DYNAMIC VOLTAGE SCALING TECHNIQUES FOR POWER-EFFICIENT MPEG DECODING

WISSAM CHEDID

Bachelor of Science in Electrical Engineering

Lebanese University, Lebanon

June, 2001

Submitted in partial fulfillment of requirements for the degree

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

December, 2003

This thesis has been approved

for the department of Electrical and Computer Engineering

and the College of Graduate Studies by

Thesis Committee Chairperson, Dr. CHANSU YU

Department/Date

Dr. DAN SIMON

Department/Date

Dr. YONGJIAN FU

Department/Date

ACKNOWLEDGEMENTS

I would like to thank all my professors and faculty of the Electrical and Computer Engineering department.

In particular, I want to gratefully acknowledge the help of my advisor Dr. CHANSU YU and his support throughout this challenging thesis.

ABSTRACT

The quest for enhancing microprocessor speed and integration has long been the goal of computer architects, which helped providing tremendous performance improvements over the years but at the same time created new problems. One of the important problems is the power consumption of hardware components, and the resulting thermal and reliability concerns that it raises, making power as important a criterion for optimization as performance.

Among various system components for consideration, we are primarily interested in this thesis in power consumption of a microprocessor because, in many cases, it is the most power-consuming component in a computer system. A number of research efforts have been focused to reduce energy consumption through the use of *dynamic voltage scaling* (*DVS*), which allows a processor to dynamically change its speed and voltage at run time, increasing energy efficiency without impacting the performance. Our motivation is to exploit the DVS methodology on video processing application dealing with MPEG stream, which is the most popular video format used in many current and emerging products (HDTV, DVD, video conferencing, etc.)

This thesis provides in-depth survey on different power management techniques for energy efficient computer systems and proposes three application-based DVS algorithms for energy efficient MPEG decoding, which further reduces energy consumption without sacrificing the perceptual quality of the video stream. The advantage of the proposed schemes is verified via extensive simulation based on state-of-

the-art *SimpleScalar* tool set with our own *MPEG power estimator* and *MPEG QoS estimator*, for power and QoS statistics respectively. According to the simulation result, our schemes show up to 83% improvement in energy as compared to the On/Off mechanism, with frames drop rates as low as 0.4%.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Background

Enhancing microprocessor performance has long been the goal of computer architects, driving technological innovations to the limits for getting the most out of every cycle as well as for reducing the cycle time. This quest for performance has made it possible to incorporate millions of transistors on a very small die, and to clock these transistors at very high speeds. While these innovations and trends have helped provide tremendous performance improvements over the years, they have at the same time created new problems. One of the important and daunting problems is the power consumption of hardware components, and the resulting thermal and reliability concerns that it raises, making power as important a criterion for optimization as performance. It is a challenge to system designers not only of low-end systems but also of high-end systems. Low-end portable systems, such as laptop computers and personal digital assistants (PDAs) draw power from batteries [4, 6-8]; so reducing power consumption

extends their operating times. For high-end desktop computers or servers, high power consumption raises temperature and deteriorates performance and reliability [16, 17].

Among various system components for consideration, we are primarily interested in this thesis in power consumption of a microprocessor because, in many cases, it is the most power consuming component in a computer system. The simplest way of reducing power consumption of a microprocessor is to lower the supply voltage, which exploits the quadratic dependence of power on voltage. Reducing the supply voltage however increases circuit delay and decreases clock speed and thus, it may not be effective because some systems have latency critical tasks. One possible compromise is to dynamically vary the voltage according to the processor workload, which is made possible due to the recent advances in power supply technology [33, 34]. Current custom and commercial CMOS chips are capable of operating reliably over a range of supply voltages [35, 36]. For example, Mobile Intel processor has 11~12 frequency levels and 6 different supply voltage levels [42]. Transmeta Crusoe has also variable voltage and frequency settings, allowing it to continuously scale both the frequency and voltage of the processor according to instantaneous performance demand on the system [43].

The abovementioned technology is called *Dynamic Voltage Scaling (DVS)*. However, in order to maximize the benefit out of the DVS mechanism, it is essential to have fine-grained workload monitoring mechanism as well as accurate workload prediction scheme. Workload monitoring/prediction can be accomplished at many different levels. In processor-based approaches, the microprocessor itself performs this [10, 11] but it often leads to incorrect prediction of future workloads simply because the microprocessor is ignorant of the detailed information on application which it is

executing. Alternatively, workload monitoring/prediction can be accomplished at a higher level such as operating system or an application to obtain more accurate prediction of future workload. In fact, several application-based DVS algorithms have been proposed for real-time systems, which minimize energy consumption while all tasks are guaranteed to complete on or before deadlines [13, 27-32, 38-41].

