Coordinated, Distributed, Formal Energy Management of Chip Multiprocesors

Abstract— *Designers are moving toward chip-multiprocessors (CMPs) to leverage application parallelism for higher performance while keeping design complexity under control. However, to date, no power management techniques have been proposed for coordinated power control of multiple processor cores.*

In this paper, we first illustrate how the use of local, per-tile dynamic voltage and frequency scaling (DVFS) techniques can result in tiles counteracting each others' power management policies, leading to oscillations and significantly hurting chip power-performance. We then propose a coordinated DVFS scheme for CMPs, which eliminates the oscillations and ensures efficient and resilient DVFS control. Specifically, our proposed technique incorporates thread information collected at run-time across the chip. In addition, by extending a control-theoretic local DVFS control technique toward DVFS for chip-multiprocessors, our technique prescribes DVFS settings formally at each tile, thus ensuring **stable, distributed, coordinated** *DVFS control of a CMP. Experimental results show that our technique achieves a 15.5% improvement in energy-delay product over a CMP with no DVFS control, and a 7% improvement in energy-delay product against the latest state-of-the-art local DVFS scheme.*

1. Introduction

To computer architects, chip multi-processors (CMPs) are very appealing, as replicated cores allow high-performance and high-transistor-count chips to be built with manageable complexity and power-efficiency. In addition to the IBM Power 5 and various research designs, CMPs are also emerging in lower-power embedded systems; in May 2004, ARM announced the MPCore embedded processor, aimed at set-top boxes, automotive applications, and mobile systems.

Power-efficiency and thermal-efficiency are increasing concerns for both embedded and high-end systems. Thus, a range of methods have been explored to address these issues. With uniprocessor chips we know that often there is insufficient work to fully occupy the processor, due to memory latencies, lack of parallelism, or some other effect. Continuing to run the processor at full speed thus simply wastes energy. To conserve energy, one can change the operating voltage and frequency to scale down the speed of the processor to match the decreased requirements in processing performance. This voltage and frequency scaling technique, or Dynamic Voltage and Frequency Scaling (DVFS), is common in both embedded processors as well as more high-performance processors.

With multiple processors on the chip, power issues are multiplied by the presence of more processors, and magnified by the fact that the processors can interact, cooperatively processing parallel threads of a single application. However, most DVFS techniques proposed to date typically apply to single processors such as the Intel XScale or Pentium-M [6, 3, 7]. Some recent work has looked at processors with several internal clock domains—multiple-clock-domain (MCD) processors [10, 13, 14, 17, 18]—but are restricted to local solutions in which each domain is considered separately. Such local, per-tile DVFS techniques lead to unstable DVFS control of CMPs processing multi-threaded applications, worsening overall chip power-performance (See Section 2).

The majority of current DVFS work can be characterized along two general lines: the level of dynamism (online or offline) and the level of formality (ad hoc or formal). Most prior DVFS work is either a profile-based optimization approach (off-line formal) [5, 9, 10, 19] or a run-time heuristic based (online ad hoc) approach [13, 14, 8, 11]. However, CMPs pose a major problem to these techniques. First, CMPs typically execute several applications at once, making it very difficult to obtain a representative profile. Second, the number of processors and possible placements explosively increases the tuning space for ad hoc approaches.

This motivates us to investigate and propose a run-time, control-theoretic DVFS algorithm for CMPs (online formal). An online solution eliminates the need for a representative profile, and the formal approach reduces the amount of parameter tuning required while ensuring stability. Currently, the best known online formal DVFS approach is that described in [17], which uses a control-theoretic approach in the context of MCD processors. For the remainder of this paper we will refer to this style of control as *local-PID*. *Local-PID* is a purely local scheme—that is, only information local to a tile is used, with interaction across multiple tiles ignored. In this paper, we propose a distributed version of this scheme, applying the basic mechanics of the formal approach to CMPs, while realizing stable, distributed, coordinated DVFS control.

Our approach shows that compared to a baseline approach with only tile-gating, our scheme improves the energy-delay product (EDP) by 15.5%. Our scheme compares favorably to *local-PID*, which can only produce an 8.48% improvement in EDP. For applications with high variability and dependencies between threads, our scheme improves EDP by 8.8%, whereas the local scheme manages less than a 1% improvement. We also show that our scheme is more stable and more resilient in situations where *local-PID* severely breaks down.

Our paper is structured as follows. Section 2 describes the *local-PID* scheme and its drawbacks when used in a CMP. Section 3 proposes our scheme, called *dist-PID*. Next, Section 4 details our simulation setup and the benchmarks used, with Section 5 following with the results. Finally, we draw our conclusions and sketch plans for future work in Section 6.

2. Motivation

To motivate the need for distributed, coordinated DVFS, we first study the effect of local DVFS control in a CMP, where each tile's frequency (and voltage) is set independently based on local information.

