

**Power Management in Multicore Processors through  
Clustered DVFS**

**A THESIS**

**SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL  
OF THE UNIVERSITY OF MINNESOTA**

**BY**

**Tejaswini Kolpe**

**IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
Master of Science**

**July, 2010**

© Tejaswini Kolpe 2010  
ALL RIGHTS RESERVED

# Acknowledgements

I would like to thank my advisor Prof. Sachin Sapatnekar for giving me an opportunity to be a part of his research group. I attribute my academic success to his guidance, encouragement and support throughout my graduate study at the University. His emphasis on reasoning, attention to details and clarity of expression has taught me to be meticulous and will guide me in future too. He will continue to be a role model for me throughout my life. My special thanks to Shruti Patil for being patient with me and answering all the simulator and benchmark related queries. I would also like to thank Prof. Zhai, Suhail, Jieming and Pingqiang for assisting me in updating simulator files. Thanks to all members of the VLSI Design Automation lab for making it a great place to work with their cheerful and friendly disposition. Thanks to Baktash, Saket and Yaoguang for their help in setting up a workplace for me in the lab. Last but not the least, I would like to express my humble gratitude to my parents. I am indebted to them for their continuous encouragement and their role in shaping my career.

## Abstract

The need for high speed processors in recent years has increased the need to exploit more parallelism than instruction level parallelism (ILP) and thread level parallelism (TLP). As a result, chip multiprocessors have emerged as a solution for the high speed computing demands. Though a high throughput is achieved, power dissipation in chip multiprocessors is still a problem that needs to be addressed. A number of techniques for reducing both the active and static components of power exist. Dynamic voltage and frequency scaling (DVFS) is one of the schemes to reduce active power. DVFS is easy to implement for a single processor but if it has to be implemented for each core of a chip multiprocessor, a number of voltage regulators are required on chip and the area and power overheads associated with them surpass the advantages of having per-core control. On the other hand, one DVFS control for all cores cannot fully harness the potential for power reduction in each core.

In this thesis, we look at the possibility of clustering the cores of a multicore processor and implementing DVFS on a per-cluster basis. We propose a scheme to find similarity among the cores and cluster the cores based on the similarity. We also provide an algorithm to implement DVFS for the clusters. We evaluate the effectiveness of per-cluster DVFS in power reduction by considering different number of clusters and different use cases for the applications running on the cores.

# Contents

<b>Acknowledgements</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>List of Tables</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and background . . . . .	1
1.2 Dynamic voltage and frequency scaling . . . . .	3
1.3 Problem description . . . . .	7
1.4 Organization of the thesis . . . . .	8
<b>2 Offline Profiling and Voltage and Frequency Schedule</b>	<b>9</b>
2.1 Optimal frequency and voltage schedule for each core . . . . .	11
2.2 Clustering the cores . . . . .	15
2.3 Optimal frequency and voltage schedule for a cluster of cores . . . . .	18
<b>3 Experimental Setup</b>	<b>20</b>
<b>4 Results</b>	<b>29</b>
4.1 Case 1: All applications execute completely . . . . .	30
4.1.1 Target execution time of 0.9 s . . . . .	31
4.1.2 Target execution time of 1.1 s . . . . .	36
4.1.3 Target execution time of 1.3 s . . . . .	38

4.2	Case 2: Applications are penalized for noncompletion . . . . .	40
4.2.1	Target execution time of 0.9 s and $K = 2$ . . . . .	41
4.2.2	Target execution time of 0.9 s and $K = 5$ . . . . .	46
4.2.3	Target execution time of 1.1 s and $K = 2$ . . . . .	48
4.2.4	Target execution time of 1.1 s and $K = 5$ . . . . .	48
4.3	Practical nature of tuple growth . . . . .	51
<b>5</b>	<b>Conclusion</b>	<b>54</b>
	<b>References</b>	<b>55</b>

# List of Tables

3.1	Configuration of a single processor . . . . .	22
3.2	Supply voltages and supported frequencies . . . . .	23
3.3	Memory access latency and TLB miss penalties for each frequency . . . . .	23
3.4	SPEC CPU2000 benchmarks and the number of instructions with MinneSPEC inputs . . . . .	24
3.5	SPEC CPU2000 benchmarks and the execution time at different frequencies . . . . .	24
3.6	SPEC CPU2000 benchmarks and the number of instructions after a workload balance . . . . .	25
3.7	SPEC CPU2000 benchmarks and the execution time at different frequencies after a workload balance . . . . .	26
3.8	SPEC CPU2000 benchmarks and the power consumed at different frequencies after a workload balance . . . . .	27
4.1	Encoding of the frequency and voltage for the sake of simplicity of representation . . . . .	30
4.2	Per-core VFS for 0.9 s target execution time . . . . .	32
4.3	Assignment of cores to clusters for the 8-cluster configuration under Case 1 and 0.9 s target execution time . . . . .	32
4.4	Assignment of cores to clusters for the 4-cluster configuration under Case 1 and 0.9 s target execution time . . . . .	32
4.5	Per-core VFS for 1.1 s target execution time . . . . .	36
4.6	Assignment of cores to clusters for the 8-cluster configuration under Case 1 and 1.1 s target execution time . . . . .	37
4.7	Assignment of cores to clusters for the 4-cluster configuration under Case 1 and 1.1 s target execution time . . . . .	37

4.8	Per-core VFS for 1.3 s target execution time . . . . .	39
4.9	Assignment of cores to clusters for the 8-cluster configuration under Case 1 and 1.3 s target execution time . . . . .	39
4.10	Assignment of cores to clusters for the 4-cluster configuration under Case 1 and 1.3 s target execution time . . . . .	39
4.11	Per-core VFS for 0.9 s target execution time and $K = 2$ . . . . .	42
4.12	Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 0.9 s target execution time and $K = 2$ . . . . .	43
4.13	Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 0.9 s target execution time and $K = 2$ . . . . .	43
4.14	Per-core VFS for 0.9 s target execution time and $K = 5$ . . . . .	47
4.15	Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 0.9 s target execution time and $K = 5$ . . . . .	47
4.16	Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 0.9 s target execution time and $K = 5$ . . . . .	47
4.17	Per-core VFS for 1.1 s target execution time and $K = 2$ . . . . .	49
4.18	Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 1.1 s target execution time and $K = 2$ . . . . .	49
4.19	Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 1.1 s target execution time and $K = 2$ . . . . .	49
4.20	Per-core VFS for 1.1 s target execution time and $K = 5$ . . . . .	50
4.21	Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 1.1 s target execution time and $K = 5$ . . . . .	51
4.22	Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 1.1 s target execution time and $K = 5$ . . . . .	51



# List of Figures

1.1	Power management in a CMP . . . . .	5
1.2	Voltage, frequency and load current traces with DVFS . . . . .	5
2.1	An example of the growth of tuples at each time step . . . . .	13
3.1	Block diagram of the simulated 16-core CMP . . . . .	21
4.1	Per-core VFS under Case 1 and 0.9 s target execution time . . . . .	33
4.2	The 8-cluster solution under Case 1 and 0.9 s target execution time . . .	34
4.3	The 4-cluster solution under Case 1 and 0.9 s target execution time . . .	35
4.4	Comparison of the power saved from the different cluster configurations under Case 1 and 0.9 s target execution time . . . . .	35
4.5	Comparison of the power saved from the different cluster configurations under Case 1 and 1.1 s target execution time . . . . .	38
4.6	Comparison of the power saved from the different cluster configurations under Case 1 and 1.3 s target execution time . . . . .	40
4.7	Per-core VFS under Case 2 and 0.9 s target execution time and $K = 2$ .	43
4.8	The 8-cluster solution under Case 2 and 0.9 s target execution time and $K = 2$ . . . . .	44
4.9	The 4-cluster solution under Case 2 and 0.9 s target execution time and $K = 2$ . . . . .	45
4.10	Comparison of the power saved from the different cluster configurations under Case 2 and 0.9 s target execution time and $K = 2$ . . . . .	46
4.11	Comparison of the power saved from the different cluster configurations under Case 2 and 0.9 s target execution time and $K = 5$ . . . . .	48
4.12	Comparison of the power saved from the different cluster configurations under Case 2 and 1.1 s target execution time and $K = 2$ . . . . .	50

4.13	Comparison of the power saved from the different cluster configurations under Case 2 and 1.1 s target execution time and $K = 5$ . . . . .	52
4.14	Tuple growth as a function of time for different number of time steps . .	53
4.15	Tuple growth as a function of time for different number of frequencies . .	53

# Chapter 1

## Introduction

### 1.1 Motivation and background

In recent years, there has been a growing demand for high-performance computing systems capable of performing a multitude of tasks. A typical superscalar processor implements a pipelined architecture that makes use of the instruction level parallelism (ILP) in a sequential instruction stream. Multiple instructions can be issued in a single clock cycle if the pipeline resources are sufficient to support them.

Meeting these challenging performance requirements using a single processor core implies the need for enormously high clock frequencies and a growth in the size and the complexity of the processor. As device sizes shrink due to technology scaling, it is indeed possible to build larger and more complex systems on the same die area, but the increased clock speed is a much more significant challenge. First, as interconnect becomes a limiting factor in design, it is harder to distribute high-speed clock signals, and to transmit data signals within a single clock cycle. Second, running a large number of devices at a very high frequency leads to very high power dissipation. Power dissipation dictates packaging and cooling costs and is also the prime concern while designing chips for battery-operated hand-held electronic devices. Moreover, increasing the clock frequencies beyond a limit is not very advantageous because it does not scale the performance of the processor by the same factor, and the diminishing returns can be attributed to the limited amount of ILP that can be extracted from a conventional superscalar processor. The external memory access times have also not kept pace with

the increasing clock speeds and this limits the performance improvement that can be achieved if memory accesses constitute a majority of the operations performed.

The memory latency and limited ILP problems can be partially overcome by simultaneous multithreading (SMT), in which instructions from multiple threads are issued in one clock cycle on a superscalar processor. A thread is a separate process with its own instructions and data. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own [1]. SMT relies on thread level parallelism, and a pipeline stall due to memory access from any one thread can be overcome by instructions from other threads, resulting in better utilization of the available processor resources. To exploit SMT, it is necessary to build additional capability to form multiple parallel threads from an application. Parallel workloads and multiprogrammed workloads have inherent parallel threads but forming threads from a sequential program requires thread-level speculation (TLS). Although SMT processors offer area-efficient throughput enhancement [2], the high throughput results in high power dissipation and the smaller area results in high power density.

Multicore processors, or chip multiprocessors (CMP), have been adopted recently as a way to achieve high performance. These consist of several processor cores on a single die: each core is relatively simpler and easier to design and validate than a single large SMT processor. Multiple applications can be run independently on each core of a CMP, or a single application can be split into several parallel threads and can be executed on the cores simultaneously. Thus, a high throughput can be achieved without increasing the clock rate. The threads running on the cores can benefit greatly from the dedicated processor resources. Each core of a CMP is also equipped with one or more levels of private cache. The dedicated cache reduces contention for cache between multiple threads and has a significant advantage over a single SMT processor where the threads compete for shared resources. The reduced complexity and smaller sizes of the cores of a CMP eliminate the necessity for long interconnects, thus eliminating significant performance bottlenecks [3].

Replicating cores, however, comes with the potential for increased area and power overheads. The idea of migrating to architectures that exploited greater parallelism than ILP was to mitigate the power dissipation due to high clock speeds. However,

the naïve application of the CMP architecture paradigm could potentially lead to large power dissipation as multiple cores operate simultaneously. A more intelligent approach manages the spatial distribution of power in a CMP architecture while ensuring that the required performance constraints are met. Thus, a key task in working with multicore architectures is in controlling the power dissipation.

## 1.2 Dynamic voltage and frequency scaling

It is well-known that there are two components of power consumption: dynamic power and static power. The major component of dynamic power is dissipated while charging and discharging the load capacitor at the output of each CMOS gate from 0 to  $V_{dd}$  and vice versa. In addition, for a short duration of time when the input to a CMOS gate is switching, both PMOS and NMOS transistors will be on and a short-circuit current flows from power supply to ground. The resulting power is called short-circuit power and is classified under dynamic power: in a well-designed circuit, its magnitude can be controlled to be much less than switching power. Static power is attributable to the nonzero currents flowing in a transistor when it is in the cut-off state, and is a growing component of the power dissipation. Unlike dynamic power, which is dissipated when the circuit is active, static power is dissipated at all times, whether the circuit is in active or standby mode.

The switching power consumed in a circuit due to charging and discharging of the load capacitance for each gate is given by

$$P_{dyn} = C_{EFF}V_{dd}^2f \quad (1.1)$$

where  $V_{dd}$  is the supply voltage and  $f$  is the clock frequency. The term  $C_{EFF} = C_L P_{0 \rightarrow 1}$ , where  $C_L$  is the load capacitance that is switched and  $P_{0 \rightarrow 1}$  is the probability that a clock event results in a  $0 \rightarrow 1$  transition. Models for the short-circuit and leakage power dissipation may be found in [4].

Switching power is a major component of the total power and it can be reduced through the following observation: in a processor, instructions do not complete execution at a constant rate. For example, cache misses result in memory accesses that are much slower than on-chip operations and are of the order of few hundreds of cycles, or there

may be pipeline stalls while the processor waits for such slower peripherals to respond. In a multicore processor, some cores may be idle as they await messages from other cores. The instruction throughput is low during such periods of low activity, and hence, if the processor is operated at a high frequency during these times, the corresponding switching transitions are essentially wasted. To reduce the total power during such periods, the processor frequency can be dynamically reduced, and the supply voltage can be proportionately lowered. It can be seen from Equation (1.1) that this can result in a cubic reduction in  $P_{dyn}$ .

The process of dynamically altering the supply voltage and frequency is commonly referred to as dynamic voltage and frequency scaling (DVFS). The effectiveness of such a scheme is directly related to the ratio of low-activity (e.g., memory-bound) cycles to CPU-bound cycles [5]. In the traditional DVFS technique, the operating system (OS) samples the processor state at regular intervals and sends the information to the Power Management (PM) unit. At the beginning of each interval, the PM unit, based on a deterministic algorithm, decides on the  $V_{dd}$  and frequency values to be applied to the current interval and sends the appropriate information to the voltage and frequency controllers. Thus, the processor is set to operate at the estimated  $V_{dd}$  and frequency values for the interval. The block diagram in Figure 1.1 shows the interaction between the PM unit, the voltage and frequency controllers, and all cores in the CMP system.

While switching from a lower frequency level to a higher frequency level, it is necessary to switch the supply voltage first to the level that can support the desired higher frequency and then only change the frequency. On the other hand, while switching from a higher frequency level to a lower frequency level, we can switch to the desired lower frequency even before a voltage switching takes place. This ensures that there is no need to stall the execution during voltage and frequency changes. An example of a DVFS schedule and the corresponding change in load current is shown in Figure 1.2.