Thesis Outline

The motivation of this thesis is to exploit the DVS methodology on video processing application dealing with MPEG *(Moving Pictures Expert Group)* stream, which is the most popular video format and is described in detail in Chapter III. Since there is a growing interest in video applications on mobile devices, ranging from video games and movie players to sophisticated virtual reality environment, energy efficient MPEG decoding becomes extremely important. While MPEG decoding is a computationally intensive, power hungry process, there is a great degree of variance in processing requirements due to different frame types and variation between scenes. This high variability in video streams can be exploited to reduce power consumption of the processor based on the DVS technique. Processor-based DVS algorithm may fail since it is difficult to predict the next workload based on the previous workload and a wrong prediction causes frames to be dropped.

Recent studies present application-based approaches that predict the decoding times of incoming MPEG frames and reduce or increase the supply voltage based on this prediction [13, 38-41]. In an ideal case, the decoding times are estimated perfectly and all the frames are decoded at the exact time span allowed with the exact supply voltage level.

In practice, decoding time estimation includes errors that result in frames being decoded either before or after their expected playout time. When the decoding finishes early, the processor will be idle while it waits for the frame to be played, and some power will be wasted. When decoding finishes late, the frame will miss its playout time, and the perceptual quality of the video could be reduced.

This thesis provides in-depth survey on different power management techniques for energy efficient computer systems and proposes three application-based DVS algorithms for energy efficient MPEG decoding which reduces energy consumption without sacrificing the perceptual quality of the video stream. The advantage of the proposed schemes is verified via extensive simulation based on state-of-the-art *SimpleScalar* tool set [18] with our own *MPEG power estimator* and *MPEG QoS estimator*, for power and QoS statistics respectively. According to the simulation result, our schemes show up to 83% improvement in energy as compared to the On/Off mechanism (where the processor is just turned off while idle), with frames drop rates as low as 0.4%.

Thesis Organization

The rest of the thesis is organized as follows. Chapter II overviews power management techniques proposed so far in the literature and introduces our classification of those techniques. Chapter III presents background information on MPEG video format as well as MPEG decoding procedure. It is followed by the introduction of previous energy efficient MPEG decoding schemes based on DVS technique. Our decoding time estimation and the corresponding three DVS algorithms are presented in Chapter IV. The

first algorithm takes advantage of the linear regression model of the decoding-time/frame-size distribution to improve the prediction accuracy. The other two algorithms divide the decoding-time/frame-size distribution into *intervals* and make the prediction locally within each interval. On top of these voltage prediction algorithms, a voltage averaging technique is also proposed, aiming at further reducing the power consumption. Chapter V presents the experimental environments based on *SimpleScalar* as well as simulation results. Conclusion remarks are found in Chapter VI.

CHAPTER II

POWER MANAGEMENT TECHNIQUES FOR POWER EFFICIENT COMPUTER SYSTEMS

In this chapter we discuss some of the power management techniques proposed so far in the literature. They are classified as *Static Power Management (SPM)* and *Dynamic Power Management (DPM)* techniques. SPM techniques are applied at design time (off-line) and target both hardware and software implementations (Section 2.1). In contrast, DPM techniques use runtime (on-line) behavior to adjust power depending on system workload (Section 2.2). Note that the main theme of this thesis, DVS, is classified as a processor-based DPM technique. Another important thing to note is that while DPM techniques are used to optimize energy performance at runtime, SPM techniques are used to obtain energy performance information to help system designers to select the best system parameters. Table I summarizes the SPM and DPM techniques.

Table I: Classification of power management techniques.