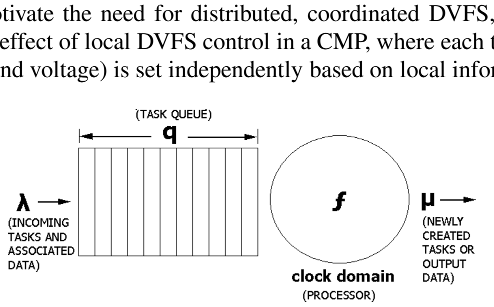

(TASK QUEUE)
q
λ
(INCOMING TASKS AND ASSOCIATED DATA)
f
clock domain
(PROCESSOR)
μ
(NEWLY CREATED TASKS OR OUTPUT DATA)

Figure 1. Each CMP tile modeled as a queueing system.

Our study is based on *local-PID*, which is representative of per-tile queue-based DVFS control policies [17, 18] and was shown to result in better energy efficiency than prior online/offline ad hoc DVFS approaches.

While used in an MCD context, the solid formal control-theoretic principles behind this algorithm can be readily applied to single-clock-domain CMPs: Each tile is modeled as a local queue model as shown in Figure 1, with each tile processor fed by its task queue, where threads scheduled on the processor await execution. The service rate is denoted as μ, which is determined by the tile processor frequency *f*. Demand is represented as λ, which in our system is the arrival rate of new tasks.[1] Queue occupancy, $\mathbf{q}$, refers to the number of task in the task queue, as well as the expected relative execution time of each task (what we term *load factor*, whose derivation will be elaborated in Section 3.3). In this framework, the queue size, $\mathbf{q}$, encapsulates the load on each tile. Conceptually, we seek to exactly match the service rate μ with the demand λ such that the average queue occupancy $\mathbf{q}$ remains constant from interval to interval; this implies that the processor has supplied just enough performance to meet the processing requirement of the application, and thus the maximum amount of energy has been saved.

Local-PID computes μ at interval *k* as:

$$\mu_k = \mu_{k-1} + K_i(q_k - q_{ref}) + K_p(q_k - q_{k-1}) \quad (1)$$

where μ_{k-1} is the service rate of the last interval, q_k and q_{k-1} the average queue occupancy of the current interval *k* and previous interval *k-1*. The remaining variables are fixed: K_i and K_p constants that are picked based on control-theoretic principles to ensure stability, and q_{ref} the steady-state desired queue occupancy constant. Essentially, the above Equation 1 predicts the needed service rate to *eventually* bring the queue occupancy to q_{ref}.

Conversely, if one were to assume a μ_k (proportional to the processor frequency), and solve for q_{ref}, then the equation produces what the eventual value of q_{ref} (target task queue size) should be in response to a frequency setting of μ_k. Solving for q_{ref} we get the following:

$$q_{ref} = (K_p(q_k - q_{k-1}) + K_i q_k - \mu_k + \mu_{k-1})/K_i \quad (2)$$

2.1 Demonstrating oscillations

Consider the following four-threaded program as shown in Figure 2. The code on the right is the main "master thread" (Thread *MT*) which launches three threads with initial parameters 2 (Thread *T(2)*), 17 (Thread *T(17)*), and 10000 (Thread *T(10k)*). The code on the left is the parallel `sum_sqrt()`. Each thread is run on a separate tile on the CMP.

At start, *MT* creates three threads, running the function `sum_sqrt`. *MT* then sends each thread their initial argument (num), before moving into a loop in which it sends a continuous stream of numbers (div) to each thread—each number is sent three times in total. Each thread then compares the value received and if it is divisible by the initial argument, adds the square root to a running sum. If the value is -1, the program ends.

From the code we see that *T(17)* has far less work to do; based on profiling, *T(17)* spends 169 cycles per element, whereas *T(2)* needs 540 cycles on average. *T(10k)* has virtually no work, and spends fewer than 100 cycles per element. *MT* inserts an entry every 106 cycles, and stalls when the queue occupancy of either destination is full. Thus, ideally, the tile executing *T(2)* should run at full speed, while the one executing *T(17)* should run at approximately one-third full frequency to realize maximum savings in energy-delay-product. *T(10k)* should quickly settle to the minimum speed.

Figure 3 shows a sample execution of the code in Figure 2 using the default settings for the stability constants (K_i=0.6 and K_p=0.2) for *local-PID*. We see from the figure that *T(2)* quickly saturates and runs at full frequency, as expected, and likewise, *T(10k)* drops to the minimum frequency, occasionally increasing its speed to compensate for small perturbations in execution. Unexpectedly however, rather than settling at a lower frequency, *T(17)* oscillates between low and high DVFS settings.

The reason is as follows: on average the processor completes one element every 169 cycles, but this is skewed, as the first 16 elements require almost no processing, while the 17th needs significant processing. While processing the 17th element, the incoming task queue fills up quickly—and since *T(17)* does not know that *T(2)* is running much more slowly, sees a large jump in queue occupancy and hurriedly raises its frequency in response. With only purely local information, there is no way that *T(17)* can know that it can slow down and match *T(2)*.

