Petri-Net based Performance Modeling for Effective DVFS for Multithreaded Programs

Arun R and Y. N. Srikant

IISc-CSA-TR-2009-12
http://archive.csa.iisc.ernet.in/TR/2009/12/

Computer Science and Automation
Indian Institute of Science, India

November 2009
Updated: July 2011

Petri-Net based Performance Modeling for Effective DVFS for Multithreaded Programs

Arun R and Y. N. Srikant

Abstract

Chip Multiprocessors (CMP's) have made their way into servers, desktop systems and even embedded systems. While power dissipation limits performance of CMP's and affects system reliability, energy efficiency is becoming increasingly important for bringing down the significant contribution of energy in organizations' budgets and the environment, besides enhancing battery life. Dynamic Voltage and Frequency Scaling (DVFS) is a very effective tool for designing tradeoffs involving power, energy, temperature and performance.

It is important that the potential benefits DVFS offers are realized. To that end, in this report, we present a novel software based DVFS method, which uses a formal Petri net based program performance model to find energy efficient voltage-frequency settings. We evaluate this, and a state-of-the-art hardware based DVFS method with two important classes of parallel applications: data parallel (SPMD style) applications, and stream multithreaded applications.

From our evaluation, we find that the Petri net performance model based software method for DVFS achieves significant $Energy/Throughput^2$ (ET^{-2}) improvements for both classes of multithreaded applications. The hardware based method achieves high ET^{-2} improvements for SPMD applications, but performs poorly in the case of stream applications. For stream applications, a simple linear interpolation based DVFS method achieves all the benefits of the Petri net based method. We also observe that support for independent voltage and frequency control for each core is essential for obtaining high ET^{-2} improvement for stream applications.

1 Introduction

Chip Multiprocessors (CMPs) continue the legacy of the potential for increasing performance for newer processor generations. While multithreaded

programs directly benefit from CMPs, single threaded programs can benefit by utilizing the larger caches. CMPs have also been used for improving system throughput. Moore's law and increasing performance demands have helped them make their way into servers, desktop machines and even embedded systems.

The advent of CMPs has renewed the interest in parallel programming, and encouraged the design of new programming languages and paradigms as well. Several scientific applications have been the targets for conventional parallel computing systems, as they exhibit SPMD (Single Program Multiple Data) parallelism (also called as data parallelism) in which the same set of operations are performed on different data elements. OpenMP [1] compiler directives can be used to parallelize sequential programs without much difficulty, for applications exhibiting SPMD parallelism. Many modern emerging applications exhibit task and pipeline parallelism as well. Stream Programming offers a convenient way to express parallelism for such applications [5].

In this work, we consider these two important classes of applications. Most of the implementations of data parallel applications we consider in this work, which include SPECOMP [3] and NAS parallel benchmarks [2], use OpenMP directives for parallelization. For stream applications, we consider implementations in the streamit language [5].

1.1 OpenMP Parallel Directives

OpenMP is a set of compiler directives and libraries used for writing explicitly parallel programs [1]. The directives are defined for C, C++ and fortran, and many modern compilers support them.

1.1.1 Programming Model:

OpenMP uses the fork-join parallel programming model, where a master thread creates a set of threads when needed. It encourages the evolution of parallel programs from sequential programs. For creating threads, it supports constructs that can be classified as

- **SPMD based:** The 'parallel' and 'for' constructs essentially allow the execution of the same code by different threads. The 'for' construct relieves the programmer from distributing the iterations of the loop among different threads.
- **OpenMP Section based:** This construct allows different threads to execute different code segments.

1.1.2 Data Sharing:

OpenMP is based on shared memory programming model, where threads communicate among themselves using data shared among them. By default, global and static file scope variables are shared. Automatic variables, and stack variables of functions called from parallel regions are private.

This default categorization can be overridden by explicit specification using the *shared, private, firstprivate, lastprivate, threadprivate* clauses. While the *firstprivate* specification for a variable allows initialization of the variable from the value of the corresponding variable of the master thread, *lastprivate* updates the value of a global variable with the value from the last iteration of the for loop associated with the specification. The *threadprivate* clause makes a global variable private to each thread. OpenMP supports special clauses for reduction operations.

1.1.3 Synchronization:

In addition to the implicit barriers at the end of 'parallel for' blocks, OpenMP provides the following synchronization constructs:

- *critical section:* Only one thread can execute the code enclosed between *critical* and *end critical* constructs
- *atomic*: Similar to *critical section*, but for a single statement
- *barrier:* Construct for explicit barrier synchronization
- *flush:* All memory operations before the *flush* must finish before any new memory operation after *flush* begins; Variables in registers must be updated in the memory
- *master:* Only the master executes the block. There is no implied barrier after this construct.
- *single:* Only a single thread executes the specified block. A barrier and flush are implied at the end of the block.
- *ordered:* It ensures that the program block within the construct is executed in sequential order

1.1.4 Library Functions:

OpenMP also supports different library routines for managing locks, controlling the degree of parallelism, and to determine the number of processors in the system.

The following is a simple c program computing dot product of a vector and itself. It demonstrates *reduction* and *for* clauses. There are implicit barriers after the parallel for loops. For the reduction operation, a local copy of *dSum* is created for each thread, and initialized to 0, since the operator is *+*.

```
#include <omp.h>
#include <stdio.h>

#define ARRAY_SIZE 400000000
double d[ARRAY_SIZE];

int main() {
  double dSum = 0.0, dProd;
  int i;
#pragma omp parallel for
  for (i = 0; i < ARRAY_SIZE; i++) {
    d[i] = i;
  }
#pragma omp parallel for
  reduction(+:dSum) private(dProd)
  for (i = 0; i < ARRAY_SIZE; i++) {
  dProd = d[i] * d[i];
    dSum += dProd;
  }
  printf("%g\n", dSum);
}
```

1.2 Stream Programming using Streamit

Stream Programming helps express more general forms of parallelism such as task and pipeline parallelism [5]. Many applications in the audio, video, and digital signal processing, encryption and decryption, and networking domains can be conveniently programmed explicitly parallel, using this model. Infact, these applications, to a large extent, have motivated the design of the Stream Programming paradigm. These applications exhibit task, data and pipeline parallelism, and Stream Programming languages such as Streamit provide language constructs that help programmers express parallelism easily.

In Streamit, a program consists of a set of autonomous actors, called filters, executing some action, termed *work* repeatedly. These filters communicate explicitly through communication channels, and the entire program

can be viewed as a stream graph. The amount of data a filter produces or consumes for performing its *work* is fixed, and is specified when the filter is defined.

For exposing task level parallelism, the SplitJoin construct can be used. In this construct, independent parallel streams originate from a common splitter filter, and merge into a common joiner filter. Actors can also be connected together in a sequence, and this exposes pipeline parallelism. A filter exhibits data parallelism, if there is no dependence between the execution of one instance of its *work* and any other instance, and this can be easily detected by a compiler.

While workload partitioning, load balancing and communication scheduling are hard challenges for the compiler, they can be more effectively solved than the problem of extracting parallelism from sequential programs - Infact, previous work [4] has shown that a sophisticated compiler has the potential to fetch impressive speedup benefits for Streamit programs.

1.3 Challenges for Performance in CMPs:

While CMPs carry a huge potential for achieving high performance for several classes of applications, they also present some challenges in realizing the potential.

Power dissipation has become a first class constraint in the design of modern processors. Temperature, along with power, limits performance of CMP's, and affects system reliability. For servers and data centers, they determine the cooling cost; for mobile systems, they check the continuous usage of these systems. Energy efficiency is critical for increasing battery life for embedded systems. It is becoming increasingly relevant for servers and workstations as well, for bringing down the significant contribution of energy in organizations' budgets, and for the environment.

1.4 Dynamic Voltage and Frequency Scaling (DVFS)

Dynamic Voltage and Frequency Scaling (DVFS) is a very effective tool for designing tradeoffs involving power, temperature, energy and performance [8–10, 14]. Several state-of-the-art CMPs available in the market [11, 12] support DVFS, but most of them require that all cores operate at the same voltage. DVFS has been used at the hardware, operating system and application level to achieve different goals such as optimizing system ET^{-2}, power constrained throughput optimization and temperature constrained power control.

In this report, we present a novel software based DVFS method, which uses a formal Petri net based program performance model to find energy efficient voltage-frequency settings. We evaluate this method with SPMD applications which exhibit data parallelism, and stream applications, which exhibit task and pipeline parallelism as well. We also share our experiences with DVFS when the number of threads is more than the number of cores. We choose to compare the DVFS techniques using the ET^{-2} metric (as in [6]), since it is a voltage invariant power-performance metric [45].

Our main contributions are

- We build a formal Petri net based program performance model for a CMP system, parameterized by architectural settings, resource configurations, and a memory system that services requests from multiple threads
- We use this model to find energy efficient DVFS settings for different classes of multithreaded applications
- Our evaluation finds several interesting results:
  - the Petri net based method achieves significant ET^{-2} improvements for both SPMD parallel and stream applications
  - the best performing hardware based DVFS controller in a recent evaluation [6] performs well for SPMD parallel applications, but achieves little ET^{-2} improvements for stream applications
  - a simple linear interpolation based scheme applicable for the stream applications achieves all the benefits of the Petri net based method
  - for stream applications, per-core voltage/frequency control is very important for achieving high ET^{-2} improvements for little performance degradation for applications exhibiting different forms of parallelism
- We have significantly advanced the state-of-the-art of m5sim, a popular multicore full system simulator [15], integrating power models, support for DVS, and software thread-specific path profiling of binaries. To this end, we have also designed and implemented an intermediate representation for binaries, that aids in the analysis of binaries at different levels: instruction, basic block, and control flow graphs. This intermediate representation is used by our software based voltage-frequency selection module for finding energy efficient voltage-frequency settings.

We have used the m5sim simulator [15] in full system mode for our experiments.

The rest of the report is organized as follows. We characterize program execution in a processor, and introduce Petri net based performance modeling in Section 2. We explain how Petri net based performance modeling can be used for finding energy efficient voltage/frequency settings, in Section 3. Section 3 also briefly explains the state-of-the-art hardware based controller for DVFS, which was originally proposed in [7] and found to be the most effective DVFS method in a recent evaluation of different DVFS controllers [6]. We describe our experimental setup, including our enhancements to the m5sim simulator [15], in Section 4. In Sections 5 and 6, we discuss our experimental results for SPMD and Stream programs, respectively. Section 7 provides information about the related work, and we conclude in Section 8, with pointers for future directions.

2 Petri net based Performance Modeling

Petri nets are natural abstractions for modeling and evaluating performance of programs. Using their expressiveness, we can capture both the properties of a program and the properties of the system in which the program executes. They have been used for finding energy efficient frequency (voltage) settings in [13] in the context of sequential programs running on a single core MCD processor, besides finding kernels in software pipelining [17].

We first define Petri nets, how they are constructed, explaining the intuition behind using Petri nets.

2.1 Petri nets: A Short Introduction

Formally, a Petri net is a three tuple (P,T,A), where P is the set $\{p_1, p_2, \ldots p_n\}$ of places, T is the set $\{t_1, t_2, \ldots t_m\}$ of transitions. Both P and T are non-empty, and disjoint. A is the multiset (bag) of arcs that connect places and transitions.

Places can hold *tokens*. A marking is a function $M : P- > I$, where I is the set of non-negative integers, indicating the number of *tokens*, a place has. A transition can *fire* if all places incident into it have a non-zero token count.

When a transition fires, the token count of each place incident into it is decremented by one, and the token count of each place incident out of it is incremented by one. A timed Petri net is a tuple (PN, Ω) where PN is a

Petri net and $\Omega : T -> \Re^+$ assigns the time required by each transition t_i in PN to fire ($\Re^+$ is the set of positive reals).

