CMP will be an *unpredictably heterogeneous* multi-core system, even if it was designed to be homogeneous.

A number of previous studies [1,5,28,29,30,33] have explored processors that can tolerate manufacturing and wear-out faults and variations, thereby keeping these chips operational. Prior research has not yet addressed the problem of ensuring that these degraded multi-core processors deliver adequate performance and power efficiency throughout their expected lifetime. Even if processors are able to continue to function in the presence of errors and variability, they may not deliver the minimum expected level of power/performance efficiency, causing them to be rendered unusable before their expected end of life. This paper addresses this issue through self-tuning operating system scheduling policies that use high-level system feedback to match application characteristics to the degraded cores so as make the performance and

power impact of hard errors and variability imperceptible to the user.

While many prior reliability efforts examined purely hardware solutions [5,17,18,28,29,30,32,33], we take a combined hardware/software approach to address this complex scheduling problem. The hardware is best capable of providing feedback on the performance and power dissipation of threads running on the degraded processors. On the other hand, the operating system is best situated to assess the overall situation and to balance the requirements of each application from a global perspective.

We explore two methodologies for attacking the scheduling problem. First, by assuming that application behavior changes slowly and that interactions between applications are limited, we reduce the scheduling problem to the Assignment Problem, which can be solved by employing the Hungarian Algorithm [21]. Our second approach is to apply iterative optimization algorithms that have been shown to be effective on many similarly difficult combinatorial problems [25,27]. These iterative techniques operate with little domain-specific knowledge, are easy to implement, have low computational requirements, and continuously improve their solution, permitting the user to trade off algorithm runtime and solution quality. To our knowledge, our study is the first work to apply iterative optimization algorithms to heterogeneous multi-core thread scheduling.

2. Related Work

A number of prior research areas relate closely to our degraded multi-core scheduling problem. First, a number of efforts address architectural techniques to tolerate permanent errors resulting from manufacturing defects or wear-out during the processor lifetime. Shivakumar et al. [29] suggest exploiting inherent redundancy in the processor for hard error tolerance. Srinivasan et al. [33] propose two methods to increase the processor lifetime: structural duplication and graceful performance degradation. Schuchman and Vijaykumar [28] develop a methodology for testing architecture level components and isolating faults. Bower el al. [5] address fault diagnosis and unit de-configuration. Distributed built-in self-testing and checkpointing techniques are devised by Shyam et al. [30] for detecting and recovering from defects. Finally, Aggarwal et al. [1] study mechanisms for isolating faulty components in a CMP and reducing an error's impact through reconfiguration. We assume such schemes are already implemented in our baseline CMP, permitting cores to function in degraded states. Our work examines the next stage of the problem: making most effective use of the resulting heterogeneous CMP by scheduling applications to match core and workload characteristics.

A second direction of prior research strives to understand, model, and mitigate manufacturing process variations (PV). Most work on PV focuses on the semiconductor device and circuit level, but a number of researchers have devised system-level approaches. Humenay et al. [10, 11] examine how parameter variations specifically impact multi-core chips. A number of studies [17,18,22] propose techniques to reduce the negative impact of variations on frequency and yield. Many of the mechanisms create heterogeneity on a CMP by disabling array elements or creating variable access times.

Other previous research uses the operating system to improve CMP energy efficiency. Juang et al. [13] argue for coordinated formal control-theoretic methods to manage energy efficiency in multi-core systems. Li and Martínez [16] investigate heuristics that adaptively change the number of cores used, and the chip voltage and frequency to optimize power-performance in parallel applications. Isci et al. [12] further develop globally aware policies to dynamically tune DVFS to workload characteristics to maximize performance under a chip-wide power constraint. While this effort has similar elements to ours, they use DVFS to improve efficiency, whereas in our heterogeneous system, we use core scheduling.

Most papers on power-aware multi-core thread scheduling are primarily concerned with thermal control in homogeneous chip multiprocessors [7,8,20,24,34]. In heterogeneous chip multiprocessors architectures [3,15], the heterogeneity is designed into the system rather than the unintentional result of hardware faults and variations. As a result, the degree and nature of heterogeneity is quite different. In Kumar et al. [15] the focus is on multi-programmed performance and applications are scheduled on cores to best match 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. Our scheduling problem is far more complex: an unpredictably large number of heterogeneous organizations can arise in term of frequency, dynamic power, and leakage currents, in addition to architectural parameters.