Voltage regulators are required to deliver power to a circuit from an energy source, and are a key element of DVFS. These regulators are present in all computing systems and deliver power at the desired fixed or time-varying voltage levels, and may lie on-chip or off-chip. On-chip voltage regulators have the advantage of providing fast switching to different voltage levels as compared to off-chip voltage regulators: specifically, the voltage transition times are of the order of tens of nanoseconds for on-chip voltage

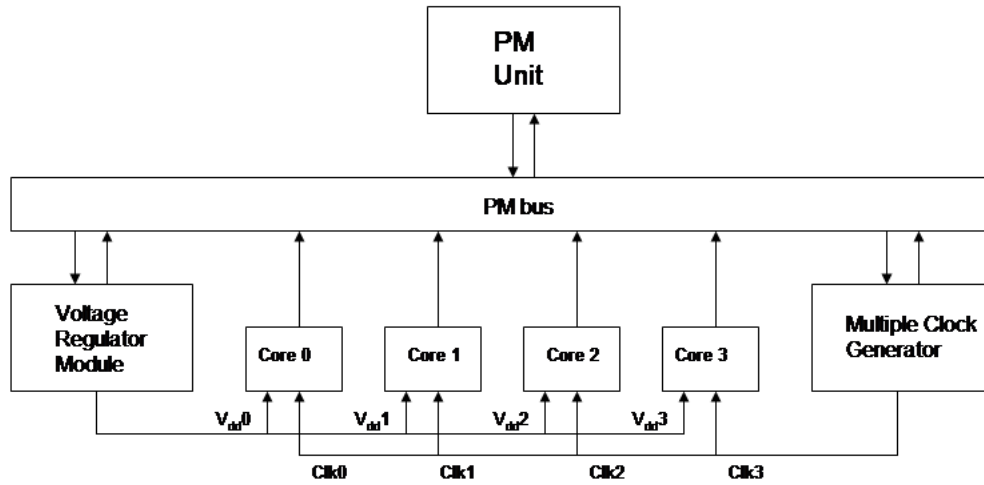


Figure 1.1: Power management in a CMP [6]

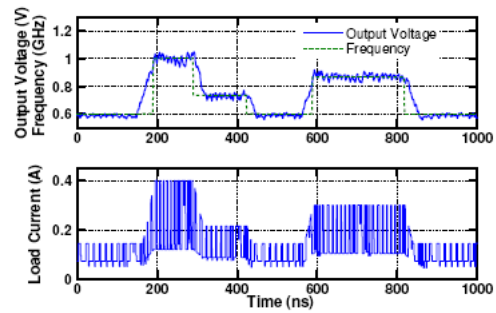


Figure 1.2: Voltage, frequency and load current changes with DVFS [5]

regulators, and of the order of tens of microseconds for off-chip regulators [5].

In principle, DVFS can be performed at the per-chip level at one extreme, with every core being bound to the same DVFS schedule, or at the per-core level at the other extreme, where an independent DVFS schedule can be applied to each core. The former requires only one voltage regulator, which may be located on-chip or off-chip, while the latter requires one regulator per core: due to the scales involved, it is essential for all of these regulators to be located on-chip. However, on-chip regulators incur significant area and power overhead by introducing large inductors and capacitors; moreover, it is difficult to build inductors with sufficient Q factors to support a large number of on-chip regulators. Hence, implementing DVFS scheme for each core in such a large multicore system is difficult in practice and the benefits are not necessarily large enough to overcome the overhead and complexity associated with such a scheme [7]. On the other hand, implementing DVFS for the entire chip simultaneously may not take full advantage of the diversity of the applications running on the cores, and may yield low power/performance improvements over running the entire chip at a constant voltage and frequency. A more reasonable middle ground is to employ a small number of on-chip regulators that drive a set of DVFS domains, or clusters, so that multiple cores are associated with each cluster. This is the problem that is addressed in this thesis.

Several prior research efforts have addressed the issue of power dissipation in a CMP. The work done in [8] explores a heterogeneous multicore architecture and describes techniques for power reduction. The cores cover a wide range of capabilities and performance levels that vary in their energy dissipation. Applications are mapped to the cores depending on the resource requirements. Differences between phases in the same application are handled by migrating the application to the core that best meets the resource demands and runtime heuristics are used for core switching. In [6], two algorithms for power management in multicore processors are discussed. One assumes discrete voltage-frequency pairs and the other assumes continuous power modes where the processor can run at any frequency and voltage within predefined upper and lower bounds. Both maximize the chip performance under a given power budget. The chip performance is measured by the total number of instructions completed by all the cores in the chip. The chip performance for each interval is predicted based on the observed performance in the previous interval. In [9] a significantly different approach to power



reduction in homogeneous multicore system called thread motion is suggested. The cores differ only in their supply voltage and operating frequencies, and applications are migrated to cores of higher or lower voltage/frequency settings, depending on the time-varying compute intensity in a program.

### 1.3 Problem description

As stated earlier, on-chip supply voltage distribution may range from per-chip DVFS to per-core DVFS, and the intermediate case of clustered DVFS is the problem studied in this thesis. Clustered DVFS keeps the overhead of multiple on-chip regulators to a reasonable amount and at the same time offers greater flexibility than per-chip DVFS. We address the problem of power dissipation in a homogeneous CMP consisting of identical cores at the design stage by clustering the cores into desired number of clusters such that the performance and power dissipation of the clusters is optimized.

To our knowledge, there has been little to no prior work in the area of clustered DVFS, since researchers have focused primarily on per-core and per-chip DVFS for CMPs. The clustering of cores in a 16-core CMP and the implementation of DVFS on a per-cluster basis has been analyzed in [7], but the clustering is not based on similarity between the cores. Two clustering methods are used: cores with consecutive indices are grouped together and cores with the same index mod 4 are clustered together. Our proposed method for core clustering, on the other hand, is based on the temporal correlation between them in the required voltage values based on running a representative set of benchmarks.

Our approach is based on running a set of representative benchmarks on the cores. We assume a multiprogramming environment where each core runs a separate program, and has a private memory and a shared main memory. Based on this representative set of user-specified benchmarks, we determine an optimal set of DVFS clusters. There can be many ways of identifying the voltage and frequency requirements of the different phases of an application and the applications could be mapped using such techniques to the appropriate DVFS clusters. Some techniques by which compiler finds regions of programs for  $V_{dd}$  and frequency adjustments have been explained in [10]. A method of embedding an architectural signature of an application in the binary of the application,

which indicates to the OS the core that it should be mapped to, in a heterogeneous CMP environment, is discussed in [11].

We divide time into intervals of equal duration and allow the possibility of dynamically modifying the voltage and frequency at the beginning of each interval. The set of voltage-frequency assignments for all the intervals is called voltage and frequency scaling (VFS) schedule. The approach first finds the optimal VFS schedule on a per-core basis using a bounded enumeration scheme, and then clusters processors together, depending on an affinity metric based on the per-core VFS schedule. Finally, the efficacy of the clustering is examined by determining the VFS schedule for each cluster.

## 1.4 Organization of the thesis

The thesis is organized as follows. Chapter 2 describes the problem formulation and the various steps of the algorithm that is used to determine the optimal DVFS clusters in a CMP. Next, Chapter 3 explains the experimental setup and the processor configuration considered. Chapter 4 presents the results for the simulated 16-core CMP in 45nm technology node, for various values of the parameters of the experiment. Finally, Chapter 5 summarizes the findings.

## Chapter 2

# Offline Profiling and Voltage and Frequency Schedule

This chapter explains our method of finding the optimal voltage and frequency schedule for each core of a CMP. As explained in Chapter 1, we look at implementing VFS for a cluster of cores of a CMP and the entire method is based on a prior knowledge of the profiles of the applications run on the cores. The steps involved and the algorithms used for each step are explained in this chapter. For offline profiling of the applications, representative benchmarks are run on the cores and information regarding the number of instructions completed and power consumed at the granularity of every  $K$  cycles is obtained, where the value of  $K$  is determined through experiments and is set to a value that captures the variations in the processor activity with good accuracy. The data obtained also gives the total number of instructions present in each benchmark. This data will be used to determine an optimal voltage and frequency assignment for each VFS interval for each processor in a CMP.

We consider a CMP executing a multiprogrammed workload where each core runs an independent application. We consider implementing VFS for such a system such that each core completes a certain fraction of the application running on it within a given duration of time and the associated cost is minimum. There are two use models that we consider that determine the fraction of completion required for all the applications and the cost function used:

1. There is a hard constraint that the application has to execute completely in a given duration  $T$  and the cost function is the power consumed
2. There is a soft constraint that the application has to complete at least 85% of the total number of instructions in it in a given duration  $T$ . There is a penalty associated with the incomplete instructions and this is built into the cost function. Cost function for this case is defined as,

$$Cost = Power(1 + K(\textit{fraction incomplete})) \quad (2.1)$$

The value of  $K$  is user defined. Higher the value of  $K$  more is the penalty for incompleteness and vice versa.

The second model above is based on the intuition that allowing a small percentage of incompleteness might allow a greater power saving since the core can operate more at lower frequencies. The decision, however, to operate at lower frequencies will have to be made in the presence of a penalty for the incomplete instructions. Thus, if there is a core running an application which does not benefit greatly from operating at high frequencies and the number of instructions completed does not increase by about the same factor as the frequency increase, then the core will choose to be at lower frequencies. The basic idea is that, for this core, the penalty for incomplete instructions is lower than the power that would be consumed if it has to completely execute all the instructions. The first model forces high frequency of operation if a benchmark has a poor rate of completion. In other words, if there are more memory-bound cycles and periods of low activity due to the benchmark, it gets assigned high frequency of operation because there is a hard constraint that it has to complete within the duration  $T$ . This is contrary to the general idea of DVFS where such a benchmark has to choose the lower frequencies because the high frequency does not result in a significant increase in throughput. Therefore, the second use model was introduced and it will be seen in the results section of the thesis that the second use model does capture the essence of DVFS.

Implementing VFS for clusters of cores in a multicore system requires a scheme to optimally cluster the cores together. Cores that have good temporal correlation in the required voltage and frequency of operation have to be clustered together. This implies that the frequency/voltage schedule that can be obtained by implementing VFS

on a per-core basis is a good metric that indicates the similarity of the cores. Such a VFS schedule gives the best assignment of voltage and frequency for each interval for the core. For example, if two cores have the same voltage/frequency assignment for each VFS interval, then it would be ideal to cluster them together. Extending the same principle for multiple cores, we look at the best VFS schedule for each core as an attribute of the core that can be used to establish the similarity among cores. This tells us that the first step in our implementation has to be finding the VFS schedule for the cores as if the VFS were performed on a per-core basis. Thus, there are three steps in our implementation of VFS for a multicore system

1. Find the optimal frequency and voltage schedule for each core of the multicore system
2. Cluster the cores using the voltage schedule obtained in step 1 as the similarity metric
3. Find the optimal frequency and voltage schedule for the clusters obtained in step 2.

We will look at each of the above steps in detail in the sections that follow.

## 2.1 Optimal frequency and voltage schedule for each core

The first step in our implementation of VFS for multicore system requires finding the optimal frequency and voltage schedule for each core such that the cost is minimum. This schedule will be used later as the basis to cluster the cores together. This section explains the algorithm used.

The multiprogrammed workload scenario requires applications to be run on a core each. Let us consider one such core. Let the target execution time for the application running on the core be  $T$ . Let this time  $T$  be divided into  $M$  equal time steps such that  $\sum_{i=1}^M t_i = T$ . Here, each time step corresponds to a VFS interval. Also, let there be a set of  $N$  discrete  $V_{dd}$ -frequency pairs available:  $(V_j, f_j)$ ,  $j \in 1, \dots, N$ . Here,  $V_j$  is the minimum supply voltage required to sustain an operating frequency of  $f_j$ . The objective is to find the assignment  $(V_j, f_j)$  for each  $t_i$  such that the associated cost is minimum.

This assignment is the optimal VFS schedule for the core. The cost function used and the constraint is determined by which one of the two use models mentioned before is considered. The cost function is power consumed and the constraint is to complete the execution of the application if first use model is considered. The cost function is given by Equation 2.1 and the constraint is to complete execution of at least 85% of the total number of instructions in the application if second use model is considered.

Algorithm 1 shows the procedure to find the optimal frequency and voltage assignment for the time steps. The algorithm starts at first time step and proceeds till the  $M^{th}$  step. At the end of each time step, we compute a set of tuples. A tuple,  $(I, C)$ , has two values.  $I$  is the number of instructions committed till the end of the time step and  $C$  is the associated cost till the end of the time step. From each tuple of the previous time step, we compute at the current time step, tuples corresponding to each of the available discrete frequencies of operation. For example, let us consider that there are two discrete frequencies of operation:  $f_1$  and  $f_2$ . Then, at the first time step we compute the tuples for each of the two frequencies. At the second step, from each of the two tuples of the first step, we calculate tuples corresponding to running at the two frequencies at the second step. For example, at the second step,

$$\begin{aligned} I_{12} &= (I_{11})_{previous} + \text{No. of instructions completed by being in } f_2 \text{ for a time } t_i \\ C_{12} &= (C_{11})_{previous} + \text{Cost of completing } (I_{12} - (I_{11})_{previous}) \text{ instructions by being} \\ &\quad \text{in } f_2 \text{ for a time } t_i \end{aligned} \tag{2.3}$$

where, the duration of time  $t_i$  is considered from the time  $(I_{11})_{previous}$  instructions have completed in  $f_2$ . There will be four tuples at the end of second time step and so on. For each tuple, we also keep track of the frequency assignment to the time steps that has resulted in the tuple. Figure 2.1 shows the growth of the tuples for the example considered.

In Figure 2.1, the tuples at each step are shown and adjacent to each tuple, the frequency assignment to the steps is shown. At each step, we need to find the number of instructions completed within a duration  $t_i$  operating in each of the discrete frequencies and add it to the previous time step's value. This computation requires the profile data of the benchmarks. The benchmarks are run at each of the discrete frequencies and the number of instructions committed and power consumed for every  $K$  cycles is reported

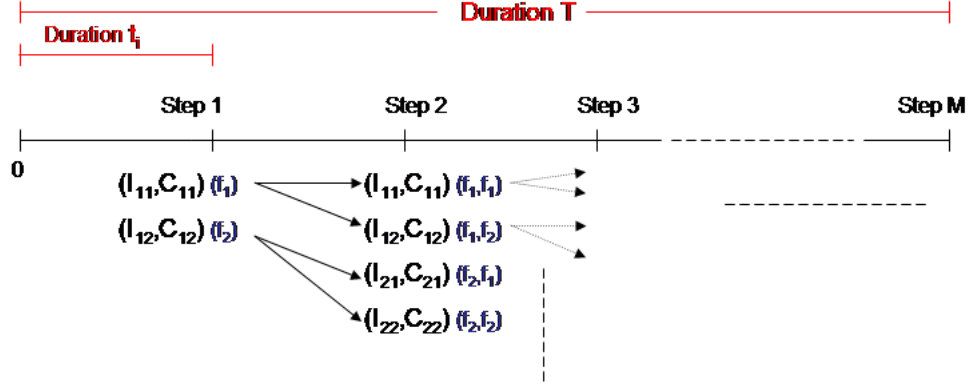


Figure 2.1: An example of the growth of tuples at each time step

for each run. The granularity  $K$  is fixed to a value that gives sufficient accuracy. We can see that the number of tuples grows exponentially with the number of steps. This can be seen in Figure 2.1, where, at the first step we have two tuples, at the second step we have four tuples. If the growth continues, then at the third step we will have eight tuples and so on. In general if there are  $N$  frequencies of operation and  $M$  time steps, the number of tuples at the end of  $M^{th}$  time step will be  $N^M$ . Eliminating provably suboptimal tuples at each time step curbs such an exponential growth. Suppose there are two tuples  $(I_1, C_1)$  and  $(I_2, C_2)$ , then, if  $I_1 \leq I_2$  and  $C_1 > C_2$ , we can be certain that  $(I_1, C_1)$  is suboptimal. This means that there is a superior tuple  $(I_2, C_2)$  that completes equal or greater number of instructions than  $I_1$  at a lower cost than  $C_1$ .  $(I_1, C_1)$  can never be a part of the optimal solution and can be eliminated. At the end of  $M^{th}$  time step, we pick from the tuples that have been maintained, the one that corresponds to the least cost such that the constraint on the number of instructions to be completed is met. The frequency assignment corresponding to the chosen tuple is the optimal frequency schedule for the core. The voltage schedule can be found from the frequency schedule by assigning the minimum voltage required to sustain each frequency in the frequency schedule.