We construct the timed Petri net model from a detailed precedence graph that captures the precedence constraints characterizing program execution, and the system resource specification.

The precedence graph is based on [16], but it models the constraints in our simulated system. Nodes of the graph represent pipeline stages, and edges represent precedence relation among them. Edges have labels that correspond to pipeline stage latencies.

From the precedence graph and the resource specification, the Petri net can be constructed [17] by

- creating a transition t_i for each node i of the graph
- creating a place $p_{i,j}$ for edge (node i, node j) of the graph; the place has t_i as an input transition and t_j as an output transition
- creating a place for each resource type; a transition using that resource is both an input and output transition for that place; the token count is initialized to the number of instances of that resource

Our construction of Petri nets from precedence and resource constraints is based on [17]. In the Petri nets that we construct, transition firing times are determined by functional unit or pipeline stage latencies. For transitions corresponding to memory access stage of memory access instructions, firing times are determined by latencies of caches and memory, and whether a particular access hits or misses in the cache, and can therefore be theoretically modeled as a random variable (and only in this case - for memory access transitions - the mapping Ω is from a transition to a random variable, instead of a fixed real [48]). Also when multiple transitions can fire at the same time, the transition that corresponds to the earlier dynamic instruction in program order gets priority.

For a Petri net model constructed like this for a program loop and simulated, the rate of firing of transitions of the Petri net corresponds to the initiation interval (II) [17] of the loop.

In the rest of this section, we will illustrate the use of Petri nets in estimating the average execution time of serial and parallel versions of a loop, and conclude how this, in turn, can be used to find energy efficient frequency (voltage) settings. In this process, we also highlight how CMP's increase the pressure on memory system.

2.2 Performance Estimation using Petri Nets

We consider the small program loop in Figure 1 to illustrate performance estimation using Petri nets. The program computes the sum of square roots of doubles in an array.

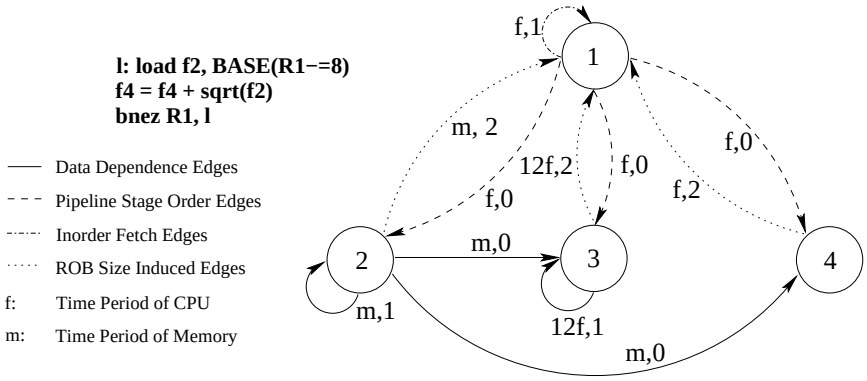

Figure 1: A Loop and its Precedence Graph

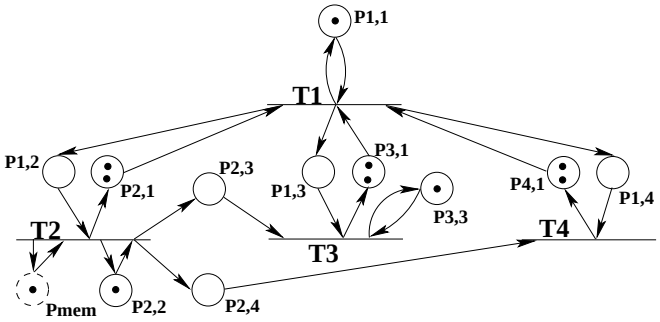

Figure 2: Petri net with Initial Marking for the Graph in Figure 1

The load instruction loads the double in the address Base + R1 to F2, and post decrements R1 by 8. There is a single instruction, that computes the square root of the source operand (F2) and adds it to F4. The loop continues to execute till R1 becomes 0.

We make several simplifying assumptions to make manual simulation of the Petri net easy. We assume that the target processor can fetch 3 instructions per cycle, has a 6 entry reorder buffer (ROB), and predicts the branch perfectly. The latency of the square-root-add operation is 12 cycles. There

is no data cache, and the memory system takes 6 cycles, on an average, to respond. The post decremented value is visible when the load finishes. The number of physical registers is large, and does not limit the throughput of the loop.

The nodes of the graph in Figure 1 represent the two pipeline stages (fetch and execute) in the processor. We show only one common fetch stage (Node 1) for all the 3 instructions. Nodes 2, 3 and 4 show the execute stages for the load, square-root-add, and the branch instructions respectively. The graph shows four different types of precedence edges: data dependence, pipeline stage flow, in order fetch and ROB size induced edges.

A label of edge (m, n) is an (l, d) tuple: It means that the $(i + d)^{th}$ iteration of n can begin, l units of time after the i^{th} iteration of m has been initiated. For instance, the edge from node 3 to node 1 has the label $(12f, 2)$. It means, for instance, that the third iteration of node 1 can be initiated only after $12 * f$ units of time have elapsed following the initiation of the first iteration of node 3. In Figure 1, f denotes the time period of the CPU, and m, the time period of memory.

Figure 2 shows the Petri net for Figure 1, following the rules for deriving the Petri net based on precedence graph and resource specification, in Section 2.1. The dashed circle around the place P_{mem} signifies that it is a place corresponding to the resource constraint. We assume that there are sufficient instances of other resources, that they can be ignored while modeling.

Table 1 shows the firing of transitions when the loop is run on a single core system.

In Table 1, we can observe the repeating sequence of firing of transitions starting from cycle 19. The bottleneck operation is the sqrt-add operation (Transition T3), and the loop takes 12 cycles on an average, for 1 iteration (12 cycles is the latency of T3).

Table 2 shows the firing of transitions in a core, when the loop is run on a 4-core CMP system. For simplicity, we assume that the memory system is available only for 25% of the time for a core, because of the requests from the other 3 cores, for this example. In practice, the availability of the memory system depends on the distribution of accesses to different banks, and the architecture of the memory system. The details of modeling requests to different banks are explained in Section 3.

We can observe the repeating sequence of firing of transitions starting from cycle 19. Here, the bottleneck becomes the memory system, and one iteration of the loop takes 24 cycles on an average - this is because, the cores can generate one request every 12 cycle (determined by the sqrt-add operation), but the memory system can service only one request of a core every 24 cycles.

Table 3 shows the firing of transitions in a core, when the loop is run on a 4 core CMP system, with the cores operating at half the frequency, when compared to the ones corresponding to Table 2. In this case, the rate at which each core generates a request (1 request in 24 (unscaled) cycles) naturally matches the rate at which the memory system can service the request of a core (1 in 24 (unscaled) cycles). The loop completes 1 iteration every 24 (unscaled) cycles, even with reduced frequency.

Time	Fired	Comments
0	T1	
1	T1,T2	T2 @ 7
7	T2, T3, T4	T2 @ 13, T3 @ 19
13	T4	
19	T1, T3	T3 @ 31
20	T2	T2 @ 26
26	T4	
31	T1, T3	T3 @ 43
32	T2	T2 @ 38
38	T4	

Table 1: Firing of Petri net in Figure 2 in a Single Processor System

Time	Fired	Comments
0	T1	
1	T1, T2	T2 @ 7; Mem @ 25
7	T3, T4	T3 @ 19
19	T1	
25	T2	T2 @ 31; Mem @ 49
31	T3, T4	T3 @ 43
43	T1	
49	T2	T2 @ 55; Mem @ 73
55	T3, T4	T3 @ 67

Table 2: Firing of Petri net in Figure 2 in a 4-core CMP System

Time	Fired	Comments
0	T1	
2	T1, T2	T2 @ 8; Mem @ 26
8	T3, T4	T3 @ 32
26	T2	T2 @ 32 Mem @ 50
32	T1, T3, T4	T3 @ 56
50	T2	T2 @ 56; Mem @ 74
56	T1, T3, T4	T3 @ 80

Table 3: Firing of Petri net in Figure 2 in a 4-core CMP System : All Cores run at Half of Max. Frequency

This example demonstrates how multiple cores increase the pressure on the memory system, and how Petri nets can be used for modeling performance.

We capture the increased pressure on the memory sub-system due to the activities of concurrent threads in our Petri net model. This is very important, because ignoring the memory sub-system can result in highly conservative voltage (frequency) settings, particularly when it is the bottleneck [13]. Our voltage (frequency) selection procedure (a binary search over available frequencies) uses the Petri net model to evaluate the performance impact of different frequency settings and chooses the least frequency setting that satisfies the performance constraints.

We elaborate on the Petri net model and the frequency (voltage) selection procedure used in Section 3.

3 Frequency Selection Methods

In this section, we will have a detailed look at the Petri net performance model based software scheme, and the Greedy Controller, the best performing hardware DVFS controller in [6].

3.1 Petri net Performance model based Method:

In this method, we first identify program segments (called *regions*) for which DVFS can be applied. We then find energy efficient voltage (frequency) settings for the identified regions, and then insert voltage (frequency) reconfiguration instructions at the boundaries of the chosen program segments.

We analyze application binaries for identifying program regions, and for choosing frequency settings. The output of our method is a file that lists the reconfiguration points (Program Counters) and the corresponding voltage (frequency) settings. The m5sim simulator is enhanced to interpret this and implement the reconfigurations.

We identify DVFS-applicable regions for a program based on its structure [10, 13, 18, 19]. For every region identified, different frequency settings are evaluated, and the best setting meeting the performance constraint is chosen. The evaluation favours lower frequency settings, because they can potentially result in more energy savings. The performance constraint ensures that the energy savings is not at the cost of an unacceptable performance loss (A huge loss in performance can also cost us in energy due to leakage energy consumption).

To evaluate the impact of a frequency setting on program performance, the Petri net based program performance model is used. With the performance model, we find the lowest frequency setting that meets the performance constraint using binary search for each identified region, and choose

it as the voltage-frequency setting for it.

Figure 3 gives a high level view of the voltage-frequency selection procedure.

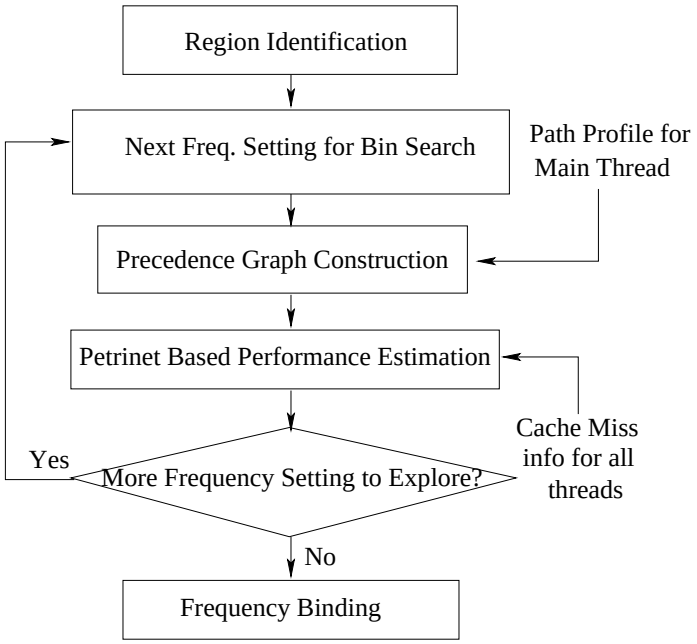

Figure 3: Overview of Frequency/Voltage Selection

Region Identification: A region corresponds to a well defined structure of a program. It is either a natural loop [20] or an acyclic graph whose nodes are regions or basic blocks [10, 18]. Boundaries of regions are potential reconfiguration points. Because there is a finite cost in terms of energy and time to effect a reconfiguration, a region is chosen for voltage (frequency) reconfiguration only if it runs long enough to offset the cost.

