To appear in the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)

Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System

Michael D. Powell Mohamed Gomaa T. N. Vijaykumar

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
{mdpowell, gomaa, vijay}@purdue.edu

ABSTRACT

Power density in high-performance processors continues to increase with technology generations as scaling of current, clock speed, and device density outpaces the downscaling of supply voltage and thermal ability of packages to dissipate heat. Power density is characterized by localized chip hot spots that can reach critical temperatures and cause failure. Previous architectural approaches to power density have used global clock gating, fetch toggling, dynamic frequency scaling, or resource duplication to either prevent heating or relieve overheated resources in a superscalar processor. Previous approaches also evaluate design technologies where power density is not a major problem and most applications do not overheat the processor. Future processors, however, are likely to be chip multiprocessors (CMPs) with simultaneously-multithreaded (SMT) cores. SMT CMPs pose unique challenges and opportunities for power density.

SMT and CMP increase throughput and thus on-chip heat, but also provide natural granularities for managing power-density. This paper is the first work to leverage SMT and CMP to address power density. We propose heat-and-run SMT thread assignment to increase processor-resource utilization before cooling becomes necessary by co-scheduling threads that use complementary resources. We propose heat-and-run CMP thread migration to migrate threads away from overheated cores and assign them to free SMT contexts on alternate cores, leveraging availability of SMT contexts on alternate CMP cores to maintain throughput while allowing overheated cores to cool. We show that our proposal has an average of 9% and up to 34% higher throughput than a previous superscalar technique running the same number of threads.

Categories and Subject Descriptors

C.4.6 [**Performance of Systems**]: Reliability, Availability, and Serviceability

General Terms

Performance, Reliability

Keywords

Power density, heat, CMP, SMT, migration

1 INTRODUCTION

Power-density problems in high-performance microprocessors refer to power, and therefore heat, concentrating in "hot spots" of highly-active microprocessor resources, such as ALUs or register files. These localized hot spots can reach a critical temperature regardless of average or peak external package temperature or chip power; therefore techniques designed to alleviate those problems are ineffective at reducing the temperature of chip hot spots. Such hot spots can lead to circuit malfunction or failure, reducing reliability. Power density continues to increase with technology generations as scaling of current, clock speed, and device density outpaces the downscaling of supply voltages and thermal ability of packages to dissipate heat [6]. Exotic technologies such as heat pipes, liquid cooling, and immersion [16] can improve the packages, but these techniques are expensive and do not scale with technology.

Two types of techniques, temporal or spatial, can manage power density within a processor. Temporal solutions either slow down computation through frequency and voltage scaling [9] or stop computation [4] for a period of time, allowing existing heat to dissipate, and then resume at full speed. This *stop-go* utilizes the resource at some fraction of its peak capacity, called the *duty cycle*. A high duty cycle means a large amount of computation per unit cooling time and implies low performance degradation. Spatial solutions reduce heat by moving computation in a hot resource to an alternate resource (e.g., a spare ALU). Spatial solutions require the presence of redundant or under-utilized resources, or *spatial slack*, to allow cooling without delaying computation.

Technology trends indicate that future processors will employ simultaneous multithreading (SMT) [15] and chip multiprocessors (CMP). SMT worsens power density because SMT increases processor-resource utilization to achieve high instruction throughput, reducing *intra-core* spatial slack and duty cycle, compared to a superscalar. CMPs worsen power density by placing more cores on the same die area that previously held one core. However, CMPs also provide a natural granularity for *inter-core* spatial slack so heat-producing computation can be migrated away from hot cores, reducing or eliminating the need to stop execution while a core cools.

Previous work has evaluated power density in a single-thread, single-core environment [1, 8, 9, 5] but does not consider the challenges and opportunities posed by SMTs and CMPs. [1] and [8, 9] tackle the power density problem for technologies where duty

cycles are above 97% for most applications and incur minimal performance impact using stop-go and voltage scaling. Unfortunately, even single-threaded processors built with future scaled technologies are predicted to approach power-densities of a nuclear reactor and have already surpassed that of a hotplate [6]. This trend combined with the above-mentioned lack of spatial slack will inevitably cause lower duty cycles (e.g., 60%) for CMP of SMTs. As such, stop-go will incur large performance degradation if as much as 30% to 40% of the time is stopped. Apart from these challenges, using spare cores is preferable to adding spare resources, such as register files [9], ALUs [5] or even issue queues [5], to superscalar for two reasons: (1) Adding spare resources (especially critical resources like register file or issue queue) solely for power-density purposes is unattractive due to worsened design and wiring complexity and increased area. (2) CMP cores, unlike spare resources, can be used to run additional threads for workloads where power density is less problematic or non-problematic.

We propose to leverage SMT and CMP for the first time to manage power density in a CMP of SMTs. We propose *heat-and-run* which uses the OS and hardware to control power density. Heat-and-run has two key components: SMT thread assignment and CMP thread migration. Heat-and-run thread assignment (HRTA) is based on the key observation that an entire core must stop execution even if *any one* essential resource (e.g., register file, issue queue) reaches critical temperature; and that cooling time does not increase much if more resources are hot (lateral heat transfer among resources is much less than vertical heat transfer away from the die [8]). Therefore, throughput can be increased if thread assignment to a core in a CMP of SMTs is done such that several resources, instead of just one, are heated to the critical temperature, and the cooling time is made more effective by allowing several resources to cool simultaneously. Thus, HRTA better utilizes the inevitable cooling time by a counterintuitive policy of *maximizing* heat generation across resources in a processor. Hence the first part of the name heat-and-run. HRTA uses the OS to assign threads to cores in a CMP of SMTs such that the threads heat complementary resources on each core and increase the amount of computation per unit cooling time. HRTA is different from symbiotic jobscheduling in the number of threads considered in thread assignment [10] and in time granularity of assignment decisions [11], as we explain later.

When an SMT core's resource reaches a critical temperature, we employ heat-and-run thread migration (HRTM) to use the OS to migrate heat away from that core and allow cooling. Hence the second part of the name heat-and-run. If there were fewer threads than CMP cores, then this migration would be trivial. Similarly, if the CMP lacked SMT, it would be trivial to add SMT and create great amounts of spatial slack. We, however, assume that the base processor already exploits SMT (but not HRTA) and has more threads than cores. When many, but not all, SMT contexts on a chip are occupied, HRTM allows threads from an overheated core to continue running by exploiting inter-core spatial slack and migrating the threads to available contexts on other non-heated SMT cores. Of course, there is no spatial slack if the number of threads running on an SMT CMP is equal to the maximum number of contexts per core times the number of cores, or *maxout* thread count. However, we show that running maxout threads without HRTM performs worse than fewer threads with HRTM. When choosing a core to migrate to, HRTM uses HRTA to match *separately* each thread from the overheated core to that non-heated core whose current threads' heat generation complements that of the incoming thread. Thus, HRTM balances heat-generation across cores to achieve high throughput. While HRTA maximizes heat generation by spreading it across *resources* within a core, HRTM maximizes heat generation by spreading it across *cores* in a CMP. By distributing threads, and thus heat, across all non-overheated cores, HRTM aims to achieve a high duty cycle, and thus high throughput.