This example is relatively simple—if, for example, *T(2)* was connected in a chain to *T(17)* in a producer-consumer fashion, the oscillations would be worse. Rather than *T(2)* falling behind rather predictably due to the steady stream of data from *MT*, instead *T(2)* would be slowing down when another burst of data from *T(17)* arrives, forcing *T(2)* to increase its speed again. Now in addition to *T(17)* oscillating, *T(2)* is fluctuating as well. Clearly, this example illustrates the limitations of oblivious, local DVFS control in a CMP, motivating the need for distributed, coordinated DVFS.

2.2 Inapplicability of local DVFS schemes

So why was local, per-tile DVFS control sufficient in a MCD chip, but not a CMP? The reason lies in several critical differences between MCDs and CMPs, which accentuate the need for coordination of DVFS control across tiles.

First, the clock domains in MCDs are not homogeneous; the integer (INT) domain tends to have much lower latencies per queue element, and the load/store (LS) domain higher. Thus the LS domain needs to be more aggressive in order to preserve performance. This is recognized in [17] as the q_{ref} values are different, 6 for INT, 5 for FP, and 3 for LS, as are the queue lengths: 20 for INT, 16 for FP, and 16 for LS. In a CMP with homogeneous processors it is impossible to determine, at design time, the different loads on each processor tile across a range of applications. Even within an application, thread placement on tiles may be scheduled dynamically at the user level or by the operating system. Thus, with a single, fixed q_{ref}, one will always encounter mismatches where q_{ref} is too lax or too aggressive. Second, the input rate λ for MCDs is always derived from a single source (all instructions fetched from a centralized instruction fetch unit), while the tiles in CMPs can be fed instructions from multiple threads and programs.

Furthermore, domains in a MCD processor essentially have an instantaneous feedback loop due to the single, shared fetch stage; local queue information is thus highly indicative of overall chip load. Conversely, task queue occupancies on CMP tiles are more loosely correlated, with spikes and ebbs at different points in time, depending on the data dependencies between threads. Thus as this simple 4-threaded program illustrates (see Figure 3), *local-PID* leads to processors adapting based on their local, internal values of q_{ref} rather than to the overall load situation across the chip.

This situation with q_{ref} is not unique to *local-PID*. Basing DVFS decisions locally and statically, whether it be q_{ref} or some other metric, will be less stable and flexible. Using synchronous-clock uniprocessor DVFS schemes, such as trying to keep IPC constant across all processors, are even less indicative of thread imbalances in between tiles. DVFS schemes for MCDs at least incorporate the sense of separate domains. For MCD processors however, the domains are tightly integrated enough such that each element transmitted between domains requires only a small bit of processing. The information drawn from queue occupancy is thus more indicative of the workload across the whole chip. In CMPs, this is less true, and therefore we need a scheme that more explicitly shares information between domains so that tiles can coordinate and produce a set of DVFS settings in a distributed fashion while preserving global chip performance.

3. Proposed Work

Our example in Section 2 points to the limitations of DVFS based

[1]In our system, λ is usually the arrival of a new thread in the task queue. However, λ can also be the arrival of a new piece of data for processing, for example, in the case of streaming applications.

```
void sum_sqrt() {
    int num, div = 0;
    float sum = 0.0;

    thread_recv(&num, sizeof(int), ANY);

    while(div != -1) {
        if((div%num) == 0) sum += sqrt(num);
        thread_recv(&div, sizeof(int), ANY);
    }

   _thread_send(&div, sizeof(int), ANY);
   _thread_terminate();
}
```

```
int main() {//Thread MT
    int i, child1, child2, child3;

    child1 = thread_create(&sum_sqrt);
    child2 = thread_create(&sum_sqrt);
    child3 = thread_create(&sum_sqrt);

    thread_send_int(2, sizeof(int), child1); //sending num
    thread_send_int(17, sizeof(int), child2);
    thread_send_int(10000, sizeof(int), child3);

    for(i = 0; i < 1000; i++) {
      thread_send_int(i, sizeof(int), child1); //sending div
      thread_send_int(i, sizeof(int), child2);
      thread_send_int(i, sizeof(int), child3);
    }

    thread_send_int(-1, sizeof(int), child1);
    thread_send_int(-1, sizeof(int), child2);
    thread_send_int(-1, sizeof(int), child3);
    thread_wait_children();
    return 0;
  }
```