Algorithm 1 has two major parts: first, finding the tuples, and second, deleting the suboptimal tuples. To find tuples, we iterate over each tuple from previous step and iterate over each of the available frequencies. We know that the number of tuples in a time step is of the order  $O(N^M)$ , where  $N$  is the number of available frequencies and

---

**Algorithm 1** Per-core VFS
 

---

```

1: for each frequency  $k$  from the list of frequencies do
2:    $(I_{0k} = 0, C_{0k} = 0)$  // Initialize
3:    $F_{0k} = 0$ 
4: end for
5: for each time step  $i \in M$  do
6:   for each tuple  $j$  of previous step do
7:     for each frequency  $k$  from the list of frequencies do
8:       Compute  $(I_{jk}, C_{jk})$  and  $F_{jk}$  where,
9:        $I_{jk} = (I_j)_{prev} +$  No. of instructions committed in frequency  $k$  within a duration  $t_i$ 
        considered from the time  $(I_j)_{prev}$  instructions complete in frequency  $k$ 
10:       $C_{jk} = (C_j)_{prev} +$  Cost of completing  $(I_{jk} - (I_j)_{prev})$  instructions in frequency  $k$ 
11:       $F_{jk} = ((F_j)_{prev}, k)$  // Frequency assignment to the time steps
12:      Insert  $(I_{jk}, C_{jk})$  into a list of tuples in the descending order of power
13:      Insert  $F_{jk}$  into a list of frequency assignments in the same order as  $(I_{jk}, C_{jk})$ 
14:     end for
15:   end for
16:   for all  $j$  such that  $j <$  number of tuples of step  $i$  do
17:     for all  $l = j + 1$  such that  $l <$  number of tuples of step  $i$  do
18:       if  $I_j \leq I_l$  then
19:         Delete tuple  $(I_j, C_j)$  as it is suboptimal
20:         Delete frequency assignment information  $F_j$ 
21:         Break from inner loop
22:       end if
23:     end for
24:   end for
25: end for
26: Pick the tuple with the least cost such that the application is completed if first use model
    is considered or such that at least 85% of the total number of instructions are completed if
    second use model is considered.
27: The frequency assignment corresponding to the chosen tuple is the optimal frequency sched-
    ule for the core.
28: For each frequency of the optimal frequency schedule, assign the minimum voltage required
    to sustain the frequency and this gives the optimal voltage schedule for the core.

```

---



$M$  is the number of time steps. This is a pessimistic case, since in practice, eliminating suboptimal tuples renders the complexity less than  $O(N^M)$ . With the pessimistic assumption for now, we find that the complexity of finding tuples at each time step is,

$$O(N.N^M) = O(N^{M+1}) = O(N^M)$$

The second part is deleting suboptimal tuples. We will have tuples arranged in descending order of power after the first step. From Algorithm 1, it can be seen that, for deleting suboptimal tuples, we iterate over each tuple at a time step and compare it with every other tuple at the time step. This leads to a complexity of  $O(N^M.N^M) = O(N^{2M})$ . But, for the suboptimality check, we usually never need to pursue comparison beyond about 100 tuples as any suboptimality that exists can be found within this number of tuples. This is an observation made during implementation and this reduces the complexity to  $O(N^M)$ . We iterate the task of finding tuples and deleting suboptimal tuples over each of the  $M$  time steps. Therefore, the complexity of the entire algorithm is  $O(MN^M)$ . We choose a small value for  $M$ . Also, the factor  $N^M$  is due to the number of tuples in a time step. This is a pessimistic assumption as mentioned before and is lower in practice due to eliminating suboptimal tuples. Therefore, the complexity of the algorithm is much lower than  $O(MN^M)$ . For the first use model where we have a hard requirement that the application has to completely execute in a given duration  $T$ , we can eliminate additional tuples at each time step. This is based on the observation that if the application has to complete by the end of  $M^{th}$  time step, it has to complete approximately  $\frac{i}{M}^{th}$  of the total number of instructions by the end of  $i^{th}$  time step. A tolerance factor is included along with this expected fraction of completion and any tuple that has completed fewer instructions than this limit is eliminated at each step. An example of growth of tuples seen in practice is shown Section 4.3.

## 2.2 Clustering the cores

The per-core voltage schedule obtained for each core of the multicore system represents the best way to vary the voltage for the core. When two or more cores have to be combined together, we need to ensure that they correlate well in their requirement of the voltage in each time interval. We use K-means clustering algorithm to cluster the cores. This section explains the algorithm used.

The voltage assignment for each core obtained as in Section 2.1 will have voltages assigned for each of the  $M$  time steps. Thus, the voltage schedule can be seen as a point with  $M$  dimensions. Let us consider  $n$  cores. After the per-core VFS, we get  $n$  voltage schedules and each schedule can be seen as a  $M$ -dimensional point. Let these be denoted as  $x_1, x_2, \dots, x_n$ . The K-means clustering algorithm partitions these  $n$  points into  $K$  sets ( $K < n$ )  $S = S_1, S_2, \dots, S_K$  such that the sum of the squares of the distances between the points and their respective cluster centres is minimum. The objective function that is to be minimized can be written as,

$$J = \sum_{j=1}^K \sum_{x_i \in S_j} \|x_i - \mu_j\|^2 \quad (2.4)$$

where,  $\mu_j$  is the mean of the points in  $S_j$ .

Algorithm 2 illustrates the clustering algorithm used. The K-means clustering algorithm is an iterative algorithm. From the set of points to cluster,  $K$  random points are chosen initially as the cluster centres. The distances of each point from the  $K$  cluster centres are calculated. Each point is assigned to the cluster whose centre is the closest to the point. The centres of the clusters so obtained are recalculated. This is followed by reassignment of the points to the appropriate clusters as done before. The process is iterated till the centres of the clusters do not change from their previous iteration values. The algorithm has the limitation that it converges to a local minima and the result depends largely on the initial centroids chosen. Hence, many trials with random initial centroids are done. This is an attempt to find global minimum but it may or may not be successful since the search is limited by the number of random trials. Nevertheless, it is better than a single run of K-means algorithm. The square root of the sum of the squares of the distances of all the points from their respective cluster centres is calculated for each trial. This is considered a quality metric of the clusters obtained and the trial with the minimum value of this metric is chosen and the result is the final cluster. In practice, there will be a fixed number of voltage regulators or voltage domains on chip. Each cluster can be thought of as one voltage domain. These voltage domains have to appropriately loaded and hence the sizes of the clusters should be balanced. We therefore set upper and lower limit on the number of cores per cluster. An upper limit can also be due to the regulator's power delivering capability.

---

**Algorithm 2** K-means clustering
 

---

```

1:  $T$  = Number of K-means trials.
2:  $Max$  = Upper limit of number of points per cluster.
3:  $Min$  = Lower limit of number of points per cluster.
4: for each trial  $t \in T$  do
5:    $X = x_1, \dots, x_n$  Points.
6:    $K$  = Number of clusters.
7:   for each cluster  $j \in K$  do
8:      $\mu_j$  = A random point from  $X$  // Initialize
9:   end for
10:  for each point  $x_i \in X$  do
11:    for each cluster  $j \in K$  do
12:      Find  $d_{ij} = \|x_i - \mu_j\|^2$ 
13:    end for
14:  end for
15:  for each point  $x_i \in X$  do
16:    Find min  $d_{ij}$ 
17:     $k' = j$  corresponding to min  $d_{ij}$ 
18:    Assign point  $x_i$  to cluster  $k'$ 
19:  end for
20:  for each cluster  $j \in K$  do
21:     $\mu'_j$  = Centre of cluster  $j$ 
22:  end for
23:  for each cluster  $j \in K$  do
24:    if  $\mu'_j \neq \mu_j$  then
25:      Goto step 10.
26:    end if
27:  end for
28:  if All clusters satisfy  $Max$  and  $Min$  constraints then
29:    Find  $QualityMetric_t = \sqrt{\sum_{j=1}^K \sum_{x_i \in S_j} \|x_i - \mu_j\|^2}$  for trial  $t$ .
30:    Store the clusters obtained for trial  $t$ .
31:    Goto step 4.
32:  else
33:    Check for clusters with  $> Max$  points. Limit one of these to  $Max$  points,  $K = K - 1$ ,
    cluster the remaining points again to  $K$  clusters.
34:    After  $\frac{K}{2}$  clusters are limited to  $Max$  points, check for clusters with  $> Min$  points.
    Limit one of these to  $Min$  points,  $K = K - 1$ , cluster the remaining points to  $K$ 
    clusters.
35:    Find  $QualityMetric_t = \sqrt{\sum_{j=1}^K \sum_{x_i \in S_j} \|x_i - \mu_j\|^2}$  for trial  $t$ .
36:    Store the clusters obtained for trial  $t$ .
37:    Goto step 4.
38:  end if
39: end for
40: Pick the trial with min  $QualityMetric_t$  among all  $t \in T$ .

```

---

## 2.3 Optimal frequency and voltage schedule for a cluster of cores

This section explains the procedure to find the frequency and voltage schedule for the clusters. Algorithm 3 shows the steps followed. The problem formulation is similar to that of implementing per-core VFS as explained in Section 2.1. Let the target execution time for the applications running on all the cores in the cluster be  $T$ . Let this time  $T$  be divided into  $M$  equal time steps such that  $\sum_{i=1}^M t_i = T$ . Let there be a set of  $N$  discrete  $V_{dd}$ -frequency pairs available:  $(V_j, f_j), j \in 1, \dots, N$ . The objective is to find the assignment  $(V_j, f_j)$  for each  $t_i$  such that the sum of the associated costs of all the cores in the cluster is minimum under the constraint that a certain percentage of the total number of instructions present in each of the applications is completed. The procedure to find the frequency and voltage schedule for a cluster proceeds in the same way as implementing VFS for a core. The difference is that instead of calculating tuples for one core at each time step, we calculate tuples for all the cores in a cluster at each time step. Another difference is that the suboptimal tuple for a core can be eliminated only if the corresponding tuples for all other cores in the cluster are also suboptimal. From Algorithm 3, it can be seen that it is very similar to Algorithm 1 except that while finding the tuples, we now iterate over each core in the cluster. This can be seen from line 8 of Algorithm 3. Also, a suboptimality check has to be done across all the cores for a tuple if it has to be deleted. This can be seen from line 23 of Algorithm 3. The complexity of the algorithm is the same as that of Algorithm 1 since we do essentially the same but for multiple cores at a time. The complexity is thus given by  $O(MN^M)$ . As discussed before for Algorithm 1, this complexity is generally lower in practice because of eliminating tuples.

---

**Algorithm 3** Per-cluster VFS
 

---

```

1: for each core  $c \in Cluster$  do
2:   for each frequency  $k$  from the list of frequencies do
3:      $(I_{c0k} = 0, C_{c0k} = 0)$  // Initialize
4:      $F_{c0k} = 0$ 
5:   end for
6: end for
7: for each time step  $i \in M$  do
8:   for each core  $c \in Cluster$  do
9:     for each tuple  $j$  of previous step do
10:      for each frequency  $k$  from the list of frequencies do
11:        Compute  $(I_{cjk}, C_{cjk})$  and  $F_{jk}$  where,
12:         $I_{cjk} = (I_{cj})_{prev} +$  No. of instructions committed in frequency  $k$  in core  $c$  within a
        duration  $t_i$  considered from the time  $(I_j)_{prev}$  instructions complete in frequency
         $k$  in core  $c$ 
13:         $C_{cjk} = (C_{cj})_{prev} +$  Cost of completing  $(I_{cjk} - (I_{cj})_{prev})$  instructions in frequency
         $k$  in core  $c$ 
14:         $F_{cjk} = ((F_{cj})_{prev}, k)$  // Frequency assignment to the time steps for core  $c$ 
15:        Insert  $(I_{cjk}, C_{cjk})$  into a list of tuples in the descending order of power for first
        core. For all other cores, follow the same order for tuples as the first core.
16:        Insert  $F_{cjk}$  into a list of frequency assignments in the same order as  $(I_{cjk}, C_{cjk})$ 
        for all cores.
17:      end for
18:    end for
19:  end for
20:  for all  $j$  such that  $j <$  number of tuples of step  $i$  do
21:    for all  $l = j + 1$  such that  $l <$  number of tuples of step  $i$  do
22:      if  $I_j \leq I_l$  for first core then
23:        if  $(I_j, C_j)$  is suboptimal in all other cores then
24:          Delete tuple  $(I_j, C_j)$  for all cores
25:          Delete frequency assignment information  $F_j$  for all cores
26:          Break from inner loop
27:        end if
28:      end if
29:    end for
30:  end for
31: end for
32: Pick the solution with the least sum of costs of all the cores such that the applications
    that are run on all the cores in the cluster meet the required completion constraint. The
    applications must completely execute if first use model is considered or they must complete
    at least 85% of the number of instructions in them if the second use model is considered.
33: The frequency assignment corresponding to the chosen tuple is the optimal frequency sched-
    ule for the core.
34: For each frequency of the optimal frequency schedule, assign the minimum voltage required
    to sustain the frequency and this gives the optimal voltage schedule for the core.

```

---

## Chapter 3

# Experimental Setup

This chapter provides an overview of our experimental setup, and describes the configuration of the processor that is simulated, and a summary of a basic set of simulations that underpin our algorithm. Recall that our approach is based on determining the optimal per-core VFS schedule and using this to create a set of DVFS clusters. This implies that it is essential to determine the performance of each core if it is provided with each of the available voltages/frequencies. These single voltage/frequency simulations are translated into a set of reports that capture the processor state at regular intervals in each of the available frequencies, using which we can determine the frequency scaling pattern for an application. These reports from simulations are also used after clustering, to determine the degradation in performance after clustered VFS, as compared to per-core VFS.

We consider a homogeneous 16-core CMP in a 45nm technology. Figure 3.1 shows the organization of the CMP that is considered. It has 16 processing cores, and each core has its private L1 and L2 cache. The highest level cache is L3 and is shared. Under our assumption of a multiprogramming environment with individual programs running on separate cores, we use a cycle-accurate out-of-order uniprocessor simulator, SimpleScalar [12], to simulate each individual core of the CMP. Clearly, using a uniprocessor simulator such as SimpleScalar to model a CMP system is an approximation: in particular, the shared cache cannot be modeled using this simulator. This approximation is reasonable under the assumption that the contention for shared cache is small and each processor runs an independent application. The simplicity and ease of use

of SimpleScalar drives our choice, and for the purposes of demonstrating the solution techniques, this is adequate. We note that the essential idea of our algorithm can also be exercised with any other multicore simulator. For power estimation, we employ Wattch, an architectural-level power modeling tool that is integrated with SimpleScalar. The interaction between SimpleScalar and Wattch is that the former, as a performance simulator, provides the cycle-by-cycle hardware access counts to the latter. This information is used by Wattch to compute the power dissipation [13]. The technology-specific parameters provided with Wattch were originally taken from CACTI 1.0 tech report, but these parameters are outdated today. As a part of the implementation, we updated these parameters based on the ORION 2.0 [14] technology file.