The key contributions of this paper are:

- We propose heat-and-run thread assignment (HRTA), which distributes threads among the SMT cores of a CMP to maximize heat generation in each core and increases per-core computation per unit cooling time.
- We propose heat-and-run thread migration (HRTM), which migrates an overheated core's threads to other non-heated cores to balance and maximize heat generation across cores and increases overall CMP throughput.
- SMT aggravates power-density problems for future designs by increasing heat within a core, reducing duty cycle compared to single-threaded runs by as much as 30% to 50%. Applying previous techniques such as stop-go while running maxout number of threads hurts instruction throughput for many applications.
- While running *fewer* than maxout threads to allow for spatial slack, HRTA and HRTM achieve *better* throughput in an SMT CMP. Using a subset of SPEC2000 benchmarks and running 5 threads on a 4-core SMT CMP, we show that HRTA and HRTM achieve an average of 9% and up to 34% higher instruction throughput than stop-go and an average of 6% and up to 27% higher instruction throughput than dynamic voltage scaling, when all the techniques run the same number of threads.

The rest of this paper is organized as follows. In Section 2, we discuss the power density problem in microprocessors. In Section 3, we discuss HRTA and HRTM. Section 4 discusses experimental methodology and Section 5 results. In Section 6 we discuss related work, and we conclude in Section 7.

2 MICROPROCESSOR POWER DENSITY

In this section, we discuss background for the power-density problem in microprocessors. First we briefly discuss on-chip heat sources and dissipation. The details of on-chip heat generation and dissipation are covered in [9] and are covered only briefly here as background for our techniques. Then we discuss spatial and temporal granularity of power density.

2.1 Heat generation and removal

In this section, we discuss the dissipation of heat and how inadequacies in heat removal create the power density problem. We describe the situation when heavy use of an individual resource causes heat production to exceed the ability of the package to remove heat, creating a hot spot and possibly a reliability problem.

Energy is dissipated and heat is produced by circuit activity within microprocessor resources. (The granularity of "resource" is discussed in the next subsection; for now resource is generic.) Fundamentally, if, over long time periods, heat is not moved away from a resource at an equal or greater rate than it is produced, temperature of the resource increases.

This process of heat dissipation can be modeled similar to an RC electrical circuit with temperature in Kelvin (K) analogous to

voltage as detailed in [8, 9]. Elements that conduct heat from one point to another are modeled as thermal resistances (units of K/W); a higher resistance indicates a worse conductor of heat. Elements that store heat are modeled as thermal capacitors (units of J/K); a large thermal capacitance stores heat energy with small temperature change in the same way that a large electrical capacitor stores large charge with small voltage. Thermal circuits also exhibit an exponential time constant equal to the value of RC. For the rest of this section, resistance and capacitance refer to thermal, not electrical values.

Heat generated within a microprocessor resource may stay put, dissipate through lateral resistance to an adjacent area of the chip, or dissipate through vertical resistance away from the chip. Because package designers want heat to move away from the chip instead of laterally within the chip, low resistance packaging materials, thermal grease, and large heat sinks are placed between the on-die resources and the ambient air to lower vertical resistance. Active components such as fans are used to increase heat transfer (lower thermal resistance) between heat sinks and the ambient air.

Because physically large components such as heat sinks have capacitances and time constants orders of magnitude larger than those of individual processor resources (seconds versus tens to hundreds of microseconds), their temperature changes slowly compared to that of individual resources. Heat can dissipate from the individual resources only as fast as allowed by resistance between the resource and the rest of the package, and the small capacitance of individual resources means a relatively small amount of energy (compared to the whole chip) can cause large temperature changes. Therefore, even a heat sink at a safe temperature dissipating heat at an adequate rate for processor as a whole can allow individual resources, with individually small thermal capacitances, to overheat dangerously.

Various technologies exist or have been proposed to reduce thermal resistance and improve heat flow away from the chip, which would increase duty cycle. These include high-airflow designs, liquid cooling, or even phase-change cooling (i.e., boiling a coolant to remove heat) [16]. Low-resistance technologies, such as heat pipes, can help move heat away from individual processor resources [16]. However, these techniques have several limitations. 1) Their effectiveness is limited by physical thermal characteristics which do not scale or improve with technology generations, while power and heat continue to grow with Moore's law. 2) The thermal characteristics dictate the physical size of the heat-removal system to achieve adequately low resistance. Volumes on the order of one hundred cubic inches may be necessary to achieve lower resistances with air-cooled systems [16], which may be prohibitive for servers and workstations, let alone mobile devices. 3) Exotic technologies which do not require such large volumes, such as liquid cooling, are expensive and complex to implement [16].

2.2 Spatial Granularity

Power density can be sensed over a spectrum of spatial resource granularities ranging from an entire chip down to one transistor. Granularity is limited by conceptual as well as practical sensor limitations and affects ability to react to power-density events.

Conceptually, if resource granularity is too coarse, opportunity for adaptation may be lost. For example, if only chipwide power-density is monitored on a CMP, then all processor cores require action if the chipwide temperature is too high. Increasing granularity to the core level allows other cores to run unaffected if one overheats. Increasing granularity to the functional-unit or pipeline-stage level, however, may not be beneficial if the entire core requires action when only a single resource overheats.

Granularity is also limited by ability to place thermal sensors. From a thermal monitoring standpoint, fine granularity allows greater tolerance for heat. For example, if only one sensor may be placed on a chip, the trigger temperature must be set low enough to detect the small amount of heat transmitted by a single overheated functional unit to the large, chipwide thermal capacitance. In contrast, fine-grained sensors on smaller thermal capacitances (e.g., near cores or functional units) may trigger at higher temperatures because they can detect localized hot spots.

2.3 Temporal Granularity

We characterize power density in terms of temporal resource utilization at processor-core granularity because cores are a natural granularity for managing power density in CMPs. If a core running an application generates heat faster than heat is dissipated, then it can slow the rate of heat generation for that core by using a duty cycle less than 100%. The duty cycle, defined in Section 1, is a characteristic of both the processor and the application(s) executed. In this section we discuss duty cycles and temporal granularity,

We use the duty cycle and the *operating period*, which is the sum of the heating and cooling times within a duty cycle (or the time between initiation of cooling intervals), to characterize temporal granularity. For example, the slowing of heat generation can be accomplished by running the processor core at full capacity until it approaches a critical temperature and then stopping until it has adequately cooled as in global clock gating [4]. With this coarse granularity, the duty cycle is the fraction of time spent in operation, and the operating period is the maximum allowed without overheating. The operating period can be shortened by subdividing the stop and go periods, eliminating long pauses and achieving finer temporal granularity. However, the overall duty cycle and net throughput are the same.

The finest temporal granularity is to slow the clock frequency enough to delay the heat generation as with dynamic frequency scaling (DFS). In that case, the operating period is one cycle, and the duty cycle is the fraction of the original clock frequency. If all other external characteristics were equal (e.g., memory behavior), DFS will achieve the same throughput as global clock gating for the same duty cycle.

It is important to note that changing temporal granularity though short operating periods or applying DFS does *not* alter the stop-go duty cycle for a given application running on a core. The duty cycle is based on equalizing the rate of heat generation and dissipation, and the spectrum of techniques from stop-go to DFS temporally spread the heat generation but do not fundamentally change the amount of heat generated. The techniques also do not increase resource utilization or change how many resources are heated before the core must be cooled. In the next section, we describe how HRTA allows additional resources to be utilized in an SMT before cooling is necessary and HRTM exploits both core-level spatial granularity and SMT to increase throughput.