As in [13, 18], if at least 10000 instructions are executed on an average in a region each time it is entered, then the region will be chosen for reconfiguration (In our experimental setting (Section 4), it takes about .8 μs to transition from the highest frequency setting to the least setting, in which a CPU core can execute atmost 12800 instructions when operating at full speed.)

Regions can be nested, and in such cases, an outer region is chosen either if

1. no inner region is a reconfiguration point, and the average number of instructions executed in it and its descendants, for each entry into the region is atleast 10000

2. the average number of instructions executed in the basic blocks exclusive to it, and in the descendants that are not chosen for reconfiguration is atleast 10000

These rules help minimizing ineffective reconfigurations (a setting which is overridden by another, before the former takes effect) [13].

Thread-Specific Path Profiling: We use hierarchical Ball Laurus Path Profiling [21, 22] to get information about the frequency of execution of each path in a region, for every function. A path is a sequence of nodes, which can be basic blocks or loops (regions). Regions that are parts of nodes in a path are expanded recursively during Petri net simulation.

From the Hierarchical Path Profile (HPP), we first arrange paths in the descending order of their execution frequencies, and generate instruction traces for these paths, for constructing precedence graphs. To simulate the execution of different paths for outermost regions of a function (whose average trip count is always 1), paths in a region are chosen in a round-robin fashion, whenever the region is simulated. The number of times a path is simulated is determined by the execution frequency of the path, and the average trip count for the region.

Because the HPP subsumes basic block and edge profile, it is also used for identifying long running regions.

The HPP is recorded for each software thread of the application non-intrusively, by making changes to the simulator. Details of this are present in Section 4.1.

Thread-Specific Cache Hit/Miss Profiling:

Misses to memory provide excellent opportunities for scaling down frequencies without sacrificing performance too much. It is critical to get an accurate estimation of misses, because a lower estimate of misses adversely affects competence whereas a higher estimate could inflict severe performance loss.

Currently PDVFS concentrates on misses which occur at regular intervals. Each memory access instruction is associated with a single period of occurence of miss, for each thread. With this information, the Petri net model delays the completion of the firing of the MemAccess transitions of frequently missing memory access instructions, periodically, driven by the Cache Miss profile.

In general, during the program execution, the period of occurence of misses for a memory access instruction could vary. For the Petri net simulation, PDVFS considers only the members of the smallest set S of periods which totally contribute to atleast 80% of the miss periods. We need to classify a memory access instruction as a frequent hit or a frequent miss, and

the period of occurence of miss, for frequently missing accesses. For each miss period $p \in S$, we account for $p-1$ partial hits, and we treat all partial hits as misses. We classify an access as a frequent miss, if the number of misses computed this way is atleast 70% of the total number of accesses. We conservatively set the period of occurence of miss as the largest member in S.

For instance, let $\{M, H, H, H, H, H, H, H\}^{10}$ be the outcome of cache accesses for a load instruction for a particular thread, where M denotes a miss and H denotes a hit. Here, there are 10 misses, 70 hits and the miss period 8 occurs nine times. Therefore this this heuristic will decide that there are $(8-1)*9 = 63$ overlapped hits. So there are 73 misses (including overlapped hits), which is more than 56 (70% of 80) and PDVFS classifies this load instruction as a miss, with period 8 (This means that every eighth instance of the transition corresponding to the memory access stage of the load instruction will have a firing time equal to the sum of memory access and cache access latencies). We found that this heuristic identifies frequently missing instructions with regular strides and miss periods reasonably well.

We record the per thread miss profile for each memory access instruction. We will shortly see how this information is used in Petri net simulation.

Precedence Graph Construction:

The precedence graph captures precedence constraints characterizing instruction execution in a processor. It is based on [16], but it reflects the constraints in the m5sim simulator (simulating Alpha ISA). Nodes in the graph correspond to pipeline stages and edges, the precedence relation among them. Tables 4 and 5 specify the graph. Edges encode the latency of the pipeline stages (nodes) they connect. In Table 5, edges connecting consecutive pipeline stages are not specified. An important difference between our graph and [16] is that certain resource constraints that can not be captured by a graph (eg. finite number of adders) are modeled separately in the Petri net simulation [13].

Node	Microarchitectual Event
Dispatch (D)	Register Renamed and routed to the corresponding issue queues; IQ, ROB, and Reg. Rename Table entries are consumed
Ready (R)	Wait in the queue, for the operands to be available
Execute (E)	A functional unit (FU) is reserved and the instruction gets executed in the corresponding FU
Commit (C)	Free up ROB entry
FreeQEntries (F)	Load, Issue Queue and Rename Table entries are freed
FreeSQEntry (S)	Store Queue Entry is freed, and Cache access is made for Stores

Table 4: Description of Precedence Graph Nodes

Construction of precedence graph is guided by the path profile. From the path profile, we generate instruction traces, resolving conditional branches

Edge Name	**Constraint Modeled**	**Edge**	**Comments**
ID	In Order Dispatch	$D_{i-1} \rightarrow D_i$	
DBW	Dispatch Bandwidth	$D_{i-dbw} \rightarrow D_i$	dbw = maximum num. of insts. that can be dispatched per cycle; lat = 1 cycle
IC	In-order Commit	$C_{i-1} \rightarrow C_i$; $F_{i-1} \rightarrow F_i$	
CBW	Commit Bandwidth	$C_{i-cbw} \rightarrow C_i$	cbw = Commit BW; lat. = 1 cycle
ROB	Finite ROB Size	$C_{i-w} \rightarrow D_i$	w = #ROB entries
DD	Data Dependences	$E_j \rightarrow R_i$	Added if inst. j produces source for inst. i; lat. depends on source instruction
RR	Finite Rename Registers	$F_i \rightarrow D_j$	there is rrnum number of new results from inst. i to the inst. j; rrnum = # extra rename regs
BL	Branch Resolution Latency	$E_i \rightarrow D_{i+1}$	For all but loop closing branches to fetch targets lat. depends on source
IQS	IQ Slot Avail.	$F_{i-NIQE} \rightarrow D_i$	NIQE = Num. Issue Q Entries; Edge from dyn. instrs. $i - NIQE$ to i
LQS	LQ Slot Avail.	$F_{i-NLQE} \rightarrow D_i$	NLQE = Num. Load Q Entries; Edge from $NLQE^{th}$ dyn. load instr. before i to i
SQS	SQ Slot Avail.	$S_{i-NSQE} \rightarrow D_i$	NSQE = Num. Store Q Entries; Edge from the $NSQE^{th}$ dyn. store instr. before i to i

Table 5: Description of Precedence Graph Edges

based on the distribution of frequencies of different paths. We examined the path profile of each thread for all the benchmarks, and found that it is almost similar for all threads, for data parallel benchmarks - this is because, for those benchmarks, different threads execute the same code, but operate on different data. The only significant difference among the path profiles of different threads is that the main thread uniquely tracks the paths in the sequential regions. In all data parallel programs, the main thread is also involved in computation. For this reason, the precedence graph is constructed only for the main thread for those benchmarks, because it covers almost all paths traversed by the other threads. For stream applications, we consider the path profile of each thread separately.

Petri net Simulation: From the precedence graph and resource availability, the Petri net model is constructed (Section 2, [17]). Petri net simulation is done for the frequency chosen by the binary search, to estimate the execution time of the region. For simulating the Petri net, cache hit/miss profile is used.

An important difference between the Petri net simulation in [13] and in this method is that we model the increased pressure the memory system is subjected to, due to the activities of concurrent threads.

We believe that for SPMD (data parallel) programs, a single voltage (frequency) setting for all threads (cores) would fetch most of the benefits of a per-thread (or per-core) DVFS scheme. Moreover, in their evaluation of commercial and scientific workloads, [6] point out that per core DVFS does not give huge benefits that could offset the design complexity. Therefore, for SPMD applications, we use a common frequency/voltage setting for all the cores, and always operate L2 cache and the memory system at a fixed speed. However, for general parallel programs where different threads execute different code segments, this may not be the most effective method, and our method considers different frequency setting for different threads.

Whenever a memory access instruction is encountered in the Petri net simulation, based on the cache hit/miss statistic for each thread, memory accesses are generated for all the threads - One access is a 'real' access (by the CPU simulated by the Petri net) and the rest are 'shadow' accesses (by the other CPUs which would have generated the accesses if they were also simulated). The simulated CPU waits only for the accesses it generated; the only purpose of the shadow accesses is to load the memory system modeled.

Because we do not know the memory bank to which a memory access goes, in our Petri net simulation, we assign a random memory bank (bank 0 to bank 7) for the first memory access corresponding to a frequently missing instruction. Subsequent accesses of the instruction will go to the successive banks in a round robin fashion. This is a conservative simplification that we have made, assuming that for frequently missing memory access instructions,

Acronym	Meaning
MD	Move Down
AL	Always at Least Frequency
SHU	Start Holding Up
HU	Hold Up
MU	Move Up
AH	Always at Highest Frequency
SHD	Start Holding Down
HD	Hold Down
S	Start State

Table 6: Description of States in Figure 4

accesses are array based, sequential, and therefore sweep successive memory banks. We repeat the Petri net simulation for 10 times, and choose the least frequency setting. All these take atmost 1 hour per benchmark.

Finally the frequency setting for each voltage/frequency reconfiguration point is written to a file. We have enhanced the simulator to interpret the file and effect the reconfigurations.

3.2 The Greedy Hardware Controller:

This controller was originally proposed for single core systems, in [7]. Later, it was adapted for multicore systems in [6], which evaluates two other DVFS techniques too. This simple controller performs the best among two other competing hardware controllers [6].

There is a separate controller for each core. The controller observes the ET^{-2} metric over a time interval, compares it with the previous interval, and then adjusts the voltage and frequency of the core based on the result of comparison, as specified by the state transition diagram in Figure 4.

As in [6], we assume that the CPU core provides counters that estimate the energy consumed by the core, and the core has performance counters for finding the application throughput (number of application instructions committed per cycle).

The controller initially starts at the highest voltage (frequency) level and then unconditionally reduces the voltage by one step after one interval. For every interval, the controller compares the ET^{-2} value over the current interval and the previous interval. If it is less, the controller remains in the same state and continues to move in the same direction as the previous. If it increases, the controller flips the direction, 'holds' the new state for a specified number of intervals (number of intervals to 'hold'), and then again moves one step in the flipped direction. The unambiguous state transition diagram is in Figure 4, and Table 6 explains the states of the controller.

In addition to this, we have evaluated this per-core controller in a different

configuration, where there is only one system-wide controller that chooses the same DVFS settings for all cores, based on the overall system ET^{-2} metric. We call this configuration Global Greedy.

4 Experimental Setup

Our implementation of the Petri net based DVFS method is very similar to compiler passes; Infact the only reason we chose to not use any existing compiler framework was to expedite the implementation. Figure 6 gives a high level view of the flow of our analysis framework.

We implemented parsers for converting the disassembled binary into the IR we designed. gcc 4.2 (linux-alpha) is used to generate binaries from source and the disassembled files.

We have used m5sim simulator for executing the benchmarks [15]. The m5sim simulator is a powerful tool capable of simulating CPU cores, the memory, I/O and network subsystems. It can be used in full-system simulation mode where in addition to the application code, OS code can also be executed. It supports the simulation of alpha binaries with linux operating system, and alpha's Privileged Architecture Library (PALcode). The simulator can be used in functional mode, timing mode and detailed mode.