Figure 2. Sample source code

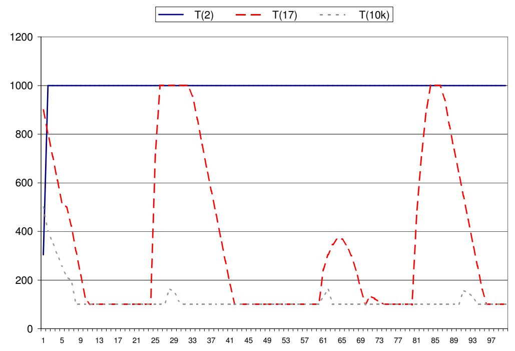


Figure 3. Frequency selection for execution of code in Figure 2

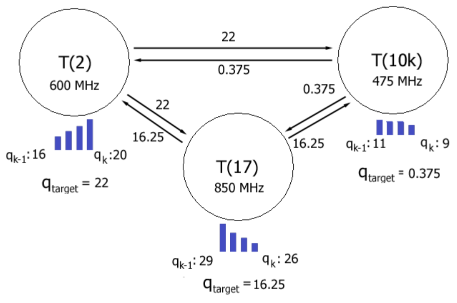


Figure 4. Example of how *dist-PID* assesses the code in Figure 2

on purely local information—more specifically, the target q_{ref} of the formal control method which dictates the target processor frequency needs to (1) adapt at runtime to match thread behavior; (2) be set based on global information, rather than fixed locally at each tile; and (3) be set to preserve high performance. Here, we detail how we build a formal online method that supports stable, distributed, coordinated DVFS control in a CMP, which we call *dist-PID*.

The intuition behind *dist-PID* lies in the observation that to preserve application performance while maximizing energy savings, it is necessary to identify the threads that lie on the *critical path* of an application. Those threads should be run at maximum speed to preserve performance, while others can be slowed to maximize energy savings without impacting performance. In parallel applications, the thread that is the last to reach a synchronization point is that which lies on the critical path. Hence, intuitively, if each tile knows the expected execution time of the longest-running-thread, they can DVFS their processing speeds to match that, optimizing energy-delay-product.

Dist-PID operates in three steps:

1. At each tile, estimate future queue occupancy (tile workload) using Equation 2, assuming the maximum service rate, and renaming q_{ref} as q_{target}:

$$q_{target} = (K_p(q_k - q_{k-1}) + K_i q_k - \mu_k + \mu_{k-1})/K_i \quad (3)$$

2. Through pair-wise communications, each tile identifies the tile scheduled with the critical-path-thread through keeping track of the highest q_{target} across the chip[2].

3. With this information, each tile then re-solves using Equation 1 to determine the new service rates (tile frequency settings), essentially slowing down tiles not executing critical-path-threads.

Figure 4 shows a snippet of execution near the beginning of the code in Figure 2. Equation 1 calculates service rates in terms of 0.0 to 1.0. We then scale μ to frequency; in this example, we shall scale frequency up to 1000 MHz. We see that in tile *T(2)*, the queue occupancy increased as the tile is unable to keep up with the processing requirements, while the reverse is true at *T(17)* and *T(10k)*. Thus we expect to lower the frequency for *T(17)* and *T(10k)*, and increase it in *T(2)*. Equation 2 estimates the q_{target}s seen for each tile, and as we expect, tile *T(2)* which is experiencing the highest load will have the highest q_{target}.

Thus each tile will select *T(2)*'s q_{target} and use this as the new q_{ref} to re-solve the other tiles using Equation 1. Mathematically *T(2)* must equal the maximum frequency (1000), running the tile at its highest frequency on our identified critical path. In all other tiles, the equation will produce a frequency lower than maximum, as their original q_{target}s are lower than q_{ref}.[3]

It is easy to see how this example holds up width-wise; that is, 4 or more child threads of *MT*. Adding a fourth thread, such as *T(250)*, would simply have the system consider a fourth q_{target}, and the system would remain stable with no impact on performance. However, consider where *MT* is the child thread of another thread, *MT+1*, i.e. it's spawned by thread *MT+1*. In order for *MT+1* to set its frequency, it must receive q_{target} from *MT* and *MT*'s siblings.

Here, the q_{target} passed to *MT+1* must not be *MT*'s, but rather *T(2)*'s, i.e. each thread passes to its parent the highest q_{target} it sees among its siblings. By induction one can see that this holds all the way up to the original parent thread. Thus, one can trace the critical path through pair-wise parent-child communications.

Once the maximum q_{target} is derived, it is then disseminated from each parent to its children. As Section 3.2 shows, the communication overhead is very low with realistic DVFS intervals of at least 50,000 cycles.

3.1 Setting of stability constants K_i and K_p

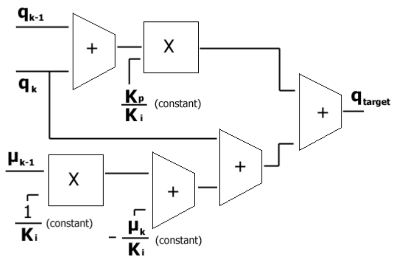
From [13, 14, 17] we set the stability constants K_i and K_p to produce a maximum swing of 4-8% in frequency per interval. To achieve this, we assume that the load factor, or difference between (q_k - q_{ref}) and (q_k - q_{k-1}) is typically between 0 and 1000. In our case the upper limit is not bounded; the controller can handle any value for load factor. This is particularly important because a compiler or programmer cannot always guarantee that the load factor can be capped at 1000. Using the stability analysis in [17], Equation 7, we choose stability constants of K_i=0.3 and K_p=0.1. These are smaller than the default stability constants of K_i=0.6 and K_p=0.2 because in general the work-

[2]In applications where there is no feedback loop, there must always exist a "master thread" with no input queue, which simply produces data for the others. In order to preserve performance, we assume it has an infinite length input queue and thus always runs at maximum speed.