3 LEVERAGING SMT AND CMP

In this section, we explain how heat-and-run leverages SMT and CMP to manage power density. First we qualitatively explain the concepts behind heat-and-run thread assignment (HRTA) and heat-and-run thread migration (HRTM). Then we give an analytical example of how our techniques can improve throughput over

stop-go techniques. Finally, we explain implementation details of HRTA and HRTM.

3.1 HRTA and HRTM concepts

3.1.1 HRTA

Rather than raising duty cycle, HRTA aims to increase utilization within an existing duty cycle by leveraging SMT to run more threads on a core, utilizing and heating more resources. HRTA determines which threads are co-scheduled on individual SMT cores on an SMT CMP. Similar to how SMT can improve throughput over single-thread by using more pipeline resources, SMT can heat more pipeline resources.

Ideally, HRTA would co-schedule threads using complementary resources such that the core would heat and cool at the same rate as the hottest resource under single thread (i.e., the heat increases without reducing the core duty cycle compared to that of a single thread). In reality, there is some reduction for two reasons. First, heating additional resources causes the entire core to heat faster because there is more heat to remove and because of lateral conductance of heat between adjacent resources. However lateral thermal resistance between adjacent resources, while not so high it can be ignored, is large compared to vertical thermal resistance [8]. Because vertical heat conduction away from the core dominates, lateral heat conduction does not drastically reduce duty cycle.

The second reason is more serious; co-scheduled threads compete for certain resources, such as the integer register file and issue queue and data cache, and may cause those resources to heat more quickly, reducing the duty cycle. This competition may not be a concern for certain resources. For example, if the issue queue is already waking up and selecting a near maximum number of instructions per cycle with a single-thread, running multiple threads will not cause it to heat any faster. However, this competition can cause execution resources such as ALUs to heat quickly. Avoiding this increased heating or offsetting it with increased throughput is a key component of HRTA.

HRTA employs several strategies to avoid large reductions in duty cycle or offset reductions with increased throughput. All strategies aim to co-schedule complementary threads that will not stress the same resources. When evaluating resource utilization for an application, we consider metrics based on the *execute* IPC, as opposed to the *commit* IPC, of both individual resources and the entire application. Execute IPC includes misspeculated instructions, which generate no less heat than instructions that commit, and can easily be determined through run-time hardware profiling. For example, an application may have an overall commit IPC of 2.0 and a d-cache commit IPC of 0.5, but due to misspeculation, the execute IPC of the application might be 4.0 with a d-cache execute IPC of 1.0. For the remainder of this section, IPC refers specifically to execute IPC unless stated otherwise.

The first and most obvious strategy is to co-schedule integer and floating-point threads, which correspondingly utilize complementary issue queues, register files, and execution resources. (Of course, floating-point programs still have many integer instructions for control flow and address calculations but not as many as an integer program). However, this strategy alone is inadequate for two reasons. 1) There may be no floating-point (or integer) threads available. 2) Pairing high-IPC floating-point and integer applications may stress shared resources (e.g., d-cache). Our other strategies aim to remedy these problems.

Our second strategy is to co-schedule high-IPC applications with low-IPC applications. Low IPC applications are likely to be cache-miss bound, and therefore place high stress on the d-cache but minimal stress on other execution resources (Recall that because we are using execute IPC, a non-memory-bound application with high misspeculation and low commit IPC would be considered *high* IPC). Pairing a low-IPC application with a high-IPC application allows both to maintain high throughput while resource heating occurs at a similar rate to the high-IPC application alone.

Our third strategy is to evenly distribute the IPC of all threads across available SMT cores to avoid one core from heating substantially faster than others. Uneven heating rates creates a low duty cycle for the hot cores. Even if two high-IPC threads could be reasonably co-scheduled (e.g., one is integer and the other is floating-point), it would not make sense to have simultaneously another core executing only low IPC threads.

Our fourth strategy is to co-schedule applications based on the IPC of specific resources. Co-scheduled applications should not heavily use the same resource. This strategy is necessary when high (or low) IPC applications are to be scheduled together, and classification as integer and floating point is insufficient. For example, one high-IPC application may heavily use the d-cache and multipliers, while another heavily uses the ALUs. Co-scheduling these applications makes sense. In reality, this fourth strategy supersedes the first strategy because it will automatically co-schedule integer and floating-point applications.

3.1.2 HRTM

HRTM exploits spatial slack in an SMT CMP core by migrating threads away from an overheated core and using HRTA to match separately each thread to a non-heated core with a complementary workload. Cores are a natural level of spatial granularity because while the heating of intra-core resources can be sensed, but the overheating of key core resources (i.e., issue queue) requires that the entire core stop and cool. Alternate cores are far enough away to be thermally unaffected by a hot core. The key parameter of HRTM is the frequency of migration. Frequency of migration is determined by the operating period (defined in Section 2.3).

Because migrating threads away from an overheating core incurs some overhead (discussed in detail in Section 3.3), we wish to migrate as infrequently as possible. Infrequent migration implies a long operating period, because migration is necessary at the end of each period. Recall from Section 2.3 that any duty cycle can be achieved using short or long operating periods. To achieve the longest operating period, we wish to heat the microprocessor to the critical temperature and then allow it to cool, rather than heating and cooling in short spurts.

The remaining question is how long to allow cooling before reassigning threads to a processor core. Too short a cooling time will not increase duty cycle because the core will reheat quickly. In addition, quick reheating of the core shortens operating period. Too long a cooling period decreases duty cycle, and the core cools less quickly as it approaches the temperature of the adjacent heat spreader and heat sink according to the exponential decay of the thermal RC time constant. Fortunately, the RC time constant provides guidance as to the most effective cooling time for an exponential system, and we use that value as the cooling time.

The long operating period HRTM is similar to global clock gating [4] or simple fetch toggling [8] and may seem heavy-handed compared to short-period techniques like DFS or control-theoretic fetch toggling [9]. However, the CMP environment favors long

operating periods because of design considerations such as migration overhead that are not present in superscalar. Techniques such as DFS may also be difficult to implement on a per-core basis within a CMP because multiple frequency (or voltage) domains, one per core, would be necessary on one die. Implementing these techniques on only a chipwide basis would require slowing the entire chip when a single core is overheated.

3.1.3 Difference From Symbiotic Jobscheduling

HRTA is different from symbiotic jobscheduling in the number of threads considered in thread assignment [10] and in time granularity of assignment decisions [11]. Given a set of runnable threads and k SMT contexts, Symbiosis identifies subsets of k threads that are complementary in pipeline resource usage so that co-scheduling them in an SMT achieves high throughput. However, Symbiosis considers only k-thread subsets. If a thread has a power density problem then running it with *fewer* co-scheduled threads may be better. Consequently, HRTA may run a power-density-constrained thread with fewer than k-1 other threads on a k-context SMT. Modifying Symbiosis to consider up to k threads is not easy because doing so would require considering exponentially more schedule combinations.

In another paper [11] Symbiosis is extended to ensure fair CPU usage in an SMT. In [11] a non-symbiotic thread may run alone to ensure that the threads get its fair share. Fairness is ensured by sampling and comparing each thread's instruction throughput alone and co-scheduled with other threads. The schedule that achieves fair shares for the threads involved is chosen to run until the next sampling period. Because running threads alone on an SMT results in underutilization of the pipeline, the sampling phase (e.g., one-two OS quanta) is must be much shorter than the running phase (e.g., tens of quanta). However, the coarse granularity of the run phase implies that the schedule does not change in this long time even if power-density problems arise. In contrast, HRTA's schedule granularity is much finer (e.g., one-tenth to one quantum) so that HRTA goes over several schedules during one run phase. One could consider HRTA to be finer-grained scheduling occurring within [11]'s run phases.