For path profiling, we have used the simulator in the functional mode, since we are interested only in the paths tracked by an application. For cache

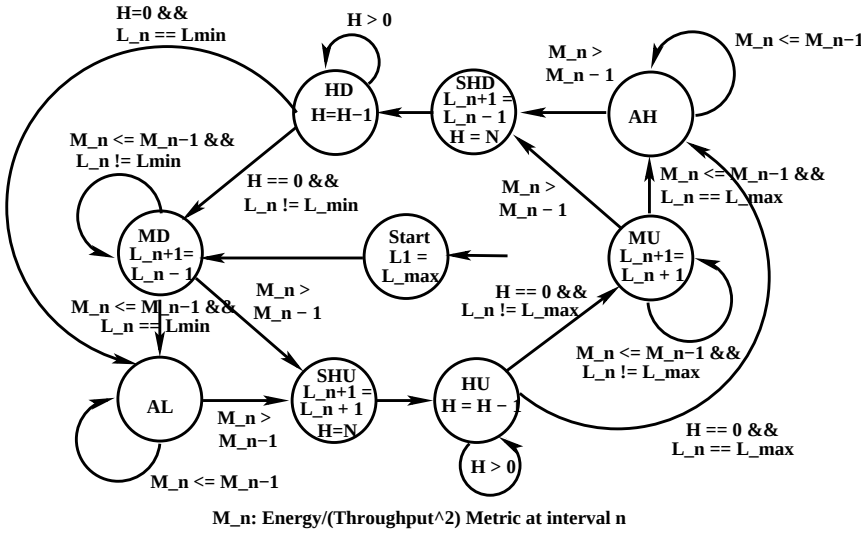

Figure 4: The Greedy VF Controller

profiling, we have used the timing mode which allows us to simulate cache hierarchies. For the actual timing and energy evaluation of the proposed DVFS scheme, we use the simulator in detailed mode, which allows cycle accurate simulation of out-of-order CPUs, along with the memory hierarchy. We always use the simulator for full system simulation, with the operating system.

The simulator allows fast forwarding of the specified number of instructions initially before switching to timing or detailed mode. In fast forward (FF) mode, only functional simulation is done. This helps us skip initialization phases of applications and simulate only those phases which characterize the properties of application.

4.1 Simulator Enhancements

We have made a number of enhancements to the m5sim simulator, and advanced the state-of-the-art of it significantly:

1. integrated Cacti 5.3 [23] with m5sim for modeling energy consumption by storage structures (caches, TLB's, register files, queues, branch predictors, BTBs); this version of cacti incorporates non-linear scaling models for process and device parameters based on the International Technology Roadmap for Semiconductors (ITRS)

2. integrated wattch [24] based power model for other structures (ALUs, buses, clock, wakeup/selection)

3. implemented voltage and frequency scaling; added a new module (Special PC Manager - SPM) that initiates voltage (frequency) scaling whenever voltage (frequency) reconfiguration points are encountered

4. enhanced it to support thread-specific hierarchical path profiling [21] and cache hit/miss profiling.

We have also used DRAMSim [25], a simulator for accurately modeling DRAMs, with the simulator.

4.1.1 Power Models:

Based on the underlying power models (wattch or cacti), we estimate the energy consumed by each hardware component during initialization, and whenever there is a change in the operating voltage and frequency. Each component updates the energy consumed by itself and its constituent components every cycle, and provides an interface for accessing energy consumption. The

information about energy consumption is propogated up hierarchically when the component at the topmost level (CPU) queries its subcomponents.

For implementing this, we changed the default implementation of the respective components (like branch predictors, caches, fetch, issue-execute-writeback units, etc.) in the simulator. The modifications, are similar to the ones in sim-wattch [44], and almost all of the components the out-of-order (O3) CPU are changed.

4.1.2 Dynamic Voltage and Frequency Scaling:

This is implemented by changing the value of the clock attribute of the CPU class of the simulator, and by changing the value of the operating voltage. Every time this is done, power is recomputed. When the greedy hardware controller is enabled, DVFS transitions are initiated by the controller's state machine. Otherwise, they are initiated by a new module we have added, called the Special PC Manager (SPM).

The SPM module accepts as inputs, a PC range and instruction count, and a list of tuples of PC, thread id, voltage/frequency setting and instruction count. The PC range and instruction count is used to terminate the simulation if the specified count of instructions in the specified PC range have been executed. A tuple of PC, voltage/frequency setting, thread id and instruction count specifies the voltage/frequency setting, the CPU should transition to, after committing the instruction at the specified PC on behalf of the specified thread id. Simulation is terminated, if the instruction at the PC is committed for the specified number (count) of times, when executed on behalf of the thread. The inputs to the SPM is specified in a DVFS configuration file.

Each CPU gets a handle of the unique instance of the SPM module, through the m5sim configuration file. After committing an instruction, a CPU uses this handle to inform the SPM, the details about the instruction it just committed. The SPM uses this information to decide if voltage/frequency transition must be made, or simulation has to be terminated.

4.1.3 Support for Path Profiling:

To support path profling, we have designed a low level intermediate representation (IR) for the binaries of the benchmarks. The IR is constructed from the disassembled binaries, and provides a rich set of interfaces that aid in the analysis of binaries at different levels: function, loop, basic block and instruction; however, it does not have any mechanism to modify the contents of the disassembled binary or the binary itself.

We have designed a new Path Profiler module for the simulator. For implementing HPP, we generate instrumentation code, which would be compiled as a part of the path profiler. This is done by writing the the instrumentation points (PC's) and the corresponding instrumentation information (path sum initialization, path sum read) to a file, which will be embedded in the path profiler code.

We have modified the CPU module of the simulator to send the PC and the software TID of the currently running thread to the path profiler module once an instruction completes execution, when path profiling is enabled. Given the PC and thread id, the path profiler takes the appropriate action which is determined by the instrumentation hints for the path profiler.

In addition to the Ball Larus instrumentation instructions [22], on function entries and exits, path sum and region information are pushed to/popped from a separate stack by the path profiler, so that path sum and region information is maintained for each function separately.

For keeping track of thread id, the simulator mimics the action performed when the function pthread_self is called (two memory reads - one in the physical address space and the other in the virtual address space of the thread), whenever the Internal Processor Register that stores the unique identifier for the current thread is written [28].

Clearly, the path profiler (and hence the simulator) needs to be recompiled if it must support path profiling for a different benchmark, but the advantage is that binary instrumentation is avoided, and yet hierarchical path profile is available.

4.1.4 DRAMSim Integration:

We have modified the default physical memory module in the m5sim to interface the simulator with the DRAMSim memory simulator [25]. Whenever there is a memory access, the physical memory module calls the DRAMSim interface for completing the access, specifying the callback function DRAMSim should call once the access is complete. Because both m5sim and DRAMSim are event driven, and m5sim provides asynchronous access functions for caches and memory, integrating DRAMSim with m5sim is reasonably straight-forward.

Figure 5 shows a high level view of the enhanced simulation infrastructure. Elliptical components are added by us; The profiler can be a path or cache hit (miss) profiler, and it is a separate thread different from the simulator thread. The SPM module takes Voltage Frequency reconfiguration points and values as input.

Table 7 lists the configuration of the simulated CMP system.

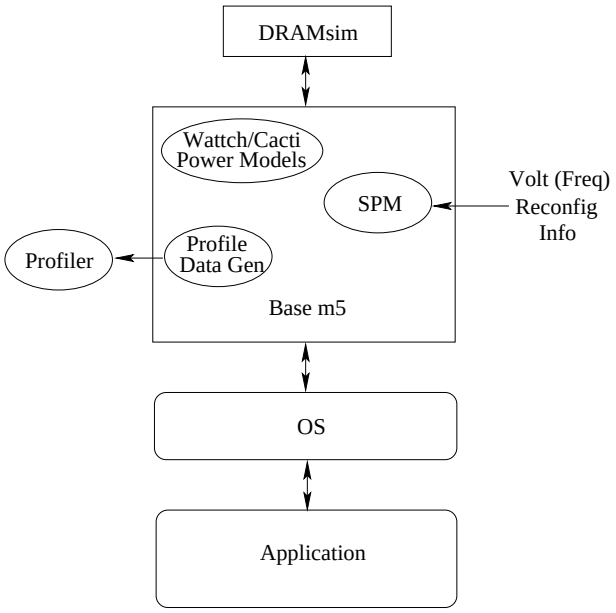

Figure 5: Simulation Framework

Since we have used Cacti 5.3 for accounting for energy consumed by storage elements and scaled the old wattch model for accounting for energy consumed by non-storage elements, we report the metrics for the two classes of components separately.

5 Data Parallel Workload: Experimental Results

In this section, we discuss the results for SPMD based data parallel workloads.

We use subsets of SPEC OMP 2001 [3], NAS Parallel Benchmarks [2] and splash 2 [26] benchmarks for evaluating our method. For the SPEC benchmarks, we use the train inputs for profiling, and ref inputs for reporting the results. For mg and art, we use two intervals to cover the sequential regions in which a significant fraction of time is spent and report the weighted average of the results for the sequential and parallel regions.

For all benchmarks, we simulated atleast 2 billion **application** instructions (library instructions are excluded for counting), after fast forwarding a few billion instructions.

Figure 7 shows the performance degradation, energy savings, and ET^{-2}

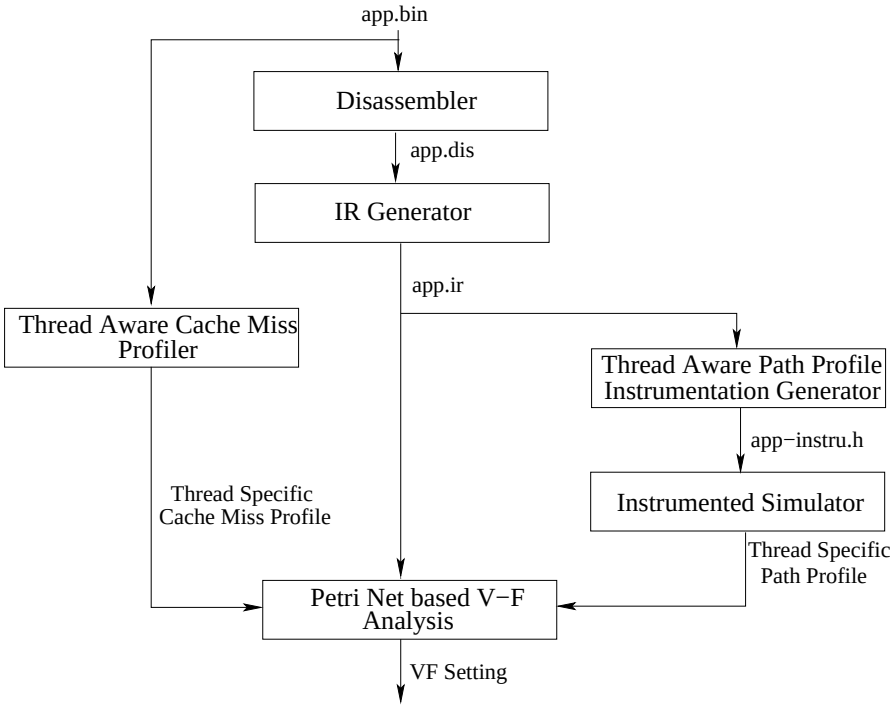

Figure 6: Our V/F Selection Framework

Parameter	Value
Volt/Freq & Tech	
Technology	45 nm
Frequency Range	3GHz - 4GHz
Domain Voltage Range	0.8V - 1.0V
Number of Steps	5; .2 μs per step
Branch Predictor	
Branch Predictor	Tournament
Local Predictor Size	2048 entries, 2b ctrs
Local History Table Size	2048 entries, 11 bit hist.
Global Predictor Size	8192 entries, 2b ctr, 13b hist.
Choice Predictor	8192 entries, 2 bit ctr
BTB Size	4096 entries, 16 bit tag
Memory System	
L1 DCache	32K 4W SA 64B blk 4 cycle lat
L1 ICache	32K 4W SA 64B blk 4 cycle lat
L2 UCache	4M 4W SA 64B blk 40 cycle lat
Memory Access latency	50 ns
Other Resources	
Decode/Issue/Retire Width	4/4/4
Integer ALUs	3 + 2 multiplier
FP ALUs	2 + 2 FPMult/Div
Issue Queue Size	64 IQ Entries, 32 loads, 32 stores
ROB Size	128
Extra Rename Registers	128 Int, 128 FP

Table 7: Simulator Configuration

improvement as percentages, over the baseline where all CPUs execute at the highest frequency. We define throughput as number of **application (excluding libraries)** instructions committed per cycle, similar to [6]. Therefore the number of library or OS instructions committed does not increase the throughput.