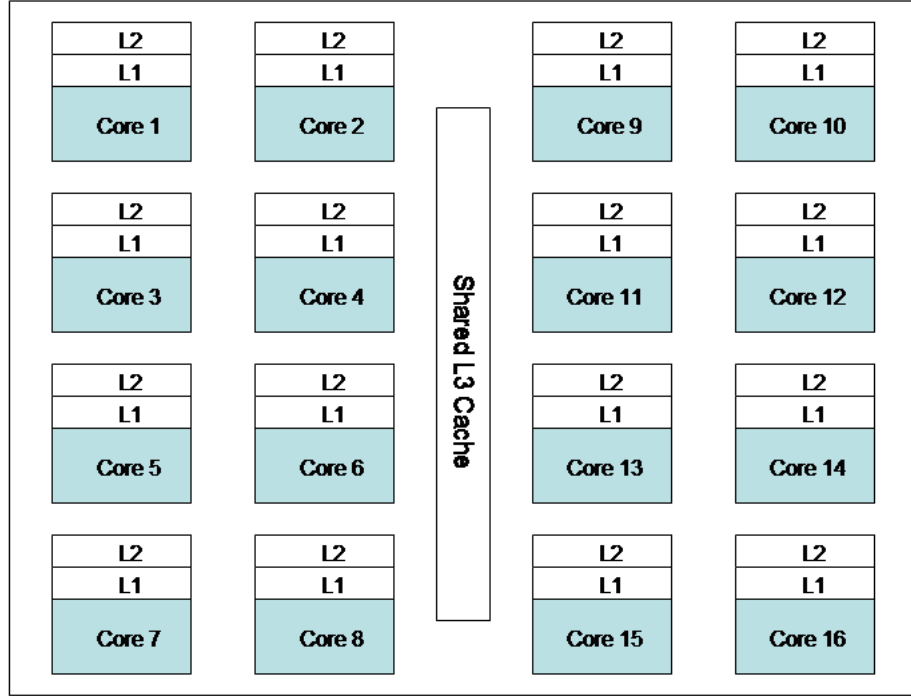


Figure 3.1: Block diagram of the simulated 16-core CMP

The first step in our implementation consists of gathering the profiles of representative benchmarks. Sixteen benchmarks from the SPEC CPU2000 suite were used; each is assumed to run on a separate core. Of these, nine are integer and seven are floating point benchmarks. These benchmarks were run on the SimpleScalar simulator

with the reduced MinneSPEC input sets. It is widely accepted that simulations with MinneSPEC inputs may not always match the profiles obtained with SPEC CPU2000 reference inputs but can provide another workload suitable for producing reportable results for simulation-based studies, and with significantly lower run-times [15]. From the benchmark runs, we report the number of instructions committed and power consumed for every 1000 cycles. The SimpleScalar source code was modified slightly to incorporate the capability to obtain such reports.

Table 3.1: Configuration of a single processor

Fetch/Decode/ Issue/Commit width	4/4/4/4 (instructions/cycle)
RUU size	64 entries
LSQ size	32 entries
Private L1 Data cache	16KB, 4-way set associative, 32B block size
Private L1 Instruction cache	16KB, 4-way set associative, 32B block size
Private L2 Unified Data and Instruction cache	512KB, 8-way set associative, 64B block size
Memory access bus width	8 bytes
Data Translation Lookaside Buffer	512KB, 4-way set associative, 4KB block size
Instruction Translation Lookaside Buffer	256KB, 4-way set associative, 4KB block size
Number of integer ALUs	4
Number of integer multiplier/dividers	4
Number of floating point ALUs	2
Number of floating point multiplier/dividers	2
Number of memory system ports available to CPU	2 (1 read, 1 write)

The processor configuration modeled in SimpleScalar is as shown in Table 3.1. Each processing core is an out-of-order processor and has private L1 data and instruction caches and a private unified L2 cache. The L1 data and instruction cache access latency is 1 processor cycle, and the L2 unified cache access latency is 4 processor cycles. Since



the L1 and L2 caches are private for each core, their access latency in terms of the number of processor cycles is the same irrespective of the frequency of operation of the processor. As stated before, the shared L3 accesses are assumed to be low because each core runs an independent application, and hence ignored. The main memory operates at a constant voltage and frequency that is unrelated to any DVFS operations on the core, and hence memory access and response times are constant over any DVFS operation. If the processor frequency is altered, this constant time translates to a varying number of cycles required for main memory access.

Table 3.2: Supply voltages and supported frequencies

Processor frequency (in GHz)	0.5	0.7	0.9	1	1.1
Supply Voltage (in V)	0.8	0.9	1	1.1	1.2

We assume a set of discrete  $V_{dd}$ -frequency pairs available from which the voltage and frequency assignment can be done. In practice, it is difficult to obtain a relation in closed form between delay of a logic gate and the supply voltage. Hence, we cannot establish relation in closed form between the supply voltage and the design frequency it can support. The gates are usually characterized for discrete values of supply voltage. Therefore, we assume discrete  $V_{dd}$ -frequency pairs. Table 3.2 shows the 5 discrete voltage-frequency pairs that are assumed for each processor. These  $V_{dd}$ -frequency pairs shown in the table are inferred from Figure 5.7.6 of [16]. The memory access latency and Translation Lookaside Buffer (TLB) miss penalty corresponding to each of the processor frequencies from Table 3.2 are shown in Table 3.3.

Table 3.3: Memory access latency and TLB miss penalties for each frequency

	Processor frequency (in GHz)				
	0.5	0.7	0.9	1	1.1
Memory access latency (in processor cycles)	25	35	45	50	55
TLB miss penalty (in processor cycles)	30	42	54	60	66

The sixteen SPEC CPU2000 benchmarks are run on SimpleScalar at each of the five frequencies listed in Table 3.2. Table 3.4 shows the number of instructions in each of the sixteen benchmarks. The execution time for these benchmarks, at each of the five frequencies, is shown in Table 3.5. It can be seen from the table that the execution time varies over a wide range across the different benchmarks, but we assume that the

Table 3.4: SPEC CPU2000 benchmarks and the number of instructions with MinneSPEC inputs

Benchmark name	Number of instructions
gap	761345807
apsi	340293846
equake	1021608702
parser	4527010000
eon.cook	854124533
eon.kajiya	1070280973
eon.rushmeier	933324650
bzip2.program	2159340000
bzip2.graphic	2643940000
bzip2.source	1819780020
mgrid	114853275
applu	88149305
mesa	1608605433
galgel	347612000
art	1660421875
twolf	972728045

Table 3.5: SPEC CPU2000 benchmarks and the execution time at different frequencies

Benchmark name	Time in 0.5 GHz (in s)	Time in 0.7 GHz (in s)	Time in 0.9 GHz (in s)	Time in 1 GHz (in s)	Time in 1.1 GHz (in s)
gap	0.843	0.633	0.517	0.473	0.436
apsi	0.336	0.241	0.188	0.169	0.154
equake	0.918	0.682	0.550	0.501	0.460
parser	5.428	4.225	3.566	3.291	3.065
eon.cook	0.883	0.631	0.491	0.441	0.401
eon.kajiya	1.181	0.843	0.656	0.591	0.537
eon.rushmeier	1.012	0.723	0.563	0.506	0.460
bzip2.program	1.967	1.480	1.210	1.107	1.023
bzip2.graphic	2.352	1.769	1.445	1.321	1.220
bzip2.source	1.740	1.337	1.113	1.024	0.952
mgrid	0.095	0.077	0.067	0.062	0.058
applu	0.077	0.059	0.049	0.045	0.042
mesa	1.298	0.931	0.727	0.655	0.596
galgel	0.267	0.192	0.150	0.136	0.124
art	2.683	2.542	2.472	2.356	2.261
twolf	1.044	0.746	0.580	0.522	0.475

same benchmark repeats after completion, to ensure that the sixteen cores are fully occupied throughout the simulation interval<sup>1</sup>. We choose to simulate the core for a set of instructions that takes 1.5s of execution time at 0.5 GHz. Table 3.6 shows the number of instructions in the same SPEC CPU2000 benchmarks after this workload balance.

The execution times in each of the five frequencies for these workload balanced benchmark runs is shown in Table 3.7: it is easily seen, as expected, that all entries in the first column correspond to 1.5s. The power consumed by these workload balanced benchmark runs in each of the five frequencies is shown in Table 3.8.

Table 3.6: SPEC CPU2000 benchmarks and the number of instructions after a workload balance

Benchmark name	Number of instructions
gap	1366098626
apsi	1526024327
equake	1767030874
parser	1247259802
eon.cook	1442820582
eon.kajiya	1358866813
eon.rushmeier	1378284359
bzip2.program	1643565536
bzip2.graphic	1695866443
bzip2.source	1581854527
mgrid	1817034005
applu	1706891547
mesa	1855082402
galgel	1952268497
art	928477187
twolf	1394783964

The tables described above present a summary of the simulation runs. More detailed reports of the number of instructions committed and power consumed, at intervals of every 1000 cycles, are obtained from each of the simulation runs and the reports will be used by our algorithm to implement VFS. This information is required by our algorithm,

<sup>1</sup> Note that this is a reasonable assumption that appropriately simulates the representative mix of benchmark programs and assigns it to a specific cluster. In particular, if a program were to end early, it would no longer need DVFS capability, and the inclusion of its core into the chosen cluster would not affect the power dissipation of the core, particularly if it could be power-gated.

Table 3.7: SPEC CPU2000 benchmarks and the execution time at different frequencies after a workload balance

Benchmark name	Time in 0.5 GHz (in s)	Time in 0.7 GHz (in s)	Time in 0.9 GHz (in s)	Time in 1 GHz (in s)	Time in 1.1 GHz (in s)
gap	1.5	1.129	0.925	0.846	0.781
apsi	1.5	1.075	0.840	0.756	0.688
equake	1.5	1.099	0.877	0.795	0.728
parser	1.5	1.166	0.983	0.906	0.843
eon.cook	1.5	1.072	0.834	0.750	0.682
eon.kajiya	1.5	1.072	0.834	0.750	0.682
eon.rushmeier	1.5	1.072	0.834	0.750	0.682
bzip2.program	1.5	1.129	0.923	0.844	0.780
bzip2.graphic	1.5	1.127	0.920	0.841	0.777
bzip2.source	1.5	1.145	0.949	0.872	0.809
mgrid	1.5	1.216	1.059	0.983	0.921
applu	1.5	1.142	0.948	0.870	0.806
mesa	1.5	1.077	0.843	0.760	0.692
galgel	1.5	1.080	0.846	0.764	0.697
art	1.5	1.420	1.380	1.315	1.262
twolf	1.5	1.072	0.833	0.750	0.682

Table 3.8: SPEC CPU2000 benchmarks and the power consumed at different frequencies after a workload balance

Benchmark name	Power in 0.5 GHz (in W)	Power in 0.7 GHz (in W)	Power in 0.9 GHz (in W)	Power in 1 GHz (in W)	Power in 1.1 GHz (in W)
gap	0.6213	0.9924	1.4799	1.9601	2.5273
apsi	0.6497	1.0792	1.6769	2.2231	2.9079
equake	0.6754	1.0941	1.6815	2.2208	2.8809
parser	0.5782	0.8982	1.3175	1.7298	2.2304
eon.cook	0.6430	1.0525	1.6350	2.1637	2.8230
eon.kajiya	0.6392	1.0517	1.6123	2.1488	2.7965
eon.rushmeier	0.6395	1.0363	1.6111	2.1459	2.7892
bzip2.program	0.6486	1.0213	1.5558	2.0422	2.6311
bzip2.graphic	0.6563	1.0409	1.5673	2.0724	2.6842
bzip2.source	0.6390	1.0077	1.4976	1.9742	2.5469
mgrid	0.7208	1.1036	1.5906	2.0881	2.6859
applu	0.6953	1.1109	1.6503	2.1827	2.8125
mesa	0.6708	1.0967	1.6845	2.2452	2.9250
galgel	0.7592	1.2407	1.9122	2.5433	3.3182
art	0.5261	0.7294	0.9970	1.2885	1.6364
twolf	0.6024	0.9849	1.5360	2.0234	2.6602

which involves steps that must have knowledge of the number of instructions committed in a VFS interval, at various frequencies.

# Chapter 4

## Results

We implement per-cluster VFS for the 16-core CMP described in Chapter 3. The VFS is implemented for the two use models:

1. The applications running on the cores of the CMP are required to execute completely
2. The applications running on the cores of the CMP must complete at least 85% of their total number of instructions, and are penalized for noncompletion.

In both cases, the objective is to minimize the corresponding cost function, as defined in Chapter 2. As described earlier, the implementation consists of 3 steps:

1. Implementing per-core VFS to obtain the optimal voltage and frequency schedule for each core
2. Clustering the cores into the required number of clusters using the optimal voltage schedule obtained from step 1
3. Implementing per-cluster VFS and obtaining the optimal voltage and frequency schedule for the clusters.

We evaluate configurations of 1, 4, 8 and 16 clusters. The 1-cluster configuration corresponds to the per-chip VFS case and the 16-cluster configuration corresponds to the per-core VFS case. For every cluster configuration, we set upper and lower limits on the number of cores per cluster. Note that each cluster corresponds to a separate voltage

regulator: the goal of placing these bounds is to ensure that no regulator is excessively overloaded or underloaded. For the 4-cluster configuration, if an equal distribution of the cores to the clusters is desired, then ideally, each cluster should have  $N_{ideal} = \frac{16}{4} = 4$  cores. We allow a variation around this ideal number, setting upper and lower limits as

$$\lfloor 0.75N_{ideal} \rfloor < \text{Number of cores per cluster} < \lceil 1.25N_{ideal} \rceil \quad (4.1)$$

In other words, for the 4-cluster case, the number of cores per cluster must be between 3 and 5; for the 8-cluster case, where  $N_{ideal} = 2$ , this number must be between 1 and 3.

It is obvious that the per-core VFS, corresponding to the 16-cluster configuration here, gives the best power saving, subject to limitations in the optimality of our heuristic algorithm. Our experiments will evaluate the variation in the power as we move from per-core to per-chip VFS, and we examine how much performance we must sacrifice to match the power dissipation of the per-core case.

Our VFS computations use a variety of allowable voltage and frequency combinations, as taken from [16], and are summarized in Table 4.1.

Table 4.1: Encoding of the frequency and voltage for the sake of simplicity of representation

Frequency	Voltage	Encoding
0.5 GHz	0.8 V	1
0.7 GHz	0.9 V	2
0.9 GHz	1 V	3
1 GHz	1.1 V	4
1.1 GHz	1.2 V	5

## 4.1 Case 1: All applications execute completely

This section presents evaluation results for the first use model, where all applications are required to run to completion. To determine a reasonable target execution time, we observe from Tables 3.7 and 3.8 that as the frequency increases, the execution time decreases and the power increases monotonically. To avoid trivial solutions, choose a target execution time that is intermediate to the two extremes. Note that since the workload is balanced, this intermediate execution time is reasonable for all processors.



From Table 3.7, we can see that the slowest execution time is 1.5 s and the fastest is 0.682 s. We consider target times in between these extremes. The execution times considered are 0.9 s, 1.1 s, and 1.3 s.

The target time is divided into 10 VFS intervals of equal duration, and our algorithm is tasked with determining the best possible frequency assignment to each of the intervals such that the cost function, i.e., the power, is minimized. Changes in the voltage and frequency may only be made at the beginning of each interval.