3.2 HRTA and HRTM Analytical Example: 2 cores

In this section, we provide a analytical example of how HRTA and HRTM leverage spatial slack to achieve higher throughput than a stop-go-based technique through SMT.

Our techniques rely heavily on the ability of SMT to achieve high throughput through inter-thread symbiosis and require us to quantify that capacity. Of course, for single-thread throughput, it is better to run fewer threads per core, even on an SMT. When two threads are running, we define the *SMT factor*, α, as the fraction of throughput for a thread compared to that if it were running alone on the same core. (We note that α is related to weighted speedup as defined in [10].) In general, α is different for different threads, but to simplify our example we assume α to be the same for both threads. An α of 1 is ideal and implies that two threads running on the same core each achieve throughput equal to that of running alone on different cores. An α of 0.5 implies that two equally-long (in terms of execution time) threads running together achieve the same throughput as if they were run sequentially alone (i.e., SMT has no benefit). An α less than 0.5 implies SMT hurts throughput.

In our example, we have a CMP with 2 cores and 2 contexts per core. There are 2 threads available to run. Because the maxout number of threads (defined in Section 1 as the number of cores multiplied by the number of contexts per core) is 4, there is spatial slack available. We assume that the duty cycle is a constant d, regardless of if the core is running single thread or SMT, and that d is greater than 0.5. A throughput of 1 is equivalent to uninterrupted execution of a single thread on a single core, and we ignore migration overhead in this example but include it in our experimental evaluation. While this is a simple example, it serves well to illustrate our techniques. The example is illustrated in Figure 1.

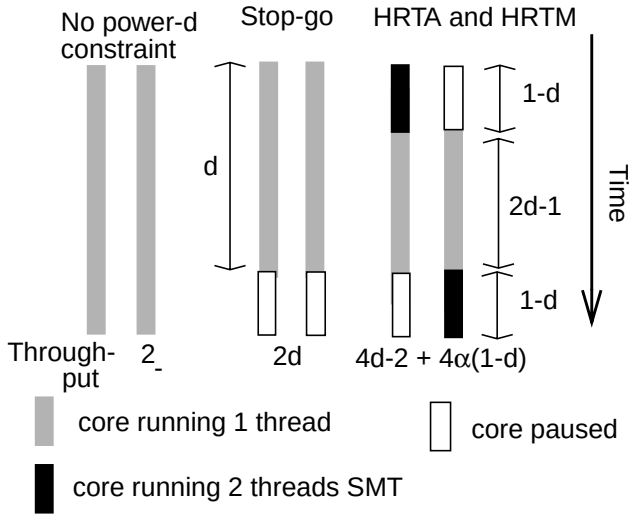

FIGURE 1: Execution profile of one operating period on 2 core CMP running 2 threads.

Using stop-go, throughput is simply the sum of the throughputs for each core or the duty cycles of each core added together.

$$d + d = 2d$$

Using HRTA and HRTM, when one core is paused due to heat, we run its threads on another core. We examine the execution profile of the first core. The second core has a duty cycle of d and is therefore paused at the fraction (1-d) of the time. The first core must run two threads during that time. Of the remaining time when the first core is not paused

$$d - (1 - d) = 2d - 1$$

the core runs a single thread to maximize throughput. Throughput for the core is

$$(2d - 1) + 2\alpha(1 - d)$$

The execution and throughput for the second core is the same; when the first core is paused, the second core runs two threads (see Figure 1). Simplifying the throughput and multiplying by the number of cores gives total throughput of

$$4d - 2 + 4\alpha(1 - d)$$

Now we wish to determine when this throughput is greater than throughput in the stop-go case

$$4d - 2 + 4\alpha(1 - d) > 2d$$

The duty cycles cancel in this inequality, yielding

$$\alpha > 0.5$$

This result is important for two reasons. First, it indicates that as long as α is greater than 0.5, HRTA and HRTM outperform stop-go for our example. Recall that an α is 0.5 or less only if there is zero or negative throughput symbiosis for the SMT threads. For many thread pairings, α will be 0.6 or higher. Second, duty cycle cancels in the equation. In our example, HRTA and HRTM outperform stop-go regardless of duty cycle as long as there is some IPC symbiosis from SMT. For example, with a duty cycle of 80% and α of 0.7, throughput with stop-go is 1.60 while throughput with HRTA and HRTM is 1.76.

We can extend our analysis beyond the simple example. First, we assumed that SMT and single-thread duty cycles are the same. If they are not, duty cycles do not cancel in the equation, and a large decrease in duty cycle will degrade performance unless offset by a high value of α. Second, we assumed the duty cycle is greater than 0.5, which is reasonable for almost all cases without extreme power-density problems. However, with a duty cycle of 0.5 or less, it is optimal to run the threads together all of the time migrating between cores when one overheats and stalling when both are overheated. Total throughput using migration is $2(2\alpha d)$ versus $2d$ using stop-go, making migration better if $\alpha > 0.5$. In the case of a duty cycle of 0.5, exactly one core is idle at all times.

It is important to note the uniqueness of the duty cycle 0.5 for the two-core case. 0.5 is (n-1)/n where n is the number of cores and n = 2. We define (n-1)/n as the *natural duty cycle* for HRTA and HRTM. The natural duty cycle helps to extend our example to multiple cores. For example, with four cores, the natural duty cycle is 0.75. If the duty cycle is equal to the natural duty cycle and the number of threads is less than maxout by the number of contexts on one core, then it is optimal to leave one core idle at all times, rotating the idle core to achieve cooling. If the duty cycle is less than the natural duty cycle (and/or if there are more threads), then there are times when more than one core is idle. If the duty cycle is greater than the natural duty cycle (and/or if there are fewer threads), then there are times when all cores should be active, as in our example.

3.3 HRTA and HRTM Implementation Details

Thread assignment and migration occur though the operating system using guidance from hardware counters and temperature sensors to assign threads. Thread assignment is conventionally performed at the OS level and incurs some overhead. We must enable fast migration to minimize overhead of HRTA and HRTM compared to core operating periods which are determined by heating rates and cooling times and are generally in the range of milliseconds.

Hardware temperature sensors, as discussed in Section 2.2, indicate when a resource on a core has overheated and the core must cool. [9] places thermal sensors on key pipeline components and functional units, such as register files and ALUs, for a superscalar core. We assume the same per-core sensor granularity in our design, but migrate computation away from the entire core when a resource reaches a critical temperature. To reduce migration overhead, we also assume that the sensors trigger a fast trap, which consumes at most a few microseconds.

Upon a fast trap, the OS decides where to assign the threads using the strategies discussed in Section 3.1, which are implemented using hardware counters of execute IPC. Threads from overheated cores are then migrated to their destination by copying register state and assigning their program counter to a free context on the destination core.

Table 1: System parameters.