The performance degradation threshold constraint is set to 5%.

We see that on an average, all three methods achieve significant ET^{-2} improvements - 14.98%, 14.24% and 14.49% in the storage structures for the Petri net based, greedy and global greedy controller, respectively, and 37.32%, 37.37% and 37.35% in the non-storage structures - for a very small (2.25%, 2.16% and 2.58% for the Petri net based, greedy and global greedy controller, respectively).

Ocean has a very high L1 miss rate (20%), but it also has one of the least L2 miss rates (52%), which indicates that the effectiveness of L2 cache. In this case, DVFS degrades performance slightly, and because of this, ET^{-2} improvements are slightly lower, even though there is a significant savings in energy.

In all other benchmarks, the performance degradation is well within the threshold, and all schemes achieve significant ET^{-2} improvements.

Load is fairly well balanced across all cores in the parallel regions - the

Benchmark	Input	Simulation Interval
SPEC OMP 2001		
art	ref	FF 1G; 4G Simulation
		FF 6G; 2G Simulation
swim	ref	FF 5G; 4G Simulation
applu	ref	FF 15G; 4G Simulation
NPB		
mg	size A	FF 3G; 2G Simulation
		FF 4G; 4G Simulation
Splash2		
ocean	1026 x 1026 grid;	FF 3G; 2.5G Simulation
	9600 s relaxations	
	20K res., err. tol. 1e-7	

Table 8: Workload Details

most lightly loaded core spends just 3.7% of the time in idle mode, which indicates that there is not much potential for a system with an independent voltage/frequency control for different cores. This is not unexpected, since all these programs are SPMD style, where all threads execute the same code, but operate on different data.

Our findings for data parallel programs agree with the findings of [6], where they report high ET^{-2} improvements for the greedy controller, and conclude that independent voltage/frequency control for different cores does not offer huge benefits.

5.1 Comparison with Optimal Frequency Setting

We have also compared the energy savings of the Petri net based method with the energy savings obtained by an optimal setting. Finding an optimal setting for a program with n regions would require 5^n simulations (There are 5 frequency settings). To make the study feasible, we chose the longest running region in the procedure in which most of the time is spent. We enable DVS only in that region, and report the IPC degradation and energy savings only for that region. Table 9 shows the results. The performance degradation is chosen not to exceed 5%.

Benchmark	**IPC Ratio**	$St.ET^{-2}Ratio$	$NonSt.ET^{-2}Ratio$
art	1	1	1
swim	1	1	1
mg	1	1	1
ocean	0.95	1.01	1
applu	1.02	1	1
Mean	0.99	1.00	1.00

Table 9: Comparison with an Optimal DVS Setting (% wrt. baseline)

We can see that the Petri net based method achieves ET^{-2} improvements

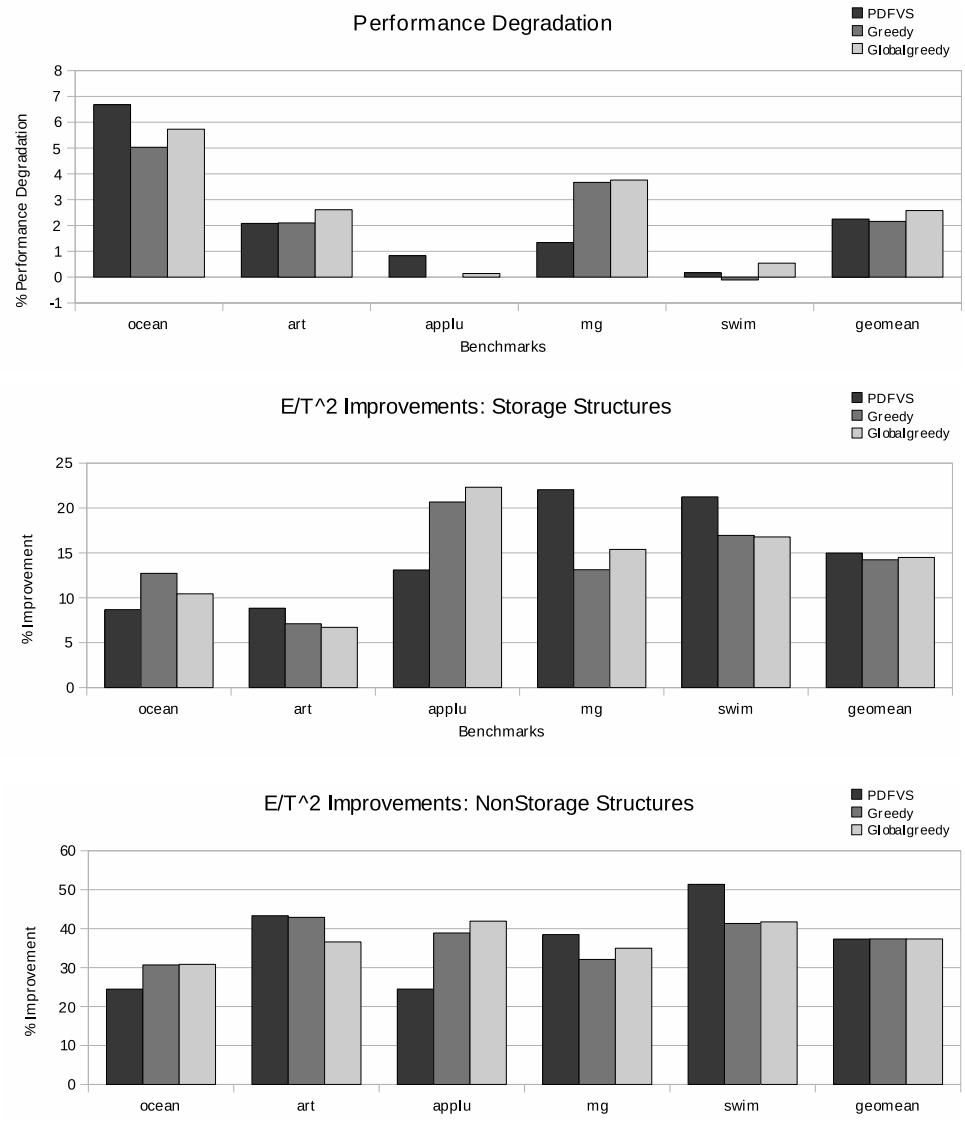

Figure 7: Performance Degradation and ET^{-2} Improvements for Data Parallel Workloads

very close to that of the optimal setting, deviating from it, by less than 1% of the optimal ET^{-2} metric in both the storage and non-storage structures respectively.

6 Stream Workload: Experimental Results

We discuss our results for stream programs, in this section. We use the streamit language infrastructure [29]. The infrastructure has a cluster library which can be used for building binaries for CMPs.

The streamit compiler generates c++ source code from streamit programs. We instruct the compiler to generate code for a 4-core machine. Since threads are created using pthreads, we could bind a thread to a core, using the sched_setaffinity system call, in the c++ code generated by the streamit compiler. (For OpenMP programs, we could not do this, because the compiler directly does not provide an intermediate representation that we could manipulate). In the rest of the section, we use threads and cores interchangeably.

In addition to data parallelism, stream programs exhibit task and pipeline parallelism as well. The streamit compiler takes as input, the number of threads to generate, and tries to partition the work across the different threads. In general, the tasks performed by the different threads are different, and therefore stream programs are more general than SPMD programs.

Such general parallel programs offer more opportunities for DVFS, because any residual imbalance in load can be used for performing DVFS. Performance is usually limited by the most heavily loaded core, and the rest of the cores could operate at speed that matches the rate of completion of the work assigned to the heavily loaded core.

We evaluate three different DVFS schemes for stream programs - the Petri net based method, the greedy controller (both explained in Section 3) and a simple linear interpolation model based method. We also use two different configurations - one in which the cores can run at independent voltage and frequencies, and the one in which cores can operate only at a common voltage and frequency. Most of the state-of-the-art CMPs [11, 12] do not support per-core DVFS, and evaluating these helps us estimate any benefits per-core DVFS schemes could provide over the support for a common DVFS setting for all cores.

Before discussing the experimental results, we briefly describe the linear interpolation based method, and mention a few words on how the Petri net performance model has been used for performing DVFS for stream programs.

6.0.1 Linear Scaling (LS):

In this scheme, we first run all cores at the highest speed and profile the time spent by each core for the thread it is assigned to. This excludes the time spent when the CPU is idle.

We find this by intercepting the writes to the Internal Processor Register of the alpha CPU that stores the unique identifier for the thread that is currently running on the core [28]. In the simulator, we keep track of the time that is spent between two writes. Along with this information and the actual values written during every write, we can get a good estimate of the amount of time a core spends, for a thread (In real linux systems, this estimate can be obtained from the proc filesystem's stat files for each thread).

At the end of the profile run, we aggregate the time spent for each thread. We normalize this time, using the highest time taken of all threads, and scale the frequency based on the normalized time. We assume that the time taken by a core to complete its workload is inversely proportional to the frequency at which the core operates.

With this assumption, we can easily derive the ideal frequency (f_{ideal}) with which a core should run, so that its rate of completion matches the slowest rate. It is given in Equation 1.

$$f_{ideal} = norm\_time * f_{max} \tag{1}$$

In Equation 1, $norm\_time$ is the normalized execution time, and f_{max} is the maximum operating frequency of the core.

From Equation 1, we get the actual frequency with which a core will be run, by rounding f_{ideal} up to the smallest frequency supported by the core, that is at least f_{ideal}.

This is one of the simplest performance models used in DVFS and scheduling algorithms (eg. [27]) and works well when there are few cache misses.

6.0.2 Petri net Performance Model based DVFS (PDVFS):

For this scheme, we first estimate the execution time of each thread with the Petri net based performance model. We then evaluate different frequency settings for each thread, and choose the least frequency that does not increase the execution time of the thread beyond the execution time of the performance limiting thread. For evaluating a frequency setting, we use the Petri net based performance model. The Petri net model is the same that is described in Section 3.

This method needs the application code (in our implementation, this is in an intermediate form obtained from disassembling binaries) to estimate

performance. Because of this reason, we could not analyze threads which perform I/O, and which involves the OS. For such threads (and for code that performs inter-thread communication), we profile to estimate execution time.

We first estimate the execution time of all threads using Petri net simulation (or profiling, for I/O threads), and find the longest execution time among all threads. We then keep scaling down frequency of each thread and estimating the new execution time as long as it is below the longest execution time, and choose the least frequency that keeps the estimated execution time, just below the longest execution time, as the frequency setting for a thread. For I/O threads and for code that implements inter-thread communication, we conservatively scale the execution time inversely with frequency, whereas for the other threads, we estimate execution times for each frequency setting we evaluate, using Petri net simulation.

