Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors*

Radu Teodorescu and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Abstract

Within-die process variation causes individual cores in a Chip Multiprocessor (CMP) to differ substantially in both static power consumed and maximum frequency supported. In this environment, ignoring variation effects when scheduling applications or when managing power with Dynamic Voltage and Frequency Scaling (DVFS) is suboptimal.

This paper proposes variation-aware algorithms for application scheduling and power management. One such power management algorithm, called *LinOpt*, uses linear programming to find the best voltage and frequency levels for each of the cores in the CMP — maximizing throughput at a given power budget. In a 20-core CMP, the combination of variation-aware application scheduling and *LinOpt* increases the average throughput by 12–17% and reduces the average ED^2 by 30–38% — all relative to using variation-aware scheduling together with a simple extension to Intel's Foxton power management algorithm.

1. Introduction

As integrated-circuit technology continues to scale, process variation — the divergence of process parameters from their nominal specifications — is becoming an issue that cannot be ignored at the architecture and system levels. Indeed, variation has major implications, such as increased leakage power consumption in the chips and limited processor frequency improvements [2].

In the context of Chip Multiprocessors (CMP), within-die process variation in current and near-future technologies causes individual cores in the chip to differ substantially in the amount of power that they consume and in the maximum frequency that they can support. This effect, which has been reported elsewhere [11] and will be confirmed in this paper, suggests that it is no longer accurate to think of large CMPs as homogeneous systems.

In these environments, it is suboptimal to schedule applications ignoring variation effects. Instead, if applications are scheduled in a variation-aware manner, taking into account the different power and frequency characteristics of each individual core, substantial savings in power or large increases in throughput are attainable. Similarly, it is suboptimal to perform power management based on Dynamic Voltage and Frequency Scaling (DVFS) assuming that all cores have the same properties. Instead, if DVFS is applied in a per-core manner, fully aware of the heterogeneity of the cores, substantially more cost-effective working points can be obtained.

Interestingly, the technology for variation-aware application scheduling and power management is now available. Indeed, sophisticated on-chip power monitors and controllers such as those in Intel's Foxton technology [22] can be used to measure and manage the power heterogeneity of the cores. Moreover, the ability to support multiple on-chip frequency domains and change the frequency of each core independently as in AMD's Quad-Core Opteron [6] can be used to exploit the frequency heterogeneity of the cores.

In this paper, we propose simple, variation-aware algorithms for application scheduling in a CMP to save power or improve throughput. Moreover, we complement these algorithms with variation-aware power-management DVFS algorithms to maximize throughput at a given power budget. One such power-management algorithm, called *LinOpt*, uses linear programming to find the best voltage and frequency levels for each of the cores in the CMP. *LinOpt* runs on-line periodically, using profile information provided by the chip manufacturer and by on-chip power and IPC sensors. In a 20-core CMP, the combination of variation-aware application scheduling and *LinOpt* increases the average throughput by 12–17% and reduces the average ED^2 by 30–38% — all relative to using variation-aware scheduling together with a simple extension to Foxton's power management algorithm. Moreover, *LinOpt*'s throughput is within 2% of that of a simulated annealing algorithm, which has a computation time orders of magnitude higher.

This paper is organized as follows. Section 2 describes related work; Section 3 gives background on process variation; Sections 4 and 5 describe the scheduling and power management algorithms; Sections 6 and 7 evaluate them; and Section 8 concludes.

2. Related Work

The contributions of this paper are related to several areas of work, which we consider in turn.

Handling Core-To-Core Process Variation. Humenay *et al.* [11] examine process variation in a CMP and point out the core-to-core variation in frequency. They estimate the maximum difference in core frequencies to be approximately 20%. They suggest Adaptive Body Bias (ABB) and Adaptive Supply Voltage (ASV) to reduce some of this variation — at the cost of increasing power variation. Their work is complementary to ours. Donald and Martonosi [5] also examine process variation in a CMP and focus on the core-to-core variation in power. They suggest turning off cores that consume power in excess of a certain computed value, with the goal of

* This work was supported by the National Science Foundation under grants CCR-0325603 and CNS-0720593, and by SRC GRC under grant 2007-HJ-1592. Radu Teodorescu was supported by an Intel PhD Fellowship.

maximizing the chip-wide performance/power ratio. We combine application scheduling and global DVFS power management.

Scheduling for Heterogeneous Architectures. Balakrishnan *et al.* [1] study how heterogeneous CMPs impact parallel workloads. They suggest fine-granularity threading as a solution for alleviating the performance instability caused by heterogeneous CMPs. Kumar *et al.* [18] propose a CMP with different-complexity cores and the same ISA. The goal is to reduce power consumption by using the simpler, more power-efficient cores to run memory-bound applications. They schedule applications to cores based on application ILP, and adapt to phase changes within an application. Our work considers design-identical cores that are affected by variation. Since we do not know beforehand the power and frequency of each core, this information is obtained post-manufacturing and is unique to each CMP.

Scheduling for Power Management. Li and Martinez [19] optimize a parallel workload running on a CMP by dynamically changing the number of active processors and the voltage and frequency levels at which the CMP runs. They apply DVFS chip-wide rather than independently per core, which reduces the flexibility and impact of the optimization. Kadayif *et al.* [15] enable embedded systems to exploit the heterogeneity of workloads. Specifically, they use the compiler to assign different voltages and frequencies to different processors depending on the characteristics of the workload.

Isci *et al.* [12] consider a 4-core CMP with core-level DVFS (i.e., the voltage and frequency levels are changed independently in each core). They examine different DVFS policies for high performance and for power efficiency. Their solutions are primarily based on the exhaustive search of the solution space. Because of this, the solutions are not scalable to large systems. In this paper, we consider a large CMP with process variation. As a result, both application scheduling and power management have to be variation-aware. Moreover, given the large design space, we need to use an intelligent way to prune the design space.

Herbert and Marculescu [10] examine the tradeoffs of using different DVFS granularities (i.e., the number of cores in the same voltage-frequency domain). They find that having a small number of cores per domain produces the most complexity-effective design.

Other researchers like Heo *et al.* [9] and Stavrou and Trancoso [34] minimize power density or temperature hot spots by judiciously scheduling jobs or migrating them from core to core.

In embedded systems, linear programming has been used to solve the problem of scheduling tasks on CMPs [33]. The goal is to minimize power while meeting strict timing constraints. Our optimization approach is different, since we do not have timing constrains. Moreover, due to variation, our cores are heterogeneous.

Core-Level DVFS and Power Controllers. State-of-the-art processor chips often have multiple voltages — for example, giving memory arrays a slightly higher voltage than the core for reliability reasons. However, the first general-purpose CMP to support a form of core-level DVFS is the AMD Quad-Core Opteron [6]. In this chip, the frequency of each core can be set independently, although all cores have the same voltage. Currently, multiple on-chip voltages are provided by off-chip voltage regulators, which are bulky and costly. Recent work by Kim *et al.* [16] describes designs of on-chip regulators. They are able to perform voltage changes in nanoseconds rather than in microseconds, and consume little energy. Designs similar to these will enable wide use of core-level DVFS.

Given the importance of power, power management control is an area of high interest. Perhaps the most sophisticated design is Intel's Foxton technology [22], which has been implemented in the Itanium II processor. Foxton is a control system that maximizes performance while staying within target power and temperature. It consists of power and temperature sensors, and a small on-chip hardware controller. If the power consumption is less than the target, the controller increases the core voltage — and the frequency follows. The opposite occurs if the power is over the target. Both cores in the Itanium II have the same voltage and frequency.

3. Background: Process Variation

Process variation refers to changes in transistor parameters beyond their nominal values, and results from manufacturing difficulties in very small feature technologies [2]. For the purposes of modeling and analysis, variation is generally broken down into die-to-die (D2D) and within-die (WID). WID variation can be further divided into systematic and random effects. In this work, we focus on WID variation, which has a direct effect on core-to-core variation in power and frequency within a CMP.

