When Homogeneous becomes Heterogeneous

Wearout Aware Task Scheduling for Streaming Applications

David Roberts, Ronald G. Dreslinski, Eric Karl
Trevor Mudge, Dennis Sylvester, David Blaauw
{daverobe,rdreslin,ekarl,tnm,dmcs,blaauw}@eecs.umich.edu
Advanced Computer Architecture Lab
University of Michigan
2260 Hayward Ave
Ann Arbor, MI 48109-2121

ABSTRACT

Recent trends in process technology suggest the need to monitor transistor wear-out in future processes. Because of within-die variation and the different computations being run on each core in a multi-core chip, this wear-out causes further imbalance to initial core frequencies as time progresses. Furthermore, manufacturing defects mean that cache sizes can vary between cores, adding further imbalance to a system. If we allow different cores to independently control their operating frequency we can achieve the best possible performance for their part of the die. Other parts of the system with slowly degrading performance can include interconnects and Flash-based file caches. In this paper we first explain how conventionally homogeneous multi-core processors can become heterogeneous over time. We discuss possible operating system based solutions to maximize the performance of a system as it wears out and present illustrative theoretical results based on linear programming. We demonstrate that for a class of streaming applications, an intelligent scheduling scheme recovers a significant amount of performance lost through wear-out. We advocate the need for multiple accurate performance measurements for effective scheduling in a wearout-aware multicore chip.

1. INTRODUCTION

The objective of this paper is to consider which techniques to apply for system management when performance levels of processors on a homogeneous multi-core chip become imbalanced. This situation arises due to variation and wear-out in the cores themselves, their caches or interconnect circuitry. The conventional design approach is to assume that all cores run at a global maximum frequency dictated by the slowest core [1]. This removes the need for complex synchronization for inter-core communication. However, allowing cores to operate at different frequencies allows the most performance to be extracted.

Multi-core chips are being built with smaller and less reliable process technologies. Intra-die variation means that different areas of a single chip can sustain different maximum frequencies, at a given supply voltage [1]. Alternatively, voltages could be adapted to maintain a consistent frequency for each core or group of cores [2]. Coupled with local caches that have different sizes at different voltages due to repaired defects [3], each thread of execution can run at a different performance level. This scenario is put forward by Sylvester et al. with their ElastIC architecture [4]. They propose an array of computation units with a central management unit to monitor and respond to the gradual wear-out of a future silicon chip. These wear-out effects include electromigration, gate oxide breakdown, negative-bias temperature instability (NBTI) and hot carrier injection, and will be discussed in a later section. There have also been publications on how to gracefully retire failed cores for continued operation on a conventional multi-core chip [5]. The overall effect of wear-out is for initially homogeneous multicore chips to become heterogeneous over time.

For multicores on unreliable process technologies, a network-on-chip (NoC) interconnect is beneficial, providing multiple routing paths in case of faults as well as support for different frequency domains [6]. To operate similarly to conventional multi-core architectures, the NoC needs to support cache coherence. Cache coherence in NoC systems is discussed in [7, 8]. In addition, defect-tolerant switches for communication between cores are proposed in [9].

Figure 1 shows an example of a system exposed to wear-out. The purpose of this diagram is to show the types of system component that can be affected by wear-out based on the most recent research ideas, rather than a contemporary architecture. Cores in the main voltage domain have their frequencies adjusted to match their individual f_{max} values. A core in a separate voltage domain has its voltage adjusted to meet a performance level dictated by the demands of software running on it, to save energy. The cores communicate through a set of fault-tolerant switches, which may employ a directory protocol and dynamic routing around failed nodes. There is external communication to off-chip DRAM memory, and a Flash-based disk cache [10]. All components of the system are subject to wear-out over time. Management at the system level is therefore important to obtain maximum throughput and lifetime. At a given point in time management of the system can be handled the same way a heterogeneous systems are handled. However, the change in the system over long periods of time means that traditional programming and compilation techniques available to heterogeneous scheduling may not be able to adapt. It will be beneficial for the OS to have a variety of performance counters to adapt to the long term changes that occur within the system due to wearout.

By deciding which software threads run on which cores, it

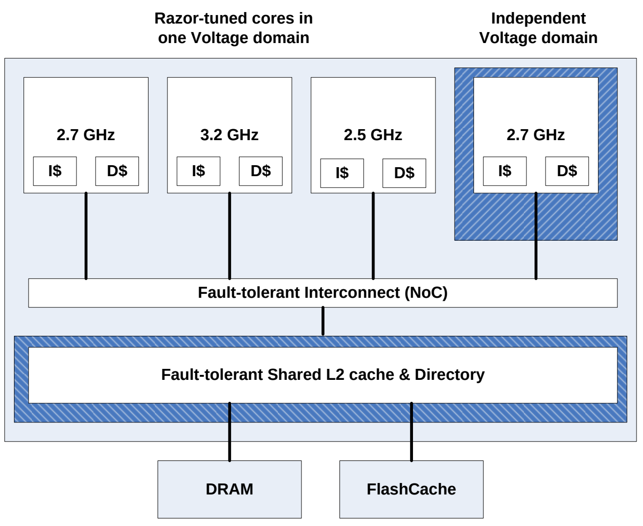

Razor-tuned cores in one Voltage domain
Independent Voltage domain
2.7 GHz
I$
D$
3.2 GHz
I$
D$
2.5 GHz
I$
D$
2.7 GHz
I$
D$
Fault-tolerant Interconnect (NoC)
Fault-tolerant Shared L2 cache & Directory
DRAM
FlashCache