SPM (off-line optimization)				
System/ Component Under Test (SUT/CUT)	Level of Detail	Evaluation Methodology	Description	Section
CPU	Cycle level or RTL	Cycle-level simulation	*PowerTimer* [1], *Wattch* [2] and *SimplePower* [3] energy models	2.1.1
	Instruction level	Instruction-level simulation	Power Profiles for *Intel 486DX2*, *Fujitsu SPARClite'934* [4] and *PowerPC* [5]	2.1.1
System	Hardware component level (e.g. hardware state: CPU sleep/ doze/busy, LCD on/off etc.)	Functional simulation (Parameters via measurements)	*POSE* (Palm OS Emulator) [6]	2.1.2
	Software component level (procedure/process/task)	Measurements (with monitoring tools)	Time driven sampling, *PowerScope*[7] and Energy driven sampling [8]	2.1.2
	Hardware & Software component level	Complete system simulation (CPU, Disc, Memory, OS, Application)	*SoftWatt* built upon *SimOS* system simulator [9]	2.1.2
DPM (on-line optimization)				
(SUT/CUT)	Implementation level	Methodology	Description	Section
CPU	CPU and System software	DVS (Dynamic Voltage Scaling)	Interval-based scheduler [10,11] and Real-time schedulers (Inter-task [12,13], Intra-task [19-23])	2.2.1
System	Components hardware (Disks, network interfaces, displays, I/O devices, etc.) and system software	Low power mode of operation	Shutdown/low- power unused devices [15,16]	2.2.2
Cluster system	Multiple systems coordination (server clusters…)	CVS (Coordinated Voltage Scaling)	Coordinated DVS between multiple nodes [17]	2.2.3

2.1 Static Power Management (SPM) Techniques

Power dissipation limits have emerged as a major constraint in the design of microprocessors, and just as with performance, power optimization requires careful design at several levels of the system architecture. Different energy models were presented in previous studies and integrated with already known simulators and

measurement tools to provide power estimation, measurement and optimization at design time [1-9]. Section 2.1.1 describes processor-based SPM techniques that estimate power consumption of a microprocessor at cycle or instruction level. Section 2.1.2 discusses system-based SPM techniques.

2.1.1 CPU-based SPM

Cycle level

Energy consumption of a processor can be estimated by using an architecture simulator. In particular, *cycle-level* or *register-transfer level* (RTL) simulators can provide accurate performance metrics by identifying the activated (or busy) microarchitecture-level units or blocks during every execution cycle of the simulated processor [1-3]. We can use these cycle-by-cycle resource usage statistics, available from a trace-driven or execution-driven architecture simulator, to estimate the power consumption. Energy models describing how each unit or block consumes energy are indispensable for any power estimation tool. Different energy models were presented in [1-3] and used in conjunction with RTL processor models creating power-aware *cycle-level* simulators.

Brooks and al. presented two types of energy models for their *PowerTimer* simulator [1]: (i) power-density-based models, used whenever detailed power and area measurements are available for a given chip, and (ii) analytical energy models, based on simple chip area factors and microarchitecture-level design parameters such as cache size, pipeline length, number of registers and so on. These energy models were used in conjunction with *Turandot*, a generic, parameterized, out-of-order superscalar processor

simulator, creating the power-aware *PowerTimer* simulator. Using *PowerTimer*, researchers in [1] studied the power-performance trade-offs of different techniques proposed in the literature and their ability to help building power-aware microarchitectures.

The next two CPU-based SPM techniques are based on S*impleScalar* [18], which is the most popular architecture simulator and will be discussed in detail in Chapter V. For *Wattch* [2], the energy model in use depends, particularly, on the internal capacitances for the circuits that make up each unit of the processor. Each modeled unit, and depending on its structure and functionality, fall into one of these four categories: array structures, memories, combinational logic and wires, and the clocking network. A different power model is used for each category and integrated in the *SimpleScalar* simulator. *Wattch* provides a variety of metrics such as power, performance, energy and energy-delay product, and it can be used to perform both architectural and compiler research.

Another *SimpleScalar*-based RT level energy estimation tool, *SimplePower*, is presented in [3]. It was developed based on transition-sensitive energy models, where each functional unit has its own energy model from a table containing the power consumed for each input transition. *SimplePower* provides cycle-by-cycle energy estimates and switch capacitance statistics for the processor datapath, memory and onchip buses. The major components of *SimplePower* are: *SimplePower* core, RTL power estimation interface, technology dependent switch capacitance tables, cache/bus estimator, and loader. *SimplePower* can be used to study different architectural optimizations.

Figure 1 illustrates a high-level block diagram of the three power-aware cycle-level simulators described earlier.