Several transistor parameters are affected by variation. Of key importance are the threshold voltage (V_{th}) and the effective gate length (L_{eff}). These parameters directly impact a transistor's switching speed and leakage power. The higher the V_{th} and L_{eff} variation is, the higher the variation in transistor speed across the chip is. This typically results in slow processors, since the slower transistors end up determining the frequency of the whole processor. In a CMP, the result is that different cores support different frequencies. Also, as V_{th} varies, transistor leakage varies across the chip. However, low-V_{th} transistors consume more power than high-V_{th} transistors save. As a result, with variation, chips consume substantially more leakage power. In a CMP, different cores leak different amounts.

Chip manufacturers hardly release any measurements on process variation for current or future technologies. As a result, we rely on statistical models of variation (e.g., [11, 20, 21, 28, 30, 37]) driven by values projected by the ITRS [14]. In this paper, we use the VARIUS model [30, 37], which we briefly summarize here.

To model systematic variation, the chip is divided into a grid. Each grid point is given one value of the systematic component of the parameter — assumed to have a normal distribution with $\mu = 0$ and standard deviation σ_{sys}. Systematic variation is also characterized by a spatial correlation, so that adjacent areas on a chip have roughly the same systematic component values. The spatial correlation between two points $\vec{x}$ and $\vec{y}$ is expressed as $\rho(r)$, where $r = |\vec{x} - \vec{y}|$. To determine how $\rho(r)$ changes from $\rho(0) = 1$ to $\rho(\infty) = 0$ as r increases, the spherical function is used. The distance at which the function converges to zero is when there is no significant correlation between two transistors. Such distance is called ϕ.

Random variation occurs at the level of individual transistors. It is modeled analytically with a normal distribution with $\mu = 0$ and standard deviation σ_{ran}. Since the random and systematic components are normally distributed and independent, their effects are additive, and the total σ is $\sqrt{\sigma_{ran}^2 + \sigma_{sys}^2}$. In the model, V_{th} and L_{eff} have different values of σ.

Algorithms to Minimize Power	
Random	Map threads on cores randomly
VarP	Map threads randomly on cores with lowest static power
VarP&AppP	Map threads with highest dynamic power on cores with lowest static power
Algorithms to Maximize Performance	
Random	Map threads on cores randomly
VarF	Map threads randomly on cores with highest frequency
VarF&AppIPC	Map threads with highest IPC on cores with highest frequency
Algorithms to Maximize Performance at a Power Budget (P_{target})	
*Random+Foxton**	Map threads on cores randomly and reduce (V_i,f_i) of cores round robin to meet P_{target}
*VarF&AppIPC+Foxton**	Map threads on cores with *VarF&AppIPC* and reduce (V_i,f_i) of cores round robin to meet P_{target}
VarF&AppIPC+LinOpt	Map threads on cores with *VarF&AppIPC* and use *LinOpt* to meet P_{target}
VarF&AppIPC+SAnn	Map threads on cores with *VarF&AppIPC* and use *SAnn* to meet P_{target}

Table 1. Algorithms for application scheduling and power management.

4. Application Scheduling and Power Management under Process Variation

In an environment with process variation, each processor in a CMP typically consumes a different amount of power and can support a different maximum frequency. Given that the core-to-core differences can be substantial, it makes sense to schedule applications and perform power management as if the variation-affected CMP was a heterogeneous system.

In this environment, there are two high-level design issues (Table 2). The first is whether the different cores of the CMP have to cycle at the same frequency (Uniform Frequency) or not (Non-Uniform Frequency). The second is whether the frequency and voltage of the cores can be changed dynamically (DVFS) or not (No DVFS). For these configurations, we present a set of simple variation-aware algorithms for application scheduling and power management, aimed at minimizing power or maximizing performance. Note that the scheduling algorithms are intended to complement the other scheduling criteria used by the OS, such as priority, fairness, or starvation-avoidance. In the following, we assume that the number of threads is less than or equal to the number of cores.

	Uniform Frequency	Non-Uniform Frequency
No DVFS	Name: *UniFreq* – Minimize Power	Name: *NUniFreq* – Minimize Power – Maximize Perform.
DVFS	Name: *UniFreq+DVFS* – Maximize Perform. at a Power Budget	Name: *NUniFreq+DVFS* – Maximize Perform. at a Power Budget

Table 2. CMP configurations for application scheduling and power management under variation.

4.1. UniFreq: Uniform Frequency & No DVFS

In this configuration, although different cores support different maximum frequencies, all cores run at the frequency of the slowest one. Moreover, frequency does not change dynamically. We call this configuration *UniFreq* (Table 2). The only inter-core variation is in power consumption. Since all cores run at the same frequency, the goal of the scheduler is to minimize the power consumption at the given frequency.

The top part of Table 1 shows three possible algorithms that the scheduler can use in this configuration to minimize power. In *Random*, threads are mapped on cores randomly. This is our baseline. In *VarP*, the cores are ranked by their static power consumption from lowest to highest. Then, the *N* threads are randomly mapped on the top *N* cores (one thread per core). Finally, in *VarP&AppP*, in addition to ranking the cores as before, the threads are ranked by their dynamic power consumption from highest to lowest. Then, the highest-power threads are mapped on the lowest-power cores. The intuition is to "even out" the power consumption across cores, avoiding hot spots. Because of the exponential dependence of leakage on temperature, keeping the power consumption (and temperature) as uniform as possible saves power.

The configuration in Table 2 where the frequency is uniform and there is DVFS is called *UniFreq+DVFS*. It is a generalization of *UniFreq*, where the goal is now to maximize performance at a given power budget. We can use the same scheduling algorithms as in *UniFreq* — e.g., mapping the threads with the highest dynamic power on the cores with the lowest static power — and then reduce the frequency and voltage until the power budget is met. Since most aspects of this configuration are covered in the other configurations, we do not consider it further in this paper.

4.2. NUniFreq: Non-Uniform Frequency & No DVFS

In the *NUniFreq* configuration (Table 2), different cores run at different frequencies — the highest that each supports. However, frequencies do not change dynamically. In this configuration, two simple scheduling goals are to minimize power or to maximize performance.

To minimize power, we use the *VarP* or *VarP&AppP* algorithms discussed in the previous section. To maximize performance, we use the algorithms in the middle of Table 1. In *VarF*, the cores are ranked by the maximum frequency they support, from highest to lowest. Then, the *N* threads are randomly mapped on the top *N* cores (one thread per core). In *VarF&AppIPC*, in addition to ranking the cores as before, the threads are ranked by their average IPC from highest to lowest. Then, the highest-IPC threads are mapped on the highest-frequency cores. The intuition is that low-IPC threads

typically benefit less from high frequency — because they are often memory-bound.

4.3. NUniFreq+DVFS: Non-Uniform Frequency & DVFS

In this configuration, different cores run at different frequencies and DVFS can be applied independently to each core. We call this configuration *NUniFreq+DVFS* (Table 2). In this configuration, the most obvious optimization goal is to maximize performance at a given power budget. Given the complexity of this environment, where the algorithms for scheduling and for power management interact, a global optimization solution is required.

The lower part of Table 1 shows possible algorithms to maximize performance at the target power. To simplify the problem, we construct each algorithm in two steps. First, we select one of the scheduling algorithms that maximizes performance (*VarF* or *VarF&AppIPC*), to map the threads to cores. Then, we use a power management algorithm to find the best (V_i,f_i) pair for each active core *i*, that maximizes overall performance while keeping the total power no higher than P_{target}.

Because of the non-linear dependence of power on *V*, and the exponential size of the search space, finding the optimal solution to the second part of the problem is very expensive. Previous solutions that have looked at global optimization of DVFS on CMPs [12] have used an *exhaustive* search through the solution space. This is feasible only for very small systems and does not scale. For a system like the one we evaluate (a 20-core CMP with many per-core voltage-frequency pairs), exhaustive search is too expensive.