Figure 1: Wear-out tolerant multicore chip

is possible to either maximize throughput or reduce power and wearout effects. Thread management on heterogeneous multicores has been addressed by prior work with the aim of optimizing performance [11] or under a power budget [12]. However, these did not consider a hardware platform whose performance changes over time. Several theoretical scheduling algorithms have been proposed which increase performance by running the most appropriate threads on each core [13, 14, 15]. To be accurate, schedulers require detailed knowledge and prediction of thread behavior on each core.

Centralized management on a dedicated unit has frequently been proposed as an option. For example, ElastIC [4] has a Diagnostic and Adaptivity processing unit (DAP) that takes units off-line and tests them using test vectors and sensor devices. It can also perform active healing to reverse wear-out effects including negative bias temperature instability (NBTI) and electromigration. The alternative is to temporarily allocate the task of system management and thread scheduling to an under-utilized computation unit [15].

The rest of the paper is organized as follows. Section 2.1 provides an explanation of process variation and wear-out. Section 2.2 covers the aspects of memories (including variable-size cache and Flash) and unreliable interconnect is discussed in 2.3. From a software standpoint we then provide background on Streaming applications (2.4) and thread scheduling for heterogeneous systems (2.5). In sections 3 and 4 we provide analytical results comparing different scheduling schemes to handle wearout. We conclude in section 5 by suggesting a coordinated approach to managing all of these components under wear-out conditions.

2. BACKGROUND & RELATED WORK

The first three sections will describe ways in which initially homogeneous systems degrade and become heterogeneous over time. Then the next two sections will present background and related work on a particular task scheduling problem for streaming workloads that will later be studied in the context of a homogeneous chip undergoing wearout effects.

2.1 Process Variation and Wearout

Recent trends [16, 17] suggest that the negative bias temperature instability effect (NBTI) and dielectric breakdown will remain critical mechanisms for degradation or "soft" breakdown of integrated circuits despite the transition to new manufacturing materials. Soft breakdown events, defined as an event in which the system fails to meet previous performance specifications (typically power or delay), are particularly difficult to safeguard against using traditional corner-based reliability qualification. Systems with different workloads, usage profiles and environmental conditions will degrade at different rates and to different extents. A suitable reliability guideline for one system in operation may place an unwarranted limitation on the performance of another system under different conditions. A brief review of these two dominant soft breakdown mechanisms will illustrate the nature and magnitude of soft breakdown's impact on system performance that one can expect over system lifetime. It is this impact on system performance that will lead an initially homogeneous system to degrade into a heterogeneous one over time.

The NBTI effect is observed primarily in PMOSFET devices biased negatively (conducting current) at high temperatures. The effect is caused by trap generation (and reversed by annealing) near the oxide interface when Si-H bonds are broken by collisions with holes. The effect is time dependent, following a power law relationship with the time of applied stress ($\alpha = 0.25$) and exhibiting a strong dependence upon the electric field across the oxide. The generation of the traps leads to a reduction in saturation current of the device, increasing device and circuit delay as the transistor ages. Interestingly, the process can partially reverse the effects through natural annealing when stress is removed. Thus, the precise degradation to expect on any given system is highly dependent upon workloads and environmental conditions. In [18], the simulated effect of NBTI on a 70nm device is a 50-55mV threshold voltage (V_{th}) shift, leading to 8-10% change in circuit delay over lifetime. Maximum delay changes to NBTI effects average 8.5% across ISCAS85 benchmark circuits in another work [19]. Although delay change in these studies is limited to 10% or less, a 50 mV shift in threshold may become more significant as the baseline threshold voltage may scale in future technology generations. The timescale for the stress and recovery effects of NBTI are uniquely short; 75% of the degradation or recovery can be observed minutes after the application or removal of voltage and temperature stress.

The oxide breakdown effect is caused by interaction with tunneling charges through thin-film oxides, used on the gate of MOSFET transistors and the insulation between neighboring interconnects. Traps, or interface states created in the oxide lead to a reduced energy required for a tunneling charge to pass through the oxide. Over time, the gate leakage current of a degrading MOSFET will increase eventually violating timing or power constraints and in extreme cases, causing a permanent fault at a particular circuit node. The oxide breakdown effect is exponentially dependent upon the electric field (voltage) and the probability of defect generation also has a strong, positive correlation with the oxide temperature. This implies that temperature reduction through frequency or voltage decreases and improved cooling can increase lifetime. Oxide-related soft breakdown can cause up to 30% variation in the delay of a simple buffer circuit [20]. The timescale for degradation in delay for oxide soft breakdown is highly variable, with slow degradation to ultimate failure taking up to years to develop [21].