[3]If this derived frequency is lower than the lowest processor frequency setting, that lowest setting will be selected.

load variation in CMPs tends to be shorter and sharper compared to MCDs, and thus we need to underdamp (relatively) the response.

3.2 Hardware implementation issues



q_{k-1}
q_k
+
X
$\frac{K_p}{K_i}$ (constant)
q_{target}
+
μ_{k-1}
X
+
$\frac{1}{K_i}$ (constant)
$-\frac{\mu_k}{K_i}$ (constant)

Figure 5. Proposed additional hardware needed to calculate q_{target}

Little additional low-level hardware is needed to implement *dist-PID*. Figure 5 shows the hardware needed to calculate q_{target}. This is in addition to the hardware required for *local-PID*, which is a 16-bit counter, a 30-bit accumulator, two adders, and two multipliers.

To build the q_{target} hardware, we distribute K_i such that:

$$q_{target} = K_p/K_i * (q_k - q_{k-1}) + q_k + (-\mu_k + \mu_{k-1})/K_i$$

The multiplications cannot be done by lookup table, as in *local-PID* as our range is essentially unlimited. However, K_p/K_i is fixed, 3 in *dist-PID*, and can be done with a shift and a fixed value add. Also a constant is $-\mu_k/K_i$, and can be pre-calculated to save a multiplication. Thus, all we need is four adders and two multipliers. Since q_{target} and μ_k are calculated in different steps, multiplexers, adders, and multipliers can be shared, which brings the total to a 16-bit counter, a 30-bit accumulator, four adders, and two multipliers.

The second implementation overhead concerns the communication bandwidth used in the distribution and derivation of q_{target}. By using pair-wise communications between tiles housing parent-child threads to percolate q_{target}, the overhead can be reduced to just the sending of two floats per tile per statistics interval, which is a low 256 bits in 2,500 cycles in our assumed system. Alternatively, the distribution can be simply pre-orchestrated, such that tiles on the outside send to a tile on the inside, who will select the highest q_{target}, and forward it to its interior recipient. The resultant q_{ref} will be passed downward along the same path[4]. In a 4x4 tile CMP, for example, the maximum latency will be eight hops round-trip, which will ensure up-to-date information in the face of the 2,500-cycle statistics interval.

3.3 Estimation of queue occupancies q_i in a CMP

Our technique estimates the execution load on a tile in a CMP through monitoring the threads in the task queue of a tile. Each thread is associated with a load factor (a number ranging from 0 to 1000), provided by the compiler or programmer. Queue occupancies at each tile are then the summation of the load factors of all threads in the tile's task queue.

This load factor allows the software some control over the performance of the code. In *local-PID*, q_{ref} is used to allow the software to dictate the general performance objective—a higher q_{ref} saves more energy, a lower one preserves higher performance. In the same way, the load factor for each thread allows the software some control over the hardware power management of the CMP. Higher load factors instruct the hardware to be more aggressive in saving energy, while a lower setting aims to maximize performance.

[4]Common DVFS settings across multiprogrammed workloads are automatically supported with this protocol. However, normally each application only needs to handle its own family of threads. Thus in this case q_{target} must also be accompanied by a thread ID.

Processor Core	
Processor clock	1 GHz, 7 stage pipeline
Issue/Decode/Commit width	2/2/2 instructions per cycle
Branch predictor	4096-entry, 13 bit global history
Functional Units	2 IntALU, 2 IntMult/Div, 1 FPALU, 1 FPMult/Div, 1 MemPort
Memory Hierarchy	
L1 D-cache Size	32KB, 4-way, 32B blocks, 1 cycle latency
L1 I-cache Size	32KB, 4-way, 32B blocks, 1 cycle latency
L2	None
Memory	20 cycles
Network	
Topology	2-dimensional mesh
Channel Width	256-bit
Flit Size	256-bit
Input Buffers	15 flit buffers per port
Output Buffers	1 flit buffer per port
Crossbar	5-by-5
Router Pipeline	3 cycle latency (scouts), 1 cycle latency (others)
Link Traversal	1 cycle latency between hops

Table 1. Architectural parameters.

The load factor can be estimated in several ways. The first is through profiling. While profiling can be quite accurate, since the load factor needs only be a relative measure of load across threads, we can estimate it using simpler, static measures. For instance, the size or number of input arguments often serves as a good proxy for execution time. We show in Section 5 how *dist-PID* remains resilient even with simple load factors that approximate actual execution time.