Kumar et al. [14] study heterogeneous architectures where the cores are not restricted to a few configurations. The goal is to determine how much heterogeneity is necessary and how the cores should be designed to fit a given power budget. They focus on the

architectural design issues rather than the scheduling aspect of the problem. Balakrishnan et al. [2] study the impact of asymmetry in core frequency on parallel commercial workloads using a hardware prototype. Ghiasi et al. [9] examine heterogeneity resulting from cores running at different voltages and frequencies. While their work adapts the core voltages and frequencies, we investigate cores with unpredictably heterogeneous frequencies and power output. We adapt the workload assignment to mitigate possible negative affects.

3. Scheduling Algorithms for Unpredictable CMPs

We propose scheduling algorithms that assign applications to cores over a fixed, relatively short period of time. Scheduling decisions are periodically reassessed to account for large application phase changes, programs completing, and new applications arriving to be processed. Our best algorithms consist of an exploration phase where samples of thread behavior on different cores are observed and a steady phase during which the algorithm runs the best schedule it found during the sampling phase.

Table 1 compares the scheduling algorithms that we explore in this paper. The table specifies the complexity of each algorithm where N is the number of cores (and the number of applications). For comparison purposes, we also implement randomized and round robin scheduling, two simple algorithms that have worked well on past multi-core designs.

3.1. Hungarian Scheduling Algorithm

The Hungarian scheduling algorithm is based on the Hungarian Algorithm developed by mathematicians to solve the well-known Assignment Problem, also called Weighted Bipartite Matching in graph theory [21]. During the exploration phase, the algorithm samples application performance and power statistics on each core and picks the best scheduling assignment. In general, finding the best schedule is extremely difficult because threads interact during execution through contention for I/O and memory bandwidth as well as through heat conductivity between cores. Furthermore, program behavior is dynamic both in the short-term time frame and over large program phases, such that sample information may not reflect future behavior.

In order to simplify the problem, our algorithm assumes that there are no such interactions between threads and that program behavior is static – at least for the duration of the exploration phase. Making these assumptions eliminates the interdependence between execution samples running simultaneously, reducing the scheduling problem to the Assignment Problem.

The Assignment Problem is defined as follows. Given an N×N cost matrix where the (i,j) element represents the cost of running application i on core j, find the assignment of applications to cores with lowest total cost. In our case, the elements of the cost matrix consist of the normalized energy-delay-squared (ED^2) product obtained by first sampling the execution of applications on each core. For each application, we divide each ED^2 sample by the ED^2 obtained during the first sampling interval to obtain the normalized values. Normalization ensures that applications are treated fairly by the scheduler despite any differences in the absolute value of their performance and power data.

```
Step 1: For each row of the matrix, find
the smallest element and subtract it from
every element in its row. Go to Step 2.

Step 2: Find a zero (Z) in the resulting
matrix. If there is no starred zero in
its row or column, star Z. Repeat for
each zero in the matrix. Go to Step 3.

Step 3: Cover each column containing a
starred zero. If N columns are covered,
the starred zeros describe a complete set
of unique assignments and the algorithm
is done. Otherwise, go to Step 4.

Step 4: Find a non-covered zero and prime
it. If there is no starred zero in the
row containing this primed zero, go to
Step 5. Otherwise, cover this row and
uncover the column containing the starred
zero. Continue in this manner until all
zeros are covered. Save the smallest
uncovered value and go to Step 6.

Step 5: Construct a series of alternating
primed and starred zeros as follows. Let
Z0 represent the uncovered primed zero
found in Step 4. Let Z1 denote the
starred zero in the column of Z0 (if
any). Let Z2 denote the primed zero in
the row of Z1 (there will always be one).
Continue until the series terminates at a
primed zero that has no starred zero in
its column. Unstar each starred zero of
the series, star each primed zero of the
series, erase all primes, and uncover
every line in the matrix. Return to Step
3.

Step 6: Add the value found in Step 4 to
every element of each covered row, and
subtract it from every element of each
uncovered column. Return to Step 4
without altering any stars, primes, or
covered lines.
```

Figure 1: The six step Hungarian Algorithm

Figure 1 outlines the six steps of the Hungarian Algorithm as described in [23]. The algorithm takes the

cost matrix as input and proceeds by manipulating rows and columns through addition and subtraction to find a set of *starred* zero elements that represent the optimal assignment. During the algorithm, rows and columns are *covered* and zeroes are *starred* and *primed* to indicate special status. When the algorithm completes, there are N starred zeroes. A starred zero at location (i,j) means that the optimal solution to the Assignment Problem schedules application i to run on core j. The Hungarian scheduler then uses the best assignment for the simplified problem as the schedule for the steady-state phase.