A microprocessor and operating system can be made aware of this wear-out via several hardware techniques. Sampling-based detection [22, 2] works by either measuring signal propagation time through an artificial delay, or identifying and correcting late-arriving signals to handle the increasing logic delay mentioned above. Another option is periodic testing for correct operation under reduced delay margins in order to expose new paths whose delays have increased [23]. The operating system can periodically read the delay metrics and alter its schedule to match the modified hardware capabilities. For permanent, hard faults identified by error correction and detection circuits (in memories, for example) the OS can immediately be informed of the change in latency or storage capacity.

2.2 Memory

On-chip cache memories are becoming increasingly susceptible to process parameter variations and voltage changes [3]. This is also reflected by the adoption of cache fault-tolerance schemes such as Intel's Pellston [24], which selectively disables individual faulty cache lines. Studies have shown [25] that erratic fluctuations of minimum operating voltage can occur for random cells in 90nm SRAM arrays. SRAM reliability becomes worse with shrinking cell sizes, and there are three basic solutions. These are to maintain a higher voltage, keep SRAM cells relatively large or perform error correction. Error correction (ECC) increases memory access latency and if ECC capability is exceeded, reduces capacity by disabling blocks. The operating system must be aware of such changes to maintain maximum possible performance.

As shown in Figure 2 (taken from [3]), cache cell failure rate increases exponentially as voltage is scaled. The graph shows that for this simulated 45nm process, a small reduction in voltage can cause a large increase in defective cache lines. Supply voltage may need to be reduced to offset NBTI and oxide breakdown effects or to reduce energy consumption. As operating voltages are lowered there is a reduction in the static noise margin (SNM) of the SRAM cell. Since SRAMs are typically sized for optimal density and for use at high operating voltages, a reduction in SNM dramatically increases the likelihood that the cell value will be flipped when exposed to noise. Random dopant fluctuations (RDF) further exacerbate SRAM variability by changing device threshold voltage (V_{th}) [26]. At low voltages, there is a stronger relationship between V_{th} and device drain current, increasing the effect of RDF. Since RDF is a local process variation, individual devices within a single SRAM cell become mismatched, reducing static noise margin and increasing failure rates. The effects of RDF are expected to increase in newer process technologies since the change in V_{th} is inversely proportional to the root of the channel area. This will further increase SRAM failure rates.

As operating voltages are lowered conventional error correcting codes quickly become overwhelmed by the number of faulty bits. The only option at that point is to disable an entire cache line. This will have a negative effect on performance of which the operating system needs to be aware. In a system employing adaptive voltage scaling, this is a realistic scenario. Coupled with process variation, it is likely that processor caches will not be of equal size and their impacts should be considered. An operating system with knowledge of the current cache size or miss rate can perform more intelligent scheduling for performance or lifetime optimization.

Flash Memories are becoming a common feature of systems as a convenient, low-power non-volatile storage medium. They can also be applied in a more intensive context such as a low-power file cache [10]. As the Flash ages its write and erase latencies increase to compensate for reducing threshold voltage (V_t) margin (Figure 3) [27]. Erratic bits in Flash memories [28] have been observed when blocks of bits are erased, causing unpredictable jumps in threshold voltage. This behavior is due to positive point charges in the tunnel oxide, between the transistor channel and the floating-gate. When the number of faulty bits per block exceed the capabilities of an error-correcting code (ECC), blocks are disabled, reducing the capacity of the memory. Therefore this is another instance of wear-out affecting system performance over time, especially for file cache applications.

2.3 Interconnect

Electromigration is another reliability mechanism of interest given the trend of multi-core design and increasingly large on-chip memories. EM is an effect caused by the collision of electrons and metal atoms in interconnect leading to physical transport of the metal atoms in the direction of electron flow. Over time, the transport mechanism leads to opens or shorts in interconnect and failures in most circuits. The EM effect is pronounced with high current densities which may be observed in power networks on chip and is 1000 times more likely to lead to reliability issues in wires featuring mostly uni-directional electron flow [29]. Large bus structures between caches and processing elements or nodes in a multi-core or network on a chip are likely to result in unidirectional electron flow, since the bus wires are driven from different ends very frequently.

Hard faults may exist in the interconnect of a large multiprocessor system, and will not cause catastrophic failure if appropriate fault tolerance is in place. Soft errors can also

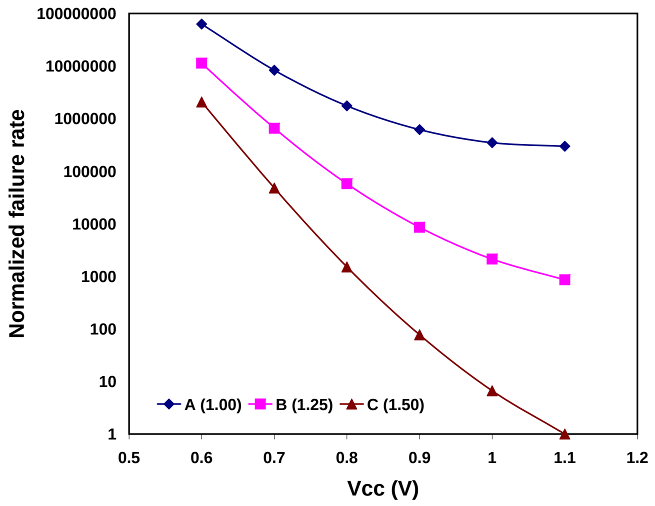


Figure 2: Cache cell failure rate for cells of increasing size (A, B and C) as a function of Voltage