6.1 Results: LS, PDVFS, Greedy and Global Greedy Controllers vs Baseline

We discuss the results of simulation in this section. We simulate 4G application instructions (which does not include OS or libraries), after fast forwarding 200M instructions.

We define throughput as the number of application instructions committed per cycle, and this excludes OS and library code.

Figure 8 shows the performance (Instructions per clock - IPC) degradation and ET^{-2} improvements for the LS, PDVFS (both with per core DVFS support), the Greedy Controller, and the Global Greedy Controller over the baseline with no DVFS, which always runs at the highest frequency. In the following discussions, unless explicitly stated, we always refer to the per-core DVFS configurations for the software based schemes, when we mention them.

As we can see, the Linear Scaling method achieves significant ET^{-2} improvements - 9.67% and 24.73%, on an average, in the storage and non-storage structures, over the baseline, with much less performance degradation of .21%.

The Petri net based DVFS method follows Linear Scaling closely, making 8.38% and 23.47% ET^{-2} improvements, for storage and non-storage structures, for a slightly higher performance degradation (0.68%).

The hardware based Greedy Controller degrades performance significantly (12.97%) on an average. Due to this high degradation, it degrades the base ET^{-2} by 6.78% and 2.1% in the storage structures and non-storage structures respectively. The global greedy configuration degrades performance even worse (15.86%), for ET^{-2} comparable to the Greedy Controller.

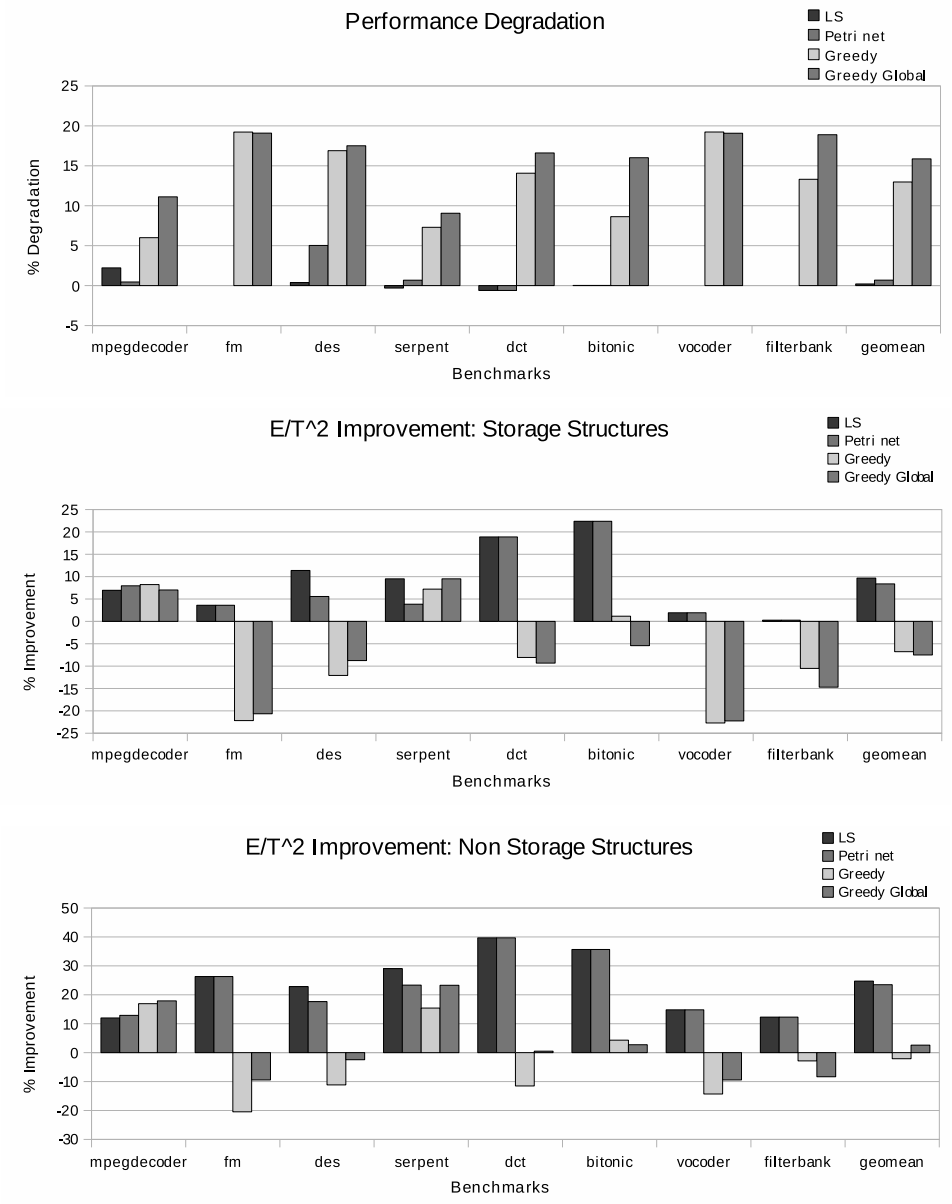

Figure 8: Results: Performance Degradation and ET^{-2} Improvement over Base

We do not show the results for the software based schemes for a system supporting a common DVFS setting for all cores in Figure 8 because both schemes choose the highest DVFS setting for all cores (which means that there is no performance degradation and ET^{-2} improvement) for all benchmarks. This is because choosing a common lower frequency for all cores would result in significant performance degradation, since there is invariably at least one thread that is the performance bottleneck, and reducing its CPU speed has a severe adverse impact on performance.

One important observation we could make for the hardware based controller (both versions) is that, for all benchmarks, the controllers in the cores hosting the performance limiting threads spend at least 70% of their time in the HoldUp and HoldDown states, with the time in these two states shared almost equally. This means that for at least 35% of the time, the performance limiting thread spends its time in a low frequency state, which explains the huge performance degradation for the hardware controllers.

For vocoder, fm and filterbank, the load distribution is heavily skewed and this gives more opportunity for DVFS. However, for the same reason, power consumption in storage structures is heavily dominated by the busy cores; Even though the idle cores operate at the lowest frequency, it does not make much difference for the ET^{-2} product of storage structures. There is a modest improvement for non-storage structures.

We must note that both the software based schemes try to keep the performance degradation to a minimum, and they succeed in their goals (0.21% and 0.68% performance degradation for LS and PDVFS, respectively). This also translates to very high ET^{-2} savings in most cases. In mpegdecoder, degrading performance a bit more fetches more ET^{-2} benefits.

While the Linear Scaling method is extremely simple, the Petri net performance based DVS method is much more complex, and it performs slightly worse than the Linear Scaling method (though it matches the Linear Scaling closely). The Petri net performance based DVS method was found to be very useful in the context of DVFS for Multiple Clock Domain (MCD) processors [13] where different parts of the CPU (Floating point units, Integer Units, Caches etc.) execute at different speed. In MCD processors, performance is a complex non-linear function of domain frequencies, and when frequencies of domains change, different schedules of instruction execution are possible [13]. The Petri net model can capture that, and performed better than a profile based approach that uses a very detailed execution trace. We also saw in Section 5 that the method performs very well in the case of data parallel programs with high memory misses too. For such programs, the simple linear scaling is not applicable, because even the most lightly loaded core spends just about 3.7% percentage of the time in idle mode. Perfor-

mance in the presence of frequent memory accesses is not a simple function of CPU frequency, as it involves an independent frequency domain (memory), which is shared by all cores.

However, for single clock domain cores, when there are few cache misses, performance is more or less a linear function of frequency. The Petri net model does not provide much advantage over the Linear Scaling model in this case.

Table 10 shows the normalized execution time of differnet threads after profiling for the linear scaling method, and the work estimated by the streamit compiler for different threads. The streamit compiler analyzes the input stream program, estimates work in each filter, and uses this information to balance load among different tasks while performing task partitioning. Although in some cases, the workload estimated by the compiler and the actual workload of threads match, in 5 benchmarks (mpegdecoder, dct, des, vocoder, bitonic), there is at least one thread for which the estimates vary significantly.

In most benchmarks, the frequency of cores hosting I/O threads have been scaled down by the software based methods, since the threads performing the actual computation limit the performance. However, bitonic and mpegdecoder are interesting in that the I/O threads are the performance bottlenecks, and at least one of the cores hosting the compute threads has been scaled down.

Table 10 also explains why per-core DVFS schemes achieve superior ET^{-2} results over common DVFS schemes. We can see that there is load imbalance in the applications. Both the software based per-core DVFS schemes are able to exploit the opportunities provided by the load imbalance well, by scaling down the frequencies of cores that are lightly loaded, while running the heavily loaded cores at high frequencies. If there is no support for per-core DVFS in the hardware, these opportunities can not be used effectively for achieving ET^{-2} improvements, and the software based schemes choose the highest frequency setting for all cores, offering no ET^{-2} benefits.

Our findings in this section are contrary to the results of [6] where they report that per-core DVFS does not offer significant benefits for multithreaded applications. We believe that this is because of the workloads being chosen. Some of the applications in [6] are data parallel (all threads exhibit similar behaviour), have high miss rates and their memory system is very slow. (There is no information about load balance in [6]).

When threads exhibit similar behaviour or all threads have high miss rates, then per-core DVFS may not offer huge benefits. The first case is obvious, whereas in the second case, we can possibly run all cores slower, because the common bottleneck is the memory subsystem. In our case, all

Benchmark	Estimate	Core 0	Core 1	Core 2	Core 3
filterbank	profile	0.01	1	1	0.01
	compiler	0	0.82	1	0
serpent	profile	0.2	0.88	1	0.16
	compiler	0	0.93	1	0
mpegdecoder	profile	0.92	0.75	0.93	1
	compiler	0.02	1.00	0.80	0.02
dct	profile	0.3	0.41	1	0.3
	compiler	0	0.72	1	0
des	profile	0.87	0.91	1	0.72
	compiler	0.01	0.94	1	0.01
vocoder	profile	0.06	0.04	1	0
	compiler	0.92	1.00	0.52	0
bitonic	profile	1	0.64	0.63	0.8
	compiler	0.03	0.86	1.00	0.03
fm	profile	0.01	0.07	1	0
	compiler	0.00	0.09	1	0

Table 10: Normalized Execution Time vs Compiler Estimated Work for different threads

our applications are pipeline parallel, suffer few L1 misses (less than 5%), and the load across cores is not uniform.

From Table 10 we can also see that there is a potential for better load balancing, and in most cases, a part of the computation can be alloted to the I/O threads too. At the same time, in some benchmarks, the work done by I/O threads becomes the bottleneck. It is not clear at the moment, if other constraints related to partitioning pose difficulties in realizing this potential. Moreover estimating the I/O workload at a high level is difficult.

6.2 Comparing the Software Schemes with the Optimal Setting

We have also conducted experiments to find the optimal DVFS setting that minimizes the system ET^{-2}, and compared the optimal ET^{-2} with that obtained by Linear Scaling and the Petri net based method. We choose the system configuration with two frequency settings (4 GHz and 3.5 GHz) for each core. For n settings, we need n^4 simulations to find the optimal setting. Even for this configuration, we need 16 (2^4, 2 for each core) simulations to determine the optimal setting. We simulate 2 billion application instructions.