Table 1: Scheduling algorithms

Algorithm	Exploration Phase (in cycles)	Complexity
Randomized	none	O(N)
Round Robin	none	O(N)
Hungarian	8 intervals of 12.5M	$O(N^3)$
Global Search	25 intervals of 4M	O(N)
Local Search	25 intervals of 4M	O(N)

3.2. Iterative Optimization Algorithms

Our other approach is to use iterative optimization algorithms inspired by artificial intelligence research [25,27]. These algorithms are highly suited to this scheduling task because they are generally simple to implement, have low computational requirements, and yet are extremely effective in practice. The simplest search algorithms are greedy: they avoid searching in directions that initially appear to have performance slowdowns or power inefficiencies even if they may hold promise in the future. Therefore, these greedy algorithms may get stuck in local minima. However, in practice, greedy algorithms are quite effective in certain problem domains and are often used due to their simplicity. In this paper, we study global search and local search.

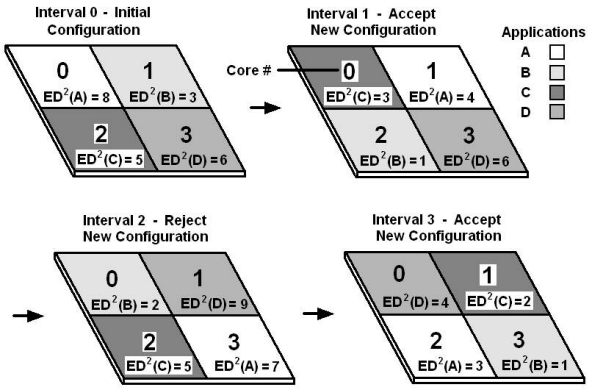

Figure 2: The global search algorithm

In global search (Figure 2), the processor is configured into a new random schedule in each interval of the exploration phase of the algorithm. The operating system keeps track of the best configuration thus far and employs this configuration during the longer, steady-state phase. Figure 2 illustrates how global search operates on a sample four core chip multiprocessor. Global search has the advantage of rapidly exploring a broad range of configurations in a large search space such as a CMP with many cores. However, it may not arrive at a near-optimal solution.

Local search defines a neighborhood of assignment options that are closely related to the current configuration. During each exploration interval, a member in the neighborhood of the current assignment is selected as the next assignment. If this new assignment performs better than the original, then it is kept and local search proceeds from this new configuration. If the new assignment does not function as well as the original, then local search reverts to searching further in the neighborhood of the original solution. We define the neighborhood of a scheduling configuration as all schedules that can be derived from the original schedule through some fixed number of pair-wise swaps.

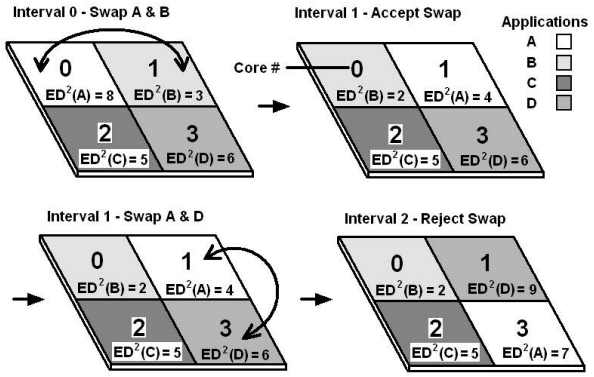

Figure 3: The one swap local search algorithm

In our results, we explore how many pair-wise swaps the algorithm should make per interval to determine the best setting. The advantage of selecting among a neighborhood of configurations that are derived from a few or even just one swap is that assignments in close proximity to the original are likely to have quite similar performance. This will lead to a more gradual search that steadily improves the solution and avoids the large changes which could lead to poor results. On the other hand, increased swapping more rapidly explores the search space of assignments. Figure 3 demonstrates how local search works when one swap is performed per iteration. Figure 4 shows a two swap version of local search and highlights a key

improvement in our algorithm which allows some of the swaps from an interval to be retained while others are discarded. We also implemented a version of local search that uses hill climbing [27] to escape local minima. We found, however, that the improvements over greedy search were minimal, indicating that the algorithms were not greatly impacted by local minima.