Architectural Parameters	
Instruction issue	6, out-of-order
L1	64KB 4-way i & d, 2-cycle
L2	2M 8way shared 12-cycle
RUU/LSQ	128/32 entries
Memory ports	2
Off-chip memory latency	150 cycles
CMP and SMT	4 cores, 2 contexts/core
Power Density Parameters	
V*dd*	1.1
Base Frequency	4.0 GHz
Convection resistance	0.8 K/W
Heatsink thickness	6.9 mm
Maximum temp	85 degrees C
Thermal RC cooling time	10 ms for a core

The overhead of our thread migration comes primarily from the fast trap and the transfer of register state between cores. While not completely negligible, this overhead should be on the order of a few microseconds compared an operating period of a few milliseconds. Furthermore, scaling trends favor the reduction of the relative overhead. The migration overhead is based on the wall time required for the fast interrupt and state migration, which shrinks with increasing clock frequency. The operating period, however, is based on heating period and thermal time constant. A dramatic decrease in heating period is unlikely because such low duty cycles would drastically impact performance, and because the thermal time constant does not scale according to Moore's law, as mentioned in Section 2.1.

3.3.1 Optimizations

An additional migration overhead occurs when threads must warm up stateful resources on their destination core, specifically the branch predictor and the caches. Branch prediction state, though not required for correctness, could be transferred along with the register state. This transfer, however, would greatly increase the volume of data transferred and gain little performance improvement. Branch predictors warm up after only a few iterations through the working-set of code, so we do not apply this optimization in our results.

A recently-migrated thread also faces cold (in the sense of state, not temperature) L1 caches. Although the cache-coherence protocol ensures correctness of memory accesses whose data is in the previous core's cache, cache-to-cache transfers from the previous core or cache-to-memory transfers may be slow. (Note that cache-to-cache transfers from an overheated core are not a power-density problem because L1-cache SRAM arrays are too large to become an overheated resource.) The cache warmup time can be mitigated by having an idle core's cache snarf bus traffic from L1 misses of the running cores to keep its own cache warm. (Again, such snarfing is not a power-density problem for the overheated

Table 2: Spec2000 applications with single-thread: stop-go commit IPC, execute IPC, and duty cycle.

	Low Execute IPC (L)					High Execute IPC (H)						
name	*ammp*	*applu*	*apsi*	*art*	*lucas*	*bzip*	*crafty*	*eon*	*equake*	*fma3d*	*galgel (f)*	*gap*
IPC (Int/Fp)	0.06	0.98 (f)	0.94 (f)	0.54 (f)	0.44 (f)	1.19 (i)	1.03 (i)	1.73 (i)	1.28 (f)	1.25 (f)	1.35 (f)	1.44 (i)
ex-IPC	0.06	0.99	0.95	0.66	0.44	3.27	3.58	3.07	4.22	4.90	3.66	2.90
duty cycle	1.00	1.00	1.00	1.00	1.00	0.58	0.53	1.00	0.38	0.46	0.40	0.68
name	*mcf*	*mgrid*	*parser*	*swim*	*vpr*	*gzip*	*mesa*	*perl*	*sixtrack*	*twolf*	*vortex*	
IPC (Int/Fp)	0.20 (i)	1.27 (f)	0.75 (i)	1.00 (f)	0.95 (i)	1.11 (i)	1.85 (i)	1.33 (i)	1.50 (f)	0.90 (i)	1.85 (i)	
ex-IPC	0.42	1.28	1.28	1.00	1.26	3.48	2.88	2.75	3.35	1.96	2.13	
duty cycle	1.00	1.00	1.00	1.00	1.00	0.61	0.79	1.00	0.48	1.00	1.00	

core's cache either.) This snarfing, however, may be unnecessary as even large L1 caches tend to warm up within a million cycles, which is orders of magnitude less than the operating period of cores using HRTA and HRTM.

4 METHODOLOGY

In this section we discuss our simulation environment, design parameters, and benchmarks. Our base simulator is Wattch [2] extended to include code from SimpleScalar 3.0b [3] to execute the Alpha ISA. We extend Wattch to include SMT and CMP capability. The architectural configuration of our simulator is shown in Table 1. Our SMT cores fetch from up to two threads per cycle and use the ICount fetch policy [14]. We implement common SMT optimizations including memory offsetting to reduce cache conflicts between threads and thread squash upon L2 misses to avoid pollution of the issue queue [13]**.** Each core has private L1 caches; the cores share a unified L2. We enable snarfing by idle cores' L1 caches as described in Section 3.3.1. We model a 5 microsecond overhead for each thread HRTM migration between CMP cores to account for the fast trap and state copy.

We use the HotSpot [9] model to extend our Wattch-based simulator for power density, sensing temperature at 100,000 cycle intervals (well under the thermal RC time constant of any resource). Circuit and packaging parameters are also in Table 1. For each CMP core, we use the single-core floorplan provided in [9] and without private L2 cache, assuming the CMP cores are laterally thermally isolated by the cooler shared L2 cache. We use a chipwide V_{dd} of 1.1 V and a clock frequency of 4.0 GHz. The parameters are consistent with estimates for high-performance designs in the next 5 years according to the ITRS [7]. They are substantially more aggressive than those of [9] due to the higher clock frequency and smaller area. Our thermal packaging is also consistent with an air-cooled, high-performance system.

For our simulations, we run multithread groupings of SPEC2000 [12]. Because our default configuration includes 4 cores with 2 contexts each, it is impossible to show all permutations of applications. Therefore we show groupings of high-IPC, low-IPC, and mixed-IPC applications as well as integer and floating-point mixes. We fast-forward each thread two billion instructions to pass initialization code and warm up the caches (cache state, not temperature). We use initial thermal conditions consistent with SMT workloads on our core. Our simulations run until one thread completes 400 million instructions, measuring instruction throughput in instructions per second (IPS). Because we show previous techniques which involve clock-frequency scaling, we need to show throughput in IPS and not instructions per cycle (IPC).

5 RESULTS

We present our experimental results in this section. In Section 5.1, we show that power-density limitations of stop-go techniques on SMT CMPs limit the number of threads to less than maxout. Section 5.2 evaluates policies for HRTA and HRTM and shows throughput compared to stop-go. Section 5.3 compares HRTA and HRTM to superscalar power-density techniques.

5.1 Throughput of stop-go

Adding threads to a power-density-constrained SMT CMP may not improve instruction throughput because the additional threads can cause cores to overheat, as mentioned in Section 1. In this section, we evaluate SMT CMP instruction throughput as we increase the number of threads. We use the stop-go power-density technique as our baseline, which is similar to global clock gating and fetch toggling with coarse temporal granularity, as described in Section 2.3. We do not expect increasing the number of threads beyond the number of processor cores to consistently improve performance across our applications, especially for high execute-IPC (i.e., high heat) application pairings. We expect running maxout threads to aggravate the power density problem and hurt throughput.

Table 2 shows our SPEC 2000 applications along with their instruction throughput (IPC) and duty cycle running as single threads using stop-go. Only nine applications have duty cycles under 100%; the rest do not exhibit power density problems in this configuration. We also show the execute IPC (ex-IPC, as defined in Section 3.1.1, not to be confused with commit IPC) for each application, which includes misspeculated instructions. For applications with duty cycles under 100%, or "hot" applications, ex-IPC reflects only active periods and *does NOT include the stopped time.*

We divide applications into two categories — high and low— based on ex-IPC and will use this categorization throughout the results. As discussed in Section 3.1, high ex-IPC reflects high core activity and can indicate potential power-density problems. All of our hot single-thread applications are in the high-ex-IPC category.