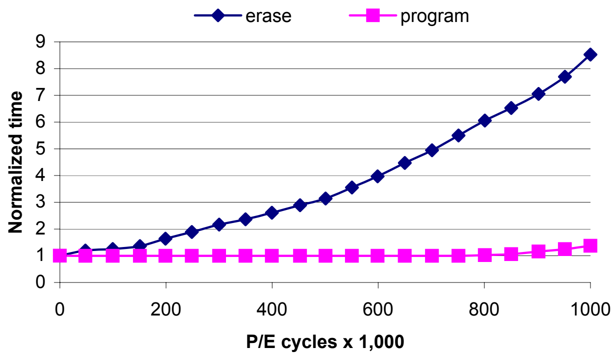


Figure 3: Increase in Flash program/erase time as flash cells wear out

occur and are often dealt with using multiple layers of error correction [6]. Because of the increased likelihood of hard faults in the future process technology where wear-out is most significant, a NoC allows connection and disconnection of faulty units. NoCs can affect performance however, especially for streaming applications where the choice of thread to CPU allocation can change the number of nodes across which data is routed.

As an example of how variable NoC latency can be handled, hardware could track coherence traffic identifying which threads communicate the most data to each other. With prior knowledge of the running software, for example through software hints, the operating system can be informed of communication patterns without extra hardware. A thread schedule can be formed whereby communicating threads are physically adjacent to each other to minimize router hops on a NoC.

2.4 Streaming Applications

Many modern applications are based on the concept of a stream. Multiple examples are presented in [30] and include software radio, network protocols, video and audio applications. They are characterized by operation on a large stream of data items that are processed by one or more algorithms in a graph of interconnected threads. The graph does not change very often for the duration of a program run.

Vadlamani et al. investigate memory hierarchy performance in the context of streaming programs on multi-threaded and multi-core hardware [31]. By measuring the coherence overhead of communication between threads within and across cores, they make architecture-specific software optimizations tailored for each architecture. For example, the synchronized pipelined parallelism model (SPPM) splits algorithms into stages allocated to each thread. These threads are forced to communicate through the cache, increasing performance for chips with shared L2 cache. However, for cores with private L2 caches, communication latency offsets the benefits of SPPM over the standard spatial decompo-

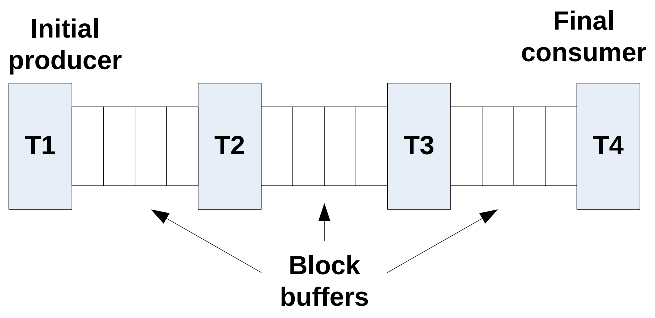


Figure 4: Four thread chain configuration

sition model (SDM) where the same algorithm operates on different data in parallel.

When the time arrives to allocate a thread to another core (to maximize performance), consumer threads should be allocated to the core that produced their input data because the producer's local cache will still contain that information. This is a similar concept to Polymorphic Threads [31] except that it would be a secondary hint applied to the throughput based thread migration scheme which we examine in our experiments (Section 3.1).

To demonstrate our wear-out aware schemes we employ a restricted type of stream program as a simple benchmark. It consists of a chain of threads running on a system with the same number of processors (Figure 4). The first thread simulates the work involved in processing a block of data by performing a number of iterations of a loop, before writing a block of data into a queue. Each pair of threads in the chain is connected by a queue from which it reads the next block of data to be processed, before writing the results to its output queue. This is similar to the benchmarks used with SPPM but with more processors and threads on a multi-core chip.

The benchmark is complete when a specified number of blocks have been processed by every thread in the chain. This simple program is adequate to show the benefits of wear-out aware scheduling.

2.5 Task Scheduling in Heterogeneous Systems

Task scheduling on heterogeneous multi-cores is a widely studied field. Theoretical as well as practical works exist, providing scheduling policies and heuristics to determine which thread to run on which processor and at what time. Our work specifically addresses streaming applications, and the following related work is relevant.

In [11] they present a dynamic assignment policy based on runtime thread behavior. Running SPEC2000 benchmarks on a system with Alpha EV5 and EV6 processors, they migrate threads between cores based on changes in IPC. Simulations are driven by execution traces on the M5 simulator. For a multi-core system where the cores have the same architecture, IPC alone is not an effective metric for scheduling because any thread will have the same IPC on any core (ignoring cache variations).