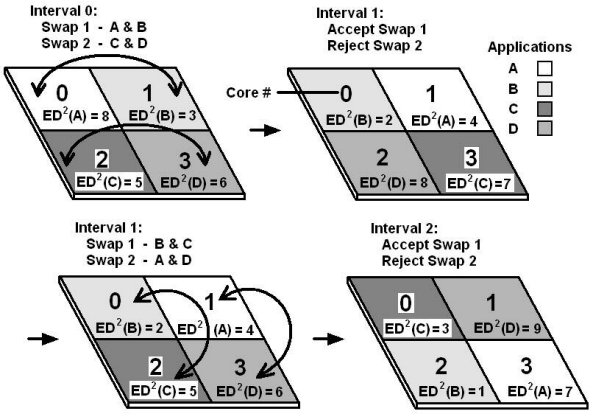

Figure 4: The two swap local search algorithm

4. Methodology

Our simulation infrastructure is based on the SESC simulator [26]. We improved the power and thermal modeling by augmenting SESC with Cacti 4.0 [35], an improved version of Wattch [6], the block model of Hotspot 3.0 [31], and an improved version of HotLeakage [36]. We extended Wattch and HotLeakage to model the dynamic and static power of all the units not addressed in Cacti 4.0, including logic structures such as the decoder, dependency check logic, issue queue selection logic, and ALUs. We assume a nominal clock frequency of 4.0 GHz and a supply voltage of 1.0V.

In order to efficiently simulate large multi-core architectures, we developed a parallel simulation framework. For this study, we focus on workloads of single-threaded applications chosen from the SPEC CPU2000 benchmarks. Multi-threaded workloads will present a unique set of additional challenges when run on a heterogeneous CMP and we leave this added dimension to future work.

With these workloads, direct interaction among applications executing on different cores is limited. While heat from one core conducted across the silicon die can cause inter-core heating effects, in our design, private L2 caches surround each core. These large caches have low and relatively uniform activity and thus act as heat sinks preventing much of the heating from another core from affecting its neighbors. The second major interaction among cores is their contention for off-chip memory bandwidth. We assume the bandwidth is statically partitioned among the cores. This avoids further complicating our already large search space of thread scheduling and core configuration options.

With these assumptions, we simulate a multi-core processor using single-core simulations to obtain performance, power, and thermal statistics that are then combined by a higher level chip-wide simulator that performs the role of the operating system scheduler. The chip-wide simulator is responsible for setting up the proper application assignments for each interval in the sampling phase, gathering and interpreting the individual core results, and applying the algorithms to determine the best schedule for the steady-state phase. A major advantage of this approach is its scalability to CMPs with a large number of cores.

Table 2: Core architectural parameters

Front-End Parameters	
Branch Predictor	Hybrid of gshare and bimodal with 4K entries in the bimodal, gshare 2nd level, and meta predictor
Branch Target Buffer	512 entries, 4-way assoc.
Return Address Stack	64 entries, fully assoc.
Front-End Width	3-way
Fetch Queue Size	18 entries
Re-Order Buffer	100 entries
Retire Width	3-way
Back-End Parameters	
Integer Issue Queue	32 entries, 2-way issue
Integer Register File	80 registers
Integer Execution Units	2 ALUs 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 assoc., 1 port, 1 cycle latency
Instruction TLB	32 entry, fully assoc., 1 port
Load Queue	32 entries, 2 ports
Store Queue	16 entries, 2 ports
L1 Data Cache	8KB, 2-way assoc., 2 ports, 1 cycle latency
Data TLB	32 entry, fully assoc., 2 ports
L2 Cache	1MB, 8-way assoc., 1 port, 10 cycle latency
Main memory	1 port, 200 cycle latency

Our baseline architecture consists of an eight core homogeneous chip multiprocessor with no degradation

due to hard failures or variations. Each core is a single-threaded, 3-way superscalar, out-of-order processor. The main architectural parameters are listed in Table 2. In order to model temperature-dependent leakage power, we created a core floor plan. Each core is surrounded by its L2 cache modeled as four banks and illustrated in Figure 5.

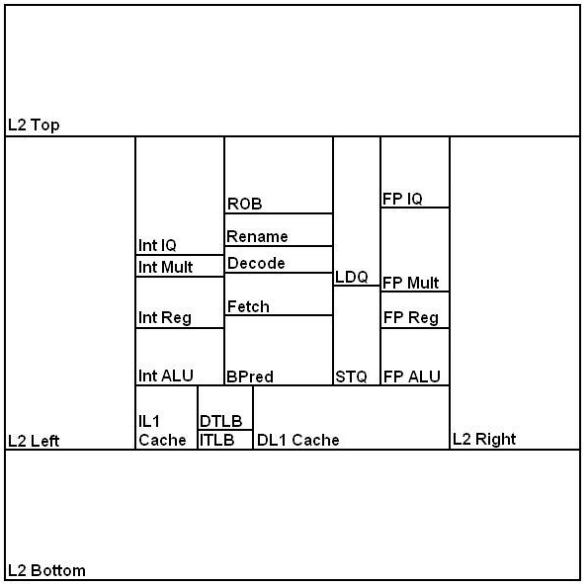

Figure 5: Processor core floor plan

The task of modeling faults and variations in an architectural simulation is quite challenging. Much of the effect from errors and variability on a chip is highly device and circuit dependent and such low level details are not available at the time of initial architectural design. In this work, we focus on the architecturally visible effects of faults and variations. We study processor configurations that have become degraded from the nominal design through manufacturing inconsistencies as well as wear-out over the lifetime of the device. For this study, the specific source of the degradation – manufacturing or wear-out – is not important because we focus on adapting the OS thread scheduling and core configuration *ex post facto*.