4. Methodology

4.1 Simulation setup

To evaluate *dist-PID* we implemented our controller on an event-based simulator modeling a 16-tile chip multiprocessor with an integrated network on-chip. Each model is based on XTREM [4], a validated SimpleScalar-ARM [2] simulator with additional modifications done to support multiprocessing and networking. The architectural parameters are listed in Table 1.

Threads are scheduled using a simple heuristic-based policy; first try to schedule onto any free processor in the system. If none are available, schedule the thread onto the processor with the lightest load. Clearly, more sophisticated scheduling policies may lead to higher performance. However, as DVFS is essentially reactive, trying to save energy only after threads have been scheduled, the specific scheduling policy used is orthogonal to the DVFS policy.

Distribution of $q_{target}s$ is done via the on-chip network, with statistics transmission prioritized. Statistics on queue occupancy and load factor are sent every 2,500 cycles. Next, to determine the DVFS interval, the regular points in time in which we invoke our DVFS algorithms, from [18] we see that the "optimal" interval lies between 1,000 and 5,000 cycles. For our machine, we cannot afford to have intervals so long that the behavior of very fast moving threads is lost in the aggregation; thus we want to choose a DVFS interval that can handle the fastest possible input rate into our machine. From micro-benchmark tests which we used to measure thread setup and launch time in our machine, we estimate that the fastest input rate into a tile will be around 3,000 cycles, or more than 30 times the rate for the INT domain in [17]. Thus we believe the lower bound for the DVFS interval to be 30,000 cycles, and the upper bound in the range of 150,000 cycles.

For the individual processors we assume ARM-like processors, and our assumptions for DVFS transition and setup delays are the same as in [17], or 73.3ns/MHz and 171ns/2.86mV. Dynamic power numbers for the CMP are obtained using Wattch [1] for the processor and memory components and Orion [16] for the network, running at a nominal voltage of 2.08V. The individual processors can select frequencies between 100 MHz (0.45V) and 1 GHz (2.08V). Voltages and frequencies are derived from the equation $f = ((V - V_t)^{\alpha})/V$, where XScale values are used for V_t (0.45) and α (1.5).

4.2 Benchmarks

We coded five multi-threaded benchmarks to evaluate *dist-PID*. Two are kernels that are aggressively multi-threaded to tax the proposed distributed DVFS technique (recursive quicksort and Othello, a game-playing algorithm. Three others are SPEC [15] benchmarks (equake, twolf, and mcf) that are hand-partitioned. The load factors here are empirically determined. As mentioned before, the load factor need only be an approximation—we show in Section 5 that *dist-PID* is resilient to major variations in the load factor, whereas *local-PID* is not.

quicksort: The multithreaded quicksort consists of threads created at each recursive invocation. Compared to the other four benchmarks, quicksort puts very high thread pressure on the system, averaging 288 cycles per thread, versus 5788 cycles on average for the other four. This creates large variance in the scheduling of threads across the tiles, requiring a nimble DVFS policy that can respond quickly as the workload on tiles vary. The load factor is based on the number of items to be sorted.

Othello: Othello is an application of minimax tree search in a classic game—each player's move involves searching the tree and selecting the best move. Each move spawns a new thread. In Othello, each move can create six or more new threads, which creates significant spatial and temporal variance in the load on each tile. As each move has roughly comparable (order of magnitude) computational complexity, the load factor is predefined at 250.

183.equake: Equake has two parallelized loops: the first is the "simulation loop" which calculates the matrix K, and the second in the "time integration loop" which is a sparse matrix access loop over K. Threads synchronize between loops. Like Othello, each thread has roughly comparable computational complexity, and the load factor is hence preset at 50 for each thread.

181.mcf: This benchmark is a specialized instance of a network minimum-cost flow problem. Due to a large shared data structure, there is significant execution variability as a result of variable-latency memory operations. We target the kernel portion of 181.mcf (function `refresh_potential`), assigning one thread to each node in the tree. The tree has arbitrary number of children, depending on the input data set, but in practice, is quite bimodal—either very small or very large. After each phase of computing the `potential` value, a new tree is re-generated, with tree size shrinking at each phase. The load factor is set by profiling the tree traversal; if there exists a grandchild, the load factor is set at 750. For all others the load factor is 0.

300.twolf: Twolf does placement and global routing. Its kernel (function `new_dbox_a`) that we target consists of a linked list traversal, where each flow does the computation for a single node (which corresponds to another list traversal). Here, we apply decoupled software pipelining [12] to pipeline-parallelize the program, so producer threads prefetch nodes from memory, while consumer threads perform computation. The load factor is set by profiling the linked list traversal, and multiplying the length by 100.

5. Results

To evaluate the efficiency of our proposed scheme, we simulated our benchmarks under a number of conditions. We evaluated three schemes, **baseline**, *local-PID*, and *dist-PID*. The baseline scheme is a processor with no DVFS, but "tile-gates" a tile when all the threads in a tile's task queue are stalled (waiting for other threads). When a tile has no threads, it is also tile-gated, and thus consumes no dynamic power. The interval length is 50,000 cycles, more of which will be discussed with Figure 7. *Local-PID* uses a q_{ref} of 300, or one-third of the maximum expected queue occupancy.