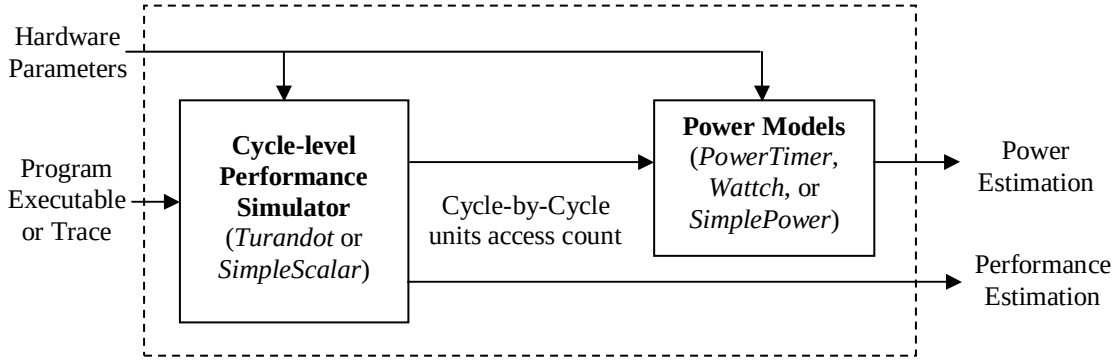


Figure 1: Block diagram of a power-aware, cycle-level simulator.

Instruction-level

As opposed to the finer grained *cycle-level* techniques, coarser grained *instruction-level* power analysis techniques were presented in [4, 5]. These techniques estimate the energy consumed by a program by adding the energy consumed by the execution of each instruction. Instruction-by-instruction energy costs are computed once for all for each target processor.

The basic steps in building energy models for any *instruction-level* simulator are the same. Only quantitative values change from one processor to another. The first step is to create the set of *base costs* of individual instructions, which is the fixed energy cost assigned to every instruction. Then, the power cost of *inter-instruction effects* should be accounted for, which is the extra power consumption due to "interaction" between successive instructions (it also includes other effects like pipeline stalls and cache misses). The experimental procedure used to determine the above costs requires a

program containing mainly a loop consisting of several instances of the targeted instruction (for *base cost* measurement) or an alternating sequence of the instructions (for *inter-instruction effects* costs). As this program is executed, current drawn by the processor under test is directly measured and a power profile is built for this specific processor. Power profiles for different microprocessors were presented in [4, 5]. Table II illustrates a subset of the *base cost* table for the Intel 486DX2 and the Fujitsu SPARClite‘934 from [4].

Table II: Subset of the *base cost* table for the Intel 486DX2 and Fujitsu SPARClite ‘394.

Intel 486DX2				**Fujitsu SPARClite ‘934**			
Instruction	**Current (mA)**	**Cycles**	**Energy (10^{-8} J)**	**Instruction**	**Current (mA)**	**Cycles**	**Energy (10^{-8} J)**
nop	276	1	2.27	nop	198	1	3.26
mov dx,[bx]	428	1	3.53	ld [10],i0	213	1	3.51
mov dx,bx	302	1	2.49	or g0,i0,10	198	1	3.26
mov [bx],dx	522	1	4.30	st i0,[10]	346	2	11.4
add dx,bx	314	1	2.59	add i0,o0,10	199	1	3.28
add dx,[bx]	400	2	6.60	mul g0,r29,r27	198	1	3.26
jmp	373	3	9.23	Srl i0,1,10	197	1	3.25

Once the *instruction-level* power model, or power profile, is constructed for a certain microprocessor, the energy cost of any given program can be easily estimated. For any given program *P*, the overall energy cost, E_P, is given by:

$$E_P = \sum_i (Base_i * N_i) + \sum_{i,j} (Inter_{i,j} * N_{i,j}) + \sum_k E_k$$

where $Base_i$ is the *base cost* of instruction *i* and N_i is the number of times it will be executed. $Inter_{i,j}$ is the *inter-instruction* power overhead when instruction *i* is followed by instruction *j*, and $N_{i,j}$ is the number of times the (*i*,*j*) pair is executed. Finally E_k is the energy contribution of other *inter-instruction effects* (pipeline stalls and data caches) that would occur during program execution.

2.1.2 System-based SPM

There is little benefit in optimizing only the CPU core if other elements participate or sometimes even dominate the energy consumption. To effectively optimize system energy, it is necessary to consider all of the critical components. Different papers [6-9] investigate the power consumption on different system levels, targeting both hardware and software on different levels of abstraction.