We focus on three forms of processor degradation. First, we model errors that cause the system to disable part of a pipeline component such as an ALU, load queue port, or set of ROB entries. We focus on large granularity errors that damage significant portions of the structure. Prior work has shown that when only a few entries in structures, such as an issue queue or register file, are damaged, the performance impact (assuming graceful degradation) is negligible, and thus adaptation is unnecessary [17]. Second, we assume core frequency degradation from manufacturing process variations that result in slow transistors in critical circuit paths [17,22]. Prior work has found that these variations can increase processor cycle time by as much as 30%, eliminating an entire technology generation's worth of frequency improvement [4]. Third, we assume leakage current variations, which are also caused by process variations that diminish the quality of the transistors, magnifying sub-threshold and gate leakage currents.

Past research concluded that excessive leakage currents will be a very serious problem, with some [4] saying that leakage variability across dies could be as high as 20X. Others [10] suggest that even at 45nm within-die variations alone could cause leakage differences among cores of as much as 45%. Following the arguments of [10,11], we focus on leakage variations that can be attributed to systematic variability. Thus, we consider leakage variations that affect an entire core as in [10] as well as those that affect a group of architectural blocks in close proximity.

Table 3: Degraded CMP configuration

Core	Structural Faults	Frequency Degradation	Leakage Increase
1	2x normal memory latency (100 ns)	–	2x in the L1 caches
2	half the nominal size integer issue queue (16)	20% (3.2 GHz)	2x for the whole core
3	half the nominal size load queue (16)	10% (3.6 GHz)	2x in the store and load queues
4	one integer ALU is disabled	20% (3.2 GHz)	–
5	integer issue queue can only issue one instruction per cycle	–	–
6	half the L2 cache is broken leaving 500KB	10% (3.6 GHz)	2x in the integer cluster
7	half the nominal ROB entries (50)	–	2x in the FP cluster
8	half the nominal size store queue (8)	–	2x in the front-end

In a CMP where cores could be affected in a multitude of ways, there are numerous heterogeneous core configurations that could arise. In this study, we assume the degraded CMP configuration shown in Table 3. We assumed each core experienced some form of faults or variation but each processor was only affected by at most a few problems.

To test the effectiveness of our scheduling algorithms, we created the four eight-threaded workloads of SPEC CPU2000 applications shown in Table 4. Each benchmark was used evenly among the four workloads. For each simulation, we fast forwarded every benchmark five billion instructions, and then executed one billion cycles in SESC, or 0.25 seconds at a nominal frequency of 4 GHz. Cores that run at lower frequencies execute for proportionally fewer cycles.

Table 4: Workloads

Workload 1	applu, bzip2, equake, gcc, mcf, mesa, parser, swim
Workload 2	ammp, apsi, art, crafty, twolf, vortex, vpr, wupwise
Workload 3	mesa, ammp, applu, crafty, vortex, gcc, wupwise, mcf
Workload 4	swim, parser, vpr, bzip2, art, apsi, twolf, equake

The OS scheduler periodically switches between the exploration and steady-state phases of the algorithm. During the exploration phase, which constitutes 10% of the total execution time, the algorithm adapts to workload changes to find the best assignment of threads to cores. During the longer steady-state phase, the CMP runs with this best configuration. The performance of the algorithm is based on both the exploration and steady-state phases. The length and number of the sampling intervals are algorithm dependent parameters and are chosen to the best advantage of each technique. For each workload, we performed five different runs with different application-to-core starting assignments, and report the average, best, and worst results.

For the simpler randomized and round robin algorithms, we modeled 10 million cycle operating system time slices, the equivalent of 2.5 milliseconds. These algorithms do not require exploration and instead they use each time slice interval to perform their reassignments.