The Synchronized Pipelined Parallelism Model (SPPM) [32] runs experiments using a streaming model similar to our own. Producer and consumer thread pairs run on a simultaneous multi-threading (SMT) processor. A bound is placed on the minimum and maximum amount of unconsumed data that has been placed in the cache. By restricting the amount of shared data spilling out of the cache, they show a performance increase. We extend this work by considering heterogeneous CMPs rather than SMT, and longer thread chains than a single producer/consumer pair. In our software model, we also allow buffer sizes to be restricted, minimizing cache spills.

The authors of paper [13] modify the "maximum utilization scheduler" algorithm to work better with processors running at different speeds. They point out that the general problem of minimizing the execution time of dependent tasks on multiple processors is NP-hard.

In [15] they propose a dynamic scheduling algorithm called Self-Adjusting Scheduling for Heterogeneous systems (SASH). They also propose the use of a dedicated scheduling processor so as not to interfere with task execution. The algorithm requires an estimate of the execution time of any task on any processor. Therefore a real implementation needs hardware performance statistics to provide this information for a more effective schedule.

3. METHODOLOGY

After understanding some of the mechanisms in which homogeneous processors wear out and a particular scheduling problem that arises for streaming applications, focus is now shifted to quantifying gains from different scheduling schemes. The following section will describe the analysis we performed.

3.1 CPU Allocation Policies

To show the benefits of intelligent thread scheduling on heterogeneous systems, we compare three different policies, as follows;

- "No wear" - the baseline where all processors run at maximum frequency
- "Static" - allocate the most work intensive thread to the fastest cores, in descending order. The allocation remains for the duration of the program run.
- "Linear" - A dynamic thread migration scheme whereby a linear program calculates the amount of time that each thread should spend on each core to equalize block throughput between threads. A thread migration schedule is then derived using a simple algorithm. This has the effect of maximizing steady-state performance.

As our "No wear" baseline we assume a system of homogeneous processors running at 2 GHz. Threads are statically allocated to cores, so the overall block throughput of the system is restricted by the thread with the longest execution time per block.

The "Static" policy also restricts execution of each thread to a single core. However, the cores run at different frequencies representing a worn-out system. The threads are allocated to cores starting with the most demanding thread on the fastest core. This reduces the amount of idle time where threads are waiting for the bottleneck thread to produce or consume data.

The "Linear" policy employs a schedule determined using several constraints and a linear program. The program specifies how long each thread should execute on each core relative to the others. The main constraints equalize the blocks per second metric (block throughput) of each thread. The objective is to obtain maximum blocks per second under this constraint. By equalizing the thread throughput, buffers are neither empty or full most of the time, reducing stalls. For our example, the schedule was derived using the MATLAB linear program solver. For a reasonable number of cores, it is feasible to periodically evaluate the schedule and re-adjust the amount of time threads spend on each core. If the workload is fairly consistent, frequent re-evaluation is not necessary and and can be performed on one of the common cores or a dedicated scheduling core.

The following simple model approximates throughput in our system.

Let each thread T_x require I_x instructions to process a unit of data (block). When running on CPU_y of a Y-core multiprocessor chip with frequency F_y, block throughput for thread x on core y is;

$$Throughput_{x,y} = \frac{IPC * F_y}{I_x} \tag{1}$$

Since we can measure cycles per block (C_x) for the core which a thread is running on, throughput can also be defined by;

$$Throughput_{x,y} = \frac{F_y}{C_x} \tag{2}$$

Assuming that any thread will exhibit the same cycles per block on any core, this can be used as a measure of how a thread will perform. However, parameters such as different IPC, bus to core clock ratios and different cache sizes can break this assumption on a complex heterogeneous system.

Our policy uses a linear programming algorithm to decide how long each thread should run on each core to obtain maximum overall throughput. As an example, if a producer runs on a fast core and completes its work before a slower consumer, the consumer can switch to the fast core to complete its work. This has two advantages;

- Some produced input data will still be present in the producer core's cache
- The consumer finishes earlier because it runs on a faster core for some of the time.

The linear programming problem is defined as follows.

The variables in our problem are the fraction of time each thread should spend on each core. Therefore with X threads and Y cores, there are $X * Y$ variables. These variables are;

$$\{v_{1,1}, v_{1,2}, ..., v_{1,Y}, ..., v_{2,1}, v_{2,2}, ..., v_{X,Y}\} \tag{3}$$

The throughput function which we want to maximize is;

$$v_{1,1} \cdot \frac{F_1}{C_1} + v_{1,2} \cdot \frac{F_2}{C_1} + ... + v_{1,Y} \cdot \frac{F_Y}{C_1} \tag{4}$$

This equation represents the average throughput of thread 1. Later, we specify constraints such that the throughput of every chained thread is equal.