Our approach is to reduce the problem to a linear optimization problem, and then use linear programming [24] to solve it. We call our algorithm *LinOpt*. Moreover, we also compare *LinOpt* to a more complex non-linear algorithm based on simulated annealing (*SAnn*), which provides a solution that is very close to the optimal one.

Overall, we examine the following algorithms (Table 1). In *Random+Foxton**, we map threads on cores randomly. Then, from among the active cores, we select one core at a time in a round-robin manner, and reduce that core's (V_i,f_i) one step. We stop when the chip-wide P_{target} constraint is satisfied and a per-core power constraint ($P_{coremax}$) is satisfied for all cores. This power management algorithm is the simplest one, and a small extension over the one implemented by the Foxton controller in the Itanium II [22] — where the two cores have the same (V,f) pair. We use *Random+Foxton** as our baseline algorithm.

In the other algorithms, we map threads on cores using *VarF&AppIPC* and then use different power management algorithms to set the (V_i,f_i) pair for each active core *i*. Specifically, we consider *VarF&AppIPC+Foxton**, *VarF&AppIPC+LinOpt*, and *VarF&AppIPC+SAnn*. To be effective, both *LinOpt* and *SAnn* have the same goal as the *VarF&AppIPC* scheduling algorithm, namely to maximize performance. In the following, we present the *LinOpt* and *SAnn* algorithms.

4.3.1. LinOpt: Power Management Using Linear Programming

Linear programming [24] is a mathematical technique for solving linear optimization problems of the following form: for N independent variables $x_1, ..., x_N$, maximize the objective function:

$$g = a_1x_1 + a_2x_2 + ... + a_Nx_N$$

subject to N primary constraints $x_1 \geq 0, x_2 \geq 0, ..., x_N \geq 0$ and to any number of additional constraints of the form:

$$b_1x_1 + b_2x_2 + ... + b_Nx_N + b \leq B$$

where $a_{1..N}$, $b_{1..N}$, b and B are problem-specific constants.

The problem we want to optimize is the following. Given a set of N cores $C_{1..N}$ that can each run at M different voltage levels $V_{1..M}$ (each with its corresponding frequency), find the best selection of voltage levels $(v_1, ..., v_N)$ for cores $C_{1..N}$ that maximizes the average throughput (*TP*) subject to the following constraints: (i) the total chip power is less than P_{target} and (ii) the power of each core is less than $P_{coremax}$.

We would like to express this problem as a linear optimization problem. To do this, we need to make sure that the objective function as well as all the constraint inequalities are linear functions. This requires some approximation.

We start with the objective function, which is the average throughput *TP*, measured in millions of instructions per second (MIPS). If we call tp_i the throughput of core i, we have:

$$TP = \frac{tp_1 + tp_2 + ... + tp_N}{N}$$

We express *TP* as a linear function of the variables we are trying to find, namely the set of optimal voltage levels $(v_1, ..., v_N)$ for the cores. By definition, $tp_i = f_i \times ipc_i$, where f_i is the frequency of core i and ipc_i is the IPC of the thread running on i. Now, f_i is largely a linear function of v_i — with parameters that depend on the core used. Moreover, to a first approximation, we can assume that ipc_i is not a function of the frequency. In reality, of course, the IPC changes with frequency — mostly because of off-chip accesses. However, we can neglect this dependence because the change in a thread's IPC as we vary frequency is typically smaller than the change in IPC across threads. Consequently, we can write $tp_i = a_iv_i$, where a_i is a constant that depends on the thread and on the core. We obtain, for a given assignment of threads to cores:

$$TP = \frac{a_1}{N}v_1 + \frac{a_2}{N}v_2 + ... + \frac{a_N}{N}v_N$$

where $v_{1..N}$ are the set of voltages we are trying to find. Next, we define the constraints of the optimization problem. Two trivial sets of constraints are the upper and lower bounds on the values of $v_{1..N}$:

$$v_1, v_2, ...v_N \leq V_{high} \text{ and } v_1, v_2, ...v_N \geq V_{low}$$

Next, we define the main constraint, which specifies that the total CMP power is less than P_{target}. For this, we need to express the total power of each core i as a linear function of supply voltage as $p_i = b_iv_i + c_i$, where b_i and c_i are both core- and thread-specific constants. In reality, the total power of a core is clearly not a linear function of supply voltage — dynamic power is quadratic in supply voltage (or cubic, if we add the corresponding change in frequency) and static power is more than linear. However, in practice, the solutions that we obtain with this linear approximation are good. They

satisfy the power constraints with little slack and provide a performance very close to that obtained with more time consuming, non-linear formulations. This will be shown in Section 7.5.

Because of variation, we cannot generate the $p = f(v)$ function analytically. Instead, we experimentally measure the power of a thread-core pair at three voltage levels, namely V_{low}, V_{high} and V_{mid} (the average of the two). Then, as shown in Figure 1, we find the values of the constants b_i and c_i that minimize the differences $dErr$ for all three points.

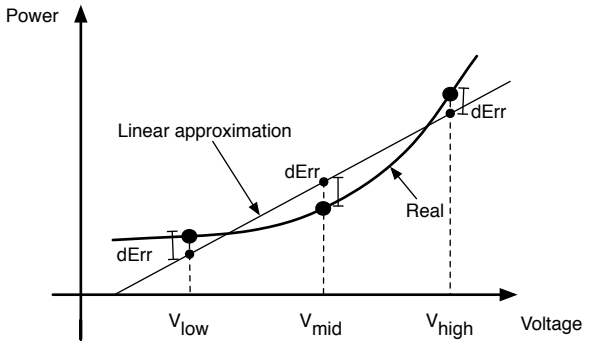

Figure 1. Linear approximation of the power dependence on voltage.

The P_{target} constraint equation can then be written as follows, where all the $c_{1..N}$ constants are folded into c:

$$b_1 v_1 + b_2 v_2 + ... + b_N v_N + c \leq P_{target}$$

Finally, the last set of constraints specifies that the power $p_{1..N}$ of each core should be less than $P_{coremax}$:

$$b_i v_i + c_i \leq P_{coremax}, \forall i \in 1..N$$

There are several techniques for solving linear programming problems. We choose the Simplex method [24] because it is relatively straightforward to implement and, in practice, it is often fast to compute.

The inputs to *LinOpt* are the constants $a_{1..N}$, $b_{1..N}$ and $c_{1..N}$, as well as the power constraints P_{target} and $P_{coremax}$. In Section 5, we show how we use profiling to compute these constants. The output of *LinOpt* is the best voltage for each core $(v_1, ..., v_N)$.

4.3.2. Other Global Optimization Solutions

We also examine solving the optimization problem of Section 4.3.1 using a non-linear algorithm. We choose simulated annealing (*SAnn*) — a well-known probabilistic algorithm for solving global optimization problems [17]. The goal of *SAnn* is the same as the one used with *LinOpt*: maximize throughput under power constraints. For a given mapping of threads to cores, the search space of the *SAnn* algorithm consists of all possible combinations of voltage levels (and their corresponding frequency levels) for each of the cores.

Unlike *LinOpt*, *SAnn* computes the power at each voltage level accurately, without the linear approximation. This should allow *SAnn* to generate a better solution. However, unlike linear programming, simulated annealing may not find the global optimum and, instead, produce a local optimum. In practice, the results of Section 7 show that *SAnn* produces better results than *LinOpt*. However, *SAnn* is orders of magnitude slower than *LinOpt*, which makes it impractical for on-line use.

5. System Implementation

Our target system is a CMP with many cores — 20 in our evaluation. The algorithms for application scheduling and power management of Section 4 are run by supervisor code. The power management algorithm can be run by either a core or a Power Management Unit (PMU) as in the Itanium II [22]. The application scheduling and power management algorithms may use profile information about core frequency and power, or application dynamic power and IPC. In what follows, we briefly outline the frequency, voltage and power control, and the profiling support.