5. Results and Discussion

In this section, we present the results of the various scheduling algorithms on our degraded eight core CMP. All comparisons are made using the energy-delay squared (ED^2) metric against a baseline with no errors or variations and an oracle scheduler which uses *a priori* knowledge to derive the best schedule among all possible options. We chose ED^2 as the metric in order to balance performance with power dissipation [19]. Section 5.1 discusses how simple schedulers compare to the non-degraded baseline. Section 5.2 shows how the Hungarian and AI search algorithms fare against the offline oracle. Finally, Section 5.3 provides an overall comparison of the scheduling algorithms.

5.1. Simple Scheduling Algorithms

We first evaluate the effectiveness of two simple scheduling algorithms – round robin and randomized – that are suitable for homogeneous CMPs and statically designed heterogeneous CMPs, on the degraded CMP of Table 3. The round robin scheduler rotates the threads on the cores at the beginning of each OS interval. This approach avoids a worst case assignment by limiting how long an application runs on any given core. The even assignment of applications to processors also avoids high power density scenarios and uneven wear-out of a core through over-activity or high temperature.

The randomized scheduler randomly assigns threads to cores every operating system interval. This approach avoids degenerate behavior that might occur with round robin such as destructive interference with program phases.

Figure 6 shows the results of these schedulers on the degraded CMP relative to a baseline with no degradation. Both approaches degrade ED^2 by over 22% on average. The final bar on the graph, the worst-case schedule, shows that an arbitrary assignment of threads to cores can degrade ED^2 by almost 45% compared to the baseline. Clearly, naïve policies can result in an unacceptable loss in power/performance that may render the degraded microprocessor unusable.

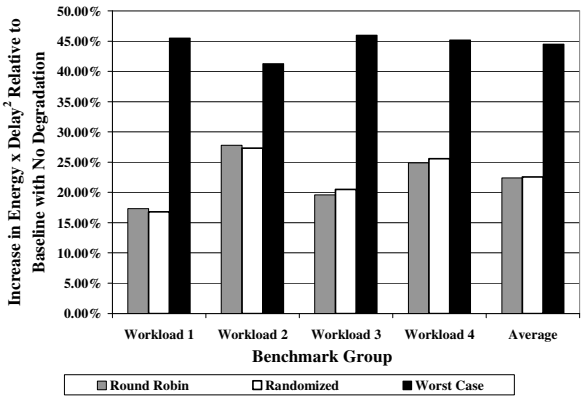

Figure 6: Comparison of simple schedulers

5.2. Hungarian Policy and Search Algorithms

The Hungarian scheduling policy samples each benchmark on each core during the exploration phase,

and then computes the best assignment among all permutations (assuming no interactions or phase behavior). For the Hungarian policy, the exploration phase is divided into eight intervals, each 12.5 million cycles long, during which the eight applications are executed once on each core, by starting with an initial assignment and then rotating the threads in a round robin fashion seven times. This allows the scheduler to generate the 8×8 cost matrix of ED^2 values to use as input to the algorithm.

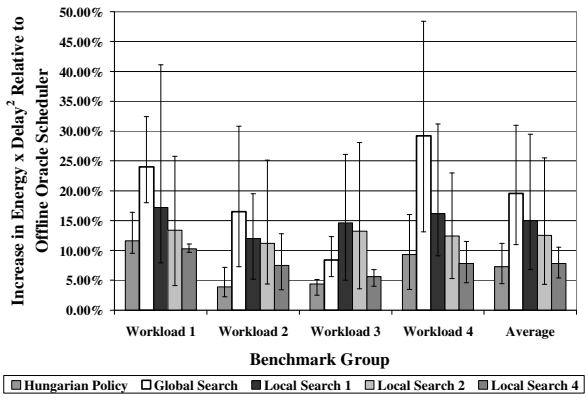

Figure 7: Comparison of advanced schedulers

Figure 7 shows the ED^2 of the Hungarian scheduling algorithm compared to the oracle scheduler. The solid bar represents the average of the five runs, and the error bars show the best and worst results. The algorithm performs well, suffering only a 7.3% increase in ED^2 relative to the oracle. The performance and power characteristics of the benchmarks during the initial 100 million cycle exploration phase are quite reflective of the overall traits of the benchmarks. Thus, using the Hungarian Algorithm to calculate the best solution among all possible scheduling permutations based on this sampling information yields a good assignment over the whole run, regardless of the starting assignment.

While effective, the Hungarian scheduling algorithm has $O(N^3)$ complexity, while the other algorithms are of O(N). We simulated the Hungarian Algorithm on our baseline core configuration and found it takes approximately 200K cycles to solve a cost matrix with eight cores, a non-trivial cost that may not scale well to larger-scale CMPs. Since the number of sampling intervals scales linearly with the number of cores, a large amount of online profiling will also be required for chips with tens or hundreds of cores. Moreover, the algorithm may not work well when there are significant interactions among applications or rapid phase changes.