In the *State-level models* approach, the energy consumption of the whole system is measured based on the state each device is in or transiting from or to. Other approaches work to identify the hotspots in applications and operating system procedures and try to reduce energy consumption by acting on the *application-, Compiler-* and *OS-levels*. Finally, a *complete system level* simulation tool, which models the CPU, memory hierarchy and a low power disk subsystem, was presented.

State-level models

As opposed to the low-level CPU simulators presented before, a high-level energy optimization technique was presented in [6]. Their proposed power model hides the complexity of the hardware state by encapsulating low-level details, but provides enough information allowing high-level optimization. This power state model accounts for the power spent in each of the device states and the transition between them. For each hardware subsystem, a set of device power states is defined (e.g. CPU: sleep, doze or busy). Each device state is characterized by the power consumption of the hardware during steady state. The relevant transitions between states occur as the result of system calls. By keeping track of system calls and measuring the transitional energy

consumption, every transition between states is assigned an energy consumption cost. The total energy consumed by the system is determined by adding the power of each device state multiplied by the time spent in that state plus the total energy consumption for all transitions.

The simulation environment was implemented as an extension of the Palm OS Emulator (POSE). POSE is a Windows based application that simulates functionality of the Palm device, emulating its operating system and instruction execution of the Motorola Dragonball processors used in the Palm. The power state model, described above, was incorporated into this existing environment.

To quantify the power consumption of a device and parameterize the simulator, experiments were held and measurements were taken using the above power model in order to capture transient energy consumption as well as steady state power consumption (results from [6] are presented in Tables III and IV). An IBM Workpad device was connected to a power supply with an oscilloscope measuring the voltage across a small resistor. The power consumption of the basic hardware subsystems of the Workpad device was measured: CPU, LCD, Backlight, Buttons, Pen and Serial link. Measurement programs, like *Power* and *Millywatt*, were used to provide a user interface to call some of the basic functions of the device for measurement intervals.

Table III: Steady state power of IBM Workpad (relative to the default mode: CPU doze, LCD on, Backlight off, Pen and Button up).

Device	State	Power (mW)
CPU	Busy	104.502
	Idle	0.0
	Sleep	-44.377
LCD	On	0.0
	Off	-20.961
Backlight	On	94.262
	Off	0.0
Button	Pushed	45.796
Pen	On Screen	82.952
	Graffitti	86.029

Table IV: Transient energy of IBM Workpad for significant system calls.

System Call	Transient Energy (mJ)
CPU Sleep	2.025
CPU Wake	11.170
LCD Wake	11.727
Key Sleep	2.974
Pen Open	1.935

Application-, Compiler- and OS-level

While hardware optimizations has been the focus of several studies and are fairly mature, software approaches for power optimization are relatively new. Software has a significant impact on the overall energy consumption being the main determinant for the hardware activity like the processor core, memory system and buses, which are, collectively, responsible for significant amount of total power dissipation. Despite this observation, to date, most of the compiler techniques consider mainly delay as their main performance metrics. With the growing demand for power-aware systems, there is an

urgent need for investigating energy-oriented compilation techniques and their interaction and integration with performance-oriented compiler optimizations.

In [19] a quantitative evaluation of the impact of different state-of-the-art high-level compilation techniques on energy consumption is presented. Different techniques, mainly targeting the widely used loop-optimizations, were evaluated vis-à-vis their impact on power consumption. As a result to this study, we find that the energy consumed in the memory system is higher than the core datapath in unoptimized code. We can also observe that most optimizations reduce the memory system energy but, on the other side, they increase the energy consumed in the core datapath, shifting the hotspot in the system from the memory to the system core, which will lead to think that more efforts should be focused to reduce the core power. Different low-level compiler techniques, applied during compile time to reduce energy consumption were proposed in [20-26], and their performance-power tradeoffs were studied using different power-aware simulators.

As an alternative approach to simulation, direct system measurement techniques were used in [7, 8] for power estimation. Using specially designed monitoring tools, these measurement-based techniques target the power consumption of the whole system and try to point out the hotspots in applications and operating system procedures. These tools mainly help programmers to produce power aware programs.