5.1. Frequency, Voltage and Power Control

A CMP with per-core frequency control requires separate PLLs to generate the clock signal for each core. These PLLs are controlled independently. Moreover, there is a synchronization mechanism between the different frequency domains, such as FIFO buffers. All this support is already present in AMD's Quad-Core Opteron [6].

To support per-core voltage control, the CMP needs per-core power grids and voltage regulators that generate the different voltages. Currently, such regulators are on the board, but new technologies will soon make it possible to place them on the processor package or even on the die [16]. In this case, voltage transition speeds will be orders of magnitude faster. In this paper, however, we conservatively assume that the voltage and frequency transition speeds are those of current systems such as Xscale [4].

To run a power management algorithm such as *LinOpt*, we need on-chip sensors that provide power consumption information as in Itanium II [22], on a per-core basis. If a PMU is used to run the algorithm, a simple on-chip design like the controller in Itanium II can be used. Such a design consumes less than 0.5W and takes up less than 0.5% of the die area.

Figure 2 shows a timeline of the execution of the algorithms. At every OS scheduling interval, the OS revisits its assignment of threads to cores using one of the scheduling algorithms — for instance, *VarF&AppIPC*. At more frequent intervals, e.g., every 10ms, the *LinOpt* algorithm runs and sets the cores to the best (V, f) pairs.

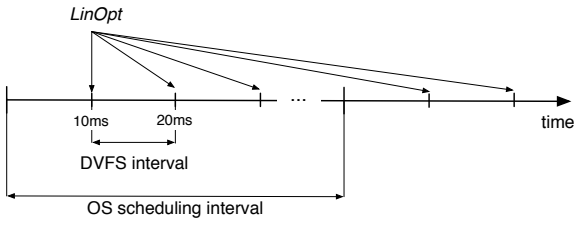

Figure 2. Execution timeline for application scheduling and *LinOpt* invocation.

Algorithm		Profile Information Required
Sched.	*VarP*	For each core: static power consumption at the maximum voltage
	VarP&AppP	For each core: static power consumption at each voltage level For each thread: dynamic power consumption while running on one random core
	VarF	For each core: maximum frequency supported at the maximum voltage
	VarF&AppIPC	For each core: maximum frequency supported at the maximum voltage For each thread: IPC while running on one random core
Power Manag.	*Foxton**	—
	LinOpt	For each core: table of (voltage, frequency) pairs For each selected thread-core pair: IPC of the thread while running on the core For each selected thread-core pair: total power at 3 (or 2) voltage levels

Table 3. Profile information needed for the application scheduling and power management algorithms.

5.2. Profiling Support

For the application scheduling and power management algorithms to be effective, they need some profile information. Some of this information is provided by the chip manufacturer, while other is provided dynamically by sensors as applications execute. Table 3 summarizes the profile information needed for each application scheduling and power management algorithm.

For the *VarP* scheduling algorithm, we need, for each core, the static power consumption at the maximum voltage. This is provided by the manufacturer, who measures the power while keeping the chip under zero load. Note that this is only an estimate of the actual static power of the cores at run time because the static power is heavily dependent on the temperature. However, it is good enough because we are only interested in a *ranking* of cores based on their static power.

For the *VarP&AppP* algorithm, we need, for each core, the static power consumption at *each* voltage level (Table 3). The reason why it is not enough to get the static power at only a single voltage will be clear later. All these values are also provided by the manufacturer. In addition, we need, for each thread, the dynamic power it consumes while running on one random core (Table 3). This information is obtained by reading the power sensors in the core for a given section of the thread's execution. The measured power is the total power, so we need to subtract the static power. Since the core may be running at a voltage different than the maximum value, we need to know the static power consumption at the current voltage — hence the need for the previous information.

Note that each thread is profiled on a potentially different core. To compare the resulting dynamic powers obtained, the power measured is scaled according to the frequency and voltage of the particular core used. We assume that the other factors that determine the dynamic power are largely constant across cores. Again, these measurements are good enough because we are only interested in a ranking of threads based on their dynamic power.

The *VarF* algorithm needs, for each core, the maximum frequency supported at the maximum voltage. This is provided by the manufacturer.

The *VarF&AppIPC* algorithm additionally needs, for each thread, the IPC it delivers while running on one random core (Table 3). The IPC is obtained by reading simple performance counters in the core for a given section of the thread's execution. Each thread is profiled on a potentially different core. As indicated in Section 4.3.1, we assume that the IPC changes negligibly with frequency and voltage changes — although a correction could be made based on the measured miss rate. We also assume that no other core property affects the IPC. Again, we are only interested in a ranking of threads based on their IPC.

We now consider the power management algorithms. *Foxton** needs no profile information. *Linopt*, as described in Section 4.3.1, needs some additional information (Table 3). First, for each core, it needs a table of (voltage, frequency) pairs. This table is supplied by the manufacturer. Then, for each of the thread-core pairs selected by the scheduling algorithm, we need the IPC of the thread while running on the core. As indicated before, this is obtained dynamically with performance counters and is assumed largely independent of the frequency and voltage. With these two sets of values, we can generate the $a_{1..N}$ constants of Section 4.3.1.

Finally, for each of the thread-core pairs selected by the scheduling algorithm, we need the power consumed at three (or at the very least two) voltages. This information allows us to generate a curve like Figure 1 for each of the selected thread-core pairs, and then generate the $b_{1..N}$ and $c_{1..N}$ constants of Section 4.3.1. This information is obtained dynamically with power sensors in the cores.

Since the IPC and power of a thread-core pair changes with time, IPC and power profiling is on all the time, and the *Linopt* algorithm is run periodically at short intervals. At longer intervals, we run the scheduling algorithm, which may change the assignment of threads to cores based on the new conditions. This is shown in Figure 2.

Note that, in all algorithms, the system continuously monitors the total power and the per-core powers. These values are compared to P_{target} and $P_{coremax}$, respectively.

6. Evaluation Methodology

We use the SESC cycle-accurate execution-driven simulator [26] to model a large CMP with 20 2-issue out-of-order cores on 32nm technology. Each core is like an Alpha 21264. Figure 3 shows the floorplan of the CMP and Table 4 summarizes the architecture configuration. In the following, we discuss the different parts of our infrastructure.

6.1. Variation Model Parameters

To model WID variation, we use the VARIUS model [30, 37] to generate V_{th} and L_{eff} variation maps. We then superimpose these maps on our floorplan as shown in Figure 3. This allows us to model how variation affects core power and frequency.

Table 4 shows some of the process parameters used. There is little public-domain information on likely values for V_{th} σ/μ and ϕ. For σ/μ, the 1999 ITRS [13] gave a design target of 0.06 for

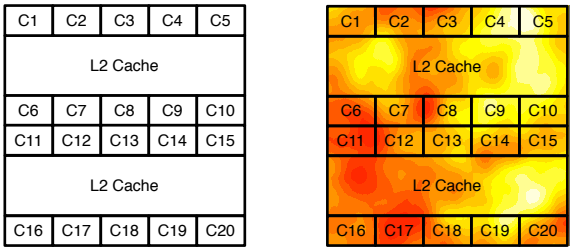

Figure 3. Floorplan of the 20-core CMP and superimposition of a V_{th} variation map.

Overall: CMP with 20 out-of-order Alpha 21264-like procs.
Technology: 32nm, 4GHz (nominal)
Branch prediction: 4K-entry BTB, 7-cycle mispred. penalty
Core fetch/issue/commit width: 4/2/2
Register file size: 80 entry; Scheduler size: 20 fp, 40 int
Private data and instr. L1: 2-way 16K each; 2-cycle access
Shared L2: 8-way 8 MB; 8-12 cycle access
Cache line size: 64 bytes all
Memory access time: 400 cycles
Die size: $340mm^2$; V_{DD}: 0.6-1V (default is 1V)
Number of dies per experiment: 200
V_{th}: μ: 250mV at 60 ^{o}C
σ/μ: 0.03-0.12 (default is 0.12)
ϕ (fraction of chip's width): 0.5

Table 4. Summary of the architecture configuration.

year 2006 (although no solution existed); however, the projection has been discontinued since 1999. Toyoda [38] presents a measured $\sigma/\mu = 0.07$ for V_{th} in chips at $130nm$ technology. In this paper, we consider a range of values for V_{th}'s σ/μ, namely 0.03–0.12, and use as default 0.12. Moreover, we assume that the random and systematic components have equal variances. For ϕ, we use Friedberg's *et al.* [8] measurement that the gate length had a correlation range close to 0.5 of the chip's width. Since the systematic component of V_{th}'s variation directly depends on the gate length's variation, we set ϕ = 0.5 for V_{th}.