For this system configuration, both the software based schemes choose the same voltage/frequency setting for all benchmarks. Table 11 shows the IPC and ET^{-2} (for storage - SET^{-2} - and non-storage structures - NSET^{-2}) of the software based schemes, normalized to the corresponding metrics of the optimal ET^{-2} setting. As we can see from Table 11, the software

Benchmark	IPC	SET^{-2}	$NSET^{-2}$
filterbank	1.0	1.0	1.0
mpegdecoder	1.0074	1.0903	1.1588
des	0.9988	1.0661	1.0964
vocoder	1.0	1.0	1.0
bitonic	1.0	1.0	1.0
fm	1.0	1.0	1.0
serpent	1.0453	1.08715	1.1020
dct	1.0	1.0	1.0
geomean	1.0063	1.0297	1.043

Table 11: Ratio of metrics of LS and Optimal

based schemes match the optimal setting in 5 of the 8 benchmarks, with the average ET^{-2} metric falling within 2.97% for storage structures and 4.3% for non storage structures.

6.3 ILP Based Partitioning and DVFS

We have also conducted experiments to see how DVFS could be done when the number of threads is more than the number of cores. In this case, the problem becomes more complex as we have to decide which threads run on which cores. If the number of threads is less than the number of cores, then the cores which are idle can be put in low power mode.

Operating systems like linux balance load dynamically. The load balancer of linux finds the runqueue of the core with the highest number of processes (it should be at least 25% more than that of the core on which the load balancer runs), and chooses the highest priority task that is not running, not bound to the core, and has not run for a relatively long time (not cache hot), and *pulls* that task to the current core. This happens as long as the runqueues are unbalanced [30]. This is done at process granularity.

To see if load balancing at a finer granularity could help better, we first profile the execution time of work unit of each thread, scaling it by the number of iterations, the work function is called during steady state. We then assigned threads to different cores, by an Integer Linear Programming based partitioning. We then applied DVFS to scale down the frequency of cores to exploit any residual load imbalance.

For the sake of completeness, we specify the program that minimizes the execution time of the most heavily loaded core, from Equations 2 to 6. T is the maximum of the sum of execution time of all threads on any core. a_{ij} is an indicator variable, which is 1, if and only if thread i is bound to core j. Equation 4 ensures that each thread is bound to exactly one core.

$$Min. \quad T \tag{2}$$

$$T \;>= \; \sum_{i=1}^{15} a_{ij}.t_i \quad \forall j \tag{3}$$

$$\sum_{j=1}^{4} a_{ij} \;=\; 1 \quad \forall i \tag{4}$$

$$a_{ij} \;>= \; 0 \quad \forall i,j \tag{5}$$

$$a_{ij} \;<= \; 1 \quad \forall i,j \tag{6}$$

To implement thread binding, we modified the streamit compiler to generate command line switches that help bind a thread to a core. After running the ILP based partitioning algorithm, we use these switches to bind threads to the cores. We use IBM's ilog [31] package for solving the ILP. In our experiments, we instructed the streamit compiler to generate 15 threads, and we ran our experiments with the same 4 core CMP configuration as in Table 7.

Figure 6.3 shows the performance degradation and ET^{-2} improvements, after ILP based partitioning, and DVS, relative to the default linux based dynamic load balancing. The static ILP based partitioning achieves a modest 1.83% performance improvement, with corresponding ET^{-2} improvements of 4.37% and 6.39% in the storage and non-storage structures respectively. DVFS marginally improves the ET^{-2} ratio, with 4.9% and 12.96% improvements in the storage and non-storage structures, respectively.

We see that DVS does not offer much benefits in this case. This is because of the fact that with more threads, load is fairly well balanced by the ILP based partitioning, leaving very few opportunities for DVFS.

7 Related Work

Dynamic Voltage and Frequency Scaling is an effective technique for designing energy/performance tradeoffs, and it has been researched well: [10, 13, 18, 32, 33, 35, 37] are a few of them.

[10, 35, 37] are for traditional single core, single clock domain systems. Hsu et al [10] and Krishnaswamy et al [37] formulate the voltage-frequency selection problem as a linear program and solve it. While [10] uses program structures to identify voltage/frequency reconfiguration points, [37] uses program phases. The latter method requires whole program simulation multiple

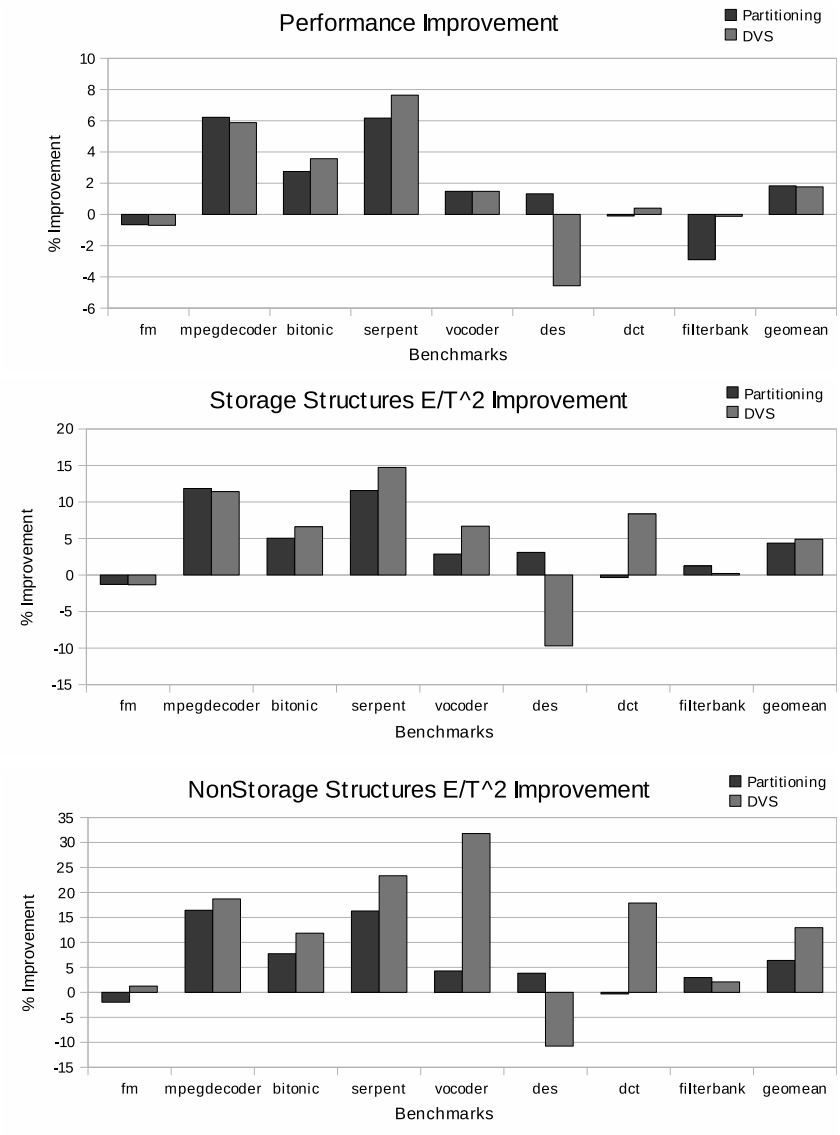

Figure 9: Results: Performance Degradation and ET^{-2} Improvement over Base for Partitioning and DVS

times, once for every voltage setting. Wu et al [35] use a dynamic compiler based method which uses a simple model to find energy efficient voltage/frequency settings.

Multiple Clock Domain Architectures (eg. [36] by Semeraro et al) have been proposed to combat the increasingly challenging clock distribution problem. MCD architectures allow chip partitioning into different clock domains, and use local clocks in each domain. Each domain can be operated at an independent voltage and frequency, and therefore, they are particularly amenable for obtaining impressive energy/performance tradeoffs: Only performance critical domains have to be run at the highest speed, and others can be run at lower speeds, without affecting the performance much.

[32, 33] are hardware based techniques for voltage (frequency) selection for multiple clock domain, uniprocessor systems; The DVFS controller of Wu et al [32] is based on control theory, and the controller of Semeraro et al. [33] is based on a set of simple heuristics, but both of them rely on queue occupancy for their decisions.

Our technique is an extension of [13], a compiler based VF selection for single core, multiple clock domain processors (for sequential programs). Like [13], we use a Petri net based program performance model to find energy efficient voltage/frequency settings. The concurrent threads in multithreaded applications increase the pressure on the memory system. We make an important value addition to [13] by modeling contention for memory.

Magklis et al [18] use a software technique for DVS control for sequential programs running on multiple clock domain processors. Their work analyzes trace files generated during a detailed simulation for guiding their choice of voltage/frequency.

DVFS has also been attempted for CMP processors. We can broadly classify the existing works based on whether they are targetting multithreaded workloads [6, 19, 34] or multiprogrammed workloads [8, 9, 14]. Our work falls in the category of methods for multithreaded workloads.

Herbert et al [6] evaluate different DVFS schemes for both commercial and scientific multithreaded applications, which do not show strong interaction. All schemes they evaluate use local information to choose voltage and frequency setting for each domain, and are interval based. Our Petri net based method captures the inherent properties of both the application and the system; the application characteristics determine reconfiguration points, while the actual frequency/voltage settings are based on the properties of both the application and the system. They report that per-core VF setting do not offer huge benefits that could amortize the design complexity. We use the best performing greedy controller in their evaluation, for our experiments, for comparison. We also find that for general multi-threaded programs, per-

core VFS control is very essential for achieving high ET^{-2} improvements for negligible performance degradation.

Juang et al [34] propose per core DVFS based on queue occupancy in *all* cores in a CMP, and point out that local information is insufficient for the choice of energy efficient voltage/frequency settings across multiple cores. Their method is based on control theoretic principles, their interval is fixed and require pair-wise communication for every DVFS interval. Moreover, their method requires the tuning of stability constants. Their method also requires setting up of per thread load factor for each thread. As we saw in Section 6, exact workload estimation at a high level itself is non-trivial. Their evaluation was done on an ARM like CMP with no L2 cache and a fast memory system, without OS. In contrast, the method we propose does not require complex controllers or extra all-to-all communication among CMPs; Our evaluation is on a full system CMP simulator, and is stronger, both in terms of number of benchmarks, and in terms of quantifying the opportunities that exist for DVFS, and how much of it is exploited by our method.

The work of Li et al [19] optimizes power consumption in a CMP system running multithreaded programs, while meeting the performance target. They use a single voltage/frequency setting for all cores, and find both the number of cores taking part in computation, and a voltage frequency assignment for them. They use a binary search to find the number of processors taking part in computation, and a highly simplified performance model to find voltage/frequency settings. Theirs is a run-time scheme, where trials for different processor counts and frequency settings are done online for each parallel region. They report that their method converges to a stable setting after about 3 trials in most cases.

Isci et al [14] propose and evaluate different global power management policies based on DVFS on a CMP. Their aim is to maximize system throughput, while meeting a given power budget. They use a hierarchical power management method where a DVFS based global power manager senses per core power-performance statistics periodically and ensures that chip-level power budgets are met using DVFS. Local monitors provide local statistics to the global manager, and employ baseline dynamic power management policies like clock gating and fetch throttling. They conclude that per-core DVFS achieves superior results over chip-wide DVFS for multiprogrammed workloads.

Teodorescu et al. [8] propose process variation aware scheduling and power management algorithms for maximizing system throughput for a given power budget. Due to process variation, each core on a many-core CMP potentially differs from others in both static power consumed, and the maximum frequency supported by it. Their scheduling algorithms map cores to ap-

plications based on IPC/power characteristics of the application, and the frequency and power characteristics of cores.

Wang et al. [9] use DVFS to control the total power of a CMP to a desired set point, while maintaining the temperature of each core below a specified limit. Their technique is based on formal optimal control theory, and their evaluation is on a multiprogrammed workload.