In [7], *PowerScope* maps energy consumption to program structure by augmenting the information gathered by a *time-driven statistical sampler*. As a result, one can determine what fraction of the total energy consumed, during a certain time period, is due to specific processes in the system. Further, we can go deeper and determine the

energy consumption of different procedures within a process. By providing such fine-grained feedback, *PowerScope* helps focusing attention on those system components responsible for the bulk of energy consumption. As improvements are made to these components, we quantify their benefits and move on to expose the next target for optimization. Through successive refinement, a system can be improved closer and closer to its energy consumption design goals. The functionality of *PowerScope* is divided among three software components. Two components, the *System Monitor* and *Energy Monitor*, share responsibility for data collection. The *System Monitor* samples system activity on the profiling computer by periodically recording information that includes the program counter (PC) and process identifier (PID) of the currently executing process. The *Energy Monitor* runs on the data-collection computer, and is responsible for collecting and storing current samples from the digital multimeter. The final software component, the *Energy Analyzer*, uses the raw sample data collected by the monitors to generate the energy profile, off-line. The analyzer runs on the profiling computer.

A similar tool was presented in [8], but this one is based on *energy-driven statistical sampling* and use energy consumption to drive sample collection. A simple 'energy counter' is interposed between the energy supply and the system under study. This counter measures the energy consumed by the system and causes an interrupt to be generated on the system whenever a predefined amount of energy, or energy quanta, has been consumed. The system handles these interrupts by executing a particular interrupt service routine that will record samples identifying the program instructions that were interrupted. The recorded samples are processed, off-line, to generate energy consumption estimates for each application, procedure, and instruction. Results show that

a non-trivial amount of energy is spent by the operating system. Additionally, there are often significant differences between the profiles generated by time and energy profiling, especially in workloads that transition quickly between multiple energy states and are undetected by the time driven sampling.

Figure 2 illustrates a high-level overview of the measurement-based power estimation technique presented above. Depending on the implementation, three, two or even one PC can perform the required tasks.

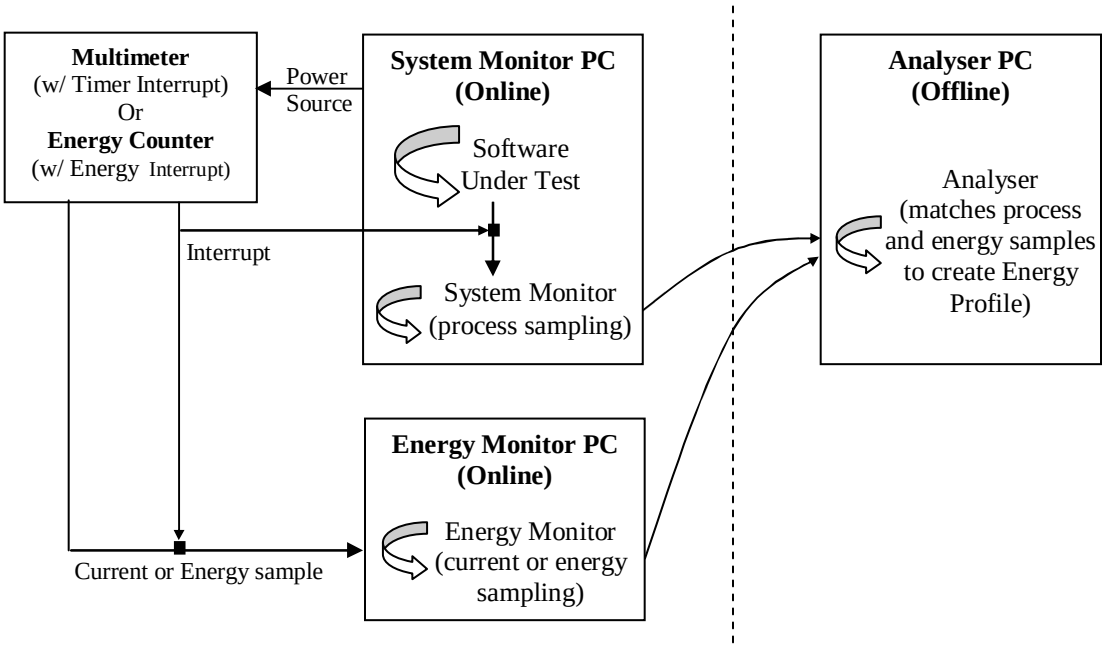


Figure 2: high-level overview of the measurement-based power estimation techniques.

Complete system level

All the simulation tools discussed earlier in this chapter focused mainly on particular hardware components such as CPU or memory, but did not capture the interaction between the different system components, and therefore, could not provide

complete description of the overall system behavior. To overcome this problem, a complete system power simulator, *SoftWatt*, was presented in [9]. It models the CPU, memory hierarchy and a low power disk subsystem and quantifies the power behavior of both the application and operating system. This tool was built on top of the *SimOS* infrastructure running the IRIX operating system, which provides detailed simulation of both, the hardware (CPU, memory and disk) and software (kernel, system and user applications). In order to capture the complete system power behavior, *SoftWatt* integrated different analytical power models into the different hardware components of *SimOS*. These power models were proposed and validated in separate previous works.