#### 4.1.1 Target execution time of 0.9 s

As stated earlier, the target execution time of 0.9 s is applied as a constraint on all of the 16 benchmarks, and the entire period is divided into 10 VFS intervals of 0.09 s each. Since the voltage transition times with on-chip regulators are of the order of tens of nanoseconds, and we can see that this is negligible when compared to the duration of each VFS interval, in all our experiments, we ignore the effect of finite voltage transition times. We then implement the first step of finding the per-core VFS schedule. Table 4.2 shows the VFS schedule obtained for each core (recall that each core runs a separate benchmark) and the respective power consumed. The VFS schedule in the 10 intervals is represented by a 10-element sequence, where each number corresponds to a voltage level, as described by the encoding in Table 4.1.

Using the VFS schedule from Table 4.2 as the points to cluster, the K-means clustering algorithm explained in Section 2.2 is implemented. Tables 4.3 and 4.4 show the assignment of core to the clusters for the 8-cluster and 4-cluster configurations, respectively.

Next, we examine the impact of clustering on the optimal clustered voltage schedule. For the 8-cluster case, the per-core VFS schedules seen in Figure 4.1 are grouped based on a temporal correlation between the required voltage values into 8 clusters using K-means algorithm and these are shown in Figure 4.2. The VFS schedule obtained for the clusters is also shown in the figure using the black line. For the per-chip VFS case, the schedule for the cluster with all the 16 cores together is shown in Figure 4.1 by the black curve. As we can see, the black curve uses the highest voltage, and is constrained by the two benchmarks *mgrid* and *art* that use the highest voltage over all intervals. A similar curve for the 4-cluster case is shown in Figure 4.3: this solution indicates

Table 4.2: Per-core VFS for 0.9 s target execution time

Core/Benchmark	VFS schedule	Power consumed (in W)
gap	3 3 3 3 4 3 3 4 3 4	1.6200
apsi	2 3 2 3 2 3 3 3 3 3	1.4933
equake	3 3 3 3 3 2 3 3 3 3	1.6115
parser	4 4 4 4 4 4 4 5 4 4	1.7774
eon.cook	3 3 3 3 2 2 2 3 3 3	1.4478
eon.kajiya	3 3 3 3 2 2 2 3 3 3	1.4324
eon.rushmeier	3 3 3 3 2 2 2 3 3 3	1.4268
bzip2.program	3 3 4 3 3 3 3 3 4 4	1.6916
bzip2.graphic	3 3 3 3 4 5 2 3 3 3	1.6956
bzip2.source	3 3 3 4 4 3 4 4 4 4	1.7847
mgrid	5 5 5 5 5 5 5 5 5 5	2.6857
applu	4 3 4 3 3 4 4 3 4 4	1.9690
mesa	2 3 3 3 3 3 3 2 2 3	1.5128
galgel	3 3 3 3 2 3 2 3 3 3	1.7457
art	5 5 5 5 5 5 5 5 5 5	1.6401
twolf	2 3 3 3 3 3 2 2 3 3	1.3571
	Total power	26.8916

Table 4.3: Assignment of cores to clusters for the 8-cluster configuration under Case 1 and 0.9 s target execution time

Cluster 1	eon.cook, eon.kajiya, eon.rushmeier
Cluster 2	galgel, apsi, equake
Cluster 3	mgrid, art
Cluster 4	bzip2.source, gap
Cluster 5	applu, bzip2.program
Cluster 6	mesa, twolf
Cluster 7	parser
Cluster 8	bzip2.graphic

Table 4.4: Assignment of cores to clusters for the 4-cluster configuration under Case 1 and 0.9 s target execution time

Cluster 1	eon.cook, eon.kajiya, eon.rushmeier, galgel, equake
Cluster 2	bzip2.source, applu, gap, bzip2.program
Cluster 3	mesa, bzip2.graphic, twolf, apsi
Cluster 4	parser, mgrid, art

that cores with a natural VFS “affinity” are indeed clustered together, but some of the clusters, such as cluster 3, are not entirely intuitive.

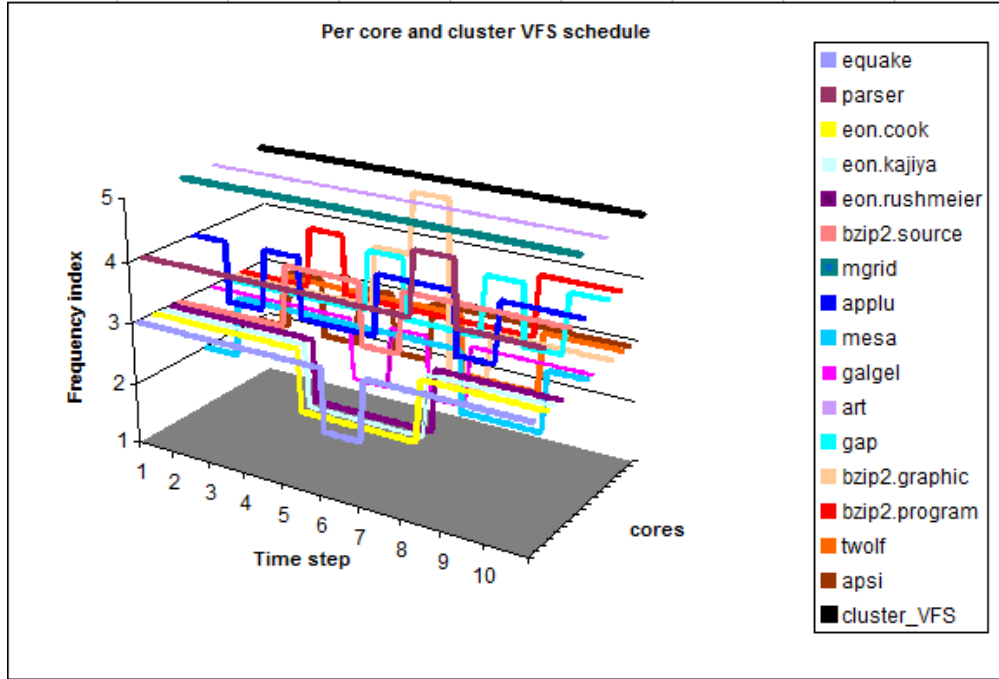


Figure 4.1: Per-core VFS under Case 1 and 0.9 s target execution time

The per-core VFS provides the potential for the best possible power saving. Though a per-core solution requires the overhead of an excessive number of voltage regulators, this solution provides a bound on the best achievable power under clustering. To evaluate the clustered solutions, we compare the power dissipated under these cluster configurations with the per-core case. We also consider the possibility of trading off performance for power, and evaluate the reduction in performance necessary to achieve the power level of the per-core VFS case.

Figure 4.4 shows the result of this evaluation, plotting the reduction in power for a performance loss. We call a relaxation in the target execution time as performance loss. The baseline for the power gain corresponds to the per-core (i.e., 16-cluster) VFS case, so that the per-core case that meets the performance specification corresponds to a point at the origin. The point at which each line crosses the x-axis corresponds

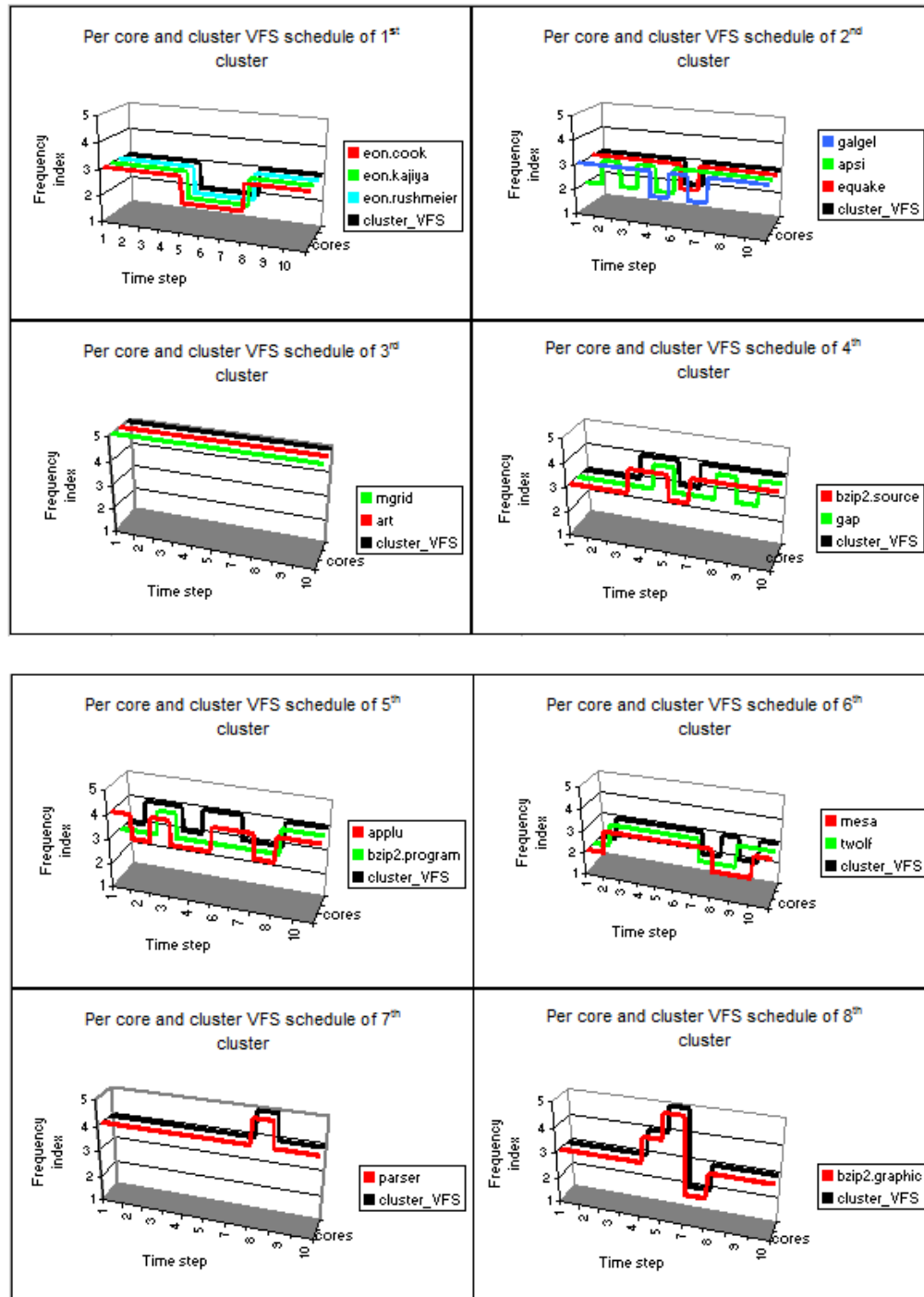


Figure 4.2: The 8-cluster solution under Case 1 and 0.9 s target execution time

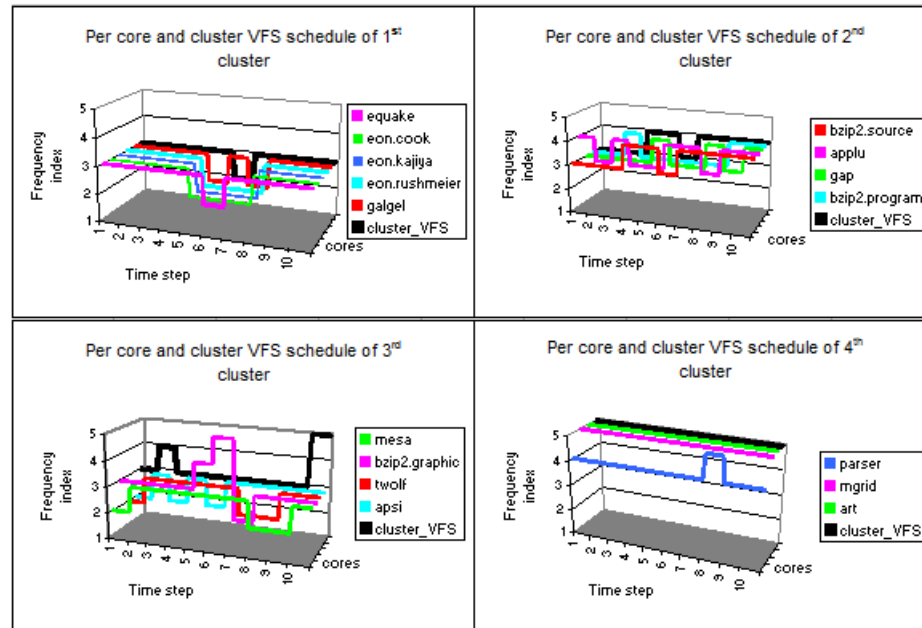


Figure 4.3: The 4-cluster solution under Case 1 and 0.9 s target execution time

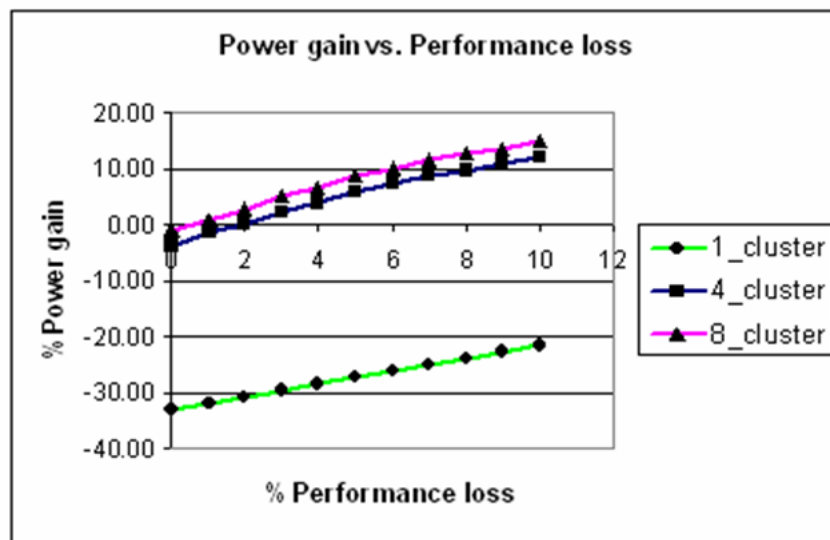


Figure 4.4: Comparison of the power saved from the different cluster configurations under Case 1 and 0.9 s target execution time

to the performance loss required to match the power of the per-core case. For the 8-cluster case, we can see that the overhead is slight, and a performance loss of under 1% is required. For the 4-cluster and 1-cluster cases, the overhead becomes progressively larger. The crossing point for the 4-cluster case corresponds to a performance loss of about 2%, while that for the per-chip (1-cluster) case is significantly larger than 10%.

#### 4.1.2 Target execution time of 1.1 s

We now set a target execution time of 1.1 s to all the 16 benchmarks and we divide 1.1 s into 10 VFS intervals of 0.11 s each. We then implement the first step of finding the per-core VFS schedule. Table 4.5 shows the VFS schedule obtained for each of the cores/benchmarks and the respective power consumed.