Figure 2 shows instruction throughput for 5 through 8 threads running on our 4-core SMT CMP using stop-go. Applications are paired to form workloads as shown on the x-axis and workloads are grouped by the ex-IPC categories of the two applications. We show the average for each group on the right of each group. We build n-thread workloads from these pairs by replicating them enough times. When applications are co-scheduled on an SMT core, each application is co-scheduled with the other in its pair and not with another copy of itself (e.g., *gap* is paired with *gzip* in the leftmost results). The bars for each workload increase in the num-

ber of threads to the right. The number of threads is shown below the x-axis. Due to space limitations, we show only 14 pairings, but we ran a total of 35 pairings which gave similar overall results. Integer and floating-point mixes are shown by the color of the bars. The duty cycle for the co-scheduled applications on a single core is shown below the bars. Instruction throughput is relative to that with 4-cores running single threads, two with each application from a pair.

For high ex-IPC workloads, adding threads does not improve throughput because of power-density problems created by running threads in SMT. For the high+high workloads, relative average throughput degrades from 0.96 with 5 threads to 0.83 with 8. Average throughput for each workload monotonically degrades as threads are added with only one exception (*crafty+gap*). All workloads except *perl+vortex* (which just perform poorly together) have duty cycles below 100%, and two are below 50%. These low duty cycles compared to single-thread runs are representative of the challenge posed by SMT for power density, as discussed in Section 1.

For mixed workloads, average throughput increases by 2%, 10%, 10%, and 17% for 5, 6, 7, and 8 threads, indicating benefits from SMT in spite of power-density constraints when pairing high and low ex-IPC applications. Three workloads (*applu+equake, eon+parser,* and *mgrid+galgel)*, have duty cycles below 100%, but each of the workloads except *eon+parser* experiences some benefit from adding threads. In addition, these three duty cycles are higher than those experienced by most of the high ex-IPC pairings.

Our low-ex-IPC workloads do not experience power-density problems. Each has a duty cycle of 100%. *mcf+lucas* experiences good SMT symbiosis, although *parser+mcf* does not. Throughput increases an average of 13% for 8 threads, and all duty cycles are 100%. Low-ex-IPC is advantageous for these workloads because it 1) does not conflict with SMT symbiosis and 2) causes fewer power density problems for the SMT configuration, as indicated by the high duty cycles. These advantages make these workloads less interesting from a power-density standpoint.

When workloads from all three groups are considered, as shown at the far right of the figure, there is no substantial benefit from adding threads due to the throughput penalties of stop-go. Averaged over all workloads shown, throughput stays within 3% below the base case as the number of threads increases to the max-out of 8, with degradations as high as 20% to 30% for some of the high-ex-IPC workloads.

Table 3: 4-application groupings.

Label	Applications	Int/FP	Ex-IPC
A	*bzip+fma3d+mesa+art*	IFFF	HHHL
B	*crafty+gzip+apsi+twolf*	IIFI	HHLH
C	*eon+swim+mcf+parser*	IFII	HLLL
D	*galgel+mgrid+twolf+lucas*	FFIF	HLHL
E	*gap+bzip+mcf+eon*	IIII	HHLH
F	*gap+crafty+vpr+gzip*	IIII	HHLH
G	*gzip+lucas+mcf+crafty*	IFII	HLLH
H	*perl+gap+sixtrack+ammp*	IIFF	HHHL
I	*vortex+perl+applu+equake*	IIFF	HHLH

Note that relative throughput does not necessarily monotonically increase/decrease as threads are added. Adding thread X to a core previously running only thread Y has a different effect on *the change* in throughput than adding thread Y to a core previously running only thread X. An example of this behavior is *galgel+lucas*. Adding the fifth thread, *lucas*, to a core running *galgel* reduces throughput compared to 4 threads. However, adding a sixth thread, *galgel*, to a core running *lucas* increases throughput.

5.2 HRTA and HRTM

We have shown that running close to maxout threads does not benefit throughput due to power-density constraints. However, intra-core spatial slack created by running fewer threads is not exploitable by stop-go, which results in reduced throughput. HRTA and HRTM make it possible to leverage the slack using migration. We expect HRTA and HRTM to outperform stop-go, and we expect the best performance from HRTA and HRTM configurations which co-schedule threads using complementary resources and spread heat throughout the chip.

5.2.1 HRTA thread-assignment policy evaluation

HRTA thread assignments affect both HRTA and HRTM.

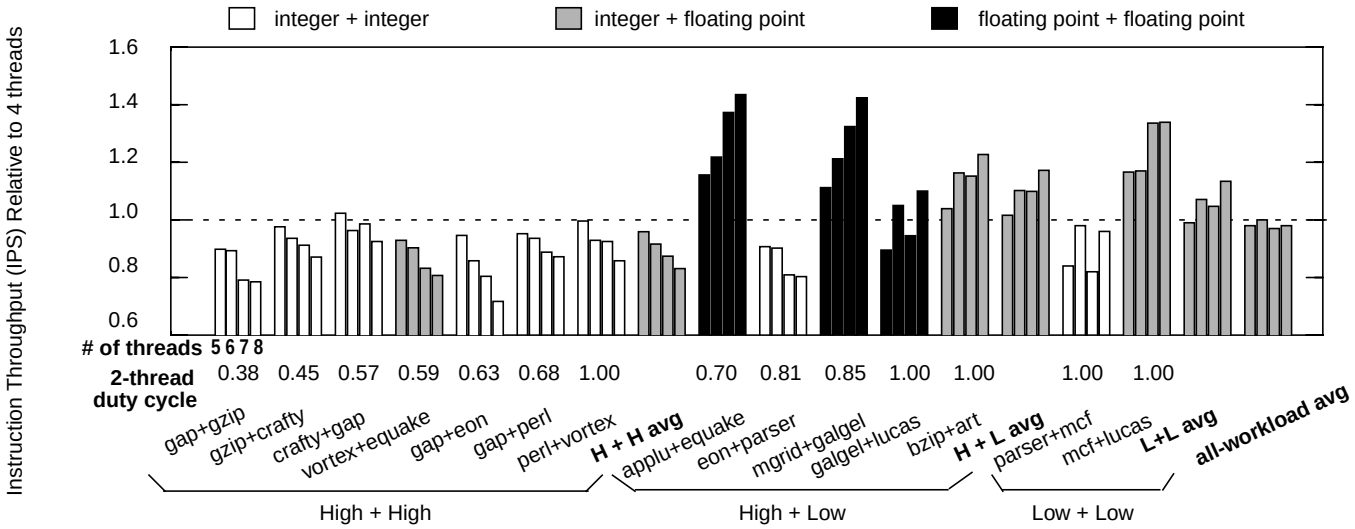

FIGURE 2: Stop-go for 5-8 threads grouped by individual-thread execute-IPC.