Based on the 1999 ITRS [13], we set L_{eff}'s σ/μ to 0.5 of V_{th}'s σ/μ. Moreover, for L_{eff}, we also assume that that the random and systematic components have equal variances, and that ϕ = 0.5.

Each individual experiment uses a batch of 200 chips that have a different V_{th} (and L_{eff}) map generated with the same μ, σ, and ϕ. To generate each map, we use the geoR statistical package [27] of R [25]. We use a resolution of 1M points per chip.

6.2. Power and Temperature Model

To estimate power, we scale the results given by popular tools using technology projections from ITRS [14]. Specifically, we use SESC, which is augmented with dynamic power models from Wattch [3] to estimate dynamic power at a reference technology and frequency. In addition, we use HotLeakage [39] to estimate leakage power at the same reference technology. Then, we obtain ITRS's scaling projections for the per-transistor dynamic power-delay product, and for the per-transistor static power. With these two factors, given that we keep the number of transistors constant as we scale, we can estimate the dynamic and leakage power for the scaled technology and frequency relative to the reference values.

We use HotSpot [31] to estimate on-chip temperatures. To do so, we use the iterative approach of Su *et al.* [35]: the temperature is estimated based on the current total power; the leakage power is estimated based on the current temperature; and the leakage power is included in the total power. This is repeated until convergence.

6.3. Critical Path Model

To determine how the frequency of the processors is affected by variation, we need to model the structure and distribution of critical paths in the processors. For this, we use the models in [30], which include models for critical path distributions in pipeline stages with logic and in stages with memory structures. The distribution of critical path delays for logic stages is obtained using experimental data from Ernst *et al.* [7], who characterized a multiplier unit. For memory stages, [30] extends the model of Mukhopadhyay *et al.* [23] for the access time of a 6-transistor SRAM cell, to include the time to access the whole SRAM structure. With this model, we can estimate the frequency of the processors. We use CACTI 4.0 [36] to estimate path layouts and wire delays, and the alpha-power law [29] to compute gate delays.

6.4. Workloads

We evaluate our algorithms with a collection of applications from SPECint (*bzip2, crafty, gap, gzip, mcf, parser, twolf,* and *vortex*) and SPECfp (*applu, apsi, art, equake, mgrid,* and *swim*). We use applications from this pool to construct multi-programmed workloads that contain from 1 to 20 applications — where each application runs on a different core. This approach to construct workloads has been used elsewhere [12, 18]. Each experiment is repeated 20 times; each time with a different set of applications. We report the average outcome of the 20 trials.

We use the simulation points present in SESC to run the most representative phases of each application. Each application runs with the reference input set for about 12 billion instructions.

6.5. Optimization Algorithms

The *LinOpt* algorithm uses the Simplex method [24] to solve the linear optimization problem. We use profile information as described in Section 5.2 to generate all the constants required. To approximate the power dependence on voltage as in Figure 1, we measure the power at 1, 0.8, and 0.6V. In our experiments of Section 7, we run *LinOpt* every 10ms.

For *SAnn*, we use the implementation of simulated annealing in the R statistical package [25]. The goal of *SAnn* is the same as *LinOpt*: maximize throughput under power constraints. In *SAnn*, the initial values of voltage and frequency for each core are determined using a simple greedy heuristic. The initial Annealing Temperature (AT) is determined based on the complexity of the problem: for a large number of threads, more randomness is needed in the initial search and, therefore, a higher value of the initial AT is used. As the number of threads decreases, we use a lower initial AT. At each AT, the next point in the solution space is generated from a Gaussian Markov kernel with scale proportional to the current AT. The algorithm automatically decreases the AT according to

a logarithmic cooling schedule. The algorithm stops after 1 million function evaluations.

We use the results of *SAnn* as an upper bound for what *LinOpt* can achieve. We therefore want to make sure that *SAnn* produces a solution as close to the optimal one as possible. Consequently, we tune the constants in *SAnn* by comparing its results, for several configurations, to an *exhaustive search* through the solution space. Since the exhaustive search is very time consuming, we can only perform it for configurations of up to 4 threads. In all these cases, the *SAnn* throughput results are within 1% of those for the exhaustive search.

6.6. Metrics

In our evaluation, we use the following metrics: total power (which includes the static and dynamic powers of processors, L1 caches, and the L2 cache), average frequency of the active cores, throughput (measured in millions of instructions per second or MIPS), and the energy delay-square product (ED^2). We also give the weighted throughput [32], which uses the weighted IPCs of the applications. The weighted IPC of an application is computed as the application's IPC normalized to the application's IPC at reference conditions. This metric gives equal weight to all the applications when measuring total system throughput.

7. Evaluation

We begin by examining the effect of process variation on core-to-core variation in power and frequency. Next, we evaluate the variation-aware algorithms for application scheduling and power management of Table 1 for the *UniFreq*, *NUniFreq*, and *NUniFreq+DVFS* configurations.

7.1. Variation Effects on Power and Frequency

To examine the potential of variation-aware algorithms, we measure the core-to-core variation. For a given die, we successively run all of our applications on a given core and compute the average power consumed by the core (which includes the L1 caches) per application. We repeat the experiment for all the cores. Then, we compute the ratio between the power consumed by the most power-consuming core to the power of the least power-consuming core. Figure 4(a) shows the resulting ratio for all the 200 dies in the form of a histogram. We see that, in most of the dies, there is 40-70% variation in total power. The average is around 53%.

We now examine core-to-core frequency variation. Since circuit delay increases with temperature, we measure the frequency of each core at the maximum temperature that any application reaches, which we measure to be around 95 ^{o}C. Then, for each die, we compute the ratio between the frequencies of the fastest and the slowest cores. Figure 4(b) shows the resulting ratio for all the 200 dies in the form of a histogram. We see that, in most of the dies, there is 20-50% variation in core frequency. The average is around 33%.

Figure 5 shows how the average ratio between maximum and minimum core power (a) and frequency (b) changes with different values of $V_{\rm th}$ σ/μ. As expected, the core-to-core variation in both power and frequency increases with larger σ/μ. Even for a small σ/μ=0.06, the variation is very significant. Consequently, variation-aware algorithms for application scheduling and power management have good potential.

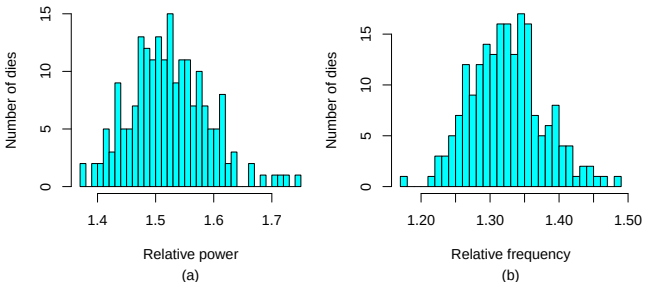

Figure 4. Histograms of the ratio between the powers consumed by the most and least power-consuming cores in the die (a) and between the frequencies of the fastest and slowest cores in the die (b).