Table 4.5: Per-core VFS for 1.1 s target execution time

Core/Benchmark	VFS schedule	Power consumed (in W)
gap	2 3 1 2 3 2 2 2 2 2	1.0550
apsi	2 2 2 2 2 2 2 2 2 2	1.0550
equake	2 2 2 2 2 2 2 2 2 2	1.0936
parser	2 2 2 2 2 2 3 2 3 3	1.0232
eon.cook	2 2 2 2 2 2 2 2 2 2	1.0253
eon.kajiya	2 2 2 2 2 2 2 2 2 2	1.0246
eon.rushmeier	2 2 2 2 2 2 2 2 2 2	1.0100
bzip2.program	2 2 2 2 3 2 2 2 2 2	1.0777
bzip2.graphic	2 2 2 2 2 3 2 2 2 2	1.0943
bzip2.source	2 2 2 2 2 2 3 2 2 3	1.1035
mgrid	3 3 2 3 3 3 3 2 3 3	1.4754
applu	2 3 2 2 2 2 2 2 2 3	1.2155
mesa	2 2 2 2 2 2 2 2 2 2	1.0745
galgel	2 2 2 2 2 2 2 2 2 2	1.2177
art	5 5 5 5 5 5 5 5 5 5	1.6377
twolf	2 2 2 2 2 2 2 2 2 2	0.9595
	Total power	18.1425

We then apply our algorithm to group the cores into 8 clusters and 4 clusters, respectively. The list of cores in each cluster for these two cases are shown in Tables 4.6 and 4.7, respectively. The assignment of cores to the clusters is slightly different than the assignment for 0.9 s execution time. The more relaxed target time of 1.1 s leads

to different VFS schedules than the 0.9 s case and hence results in different clusters. However, the cores running `eon.cook` and `eon.kajiya` are still grouped into the same cluster in the 8-cluster case. In the 4-cluster case, cores running `eon.cook`, `eon.kajiya` and `eon.rushmeier` are grouped together and is similar to observation in the 0.9 s case. The benchmarks that exhibit similarity in clustering under different execution times indicate that their behaviour varies similarly with change in target execution time.

Table 4.6: Assignment of cores to clusters for the 8-cluster configuration under Case 1 and 1.1 s target execution time

Cluster 1	<code>equake</code> , <code>eon.cook</code> , <code>eon.kajiya</code>
Cluster 2	<code>eon.rushmeier</code> , <code>mesa</code> , <code>galgel</code>
Cluster 3	<code>twolf</code> , <code>apsi</code> , <code>bzip2.graphic</code>
Cluster 4	<code>gap</code> , <code>bzip2.program</code>
Cluster 5	<code>mgrid</code>
Cluster 6	<code>applu</code>
Cluster 7	<code>parser</code> , <code>bzip2.source</code>
Cluster 8	<code>art</code>

Table 4.7: Assignment of cores to clusters for the 4-cluster configuration under Case 1 and 1.1 s target execution time

Cluster 1	<code>equake</code> , <code>eon.cook</code> , <code>eon.kajiya</code> , <code>eon.rushmeier</code> , <code>mesa</code>
Cluster 2	<code>galgel</code> , <code>twolf</code> , <code>apsi</code> , <code>bzip2.program</code> , <code>bzip2.graphic</code>
Cluster 3	<code>bzip2.source</code> , <code>applu</code> , <code>parser</code>
Cluster 4	<code>mgrid</code> , <code>gap</code> , <code>art</code>

Next, our approach is used to perform clustered VFS for the 8-cluster, 4-cluster, and 1-cluster (per-chip) cases. As in the previous section, we plot the power saved from doing voltage and frequency scaling in each of these cases as a function of performance loss, in order to assess the performance sacrifice required to match the power dissipation of the per-core case, which is a lower bound on the achievable power. The results are qualitatively similar to the case of the 0.9 s target time. The 8-cluster case matches the power at almost no loss in performance; the 4-cluster case requires a drop-off of about 6%, while the 1-cluster case requires a performance loss of well over 10%.

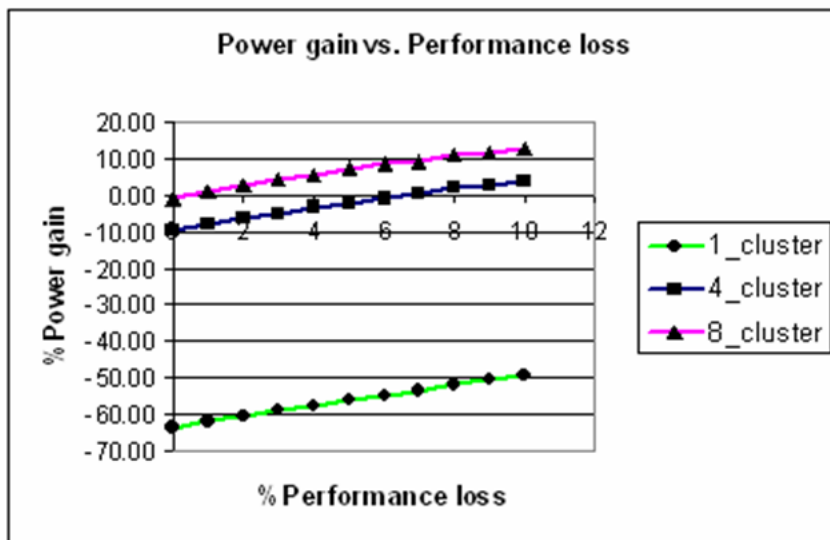


Figure 4.5: Comparison of the power saved from the different cluster configurations under Case 1 and 1.1 s target execution time

#### 4.1.3 Target execution time of 1.3 s

Finally, we repeat the same set of experiments for a target execution time of 1.3 s. The per-core VFS schedules are summarized in the Table 4.8. We can see that the VFS schedules in Table 4.8 comprise of lower voltage/frequency levels than those in Table 4.2. The longer target execution time lets the benchmarks complete execution with lower frequencies and is the reason behind the observation. The core assignments to clusters are summarized in Tables 4.9 and 4.10. We can observe differences in the clusters obtained with target time of 1.3 s when compared to those obtained at 0.9 s and 1.1 s. This is because of the varying nature of voltage and frequency requirements for different target times. As before, cores running `eon.cook` and `eon.kajiya` are still clustered together in both 8-cluster and 4-cluster cases. However, core that runs `eon.rushmeier` is not very similar in its voltage and frequency requirement to those that run `eon.cook` and `eon.kajiya` and hence does not belong to the same cluster as them. This is different from the behaviour observed at 0.9 s and 1.1 s.

We then perform clustered VFS for the 8-cluster, 4-cluster and 1-cluster cases. The



Table 4.8: Per-core VFS for 1.3 s target execution time

Core/Benchmark	VFS schedule	Power consumed (in W)
gap	2 2 1 1 2 1 1 2 1 2	0.7958
apsi	1 2 1 1 1 1 2 1 2 2	0.8172
equake	2 1 1 1 1 1 1 2 2 2	0.8467
parser	2 1 1 2 1 1 2 1 2 2	0.7409
eon.cook	1 2 2 1 1 1 1 2 1 2	0.8019
eon.kajiya	2 2 2 1 1 1 1 1 1 2	0.7996
eon.rushmeier	2 2 1 1 1 1 1 2 1 2	0.7936
bzip2.program	1 1 2 2 2 1 2 1 1 2	0.8232
bzip2.graphic	1 1 2 2 2 2 1 1 1 2	0.8343
bzip2.source	1 1 1 2 2 1 2 2 1 2	0.8175
mgrid	2 2 1 2 1 2 1 2 2 2	0.9805
applu	2 1 2 1 2 2 1 1 1 2	0.9009
mesa	2 2 2 1 1 1 1 1 1 2	0.8380
galgel	2 2 1 2 1 1 1 1 1 2	0.9498
art	4 4 4 4 4 4 4 5 5 5	1.3907
twolf	1 1 1 2 2 2 1 1 1 2	0.7505
	Total power	13.8810

Table 4.9: Assignment of cores to clusters for the 8-cluster configuration under Case 1 and 1.3 s target execution time

Cluster 1	eon.rushmeier, galgel, gap
Cluster 2	eon.kajiya, mesa, eon.cook
Cluster 3	applu
Cluster 4	parser, apsi
Cluster 5	art
Cluster 6	bzip2.source, bzip2.program
Cluster 7	equake, mgrid
Cluster 8	bzip2.graphic, twolf

Table 4.10: Assignment of cores to clusters for the 4-cluster configuration under Case 1 and 1.3 s target execution time

Cluster 1	eon.rushmeier, gap, equake, galgel, eon.cook
Cluster 2	bzip2.graphic, twolf, bzip2.program, bzip2.source, applu
Cluster 3	eon.kajiya, mesa, apsi
Cluster 4	parser, mgrid, art

comparison of the power saved with different cluster configurations is shown in Figure 4.6. As seen in the previous sections, the 8-cluster case matches the power of per-core case with almost no loss in performance. The 4-cluster case requires about 3% performance loss. The 1-cluster case shows a slightly different behaviour than the previous sections. The power approaches to that of per-core VFS case for about 10% performance loss whereas in the previous sections it required performance loss of well over 10%. The difference in behaviour is because with 1.3 s, the benchmarks art and mgrid complete well before the target time and a relaxation in target time helps assign lower voltages and frequencies. With 0.9 s and 1.1 s targets, the benchmarks could complete or be close to completion only by being at the highest frequency throughout and a performance loss did not help much.

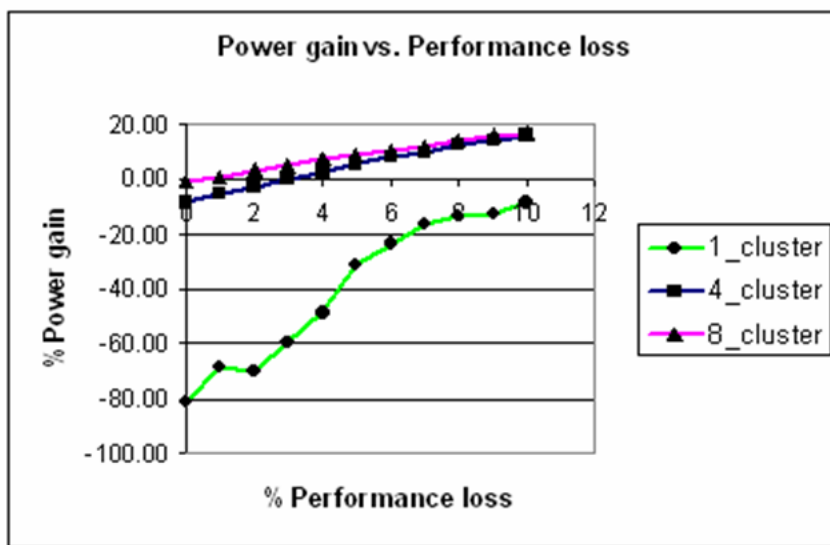


Figure 4.6: Comparison of the power saved from the different cluster configurations under Case 1 and 1.3 s target execution time

## 4.2 Case 2: Applications are penalized for noncompletion

In the second use model, we allow some applications to remain incomplete, but require a minimum threshold of instructions that must be completed within the target time. In our experiments, we require that all applications complete at least 85% of their

instructions. However, any benchmark with incomplete instructions incurs a penalty, based on the cost function given by:

$$Cost = Power(1 + K(\textit{fraction incomplete})) \quad (4.2)$$

The key idea of this cost function is to ensure that a single benchmark does not unduly affect an entire cluster, and does not degrade the performance of all other benchmarks in the cluster. At the same time, the 85% criterion sets a required bound on the performance of each individual benchmark. As can be seen from the evaluation results, this use model captures the key idea of doing DVFS, where, if an application does not benefit greatly from being at higher frequency it has to choose lower frequencies of operation.

The evaluation results for this second use model are presented in this section. We consider target execution times of 0.9 s and 1.1 s. As before, the target time will be divided into 10 equal VFS intervals. The value of  $K$  is user-defined, and we can see from Equation (4.2) that higher the value of  $K$ , greater is the penalty for incompleteness. We show results for two different values of  $K$ . It was observed that  $K = 2$  selects the lower power schedules with about 85-90% completion for almost all benchmarks and  $K = 5$  selects the lower power schedules with about 85-90% completion for those benchmarks that have a poor rate of instruction completion and higher power schedules with about 98-100% completion for those benchmarks that have a high rate of instruction completion. The value of  $K = 5$  applies a higher penalty for incomplete instructions than a value of 2. Therefore, the application chooses to stay incomplete only if the power required to complete the incomplete part is more than the penalty incurred. Thus,  $K = 5$  selects higher voltage and frequency schedules than  $K = 2$  for most benchmarks.

#### 4.2.1 Target execution time of 0.9 s and $K = 2$

Table 4.11 shows the per-core VFS schedule obtained for each of the cores/benchmarks and the respective cost incurred.

Our algorithm is applied to this case, and the resulting assignments of cores to clusters for the 8-cluster and 4-cluster case are shown in Tables 4.12 and 4.12, respectively.

The VFS schedules for various cases are shown in the following figures. Figure 4.7 shows the per-core VFS schedule for each core, which also includes a black curve that displays the cluster VFS schedule for the per-chip VFS case. It can be seen that the cluster VFS schedule for the per-chip case is no longer at the highest voltage and frequency throughout as was seen in Section 4.1.1. The relaxed constraint which allows up to about 15% incompleteness chooses lower voltage and frequency schedules for the benchmarks mgrid and art and hence, they no longer constrain the VFS schedule assignment for the cluster. Figures 4.8 and 4.9 shows the VFS assignments for the 8-cluster and 4-cluster cases, respectively. As before the individual VFS assignments are shown in various colors, and the cluster VFS assignment is represented by the black curve.

Table 4.11: Per-core VFS for 0.9 s target execution time and  $K = 2$

Core/Benchmark	VFS schedule	Cost (in W)
gap	2 2 2 2 2 3 2 2 2 2	1.3803
apsi	2 2 2 2 3 2 2 2 2 2	1.4564
equake	3 2 2 2 2 3 2 2 2 2	1.5516
parser	3 3 2 2 2 2 2 3 3 3	1.4466
eon.cook	2 2 2 2 2 2 2 2 2 3	1.4119
eon.kajiya	3 2 2 2 2 2 2 2 2 2	1.4079
eon.rushmeier	2 2 3 2 2 2 2 2 2 2	1.3890
bzip2.program	2 2 2 2 2 2 2 1 3 3	1.4448
bzip2.graphic	2 2 3 2 2 2 3 2 2 2	1.4831
bzip2.source	2 2 2 3 3 2 1 3 2 2	1.4803
mgrid	3 3 3 3 3 3 3 3 2 3	1.9787
applu	2 3 3 3 2 2 2 2 2 3	1.7185
mesa	3 2 2 2 2 2 2 2 2 2	1.4794
galgel	2 2 2 2 2 2 3 2 2 2	1.6824
art	1 1 1 1 1 1 1 1 2 2	0.7366
twolf	2 2 2 2 2 2 3 2 2 2	1.3159
	Total power	23.3633

The comparison of power saved with different cluster configurations is shown in Figure 4.10. It can be seen from the figure that in this case, per-chip VFS gives as good a power saving as per-core VFS with about 5% performance trade-off. This is significantly different from the observation in Section 4.1.1 where the power saved from chip-wide VFS was far from the optimal case, even with about 10% relaxation of target

Table 4.12: Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 0.9 s target execution time and  $K = 2$

Cluster 1	eon.kajiya, mesa, equake
Cluster 2	eon.cook, bzip2.program, applu
Cluster 3	gap, eon.rushmeier, apsi
Cluster 4	galgel, twolf, bzip2.graphic
Cluster 5	art
Cluster 6	mgrid
Cluster 7	bzip2.source
Cluster 8	parser