The global and local search algorithms divide the exploration phase into 25 intervals of four million cycles. Both start with the initial configuration and try other configurations, greedily pursuing paths that improve on the best schedule to date. Global search simply tries the initial configuration and 24 other randomly chosen ones and then selects the best among them for the steady-state phase. This strategy sometimes works quite well but can perform poorly depending on the 25 configurations pursued. Overall, global search degrades ED^2 by 19.5% over the oracle scheduler.

Three versions of the local search method were implemented which vary in the number of pair-wise swaps performed to explore a neighboring configuration. Local Search N uses N pair-wise swaps such that two benchmarks are involved in each switch for Local Search 1, while all benchmarks are swapped for Local Search 4. Local Search 1 makes a swap and then runs that schedule for the next 4 million cycle interval. If performance improves, it keeps that new configuration; otherwise, it selects another neighbor of the original solution. The comparison is made using the average of the normalized ED^2 (with respect to the ED^2 of the previous interval) of the two threads involved in the swap. Local Search 2 and Local Search 4 have an additional feature to improve their performance. Instead of collectively accepting or rejecting all the swaps made in an interval, beneficial pair-wise swaps are kept and others discarded. From the results in Figure 7, the additional pair-wise swaps of Local Search 2 and Local Search 4 significantly improves the algorithm; the ED^2 increase achieved with one, two, and four pair-wise swaps each interval is 15.0%, 12.6%, and 7.8%, respectively. Moreover, Local Search 4 significantly outperforms global search. The error bars show that Local Search 4 is also less sensitive to the initial assignment due to its ability to more rapidly search the space of possible assignments.

5.3. Overall Comparison

In Figure 8, we compare all the scheduling algorithms to the non-degraded chip multiprocessor. The offline oracle scheduler achieves 3.1% better ED^2 than the CMP without degradation. This occurs due to the fact that some of the some of the degraded cores operate at lower power, due to lower frequency or failed components that are power gated. Consequently, an omniscient scheduler can find an assignment that is more power/performance efficient than the baseline.

Moreover, both the Hungarian and Local Search 4 scheduling algorithms achieve ED^2 values very close to the non-degraded baseline – higher only by 3.2% and

3.7%, respectively – compared to the over 22% degradations with naïve schedulers. Thus, intelligent scheduling will be critical to maintaining acceptable levels of power/performance efficiency on future CMPs degraded by wear-out and variations.

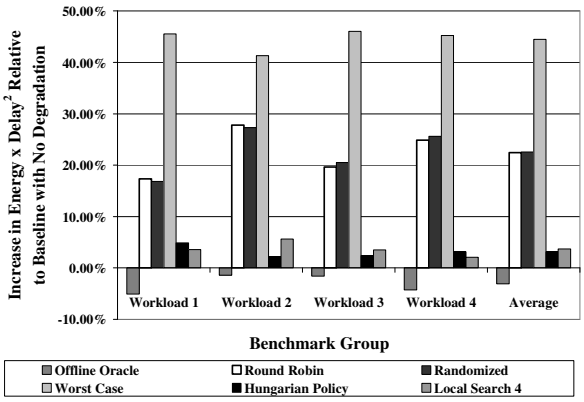

Figure 8: Overall comparison

6. Conclusions

In future CMPs, variations and hard errors will conspire to create dynamic heterogeneity among the cores. Unlike statically designed heterogeneous CMPs, the unpredictability of manufacturing defects, wear-out mechanisms, and variations will require self-tuning scheduling techniques that efficiently find a near-optimal schedule given any degraded CMP scenario, thereby making the chip degradation imperceptible to the user. In this paper, we devise a number of different scheduling algorithms for finding near-optimal thread to core assignments in a degraded CMP.

We first demonstrate that simple policies, such as round robin scheduling, degrade ED^2 to the point that the chip may be rendered unusable. Under the assumption of limited core-to-core interaction, we observe that the scheduling problem reduces to the Assignment Problem and can be addressed through the Hungarian Algorithm. We devise a scheduler based on this algorithm that achieves an ED^2 close to that of an oracle scheduler. We further develop schedulers based on AI search techniques that obviate the requirement of limited core-to-core interaction, and that better scale to large CMP organizations. The most scalable and effective of these policies rapidly arrives at a near-optimal solution that degrades ED^2 by only 3.7% over a non-degraded architecture, compared to over 22% for simple approaches.

For future work, we plan to investigate algorithms for CMPs with tens to hundreds of cores, and those that address workloads containing a mix of parallel and sequential applications.