5.1 Energy-delay-product savings

Figure 6 shows the energy-delay product for the five benchmarks. Overall the local scheme had 85.5% of the energy consumption of the baseline, but increased execution time by 6.9%, producing an energy-delay value of 97.3%. This supports our motivation that having a fixed q_{ref} independent of the system state is not the best solution; from observation gains from one tile are canceled out by losses from another. By comparison, our proposed scheme saved 20% of the energy but increased execution time by only 5.6%, giving us an energy-delay product of 84.55%.

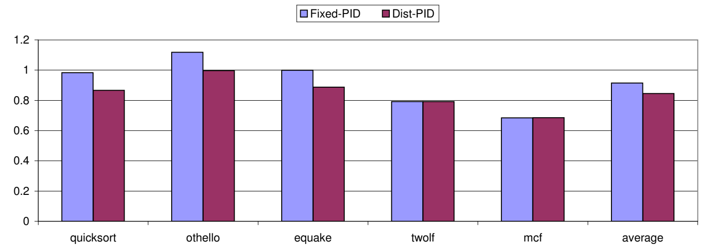


Figure 6. Normalized energy-delay product.

Othello in particular is problematic for both schemes. Othello tends to have long running threads punctuated by short outbursts where it launches many threads at once; this made it difficult for our *dist-PID* scheme to figure out the critical path, and track the oscillations faithfully. Nevertheless, *dist-PID* still outperforms *Local-PID*. *Dist-PID* saved 8% energy at the cost of 8.4% performance, whereas *Local-PID* saved 14% energy, but increased run-time by 30%.

Quicksort on the other hand maps well to *dist-PID*, as the use of the size of the input arguments as the load factor proved a good proxy for computation complexity, allowing *dist-PID* to precisely identify the critical path and slow non-critical-path threads appropriately, leading to a larger energy-delay-product savings of 86.6% as compared to *Local-PID*'s energy-delay-product of 98.3%.

As the threads of equake are fairly balanced in terms of execution complexity, the opportunities for DVFS occur largely when the parallel sections begin to wind down. *Dist-PID* outperforms *local-PID*, as q_{target}s communicate information about the other tiles that are nearing completion, while *Local-PID* continues to try to match its queue occupancy to q_{ref}. *Local-PID* is virtually the same as the baseline, whereas *Dist-PID* decreased energy consumption by 14.7% while increasing performance by only 4.1%.

Both schemes performed equally well for twolf; As twolf generates 218,000 threads with significant variation in execution times. With that kind of sustained input rate, both schemes are able to find many places to save energy. With mcf, threads typically either perform very shallow or very deep traversals of tree. In twolf the critical paths are very short, whereas in mcf the critical path is very obvious but very long. Thus both schemes were easily able to identify which tiles to slow down and save energy.

5.2 Effect of DVFS interval

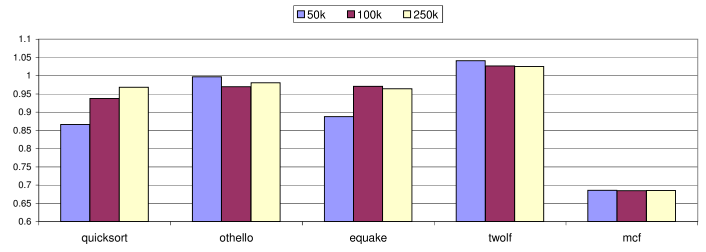


Figure 7. Energy-delay product as a result of varying the DVFS interval.

Figure 7 shows the result of varying the DVFS interval for our proposed scheme. Shorter intervals, in general, lead to better energy-delay-product, with the exception of Othello. For Othello the short interval is actually worse; longer intervals captured Othello's behavior better and performed better. This is because Othello's long-running threads require much longer DVFS intervals—short intervals tend to over-emphasize the small bursts in behavior. *Local-PID* suffers particularly from this behavior.

Quickly evolving codes like quicksort suffered from longer intervals as longer intervals hamper quick responses to changes in workloads. Equake, while much more static, suffered from slow reaction times, particularly with equake where the potential needed to be capitalized immediately.

5.3 Stability of DVFS control

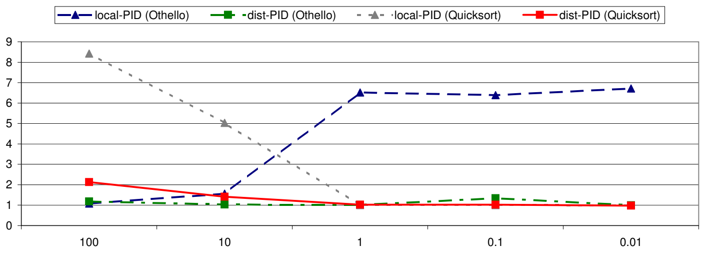


Figure 8. Run-time as a result of varying the load factor for both local-PID and dist-PID.