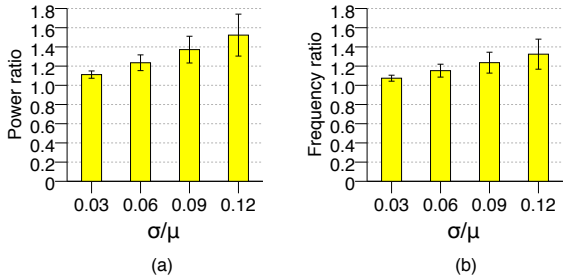

Figure 5. Average ratio between the maximum and minimum core power (a) and core frequency (b) for different values of $V_{\rm th}$ σ/μ, for 200 dies.

Due to variation, two different cores may achieve the same frequency at different voltage levels and with very different power costs. Moreover, the relative power efficiency of the two cores can change with the frequency. All this variation makes the job of a global power management scheme very challenging. As an illustration, we consider one sample die and identify the highest-frequency core (which we call *MaxF*) and the lowest-frequency one (which we call *MinF*). We run the *bzip2* application and, as we change the voltage levels, we measure the core power.

The result is shown in Figure 6. The figure shows the core power as a function of the frequency. Each core has a curve, where the dots represent voltage levels changing from 1V (top right) to 0.6V (bottom left). Power and frequency axes are normalized to the values for *MaxF* at 1V. The figure shows that, say, a 0.8 frequency can be obtained by *MaxF* at 0.7V or by *MinF* at 1V — but *MaxF* consumes less power. The figure also shows that, for frequencies below 0.74, *MinF* is more power efficient, while above that, *MaxF* is more efficient.

7.2. Application Power and IPC

Application scheduling and power management algorithms also leverage the fact that applications have a varied behavior. For example, Table 5 shows, for each of our applications, the average dynamic power of the core (which includes the L1 cache) at 4GHz and 1V, and the average IPC. From the table, we see that there is significant variation in both dynamic power (up to 2.9×) and IPC (up to 12×) across applications.

	applu	apsi	art	bzip2	crafty	equake	gap	gzip	mcf	mgrid	parser	swim	twolf	vortex
Dynamic power (W)	4.3	1.6	2.4	3.7	3.9	2.1	3.5	2.7	1.5	2.2	2.8	2.2	2.3	4.4
IPC	1.1	0.1	0.2	1.1	1.1	0.3	1.0	0.7	0.1	0.4	0.7	0.3	0.4	1.2

Table 5. Average dynamic power of the core at 4GHz and 1V, and IPC for the applications used.

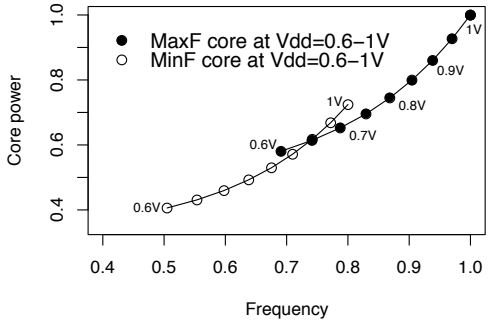

Figure 6. Core power as a function of frequency for the highest- and lowest-frequency cores in a sample die.

7.3. UniFreq: Uniform Frequency & No DVFS

The first configuration we evaluate is one with all the cores running at the same frequency and no DVFS. Figure 7(a) shows the total power consumed in the *Random*, *VarP* and *VarP&AppP* scheduling algorithms of Table 1, as we vary the number of threads in the workload. For a given number of threads, the bars are normalized to *Random*. Note that the cores that are not used are assumed to be powered off.

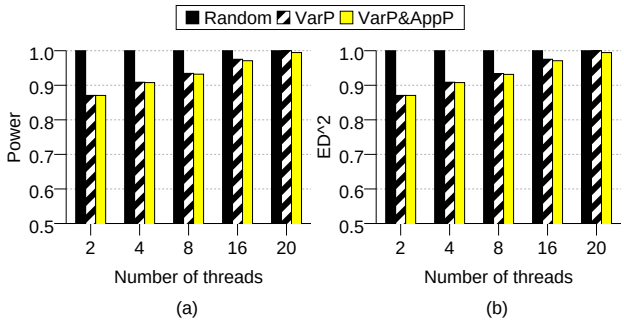

Figure 7. Total power consumption (a) and ED^2 (b) relative to *Random* in *UniFreq*.

Focusing on *VarP*, we see that for a lightly-loaded system, the savings in power are substantial — around 10% for 4 threads in the system. As the system load increases and more threads have to be scheduled, the power savings decrease. This is because more of the high-power cores need to be included in the scheduling pool. For full utilization (20 threads), *VarP* shows no power improvement.

VarP&AppP consumes the same power as *VarP*. The power averaging effect sought with *VarP&AppP* is not significant enough to reduce the temperature noticeably and, therefore, reduce the leakage power.

Figure 7(b) shows the ED^2 of the system for the different scheduling algorithms. Since *VarP* and *VarP&AppP* reduce the power at no cost in frequency, the ED^2 reduction is largely the same as the power reduction in Figure 7(a).

7.4. NUniFreq: Non-Uniform Frequency & No DVFS

NUniFreq allows each core to run at its maximum frequency. It can be shown that, under full occupancy (i.e., 20 threads), this increases the average core frequency by about 15% over *UniFreq* and increases the average power consumption by 10%. This in turn causes an average reduction in ED^2 of almost 20% for our applications. In what follows, we examine how variation-aware scheduling can further improve on these gains.

We start by evaluating algorithms to minimize power, namely *VarP* and *VarP&AppP*. Figures 8(a)–(b) are similar to 7(a)–(b) for *NUniFreq*. Figure 8(a) shows that the savings due to *VarP* and *VarP&AppP* are 14% for 4 threads. They decrease for more threads.

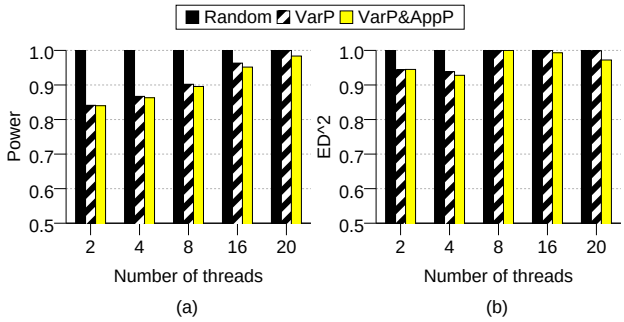

Figure 8. Total power consumption (a) and ED^2 (b) relative to *Random* in *NUniFreq*.

Figure 8(b) shows that these algorithms reduce ED^2 less than they did in 7(b). This is because in *NUniFreq*, different cores have different frequencies and *VarP* and *VarP&AppP*, by selecting the least-consuming cores, they may also end up selecting the lower-frequency ones, thus hurting ED^2.

We now consider algorithms to maximize performance, namely *VarF* and *VarF&AppIPC* (Table 1). Figure 9(a) shows the average frequency at which the threads run, normalized to that of *Random*. Recall that *VarF* and *VarF&AppIPC* select the same set of cores; therefore, their bars are the same. The figure shows that *VarF* increases the average frequency by 10% over *Random* for 4 threads. The frequency improvements decrease as the number of threads increases.

The benefits of *VarF&AppIPC* are apparent in Figure 9(b), which shows the average throughput in millions of instructions per second (MIPS) relative to *Random*. By scheduling high-IPC threads on high-frequency cores, *VarF&AppIPC* consistently delivers a higher throughput. Specifically, the throughput is 5–10% higher than *Random*. *VarF*, on the other hand, only delivers im-

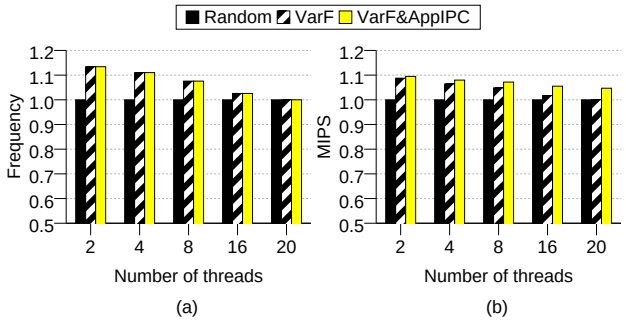

Figure 9. Average frequency (a) and average throughput (b) relative to *Random* in *NUniFreq*.

provements for lightly-loaded systems — for the 20-thread configuration, it effectively works as *Random*.