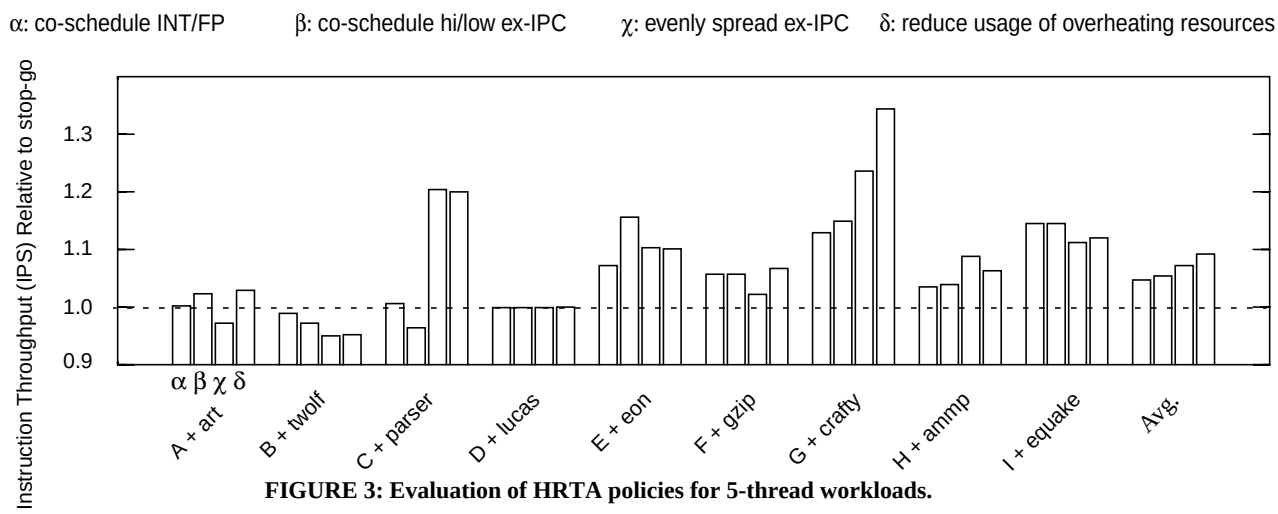

FIGURE 3: Evaluation of HRTA policies for 5-thread workloads.

Thread assignment dictates not only resource utilization for a single core but also directs migration of execution (and heat) among cores.

Figure 3 shows throughput for HRTA and HRTM relative to stop-go running the same threads. The workloads are 5-thread workloads constructed from the 4-thread groupings in Table 3 plus an additional copy of 1 of the applications as shown on the x-axis of the graph. The average over all workloads is shown at the far right. Table 3 also shows the Int/FP composition and ex-IPC category of each workload's components. We also ran 6-thread workloads but found their thread-assignment-policy results to be similar; 6-thread workloads will be shown in Section 5.3.

The four bars for each workload (α–δ) represent different thread assignment policies for migration described in Section 3.1.1: α) co-schedule applications with the most different utilization of integer and floating-point resources, β) co-schedule applications with the most disparate ex-IPC, χ) co-schedule applications to spread the ex-IPC across the chip, generating pairs with a combined ex-IPC near the chipwide per-core average, δ) co-schedule applications with small combined usage of resources prone to overheating. Nearly all overheatings we experience come from the register files, integer issue queue, and floating-point units, so we consider those units for policy δ. These policies are applied to assign threads both when a core overheats (and tries to migrate its threads elsewhere if there is spatial slack on other non-overheated cores) and when a core cools (and migrates threads from other cores). In each case, statistics since the previous migration are considered in the decision.

Co-scheduling based on different integer and floating-point utilization (policy α) experiences the smallest throughput gain over stop-go, 4.7% on average, mainly because it is effective only when appropriate applications are available to migrate. The workloads for which this policy achieves more than 1% performance improvement (E, F, G, H, and I) all have a good mix of integer and floating-point utilization, including workloads E and F, where the integer applications *gap* and *eon* have substantial floating-point utilization.

Co-scheduling based solely on difference in ex-IPC (policy β) has the second smallest throughput gain, 5.4% on average. A problem with this policy is that while it pairs applications with widely different ex-IPCs, it may not effectively spread heat across the cores because such pairings may not result in per-core ex-IPCs near the chipwide per-core average.

Co-scheduling to evenly spread ex-IPC across the cores (policy χ) is more effective at spreading heat across cores while still generating effective thread pairings. (A pairing with a combined ex-IPC near the chipwide per-core average is unlikely to include two high ex-IPC applications.) This policy achieves an average throughput gain of 7.2% over stop-go. However, this policy performs poorly for some workloads (e.g., A and F) where ex-IPC alone does not seem the best assignment policy.

Policy δ co-schedules threads based on the utilization of specific resources that overheat, not the generalized ex-IPC of the thread. This policy pairs threads with a low combined utilization of these strained resources to reduce overheating and maintain high-duty cycles. Policy δ has the best overall throughput, 9.2% higher than stop-go. While it does not outperform policy χ for all workloads, it avoids the poorer performance of policy χ in workloads A and F. We use policy δ for the remainder of our results.

There are two 5-thread workloads for which HRTA and HRTM seem ineffective regardless of policy. Workload D experiences no change in throughput because it experiences no overheating (and thus needs no migration). Workload B contains 4 high-ex-IPC threads, two of which (*crafty* and *gzip*) have low duty cycles when run in isolation, and the workload is unable to find effective pairings using our policies.

5.2.2 Cache Snarfing

All of our HRTA and HRTM results shown include snarfing by idle cores' L1 caches as described in Section 3.3.1. Snarfing aims to avoid cold L1 caches immediately after a migration. Overall, we expect snarfing to have a small effect because of the long interval between migrations (milliseconds), but it may benefit some workloads. For our 5-thread workloads using policy δ, snarfing provides only a 1% average throughput increase but provides substantial gains of 4% and 12% for workloads A and G respectively.

5.3 Comparison to superscalar techniques

Other power-density techniques have been applied to superscalar processors, such as dynamic frequency scaling (DFS) and dynamic voltage scaling (DVS) [1, 8, 9]. In this section, we compare HRTA and HRTM to these techniques applied to an SMT CMP. We expect HRTA and HRTM to outperform these techniques by exploiting spatial slack through migrating threads.

We implement DFS and DVS in our simulator using a PI controller with a gain of 10 and setpoint of 81.8 degrees C, similar to

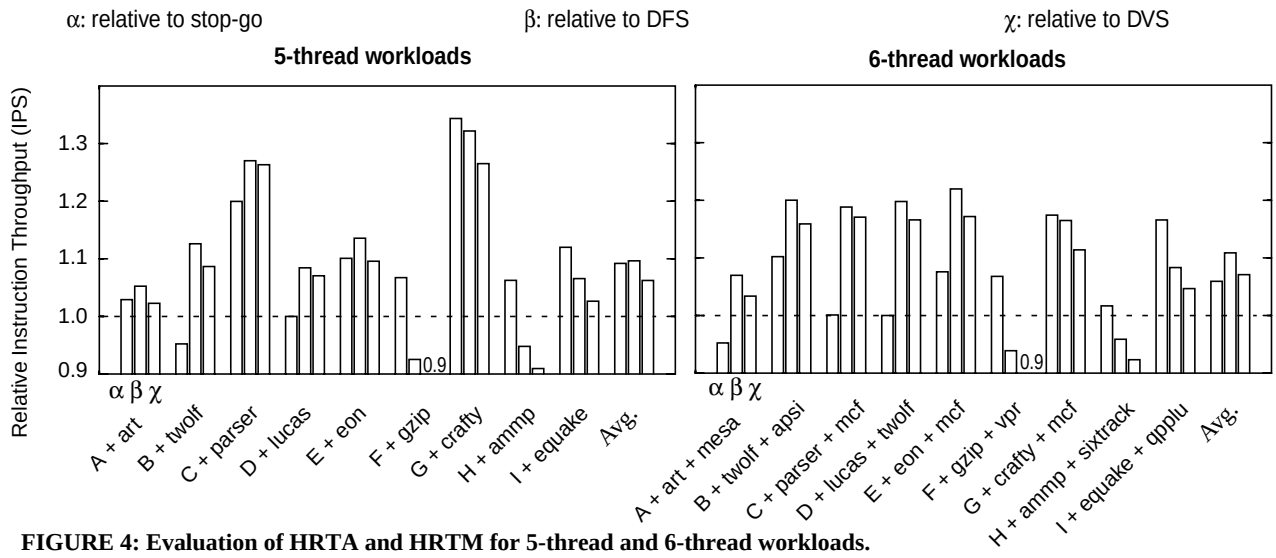

FIGURE 4: Evaluation of HRTA and HRTM for 5-thread and 6-thread workloads.