Now we define the constraints which must be met. These fall into three categories;

- Equivalence constraints specifying that every thread's throughput is equal
- Bounding the fraction of time that threads execute on any one core between 0 and 1
- Bounding the fraction of time a thread can be executed between 0 and 1

The throughput constraints are expressed between each possible pair of threads. Therefore a total of $\binom{X}{2}$ constraints are needed, specified as follows.

For each pair of threads T_a and T_b, add a constraint;

$$\begin{aligned} &\left(v_{a,1} \cdot \tfrac{F_1}{C_a} + v_{a,2} \cdot \tfrac{F_2}{C_a} + ... + v_{a,Y} \cdot \tfrac{F_Y}{C_a}\right) - \\ &\left(v_{b,1} \cdot \tfrac{F_1}{C_b} + v_{b,2} \cdot \tfrac{F_2}{C_b} + ... + v_{b,Y} \cdot \tfrac{F_Y}{C_b}\right) = 0 \end{aligned} \tag{5}$$

Another Y constraints are needed to bound the time threads spend on each core. For each thread x, add the following constraint;

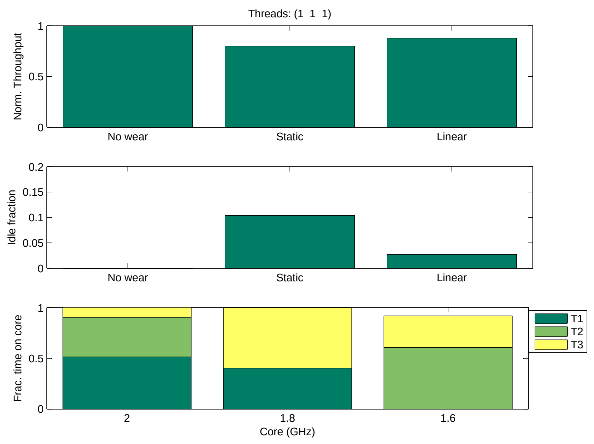
$$0 \leq \sum_{y=1}^{Y} v_{x,y} \leq 1 \tag{6}$$

Finally, to ensure that each thread is busy between 0 and 100% o the time, a constraint is added for each core y;

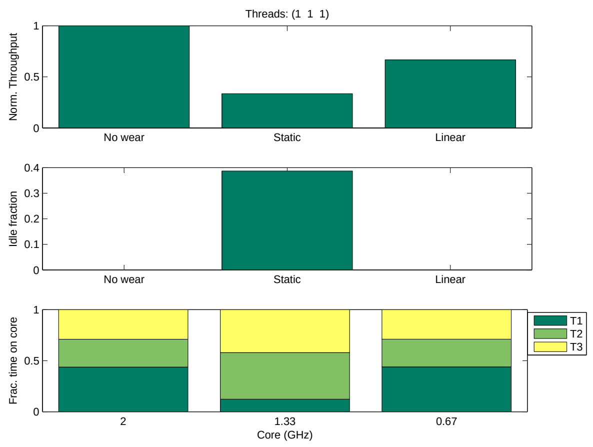
$$0 \leq \sum_{x=1}^{X} v_{x,y} \leq 1 \tag{7}$$

A further step is required to derive a schedule after the linear program provides the amount of time each thread should spend on each core. The simplest approach is to take the first CPU and allocate threads to that core in sequential order. For each successive core, threads are allocated in sequential order to the first time slot where that thread is not already allocated to another core. The linear program constraints ensure that it is possible to derive a schedule in this way, since no thread consumes more than a maximum execution period of 1.0 across any combination of cores.

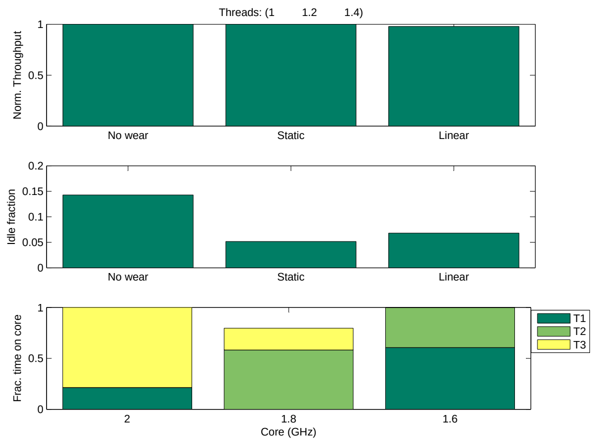
Future scheduling algorithms should look at additional performance metrics from the system that will help to handle changes in cache size and routing latencies. It will also be beneficial to detect sharing patterns and schedule threads that communicate in physically close locations or migrate consuming threads into their producers location to avoid cache warmup penalties. We leave these performance counters and scheduling algorithms for future work.