Finally, Figure 10 shows ED^2 for the same algorithms. For lightly-loaded systems (4 threads or less), the higher throughputs of *VarF* and *VarF&AppIPC* come at the cost of a higher ED^2. This is because the high-frequency cores selected dissipate more power, and the increase in throughput does not compensate. However, under higher loads (8 to 20 threads), *VarF&AppIPC* has a substantially lower ED^2 than *Random* or *VarF*. Specifically, *VarF&AppIPC*'s ED^2 is 10–13% lower than *Random*'s. The reason is that *VarF&AppIPC* increases the throughput substantially.

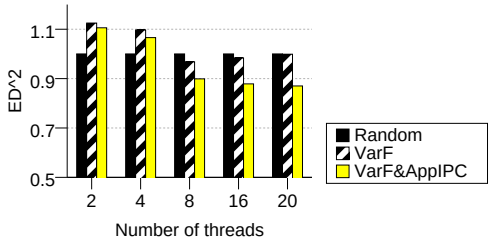

Figure 10. ED^2 relative to *Random* in *NUniFreq*.

7.5. NUniFreq+DVFS: Non-Uniform Frequency & DVFS

For *NUniFreq+DVFS*, we evaluate the algorithms in Table 1 that maximize performance at a given power budget, namely *Random+Foxton**, *VarF&AppIPC+Foxton**, *VarF&AppIPC+LinOpt*, and *VarF&AppIPC+SAnn*. We evaluate them under three Power Environments: *Low Power*, *Cost-Performance*, and *High Performance*. In these environments, the P_{target} when 20 threads are active is set to 50W, 75W, and 100W, respectively. When there are fewer threads, P_{target} is scaled down proportionally.

Figure 11(a) shows the average throughput of all the algorithms normalized to *Random+Foxton**, in the *Cost-Performance* Power Environment. We show results for different loads on the system, ranging from 4 to 20 threads. We see that *VarF&AppIPC+Foxton** only improves the average throughput by 4–6% for different numbers of threads. However, *VarF&AppIPC+LinOpt* is much more effective at boosting throughput. Specifically, it improves the average throughput by 12–17%. Moreover, its throughput is very close to that of *VarF&AppIPC+SAnn*. Indeed, *VarF&AppIPC+SAnn*'s throughput is only 2% higher. This is despite the fact that *VarF&AppIPC+SAnn* is orders of magnitude more costly in computation time.

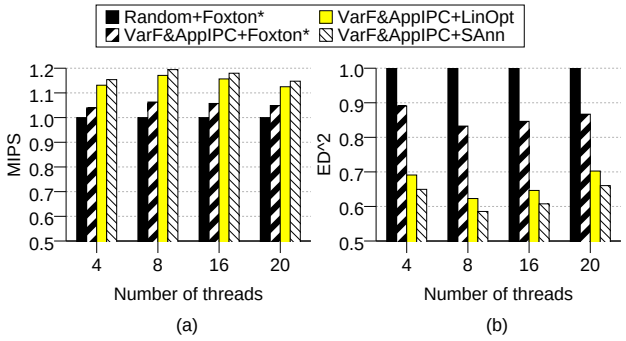

Figure 11. Average throughput (a) and ED^2 (b) for different algorithms relative to *Random+Foxton** in the *Cost-Performance* Power Environment.

Figure 11(b) shows the ED^2 for the same experiment. We see that *VarF&AppIPC+LinOpt* reduces ED^2 by 30–38%. This is a very remarkable reduction, and is very close to that of *VarF&AppIPC+SAnn*.

We now compare the three Power Environments. Figure 12 shows the average throughput of the algorithms normalized to *Random+Foxton** in the three Power Environments. All experiments are for 20-thread runs. We can see that the relative throughput gains of *VarF&AppIPC+LinOpt* are highest when the power target is low. The same is true for the other algorithms. For *VarF&AppIPC+LinOpt*, the average throughput gains in the *Low Power*, *Cost-Performance*, and *High Performance* environments are 16%, 12% and 11%, respectively.

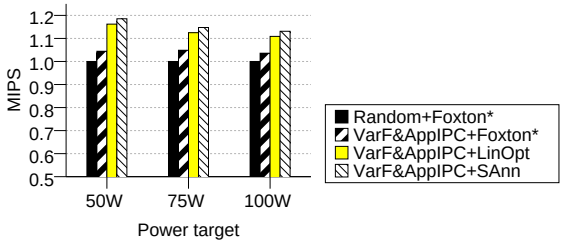

Figure 12. Average throughput for different algorithms relative to *Random+Foxton** in the three Power Environments. All experiments are for 20-thread runs.

Finally, we examine the impact of the algorithms on weighted throughput. This metric uses normalized IPC for each application and, therefore, is fairer to applications with low intrinsic IPC. Our algorithms improve throughput by adapting to IPC changes within each application — speeding up high-IPC sections and slowing down (and therefore saving power in) low-IPC sections.

Figure 13 shows the same experiments as Figure 11 but with weighted throughput as the optimization goal. We can see that the two figures are very similar, except for slightly smaller throughput improvements and ED^2 reductions in Figure 13. For example,

VarF&AppIPC+LinOpt improves the weighted throughput by 9–14% and reduces ED^2 by 24–33% — rather than by 12–17% and 30–38%, respectively, in Figure 11.

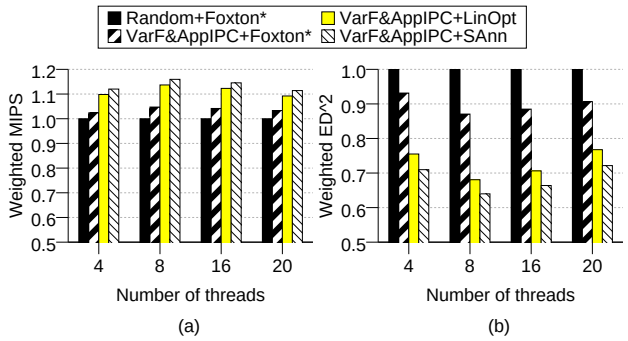

Figure 13. Average weighted throughput (a) and weighted ED^2 (b) for different algorithms relative to *Random+Foxton** in the *Cost-Performance* Power Environment.

7.5.1. LinOpt Granularity

How often we run *LinOpt* impacts our ability to keep the system power close to P_{target}. Specifically, if we use long intervals between *LinOpt* runs, the power consumed is likely to deviate more from P_{target} than if we use short intervals. Figure 14 considers different interval durations and measures the deviation between power consumed and P_{target}. Deviation is measured as follows. At every ms, the average power consumed in the past 1ms is compared to P_{target} and the absolute difference is recorded. Then, all the values recorded in the interval between two *LinOpt* runs are averaged out and plotted in Figure 14. The figure includes lines for 20- and 4-thread runs.

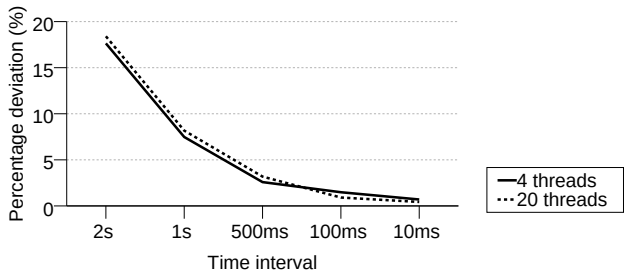

Figure 14. Average deviation of power consumption from P_{target} for different intervals between *LinOpt* runs.