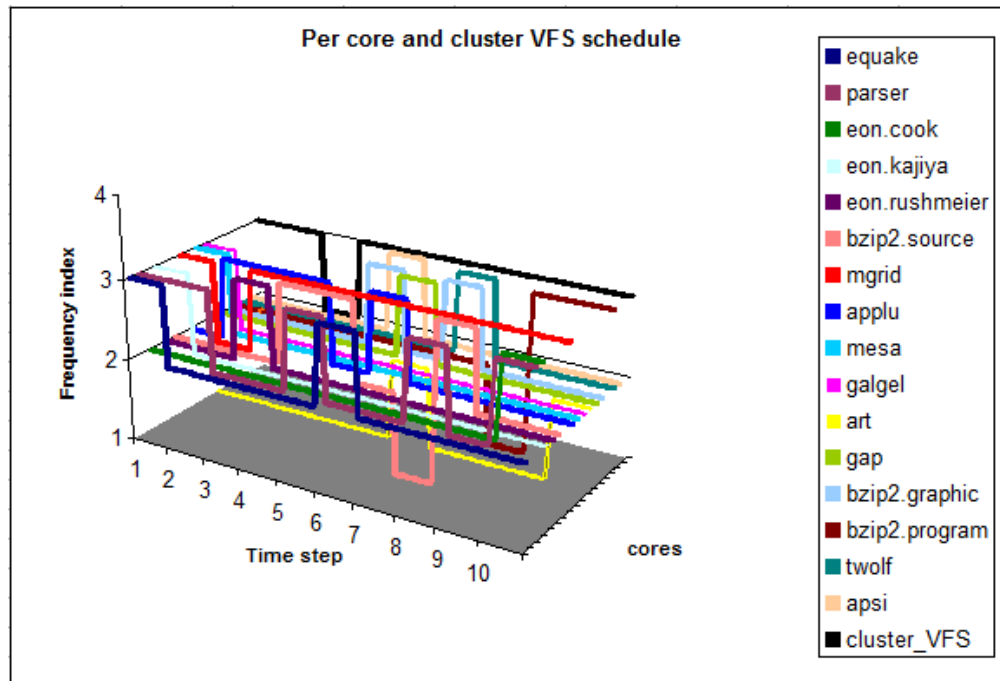


Figure 4.7: Per-core VFS under Case 2 and 0.9 s target execution time and  $K = 2$

Table 4.13: Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 0.9 s target execution time and  $K = 2$

Cluster 1	galgel, twolf, bzip2.graphic, eon.rushmeier, gap
Cluster 2	eon.kajiya, mesa, apsi, equake, bzip2.source
Cluster 3	parser, mgrid, applu
Cluster 4	eon.cook, art, bzip2.program

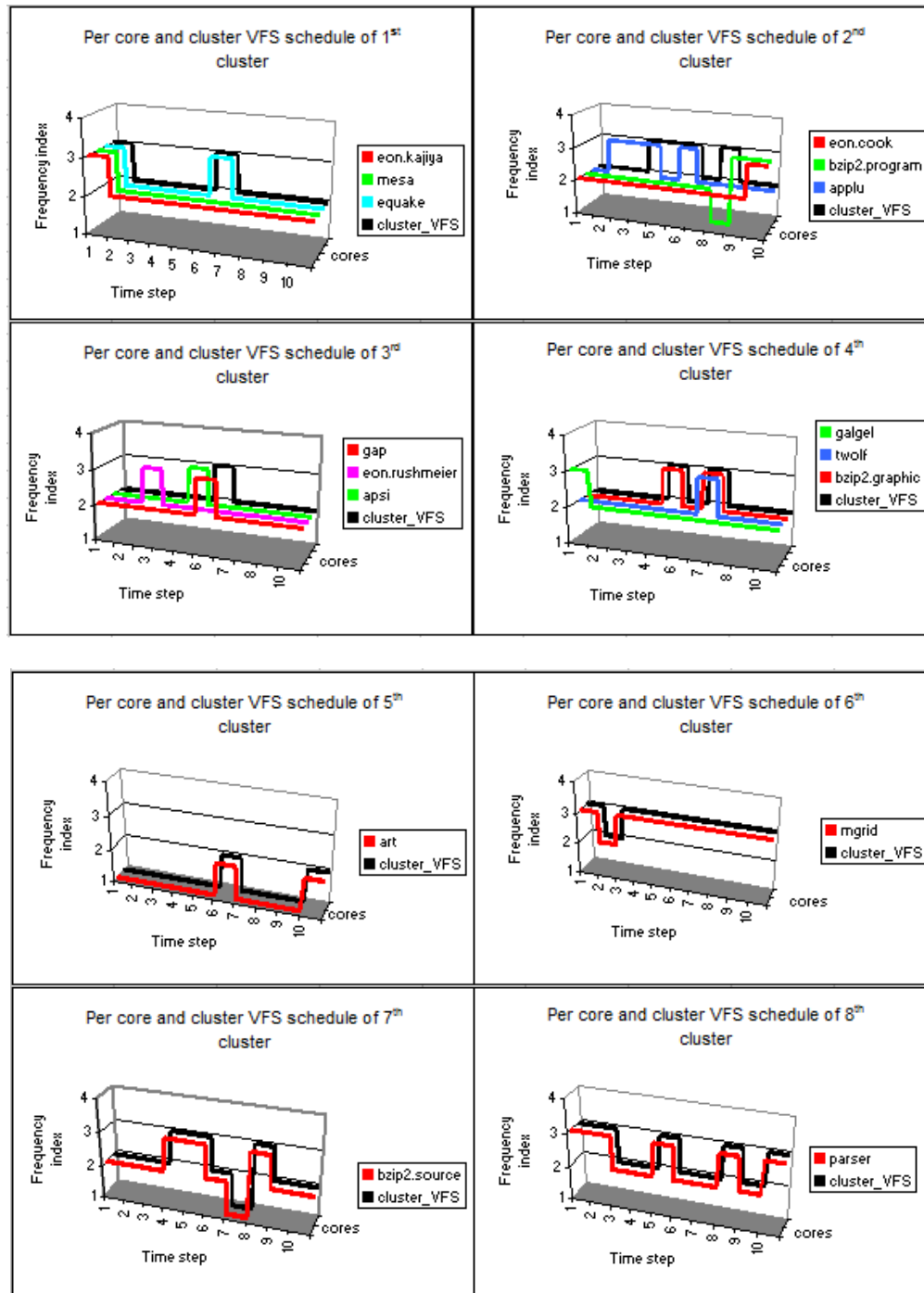


Figure 4.8: The 8-cluster solution under Case 2 and 0.9 s target execution time and  $K = 2$

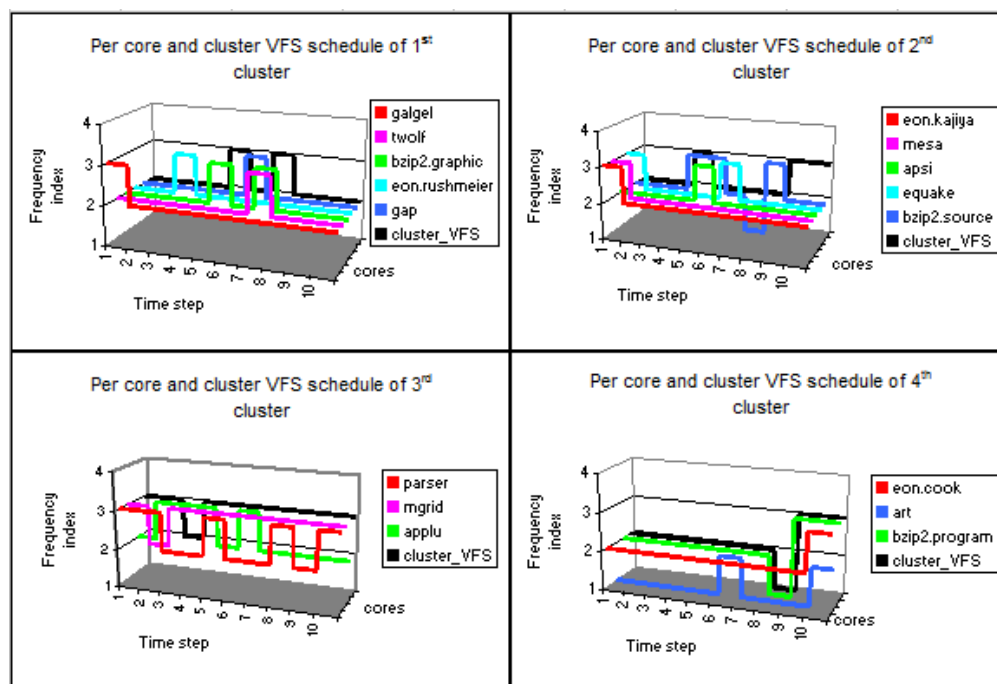


Figure 4.9: The 4-cluster solution under Case 2 and 0.9 s target execution time and  $K = 2$

time. The improvement is due to the relaxed constraint that the applications can be up to 15% incomplete.

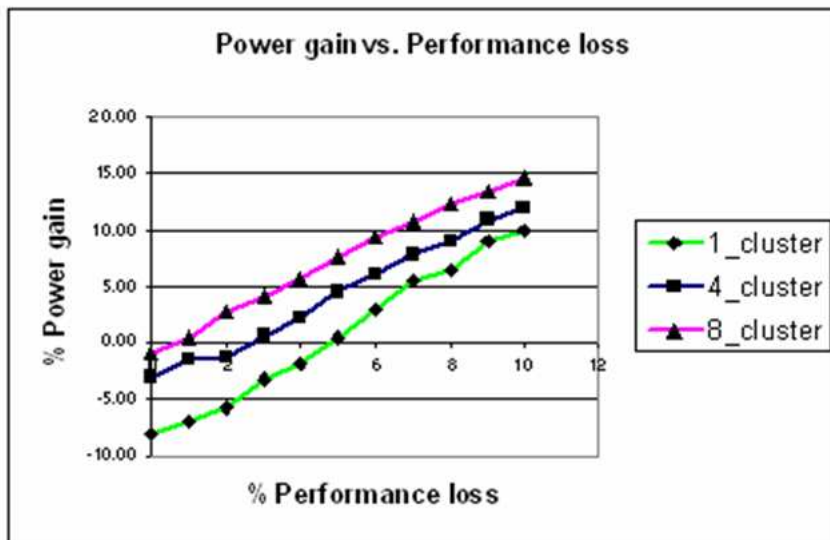


Figure 4.10: Comparison of the power saved from the different cluster configurations under Case 2 and 0.9 s target execution time and  $K = 2$

#### 4.2.2 Target execution time of 0.9 s and $K = 5$

Table 4.14 shows the per-core VFS schedule obtained for each of the cores/benchmarks and the respective cost incurred. It can be seen that the schedules in this table have greater voltage and frequency assignments than those in Table 4.11 and is because  $K = 5$  selects higher voltage and frequency values as explained before. The result of clustering these cores into 8-cluster and 4-cluster is shown in Tables 4.15 and 4.16, respectively.

As in the previous sections, we proceed to compare the power saved with different cluster configurations and is shown in Figure 4.11. The 8-cluster case requires less than 0.5% performance loss to match the power dissipation of per-core case whereas, the 4-cluster case requires slightly over 1% and 1-cluster case requires slightly over 2% performance loss.



Table 4.14: Per-core VFS for 0.9 s target execution time and  $K = 5$ 

Core/Benchmark	VFS schedule	Cost (in W)
gap	2 2 2 2 2 3 2 2 2 2	1.6200
apsi	2 3 2 3 2 3 3 3 3 3	1.4933
equake	3 3 3 3 3 2 2 3 3 3	1.6115
parser	3 3 2 2 3 2 3 3 3 3	1.7765
eon.cook	3 3 3 3 2 2 2 3 3 3	1.4478
eon.kajiya	3 3 3 3 2 2 2 3 3 3	1.4324
eon.rushmeier	3 3 3 3 2 2 2 3 3 3	1.4268
bzip2.program	2 2 2 2 2 2 2 1 3 3	1.6916
bzip2.graphic	2 2 2 2 3 3 3 2 2 3	1.6803
bzip2.source	2 2 2 3 3 2 1 3 2 2	1.7847
mgrid	5 5 5 5 5 5 5 4 5	2.6278
applu	4 3 4 3 3 4 4 3 4 4	1.9690
mesa	2 3 3 3 3 3 3 2 2 3	1.5128
galgel	3 2 3 2 3 2 3 3 3 3	1.7457
art	1 1 1 1 1 1 1 2 1 2	0.9890
twolf	2 3 3 3 3 3 2 2 3 3	1.3572
	Total power	26.1666

Table 4.15: Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 0.9 s target execution time and  $K = 5$ 

Cluster 1	mesa, twolf, apsi
Cluster 2	eon.cook, eon.kajiya, eon.rushmeier
Cluster 3	parser, galgel, equake
Cluster 4	gap, bzip2.graphic, bzip2.program
Cluster 5	art
Cluster 6	bzip2.source
Cluster 7	applu
Cluster 8	mgrid

Table 4.16: Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 0.9 s target execution time and  $K = 5$ 

Cluster 1	equake, eon.cook, eon.kajiya, eon.rushmeier, parser
Cluster 2	gap, bzip2.program, bzip2.graphic, bzip2.source, art
Cluster 3	mesa, twolf, apsi
Cluster 4	mgrid, applu, galgel

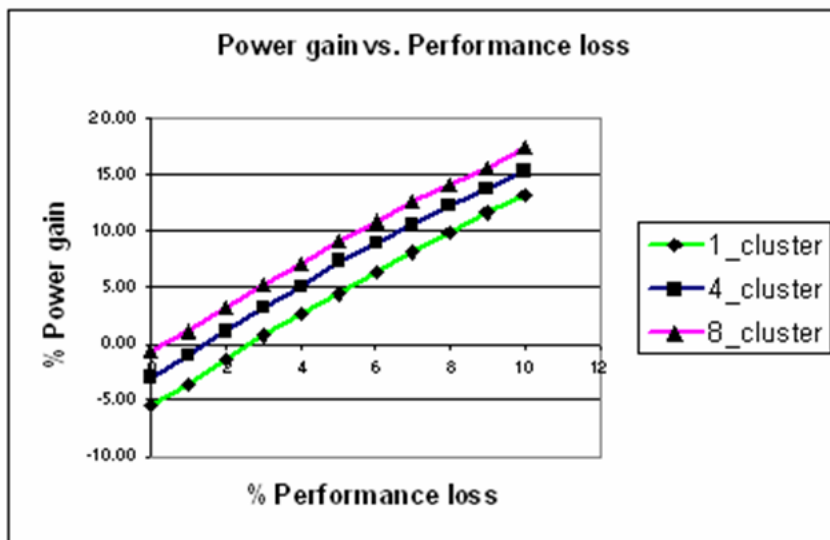


Figure 4.11: Comparison of the power saved from the different cluster configurations under Case 2 and 0.9 s target execution time and  $K = 5$

#### 4.2.3 Target execution time of 1.1 s and $K = 2$

We set a target execution time of 1.1 s and a value of 2 for the penalty co-efficient  $K$ . The per-core VFS schedules and the respective cost incurred are shown in Table 4.17. Using these schedules, the clusters obtained for the 8-cluster and 4-cluster cases are shown in Tables 4.18 and 4.19, respectively. The core assignments to clusters vary with change in target execution time and also a change in the value of  $K$ .

The comparison of power for the VFS of different cluster configurations is shown in Figure 4.12. We can see that because of a more relaxed execution time than 0.9 s and because of the relaxed constraint that allows incompleteness, the 1-cluster case also selects low voltage and frequency values and the difference in power dissipation between the 4-cluster and 1-cluster case is very small.