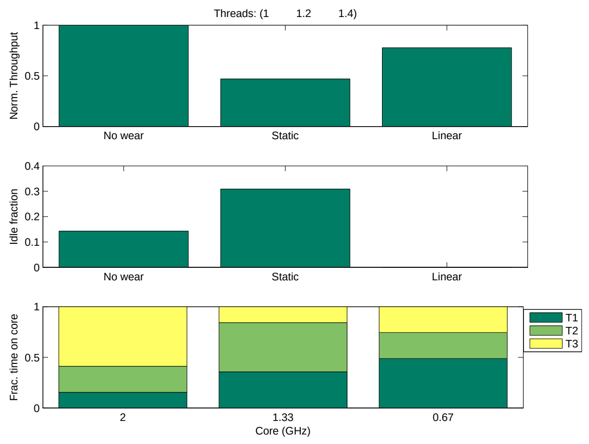
(a)



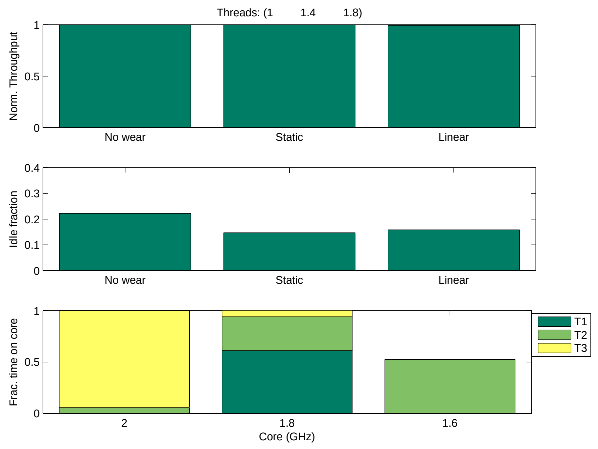
(b)



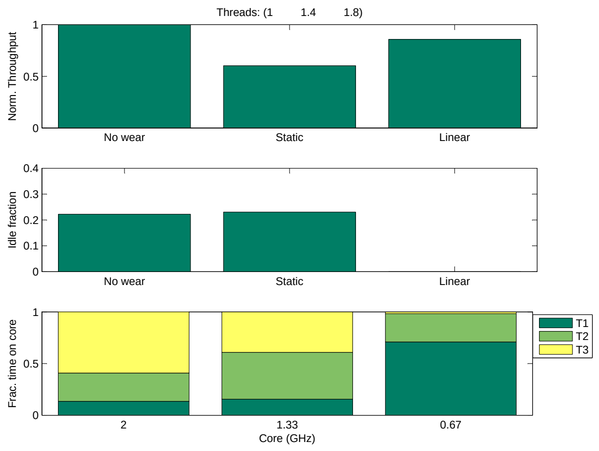
(c)



(d)



(e)



(f)

Figure 5: Data block throughput comparisons for different thread work and CPU frequency ratios

4. RESULTS

In Figure 5 we present a theoretical comparison of the scheduling policies applied to 3 threads running on 3 cores. Figures 5(a) and 5(b) assume all threads require equal work per block (hence the ratio 1:1:1 in the graph title). The first graph shows a moderately worn out system with three cores at clock frequencies 2.0, 1.8 and 1.6 GHz. The first plot on each figure is block throughput normalized to a static thread allocation on a non-worn out system with three cores, all at 2.0 GHz. For low levels of wear-out, the "Static" policy performs relatively well (Figure 5(a)). However at greater levels of cpu imbalance (5(b)) there is significant performance degradation. In all cases, a static allocation that does not assign the most demanding threads to the fastest cores would be even slower. The optimal solution found using the "Linear" policy recovers some of the performance lost due to wear-out, providing a more graceful degradation in performance. The second plot on each figure indicates the average time spent idle across all three cores. With a balanced thread workload, there is obviously no idle time in the "No wear" case. However, when core frequencies become imbalanced, faster cores are forced to become idle while waiting for the slowest thread to complete (Figures 5(c) and 5(e)). The static allocation results in a significant amount of idle time due to thread and core frequency mismatch. Periodically migrating the threads between cores (the "Linear" policy) makes use of available idle time to increase throughput. As graphs move down the page the thread workload becomes more imbalanced.

The final plot gives the output of the MATLAB linear program solver as the fraction of time each thread spends on each core. The allocation ensures that block throughput of each thread is equalized while at the same time maximizing blocks per second. Sometimes it is impossible to eliminate all of the idle time (Figure 5(e) for example) because the most demanding thread (T3) spends almost all the time on the fastest core and becomes the bottleneck. The other threads are significantly less demanding and easily match T3's throughput on the slightly slower cores leaving some slack time on core 3. The most benefits are seen when there is greater frequency deviation between cores.

5. CONCLUSION AND FUTURE WORK

In this paper we have set the scene for operating system based management of wear-out effects. We primarily addressed performance optimization considering only processor elements. A wealth of future work exists in producing a coordinated scheme to manage the interacting effects of increasing cache and memory latency and decreasing storage capacity.

Our investigation into scheduling policies for a streaming benchmark indicates that making the operating system aware of thread throughput can recover a large fraction of performance lost due to wear-out. It is therefore worthwhile to investigate systems with more cores. Increasing numbers of processors and the use of NoCs implies greater latency between physically distant cores. Information on streaming data patterns can be used by an enhanced scheduler to minimize cost by placing communicating threads close to each other. Since the software itself follows a streaming model, the communication patterns are known in advance by the operating system, which may otherwise have to be discovered by hardware.