There have been several software based DVFS algorithms for stream programs in the Embedded and Real Time Systems community [27, 38–40]. In addition to DVFS, they also consider scheduling. For DVFS, they either use a linear performance model which we have evaluated in this paper, or assume that program execution time is known for all frequencies. These works do not specify the implementation of the algorithms (eg. the level - application or OS - at which these algorithms can be implemented, implementation of the scheduling of communication among threads), and deal with the problem at more abstract levels. For evaluations, they use processor models without an operating system. Our evaluation is based on actual implementation of DVFS schemes in a detailed full system out-of-order CMP simulator capable of running the binaries of applications, along with an operating system.

Simon et al. [43] propose an adaptive DVFS scheduling scheme which chooses between optimistic and pessimistic frequency levels to optimize system energy consumption, while also ensuring timing guarantees. Their scheduler chooses frequency settings dynamically, till the chosen speed is below a threshold, and once the threshold is exceeded, the scheduler chooses a constant, pessimistic frequency to ensure that deadline is met. Tasks are modeled as events, are associated with deadlines, and the number of task arrivals in an interval of time is bound. With these assumptions, their method uses timed model checking to determine whether their DVFS scheduler should operate in the optimistic or pessimistic mode.

For multi-threaded applications whose task dependence can be represented as a Directed Acyclic Graph (DAG), Kimura et al. [41] propose an algorithm using DVFS for energy savings. Their algorithm computes the slack for a task as the difference between the latest finish time and the earliest start time for the task based on the task dependence DAG and the estimated execution times for the tasks, and uses this slack to reduce voltage/frequency settings, without increasing the execution time. They evaluate their algorithm on a real cluster, and report significant energy savings for a small performance degradation. Some important class of applications including stream applications can not always be represented as DAGs, since filters (work units in stream programs) are generally executed repeatedly. Moreover, some filters can be stateful (data dependence spanning different executions), necessitating a different approach for DVFS.

Thies et al. [5] introduce the Streamit language, characterize special properties of stream programs and explain the constructs in Streamit language. Gordon et al. [42] detail one of the earliest compilers for the Streamit language, and list the challenges in building a compiler for it. Gordon et al. [4] make significant improvements to [42] including the support for software pipelining.

Petri nets are powerful mathematical abstractions that have a wide range of applicability in modeling systems like pipelined and multi-function arithmetic units in CPUs, parallel and distributed software systems, chemical systems, and legal systems [46]. Peterson's book [46] has a good introduction to Petri net modeling, their properties and applications. Initially, Peti nets did not have the notion of time associated with them; Ramchandani augmented the basic Petri net definition by associating each transition with a firing time [48]. Gao et al. [47] used such Timed Petri nets to find software pipelined schedules for special kinds of loops, where the dependence distance is either 1 or 0.

Rajagopalan et al. [17] use Petri nets for generating software pipelined schedules for more general loops. Based on the data dependence graph for a loop and the processor resource specification, they describe a mechanical way to construct a Petri net. They fire the transitions in the Petri net, identify a repeated sequence of transition firings, and use this sequence to generate software pipelined schedules for loops. We have used their Petri net construction method for our performance estimation step in our DVFS method.

8 Conclusions and Future Work

In this work, we presented a novel Petri net performance model based DVFS technique for multithreaded applications. We showed the experimental results for SPMD and Stream applications. We also compared this with a state-of-the-art hardware based DVFS controller.

We found that the Petri net based method achieves significant ET^{-2} improvements for both classes of applications. The hardware based controller performs well for SPMD applications, but performs very poorly for stream applications. Also, for stream applications, a very simple linear interpolation based method achieves all the benefits of the Petri net based method. An important conclusion that we draw from this work is that, for general multithreaded applications, independent voltage/frequency control for each core is essential for achieving significant ET^{-2} improvements with small performance degradation.

In future, we would like to enhance the Petri net with power and temperature models, and also validate it. This would help us consider useful tradeoffs involving power, temperature and performance. Also we would like to include contention from DMA in the Petri net model. This would be very useful for architectures like IBM Cell.

References

[1] OpenMP Tutorial, `www.openmp.org/presentations/sc99/sc99\_tutorial.pdf`

[2] NAS Parallel Benchmarks, `www.nas.nasa.gov/Resources/Software/npb.html`

[3] SPEC OMP, `http://www.spec.org/omp`

[4] M.I.Gordon, W.Thies, S.Amarasinghe. Exploiting Coarse-Grained Task, Data and Pipeline Parallelism in Stream Programs. In Proc. *ASPLOS*, 2006.

[5] W.Thies, M.Karczmarek, S.Amarasinghe. StreamIt: A Language for Streaming Applications. In Proc. *CC*, 2002.

[6] S.Herbert, D.Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In Proc. *ISLPED*, 2007.

[7] G.Magklis, P.Chaparro, J.Gonzalez, A.Gonzalez. Independent Front-end and Back-end Dynamic Voltage Scaling for a GALS Microarchitecture. In Proc. *ISLPED*, 2006.

[8] R.Teodorescu, J.Torrellas. Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors. In Proc. *ISCA*, 2008.

[9] Y.Wang, K.Ma, X.Wang. Temperature-Constrained Power Control for Chip Multiprocessors with Online Model Estimation. In Proc. *ISCA*, 2009.

[10] C.H.Hsu, Ulrich Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In Proc. *PLDI*, 2003.

[11] http://support.amd.com/us/Processor_TechDocs/31116.pdf

[12] http://download.intel.com/design/processor/datashts/322909.pdf

[13] A.Rangasamy, R.Nagpal, Y.N.Srikant. Compiler-Directed Frequency and Voltage Scaling for a Multiple Clock Domain Microarchitecture. In Proc. *CF*, 2008.

[14] C.Isci, A.Buyuktosunoglu, C.Cher, P.Bose, M.Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In Proc. *MICRO*, 2006.

[15] N.L.Binkert, R.G.Dreslinski, L.R.Hsu, K.T.Lim, A.G.Saidi, S.K.Reinhardt. The M5 Simulator: Modeling Networked Systems. In IEEE MICRO, 2006, Volume 26, Issue 4.

[16] B.A.Fields, R.Bodik, M.D.Hill, C.J.Newburn. Using Interaction Costs for Microarchitectural Bottleneck Analysis. In Proc. *MICRO*, 2003.

[17] M.Rajagopalan, V.H.Allan. Specification of Software Pipelining using Petri nets. In Int. J. Parallel Program., 1994, Volume 22, Issue 3.

[18] G.Magklis, M.L.Scott, G.Semeraro, D.H.Albonesi, S.Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Microprocessor. In Proc. *ISCA*, 2003.

[19] J.Li, J.F.Martinez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In Proc. *HPCA*, 2006.

[20] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. Compilers: Principles, Techniques and Tools, Second Edition. Chapter 9. Machine Independent Optimizations. Addison-Wesley, 2007.

[21] Y.Wu, A.A.Tabatabai, D.A.Berson, J.Fang, R.Gupta. Hierarchical Software Path Profiling. US Patent 6848100, Jan 2005.

[22] T.Ball and J.R.Larus. Efficient Path Profiling. In Proc. *MICRO*, 1996.

[23] CACTI 5.3 `http://www.hpl.hp.com/research/cacti/`

[24] D.Brooks, V.Tiwari, M.Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In Proc. *ISCA*, 2000.

[25] D.Wang, B.Ganesh, N.Tuaycharoen, K.Baynes, A.Jaleel, B.Jacob. DRAMsim: A memory-system simulator. SIGARCH Comput. Archit. News, 2005, Volume 33, Issue 4.

[26] S.C.Woo, M.Ohara, E.Torrie, J.P.Singh, A.Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proc. *ISCA*, 1995.

[27] R.Xu, R.Melhem, D.Mosse. Energy-Aware Scheduling for Streaming Applications on Chip Multiprocessors. In Proc. *RTSS*, 2007.

[28] Alpha Architecture Reference Manual. Compaq Computer Corporation, 2002.

[29] http://groups.csail.mit.edu/cag/streamit/shtml/download.shtml

[30] R.Love. Linux Kernel Development, Second Edition. Chapter 4. Process Scheduling. Novell Press, 2005.

[31] CPLEX Solver `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`

[32] Q.Wu, P.Juang, M.Martonosi, D.W.Clark. Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors. In Proc. *ASPLOS*, 2004.

[33] G.Semeraro, D.H.Albonesi, S.G.Dropsho, G.Magklis, S.Dwarkadas, M.L.Scott. Dynamic Frequency and Voltage Control for a Multiple Clock Domain Microarchitecture. In Proc. *MICRO*, 2002.

[34] P.Juang, Q.Wu, L.S.Peh, M.Martonosi, D.W.Clark. Coordinated, distributed, formal energy management of chip multiprocessors. In Proc. *ISLPED*, 2005.

[35] Q.Wu, M.Martonosi, D.W.Clark, V.J.Reddi, D.Connors, Y.Wu, J.Lee, D.Brooks. Dynamic Compiler-Driven Control for Microprocessor Energy and Performance. IEEE MICRO, 2006, Volume 26, Issue 1.

[36] G.Semeraro, G.Magklis, R.Balasubramonian, D.H.Albonesi, S.Dwarkadas, M.L.Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In Proc. *HPCA*, 2002.

[37] S.Krishnaswamy, R.Govindarajan. Compiler-Directed Dynamic Voltage Scaling using Program Phases. In Proc. *HiPC*, 2007.

[38] H.Liu, Z.Shao, M.Wang, J.Du, C.J.Xue, Z.Jia, Combining Coarse-Grained Software Pipelining with DVS for Scheduling Real-Time Periodic Dependent Tasks on Multi-Core Embedded Systems. In Journal of Signal Processing Systems, 2009, Volume 57, Issue 2.

[39] Y.Chen, Z.Shao, Q.Zhuge, C.Xue, B.Xiao, E.H.M.Sha. Minimizing Energy via Loop Scheduling and DVS for Multi-Core Embedded Systems. In Proc. *ICPADS*, 2005.

[40] Y.Wang, D.Liu, M.Wang, Z.Qin, Z.Shao. Optimal Task Scheduling by Removing Inter-Core Communication Overhead for Streaming Applications on MPSoC. In Proc. *RTAS*, 2010.

[41] H.Kimura, M.Sato, Y.Hotta, T.Boku, D.Takahashi. Emprical study on Reducing Energy of Parallel Programs using Slack Reclamation by DVFS in a Power-scalable High Performance Cluster. In Proc. International Conference on Cluster Computing, 2006.

[42] M.I.Gordon, W. Thies, M.Karczmarek, J.Lin, A.S.Meli, A.A.Lamb, C.Leger, J.Wong, H.Hoffmann, D.Maze, S.Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In Proc. *ASPLOS*, 2002.

[43] S.Perathoner, K.Lampka, N.Stoimenov, L.Thiele, J.Chen. Combining Optimistic and Pessimistic DVS Scheduling: An Adaptive Scheme and Analysis. In Proc. *ICCAD*, 2010.

[44] Sim-Wattch 1.02, `http://www.eecs.harvard.edu/~dbrooks/wattch-form.html`

[45] D.M.Brooks, P.Bose, S.E.Schuster, H.Jacobson, P.N.Kudva, A.Buyuktosunoglu, J.D.Wellman, V.Zyuban, M.Gupta, P.W.Cook. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. In IEEE MICRO, 2000, Volume 20, Issue 6.

[46] J.L.Peterson. Petri Net Theory and the Modeling of Systems. Prentice Hall. 1981.

[47] G.R.Gao, Y.Wong, Q.Ning. A Timed Petri net Model for Fine-grain Loop Scheduling. In Proc. of PLDI, 1991.

[48] C.Ramchandani. Analysis of Asynchronous Concurrent Systems by Petri nets. PhD Thesis. MIT. 1973.