that in [9]. Each core is voltage and frequency scaled independent of the other cores. We assume the core must stall for 10µs on each voltage and frequency change to stabilize the PLL; this overhead is the same as that modeled in [9]. For both DFS and DVS, we allow 6 evenly-spaced frequency steps between the maximum and half of the maximum clock frequency. For DVS, we scale frequency and also allow corresponding voltage steps between 1.1V (V_{dd}) and 0.95V ($0.86*V_{dd}$).

We do not aggressively scale the voltage for DVS below 0.95V. While voltage scaling provides a desirable quadratic reduction in heat, reducing supply voltage becomes increasingly difficult for scaled, low-voltage technologies because transistor threshold voltage scales more slowly than the supply voltage [7]. As the gap between the supply voltage (e.g., 1.1V) and the threshold voltage (e.g., 0.25 V) closes, there is substantially less flexibility for DVS. Additional supply-voltage reduction may cause soft errors or prevent transistors from switching even at reduced clock frequencies. The ITRS does not predict supply voltages below 0.9 V for high-performance designs in the near term (through 2009) [7].

Figure 4 shows throughput for HRTA and HRTM relative to stop-go (α), DFS (β), and DVS (χ). 5-thread workloads are shown in the left graph, and 6-thread workloads are shown on the right. (The 5-thread results relative to stop-go are the same as the rightmost bar in Figure 3 but are kept for reference.) For 5-thread workloads, HRTA and HRTM outperform stop-go, DFS, and DVS by 9.2%, 9.6% and 6.2% respectively. For 6-thread workloads, HRTA and HRTM outperform stop-go, DFS, and DVS by 5.9%, 10.8%, and 7.1% respectively. Note that for 6 thread workloads, there is less spatial slack and thus less opportunity for HRTA and HRTM compared to stop-go.

Intuitively, DFS and stop-go (which both exploit temporal slack at different granularities) should perform about the same, while DVS, which has the advantage of voltage scaling, should do better. However, there are exceptions. DFS and DVS do not always outperform stop-go because they may scale frequency and voltage prior to a core actually overheating, such as in workload D, which gets warm but never overheats. (Recall from Section 5.2.1 that workload D experiences no migration.) In these cases HRTA and HRTM have a bigger advantage over DFS and DVS than over stop-go.

5.3.1 Other Superscalar Techniques

We do not compare against a number of other techniques which are either not generally applicable or create implementation difficulties. Temperature-Tracking Frequency Scaling (TTDFS), as proposed in [9], allows the processor to heat above its "maximum" temperature by slowing the clock and relaxing timing constraints. As stated in [9] TTDFS is effective only if the sole limitation on power density is circuit timing. TTDFS does not reduce maximum temperature or prevent physical overheating and cannot handle large increases in temperature, which may damage the chip. We do not compare to resource duplication as demonstrated for the register file in [9] or for various pipeline components in [5] because the technique adds substantial complexity within individual cores.

6 RELATED WORK

Several previous proposals address thermal management or power density in superscalars. The authors of [1] evaluate techniques to balance the rate of heat production to heat dissipation at chip-level granularity. [8] proposes toggling techniques and use of PID controllers to manage power density. [9] introduces the HotSpot temperature model used in this paper and proposes PI-controlled DFS/DVS and TTDFS, as discussed in Section 5.3. Finally, [5] proposes "ping-ponging" resource activity for various pipeline resources between duplicates within a superscalar core.

7 CONCLUSIONS

Power density in high-performance processors continues to increase with technology generations as scaling of current, clock speed, and device density outpaces downscaling of supply voltage and the thermal ability of packages to dissipate heat. Future processors are likely to be SMT CMPs, which pose unique challenges and opportunities for power density. SMT and CMP increase throughput and thus on-chip heat, but they also provide natural granularities for managing power-density. We propose *heat-and-run* which uses the OS and hardware to control power density. Heat-and-run has two key components: SMT thread assignment and CMP thread migration. Heat-and-run thread assignment (HRTA) increases processor-resource utilization before cooling becomes necessary by co-scheduling threads that use complementary resources. HRTA aims to avoid large decreases in processor duty cycle and to offset inevitable decreases in duty cycle by main-

taining high throughput. Heat-and-run thread migration (HRTM) migrates threads away from overheated cores and assigns them to free SMT contexts on alternate cores using HRTA. HRTM leverages availability of SMT contexts on alternate CMP cores to maintain throughput while allowing overheated cores to cool.

We show that running the maximum possible number of threads (maxout threads) on a power-density-constrained SMT CMP using existing stop-go techniques degrades or does not improve throughput compared to running threads equal to the number of cores. The extra heat added by running multiple threads substantially reduces duty cycle compared to single-threaded configurations. Stop-go is unable to exploit spatial slack that is available when fewer than maxout threads are run. We show that HRTA and HRTM are able to exploit the spatial slack, improving throughput over stop-go and previous superscalar techniques. For a 4-core CMP running 5 threads, HRTA and HRTM achieve 9% higher average throughput than stop-go and 6% higher average throughput than dynamic voltage scaling.

As power density worsens with technology scaling, and as voltage scaling becomes more difficult due to the shrinking gap between the supply and threshold voltages, alternate techniques like heat-and-run will become more important.

8 ACKNOWLEDGEMENTS

This research is supported in part by NSF under CAREER award 9875960-CCR, NSF Instrumentation grant CCR-9986020 and a Purdue Research Foundation Fellowship.

9 REFERENCES

[1] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Seventh International Symposium on High Performance Computer Architecture (HPCA)*, pages 171–182, Jan. 2001.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[3] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin–Madison, June 1997.

[4] S. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. In *Intel Technology Journal Q1 2001*, Q1 2001.

[5] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 217–222, Aug. 2003.

[6] F. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. In *Keynote speech: 32nd International Symposium on Microarchitecture*, Dec. 1999.

[7] SIA. *International Technology Roadmap for Semiconductors (ITRS)*. http://public.itrs.net/Files/2002Update/2002Update.htm, 2002.

[8] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Eighth International Symposium on High Performance Computer Architecture (HPCA)*, pages 17–28, Feb. 2002.

[9] 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 30)*, pages 2–13, June 2003.

[10] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 234–244, Nov. 2000.

[11] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, June 2002.

[12] The Standard Performance Evaluation Corporation. Spec CPU2000 suite. http://www.specbench.org/osg/cpu2000/.

[13] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO 34)*, pages 318–327, Dec. 2001.

[14] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, June 1996.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[16] R. Viswanath, V. Wakharkar, A. Watwe, and V. Lebonheur. Thermal performance challenges from silicon to systems. In *Intel Technology Journal 3Q 2000*, Q3 2000.