We conclude that multiple, accurate hardware performance statistics are required to get good throughput on a worn-out multi-core chip.

6. ACKNOWLEDGMENTS

We gratefully acknowledge Greg Chen and Denny Vandenberg for discussions on SRAM failure and linear programming.

7. REFERENCES

[1] Keith Bowman, Alaa Alameldeen, Srikanth Srinivasan, and Chris Wilkerson. Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors. *To appear in ISLPED*, 2007.

[2] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. A self-tuning dvs processor using delay-error detection and correction. *IEEE Journal of Solid-State Circuits*, 41:792–804, 2006.

[3] David Roberts, Nam Sung Kim, and Trevor Mudge. On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. *To appear in 10th EuroMicro Conference on Digital System Design (DSD)*, 2007.

[4] Dennis Sylvester, David Blaauw, and Eric Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design and Test of Computers*, June 2006.

[5] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman Jouppi, and James Smith. Configurable isolation: Building high availability systems with commodity multi-core processors. *ISCA*, June 2007.

[6] Giovanni de Micheli and Luca Benini. Networks on chips. 2006.

[7] E. Bolotin, Z. Guz, I. Cidon, R. Ginosar, and A. Kolodny. The power of priority: Noc based distributed cache coherency. *NOCS*, 2007.

[8] Frederic Petrot, Alain Greiner, and Pascal Gomez. On cache coherency and memory consistency issues in noc based shared memory multiprocessor soc architectures. *Digital System Design (DSD)*, 2006.

[9] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky. Bulletproof: A defect-tolerant cmp switch architecture. *International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[10] Taeho Kgil and Trevor Mudge. Flashcache: a nand flash memory file cache for low power web servers. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[11] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *CF*, May 2006.

[12] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating amdahl's law through epi throttling. *International Symposium on Computer Architecture (ISCA)*, 2005.

[13] Michael Bender and Michael Rabin. Scheduling cilk multithreaded parallel programs on processors of

different speeds. *ACM Symposium on Parallel Algorithms and Architectures*, 2000.

[14] Fabian Chudak and David Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Proceedings of teh eigth annual ACM-SIAM symposium on discrete algorithms*, 1997.

[15] Babak Hamidzadeh, David Lilja, and Yacine Atif. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7, 1995.

[16] Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, November 2005.

[17] J. W. McPherson. Reliability challenges for 45nm and beyond. *DAC*, July 2006.

[18] Bipul Paul et al. Impact of nbti on the temporal performance degradation of digital circuits. *IEEE Electron Device Letters*, 26(8), August 2005.

[19] Yu Wang et al. Temperature-aware nbti modeling and the impact of input vector control on performance degradation. *DATE*, 2007.

[20] J. H. Stathis, R. Rodriguez, and B. P. Linder. Circuit implications of gate oxide breakdown. *Microelectronics Reliability*, 43:1193.

[21] B. P. Linder, S. Lombardo, J. H. Stathis, A. Vayshenker, and D. J. Frank. Voltage dependence of hard breakdown growth and the reliability implication in thin dielectrics. *IEEE Electron Device Letters*, 23(11), November 2002.

[22] Jason Blome, Shuguang Feng, Shantanu Gupta, and Scott Mahlke. Online timing analysis for wearout detection. *Second Workshop on Architectural Reliability (WAR)*, December 2006.

[23] Jared Smolens, Brian Gold, James Hoe, Babak Falsafi, and Ken Mai. Detecting emerging wearout faults. *Third Workshop on Silicon Errors in Logic*, April 2007.

[24] J. Chang et al. The 65nm 16mb on-die l3 cache for a dual core multi-threaded xeon processor. *Symposium on VLSI Circuits*, 2006.

[25] M. Agostinelli et al. Erratic fluctuations of sram cache vmin at the 90nm process technology node. *IEEE Electron Devices Meeting (IEDM)*, December 2005.

[26] Gregory Chen, David Blaauw, Trevor Mudge, Dennis Sylvester, and Nam Sung Kim. Yield-driven near-threshold sram design. *International Conference on Computer Aided Design*, November 2007.

[27] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91, 2003.

[28] Andrea Chimenton, Paolo Pellati, and Piero Olivo. Analysis of erratic bits in flash memories. *IEEE Transactions on Device and Materials Reliability*, 1(4), December 2001.

[29] B. K. Liew, N. W. Cheung, and C. Hu. Projecting interconnect electromigration lifetime for arbitrary current waveforms. *IEEE Transactions on Electron Devices*, 37(5), May 1990.

[30] William Thies et al. Language and compiler design for streaming applications. *Parallel and Distributed Processing Symposium*, 2004.

[31] Srinivas Vadlamani and Stephen Jenks. Architectural considerations for efficient software execution on parallel microprocessors. *Parallel and Distributed Processing Symposium*, March 2007.

[32] Srinivas Vadlamani and Stephen Jenks. The synchronized pipelined parallelism model. *16th IASTED International Conference on Parallel and Distributed Computing Systems*, 2004.