Figure 14 shows that, as we decrease the interval between *LinOpt* runs, the power deviation decreases. For the 10ms-intervals that we use in our experiments, the deviation is less than 1%.

7.5.2. LinOpt Execution Time

The Simplex method used to solve *LinOpt* is generally very fast. The algorithm involves a variable number of steps, where the computation time of each step is affected by the size of the problem (the number of threads that are scheduled) and the number of constraints. Figure 15 shows the execution time of the algorithm on a 4GHz processor like the one considered in this paper. The figure shows data for different numbers of running threads and the different Power Environments.

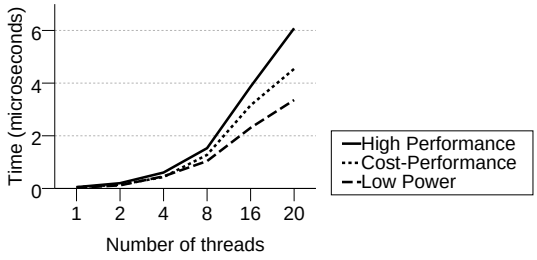

Figure 15. Execution time of the *LinOpt* algorithm for different numbers of threads in the three Power Environments.

The figure shows that the execution time increases with the number of threads. Moreover, it also increases as we go to less power-constrained environments such as *High Performance*. This is because a less strict environment increases the search space, making it harder to find a solution. Overall, the longest running time is 6μs. Since we run *LinOpt* every 10ms, the overhead is negligible.

8. Conclusions and Future Work

As a result of within-die process variation, individual cores in a CMP differ substantially in both static power consumed and maximum frequency supported. This paper showed that these effects can be leveraged with variation-aware algorithms for application scheduling and power management.

This paper proposed variation-aware scheduling algorithms to save power or improve throughput, and variation-aware power-management DVFS algorithms to maximize throughput at a given power budget. One such power-management algorithm, called *LinOpt*, uses linear programming to find the best voltage and frequency levels for each of the cores in the CMP. *LinOpt* runs online periodically, using profile information provided by the chip manufacturer and by on-chip power and IPC sensors. In a 20-core CMP, the combination of variation-aware application scheduling and *LinOpt* increased the average throughput by 12–17% and reduced the average ED^2 by 30–38% — all relative to using variation-aware scheduling together with a simple extension to Foxton's power management algorithm. Moreover, *LinOpt*'s throughput was within 2% of that of a simulated annealing algorithm, which had a computation time orders of magnitude higher.

We are working on several extensions to this work. One is to enhance our scheduling and power management algorithms with the additional goal of keeping the temperature of the CMP as uniform as possible. This can be achieved through aggressive migration of applications from active to inactive cores as in [9], and through temperature-aware mapping of applications to cores and assignment of (V,f) pairs. The result is likely to be fewer hot spots and lower power consumption, but it comes at the cost of increased complexity of the algorithms.

A second extension involves understanding how our variation-aware algorithms affect CMP wearout. Finally, we are also extending the work by analyzing the impact of the algorithms on parallel applications.

References

[1] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *International Symposium on Computer Architecture*, June 2005.

[2] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in *Design Automation Conference*, June 2003.

[3] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *International Symposium on Computer Architecture*, June 2000.

[4] L. Clark, E. Hoffman, J. Miller, M. Biyani, L. Liao, S. Strazdus, M. Morrow, K. Velarde, and M. Yarch, "An embedded 32-b microprocessor core for low-power and high-performance applications," in *Journal of Solid-State Circuits*, November 2001.

[5] J. Donald and M. Martonosi, "Power efficiency for variation-tolerant multicore processors," in *International Symposium on Low Power Electronics and Design*, October 2006.

[6] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, "An integrated quad-core Opteron processor," in *International Solid State Circuits Conference*, February 2007.

[7] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *International Symposium on Microarchitecture*, December 2003.

[8] P. Friedberg, Y. Cao, J. Cain, R. Wang, J. Rabaey, and C. Spanos, "Modeling within-die spatial correlation effects for process-design co-optimization," in *International Symposium on Quality Electronic Design*, March 2005.

[9] S. Heo, K. Barr, and K. Asanovic, "Reducing power density through activity migration," in *International Symposium on Low Power Electronics and Design*, August 2003.

[10] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *International Symposium on Low Power Electronics and Design*, August 2007.

[11] E. Humenay, D. Tarjan, and K. Skadron, "The impact of systematic process variations on symmetrical performance in chip multiprocessors," in *Design, Automation and Test in Europe*, April 2007.

[12] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *International Symposium on Microarchitecture*, December 2006.

[13] "International Technology Roadmap for Semiconductors (1999)."

[14] "International Technology Roadmap for Semiconductors (2006 Update)."

[15] I. Kadayif, M. Kandemir, and I. Kolcu, "Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessors," in *Design, Automation and Test in Europe*, February 2004.

[16] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *International Symposium on High-Performance Computer Architecture*, February 2008.

[17] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," in *Science*, May 1983.

[18] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *International Symposium on Microarchitecture*, December 2003.

[19] J. Li and J. Martínez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *International Symposium on High-Performance Computer Architecture*, February 2006.

[20] X. Liang and D. Brooks, "Mitigating the impact of process variations on processor register files and execution units," in *International Symposium on Microarchitecture*, December 2006.

[21] D. Marculescu and E. Talpes, "Variability and energy awareness: A microarchitecture-level perspective," in *Design Automation Conference*, June 2005.

[22] R. McGowen, C. A. Poirier, C. Bostak, J. Ignowski, M. Millican, W. H. Parks, and S. Naffziger, "Power and temperature control on a 90-nm Itanium family processor," *Journal of Solid-State Circuits*, January 2006.

[23] S. Mukhopadhyay, H. Mahmoodi, and K. Roy, "Modeling of failure probability and statistical design of SRAM array for yield enhancement in nanoscaled CMOS," *Transactions on Computer-Aided Design*, vol. 24, no. 12, 2005.

[24] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1988.

[25] R Development Core Team, *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2006. http://www.R-project.org.

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

[27] P. Ribeiro Jr. and P. Diggle, "geoR: A package for geostatistical analysis," *R-NEWS*, vol. 1, no. 2, 2001.

[28] B. F. Romanescu, S. Ozev, and D. J. Sorin, "Quantifying the impact of process variability on microprocessor behavior," in *Workshop on Architectural Reliability*, December 2006.

[29] T. Sakurai and R. Newton, "Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas," *Journal of Solid-State Circuits*, April 1990.

[30] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "VARIUS: A model of process variation and resulting timing errors for microarchitects," in *IEEE Transactions on Semiconductor Manufacturing*, February 2008.

[31] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *International Symposium on Computer Architecture*, June 2003.

[32] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreaded processor," in *Architectural Support for Programming Languages and Operating Systems*, November 2000.

[33] K. Srinivasan and K. S. Chatha, "Integer linear programming and heuristic techniques for system-level low power scheduling on multiprocessor architectures under throughput constraints," *Integration VLSI*, vol. 40, no. 3, 2007.

[34] K. Stavrou and P. Trancoso, "Thermal-aware scheduling: A solution for future chip multiprocessors' thermal problems," in *EUROMICRO Conference on Digital System Design*, 2006.

[35] H. Su, F. Liu, A. Devgan, E. Acar, and S. Nassif, "Full chip leakage estimation considering power supply and temperature variations," in *International Symposium on Low Power Electronics and Design*, August 2003.

[36] D. Tarjan, S. Thoziyoor, and N. P. Jouppi, "CACTI 4.0," Tech. Rep. HPL-2006-86, HP Labs, 2006.

[37] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, "Mitigating parameter variation with dynamic fine-grain body biasing," in *International Symposium on Microarchitecture*, December 2007.

[38] E. Toyoda, "DFM: Device and circuit design challenges," in *International Forum on Semiconductor Technology*, February 2004.

[39] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "HotLeakage: A temperature-aware model of subthreshold and gate leakage for architects," Tech. Rep. CS-2003-05, University of Virginia, March 2003.