Figure 8 shows the execution time (normalized against that of baseline) after varying the load factor of Othello and quicksort from 100x to 0.01x for both schemes. The *local-PID* simulations are indicated with triangular markers whereas the *dist-PID* ones are indicated with square markers.

We see that for both applications, *dist-PID* is relatively resilient toward load factor variation, remaining stable. *Local-PID*, however, is quite fragile. Unlike *dist-PID*, for Othello the performance of *local-PID* jumps quite severely at some point, which is when the load factor crosses over the default q_{ref}. We see that once the load factor consistently stays over the default q_{ref}, *local-PID* tries to preserve performance. If it is below the default q_{ref}, *local-PID* tries to save energy, without regard to the overall program state. The crossover point is not always foreseeable and not particularly predictable—here, it is very sudden and without indication, rocketing up to 6 times the normal execution time to normal. Thus we see that distributed, coordinated control not only improves EDP, but ensures stability due to inter-tile coordination.

For quicksort, *local-PID* performs well when the load factors are overestimated—for example, when the element size to be sorted is larger than what quicksort was tuned for. This tells the *local-PID* controller that we underestimated processing requirements, and thus to try to preserve performance. When the load factors are underestimated, however, performance skyrockets—much more markedly than in *dist-PID*. This leaves the programmer and compiler in an interestingly quandary. If the load factor calculation is tuned for a large quicksort, we risk the possibility that a smaller quicksort will take a massive hit in performance. Common sense would thus be to lean toward tuning for smaller quicksorts. But taken to the extreme, only very small quicksorts would be eligible for energy savings, and then we miss the point of DVFS. Again, our results show a compelling reason to have adaptable, coordinated, run-time DVFS.

6. Conclusions and Future Work

We have shown that distributed, coordinated DVFS control is necessary to overcome the possibly counter-acting DVFS actions of local DVFS. Our proposal of *dist-PID* is shown to boost energy-performance on CMPs while ensuring the stability of DVFS control. Compared to *local-PID*, it achieves up to 8.8X improvement in EDP on benchmarks with substantial variance across the chip. We also show how *local-PID* can oscillate substantially for certain benchmarks, while our proposed *dist-PID* ensures stability.

One way to improve upon *dist-PID* is to look at ways of dynamically tracking the performance of threads and to recalibrate the load factor based on this information. While *dist-PID* can handle reasonable inaccuracy in estimating the load factor, this additional ability would allow it to recover from completely incorrect load factor estimates.

As CMPs continue to be proposed and implemented, we believe that the techniques described in this paper can be used to extend online, formal approaches to DVFS from the uniprocessor realm into CMPs. Our approach is lightweight, requiring little extra hardware, and distributed, requiring little extra communication bandwidth. Overall, we believe our approach to be an effective way of improving DVFS for CMPs.

7. References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architecture-Level Power Analysis and Optimizations. In *In Proceedings of ISCA27*, ISCA 2000.

[2] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, pages 13–25, 1997.

[3] L. Clark. Circuit design of XScale (tm) microprocessors. *2001 Symposium on VLSI Circuits*, 2001.

[4] G. Contreras et al. XTREM: XScale technology research for energy modeling. *Intel Internship Report*, 2003.

[5] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *In Proceedings of PLDI-2003*, 2003.

[6] Intel Corp. The Intel XScale Processor Architecture. *http://developer.intel.com/intelxscale*, 2002.

[7] Intel Corp. The Intel Pentium M Processor. *http://www.intel.com/design/mobile/pentiumm/documentation.htm*, 2004.

[8] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. *In Proceedings of ISCA-29*, 2002.

[9] J. Lorch and A. Smith. Improving dynamic voltage scaling algorithm with PACE. *Proceedings of SIGMETRICS-2001*, 2001.

[10] G. Magklis, M. Scott, G. Semeraro, D. Ablonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *In Proceedings of ISCA-30*, 2003.

[11] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. *Workshop on Complexity Effective Design*, 2000.

[12] R. Rangan, N. Vachharajani, M. Vachharajani, and D. August. Decoupled software pipelining with the synchronization array. In *In Proceedings of PACT-13*, 2004.

[13] G. Semeraro, D. Albonesi, S. Dropsho, G. Maklis, S. Dwarkadas, and M. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. *In Proceedings of MICRO-35*, 2002.

[14] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. *In Proceedings of HPCA-8*, 2002.

[15] The Standard Performance Evaluation Corporation. WWW Site. http://www.spec.org, Dec. 2000.

[16] H.-S. Wang et al. Orion: A power-performance simulator for interconnection networks. *"In Proceedings of MICRO-35"*, 2002.

[17] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[18] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Event driven voltage and frequency control in multiple clock domain microprocessors. *In Proceedings of HPCA-11*, 2005.

[19] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. *In Proceedings of PLDI-2003*, 2003.