Acknowledgements

The authors thank Ken Birman for his valuable feedback, Paula Petrica for her help with the submission, and the anonymous referees for their useful comments. This research is supported by NSF grants CCF-0732300 and CCF-0541321.

References

[1] N. Aggarwal, P. Ranganathan, N.P. Jouppi, and J. E. Smith. Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors. *International Symposium on Computer Architecture (ISCA)*, 2007.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *International Symposium on Computer Architecture (ISCA)*, 2005.

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

[4] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter Variations and Impact on Circuits and Microarchitecture. *Design Automation Conference (DAC)*, 21.1, 2003, pp. 338-342.

[5] F. A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. *International Symposium on Microarchitecture (MICRO)*, 2005.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *International Symposium on Computer Architecture (ISCA),* 2000, pp. 83-94.

[7] P. Chaparro, J. González, G. Magklis, Q. Cai, and A. González. Understanding the Thermal Implications of Multi-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 18, 8, 2007, pp. 1055-1065.

[8] J. Donald and M. Martonosi. Techniques for Multi-Core Thermal Management: Classification and New Exploration. *International Symposium on Computer Architecture (ISCA)*, 2006.

[9] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for Heterogeneous Processors in Server Systems. *ACM International Conference on Computing Frontiers (CF)*, 2005, pp. 199-210.

[10] E. Humenay, D. Tarjan, and K. Skadron. Impact of Parameter Variations on Multi-Core Chips. *Workshop on Architectural Support for Gigascale Integration (ASGI)*, 2006.

[11] E. Humenay, D. Tarjan, and K. Skadron. Impact of Process Variations on Multi-Core Performance Symmetry. *Design, Automation and Test in Europe (DATE)*, 2007.

[12] 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. *International Symposium on Microarchitecture (MICRO)*, 2006.

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

[14] R. Kumar, D. M. Tullsen, and N.P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. *International Symposium on Parallel Architectures and Compilation Techniques (PACT)*, 2006, pp. 23-32.

[15] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. *International Symposium on Computer Architecture (ISCA)*, 2004.

[16] J. Li and J.F. Martínez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. *International Symposium on High-Performance Computer Architecture (HPCA),* 2006.

[17] X. Liang and D. Brooks. Microarchitecture Parameter Selection To Optimize System Performance Under Process Variation. *International Conference on Computer-Aided Design (ICCAD)*, 2006, pp. 429-436.

[18] X. Liang and D. Brooks. Mitigating the Impact of Process Variations on Processor Register Files and Execution Units. *International Symposium on Microarchitecture (MICRO)*, 2006.

[19] A.J. Martin. Towards an Energy Complexity of Computation. *Information Processing Letters*. 77, 2001, pp. 181-187.

[20] A. Merkel and F. Bellosa. Balancing Power Consumption in Multiprocessor Systems. *EuroSys*, 2006, pp. 403-413.

[21] J. Munkres. Algorithms for Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*. 5(1), 1957, pp. 32-38.

[22] S. Ozdemir, D. Sinha, G. Memik, J. Adams, and H. Zhou. Yield-Aware Cache Architectures. *International Symposium on Microarchitecture (MICRO)*, 2006.

[23] R.A. Pilgrim. *Munkres' Assignment Algorithm.* http://csclab.murraystate.edu/bob.pilgrim/445/munkres.html, 2008.

[24] M.D. Powell, M. Gomaa, and T.N. Vijaykumar. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004, pp. 260-270.

[25] C.R. Reeves (Editor). *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill Book Company, London, UK, 1995.

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

[27] S.M. Sait and H. Youssef. *Iterative Computer Algorithms with Applications in Engineering*. IEEE Computer Society, Los Alamitos, CA, 1999.

[28] E. Schuchman and T.N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. *International Symposium on Computer Architecture (ISCA)*, 2005.

[29] P. Shivakumar, S.W. Keckler, CR. Moore, and D. Burger. Exploiting Microarchitectural Redundancy for Defect Tolerance. *International Conference on Computer Design (ICCD)*, 2003.

[30] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[31] K. Skadron, M.R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-Aware Microarchitecture. *International Symposium on Computer Architecture (ISCA)*, 2003, pp. 2-13.

[32] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. *International Symposium on Computer Architecture (ISCA)*, 2004.

[33] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. *International Symposium on Computer Architecture (ISCA)*, 2005.

[34] K. Stavrou and P. Trancoso. Thermal-Aware Scheduling for Future Chip Multiprocessors. *EURASIP Journal on Embedded Systems*, 2007.

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

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