#### 4.2.4 Target execution time of 1.1 s and $K = 5$

The per-core VFS schedules and the respective cost incurred for each of the cores/benchmarks is shown in Table 4.20. The assignment of cores to clusters for the 8-cluster and 4-cluster cases are shown in Tables 4.21 and 4.22, respectively.

Table 4.17: Per-core VFS for 1.1 s target execution time and  $K = 2$ 

Core/Benchmark	VFS schedule	Cost (in W)
gap	1 2 2 1 2 2 1 2 2 1	1.0265
apsi	1 2 1 2 2 1 2 1 1 2	1.0479
equake	2 1 1 2 2 2 1 2 1 1	1.0878
parser	2 2 1 1 2 1 1 2 1 2	0.9696
eon.cook	1 2 2 1 1 1 1 1 2 2	1.0169
eon.kajiya	2 2 2 1 1 1 1 1 1 2	1.0160
eon.rushmeier	2 2 1 1 1 1 1 1 2 2	1.0022
bzip2.program	1 1 2 2 1 2 1 1 1 2	1.0353
bzip2.graphic	1 1 2 2 1 2 2 1 1 1	1.0449
bzip2.source	1 1 1 1 2 2 1 1 2 2	1.0327
mgrid	2 1 1 2 2 2 2 2 2 1	1.2795
applu	2 1 2 2 1 2 2 2 1 1	1.1884
mesa	1 2 2 2 1 1 2 1 1 1	1.0696
galgel	2 1 2 1 2 1 2 1 2 1	1.2155
art	1 1 1 1 1 1 1 1 2 2	0.7354
twolf	1 1 1 2 2 2 2 1 1 1	0.9514
	Total power	16.7195

Table 4.18: Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 1.1 s target execution time and  $K = 2$ 

Cluster 1	mesa, bzip2.graphic, bzip2.program
Cluster 2	equake, mgrid, applu
Cluster 3	eon.cook, eon.kajiya, eon.rushmeier
Cluster 4	galgel
Cluster 5	gap
Cluster 6	twolf, apsi
Cluster 7	parser
Cluster 8	bzip2.source, art

Table 4.19: Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 1.1 s target execution time and  $K = 2$ 

Cluster 1	bzip2.graphic, mesa, twolf, bzip2.program, apsi
Cluster 2	eon.rushmeier, eon.cook, eon.kajiya, art, parser
Cluster 3	equake, mgrid, applu
Cluster 4	bzip2.source, galgel, gap

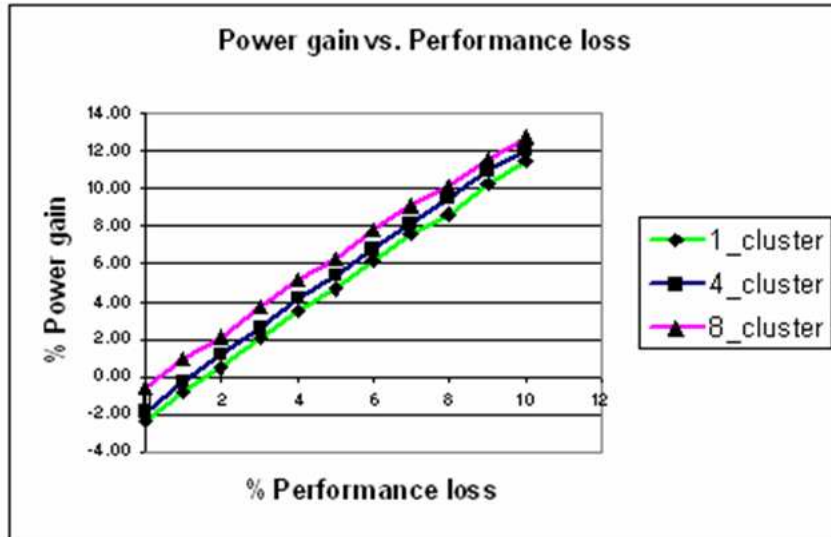


Figure 4.12: Comparison of the power saved from the different cluster configurations under Case 2 and 1.1 s target execution time and  $K = 2$

Table 4.20: Per-core VFS for 1.1 s target execution time and  $K = 5$

Core/Benchmark	VFS schedule	Cost (in W)
gap	1 2 2 1 1 2 2 2 2 1	1.0488
apsi	1 2 2 2 2 2 2 2 2 2	1.0550
quake	2 2 2 2 2 1 2 2 2 2	1.0936
parser	2 2 1 1 2 1 1 2 1 2	1.0232
eon.cook	2 2 2 2 2 1 2 2 2 2	1.0249
eon.kajiya	2 2 2 2 2 2 1 2 2 2	1.0243
eon.rushmeier	2 2 2 1 2 2 2 2 2 2	1.0095
bzip2.program	1 1 1 1 1 2 2 1 2 2	1.0777
bzip2.graphic	1 1 2 2 2 2 2 1 1 2	1.0943
bzip2.source	1 1 1 1 2 2 1 1 2 2	1.1035
mgrid	2 2 2 2 2 2 1 2 1 1	1.4573
applu	2 2 2 2 2 2 2 2 2 2	1.2155
mesa	2 2 2 2 2 2 2 2 1 2	1.0745
galgel	2 2 2 2 2 2 2 2 2 2	1.2177
art	1 1 1 1 1 1 1 1 2 2	0.9875
twolf	1 2 2 2 2 2 2 2 2 2	0.9590
	Total power	16.4113

Table 4.21: Assignment of cores to clusters for the 8-cluster configuration under Case 2 and 1.1 s target execution time and  $K = 5$

Cluster 1	applu, galgel, eon.rushmeier
Cluster 2	mgrid, mesa, eon.kajiya
Cluster 3	bzip2.source, art, bzip2.program
Cluster 4	twolf, apsi
Cluster 5	bzip2.graphic
Cluster 6	parser
Cluster 7	gap
Cluster 8	equake, eon.cook

Table 4.22: Assignment of cores to clusters for the 4-cluster configuration under Case 2 and 1.1 s target execution time and  $K = 5$

Cluster 1	applu, galgel, equake, eon.cook, eon.rushmeier
Cluster 2	parser, eon.kajiya, mgrid, mesa
Cluster 3	gap, bzip2.graphic, twolf, apsi
Cluster 4	bzip2.source, art, bzip2.program

The comparison of power for the different cluster configurations is shown in Figure 4.13. The difference between the power dissipation of the 4-cluster case and 1-cluster case has become even smaller here when compared to the previous section. The general trend of 8-cluster case being better than 4-cluster case which in turn is better than 1-cluster case, with respect to power dissipation, can be seen in the figure.

### 4.3 Practical nature of tuple growth

The algorithm to find per-core and per-cluster VFS schedules proceeds by finding tuples at each VFS time step as described in Sections 2.1 and 2.3. The worst-case time complexity of the algorithm was found to be  $O(MN^M)$ , where  $M$  is the number of time steps and  $N$  is the number of discrete voltage-frequency pairs. Further, it was also explained that, in practice, the complexity is much less than the worst-case value. The reason for reduced complexity was the elimination of suboptimal tuples at each time step. We present, in this section, an example of tuple growth for one of the benchmarks for one of the target times evaluated. We consider the benchmark galgel and 0.9 s target execution time. Similar behaviour was observed for all other benchmarks and target

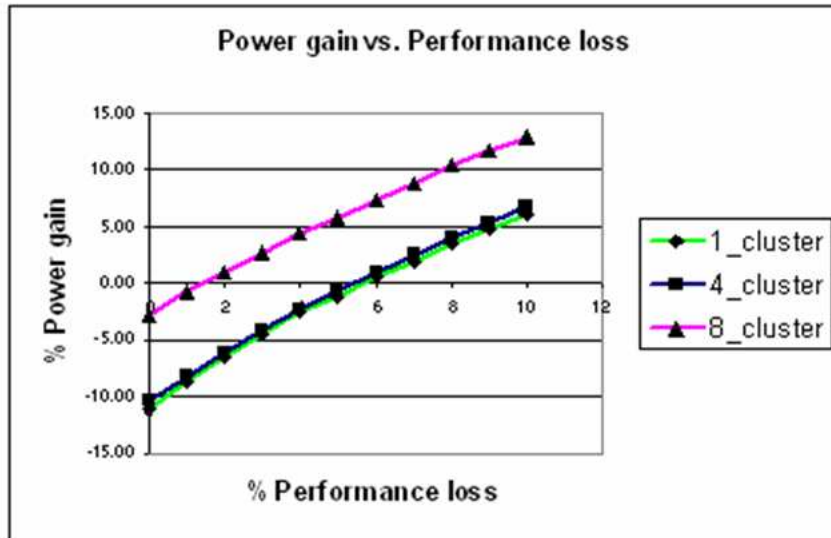


Figure 4.13: Comparison of the power saved from the different cluster configurations under Case 2 and 1.1 s target execution time and  $K = 5$

execution times. The growth of tuples plotted as a function of time for different values of  $M$  is shown in Figure 4.14. The number of frequencies is kept constant at 3. It can be seen that the growth without pruning suboptimal tuples is exponential whereas, with pruning, the growth is way less than exponential. The tuple growth plotted as a function of time for different values of  $N$  is shown in Figure 4.15. The number of time steps is kept constant at 10. Again, it can be seen that without pruning the growth is exponential. With pruning, the growth is way less than exponential.

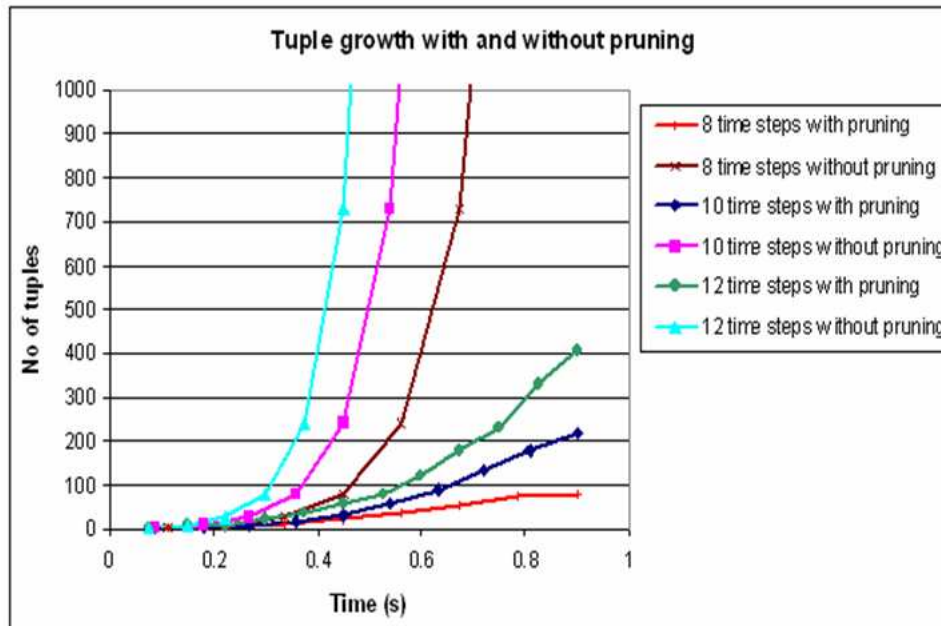


Figure 4.14: Tuple growth as a function of time for different number of time steps

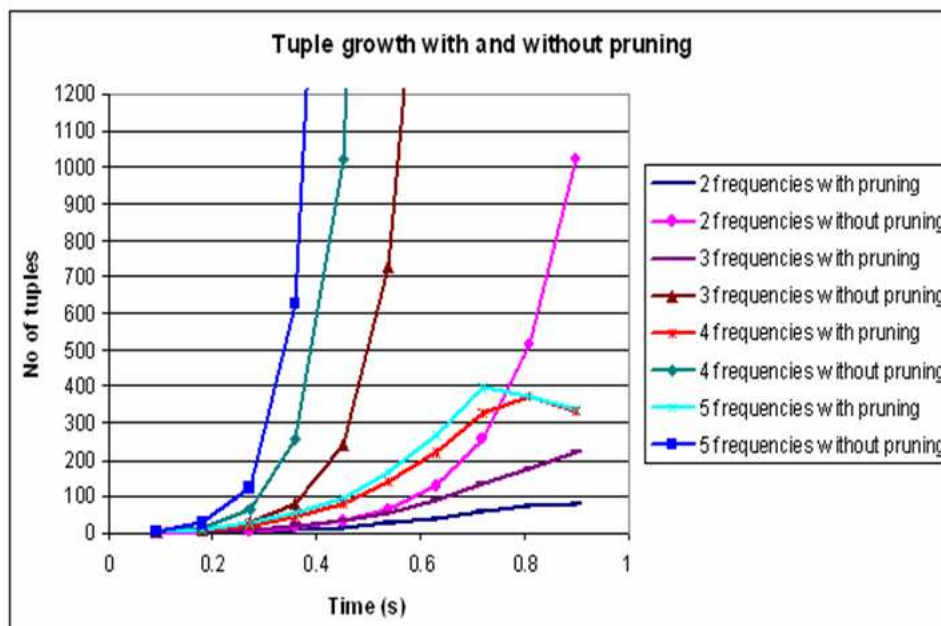


Figure 4.15: Tuple growth as a function of time for different number of frequencies

## Chapter 5

# Conclusion

Power dissipation in processors is a growing concern and there are techniques such as Dynamic Voltage and Frequency Scaling (DVFS) that help reduce the active power component. While DVFS can be easily implemented for a single processor, there is a significant overhead because of multiple on-chip voltage regulators required in implementing it for every processing core of chip multiprocessor. On the other hand, per-chip DVFS may not be effective in utilizing the varying profiles of applications running on the cores. In this thesis we have evaluated the prospects of saving power with per-cluster DVFS. We have demonstrated one way of clustering the cores using K-means clustering, a simple clustering algorithm. We have also proposed a bounded enumeration scheme for finding the optimal voltage and frequency schedule for each core and for the clusters. Under the assumption that the benchmarks chosen while implementing clustering represent all the applications that are going to be run on the cores, we demonstrate that per-cluster DVFS saves as good a power as per-core DVFS with a small performance trade-off. However, it is necessary to devise a method to map the applications to the appropriate cores and this gives direction for future work.



# References

- [1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, fourth edition, 2007.
- [2] Y. Li, K. Skadron, D. Brooks, and Z. Hu. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 71–82, 2005.
- [3] L. Hammond, B. A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer Special Issue on "Billion-Transistor Processors"*, 30(9):79–85, 1997.
- [4] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Prentice Hall, Upper Saddle River, NJ, second edition, 2003.
- [5] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, pages 123–134, 2008.
- [6] R. Bergamaschi, G. Han, A. Buyuktosunoglu, H. Patel, I. Nair, G. Dittmann, G. Janssen, N. Dhanwada, Z. Hu, P. Bose, and J. Darringer. Exploring Power Management in Multi-Core Systems. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 708–713, 2008.
- [7] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, pages 38–43, 2007.

- [8] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 81, 2003.
- [9] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread Motion: Fine-Grained Power Management for Multi-Core Systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 302–313, 2009.
- [10] M. Martonosi and S. Kaxiras. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool, San Rafael, CA, first edition, 2008.
- [11] D. Shelepov and A. Fedorova. Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. In *Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, 2008.
- [12] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25:13–25, 1997.
- [13] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, 2000.
- [14] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 423–428, 2009.
- [15] AJ KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 1:7–7, 2002.
- [16] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Paillet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Sali-hundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 Message-Passing Processor with DVFS

in 45nm CMOS. In *IEEE International Solid-State Circuits Conference*, pages 108–109, 2010.