Results from running the *Spec JVM98* benchmark suite emphasized the importance of a complete system simulation to analyze the power impact of both architecture and OS on the application execution. From a system hardware perspective, we could see that the disk is the single largest power consumer of the whole system, but with the adoption of a low-power disk, the power hotspot was shifted to the clock distribution and generation network. Also, the memory subsystem was found to consume more power than the processor core. From the software point of view, the user mode had the maximum power consumption. The kernel mode had the least power consumption overall, but due to the frequent use of kernel services, it accounted for significant energy consumption in the processor and memory hierarchy. Thus, accounting the kernel code energy consumption is critical for estimating the overall energy budget. Finally, transitioning the CPU and memory-subsystem to a low-power mode or by even halting the processor during the idle-process turns out to save a fair amount of power.

Therefore, complete system-level simulators, like *SoftWatt*, seem to be one of the most promising SPM techniques for studying and improving the power consumption of the complete computing system during design time, or *off-line*.

2.2 Dynamic Power Management (DPM)

As opposed to SPM techniques, which are applied during design time, Dynamic Power Management techniques use runtime behavior to reduce power when systems are serving light workloads or are idle. DPM can be achieved in different ways; for example, dynamic voltage scaling (DVS) changes processor supply voltage at runtime as a method of power management [10-13]. DPM can also be used for shutting down unused I/O devices [15, 16], or even unused nodes of server clusters [17].

Three Dynamic Power Management implementation levels will be discussed in this section. Subsection 2.2.1 discusses DPM techniques applied at the *CPU- level*, using DVS. In subsection 2.2.2, a more general approach uses DPM at the *system-level* to save energy of all system components (memory, hard drive, I/O devices, display…). Finally, subsection 2.2.3 generalizes DPM techniques to be used on *multiple systems*, like a server cluster, where more than one system collaborates to save overall power.

2.2.1 CPU-based DPM: Dynamic Voltage Scaling (DVS)

The intuition behind the power saving in DVS comes from the basic energy equation, which is proportional to the clock frequency and the square of the voltage. Therefore, by dynamically changing the processor speed and voltage at runtime, DVS allows more than quadratic energy saving without, theoretically, affecting performance;

extra run cycles caused by the slower speed would be spread into idle time (additional details can be found in chapter III). The main problem for applying DVS is to know when to use full power and when not to, and this requires the cooperation of a voltage scheduler. Different voltage schedulers are presented in the following subsections.

Interval-based scheduler

Interval based voltage scheduler were proposed in [10, 11], they divide time into uniform length intervals and analyze system utilization of the previous intervals to determine the voltage of the next interval accordingly.

In [10], three interval-based schedulers were proposed: (1) OPT: this algorithm assumes unlimited knowledge of the future and spreads computation over the whole trace period to eliminate all idle time. (2) FUTURE: it uses a limited future look ahead to determine the minimum clock rate and therefore voltage. (3) PAST: this policy uses the recent past as a predictor of the future.

Some more complicated algorithms were presented in [11], they estimate the future workload based on two parameters: *run_percent* and *excess_cycles*. *run_percent* is the fraction of cycles where the CPU is active in an interval. *excess_cycles* is the cycles left over from the previous interval spilled over into later intervals when speed is not fast enough to complete and interval's work. Seven dynamic speed-setting policies were explained, discussed and compared: (1) PAST: this algorithm uses the recent past as a predictor of the future. (2) FLAT: Weak on prediction, this policy simply try to smooth speed to a global average. (3) LONG_SHORT: it's a more predictive policy that attempts to find a golden mean between local behavior and a more long-term average. (4)

AGED_AVERAGES: this policy employs an exponential-smoothing method, attempting to predict via a weighted average: one which geometrically reduces the weight given to each previous interval as we go back in time. (5) CYCLE: a more sophisticated prediction algorithm. It tries to take advantage of some previous *run_percent* values that looks quite cyclical, to predict. (6) PATTERN: a generalized policy from CYCLE. It attempts to identify the most recent *run_percent* values as repeating a pattern seen earlier in the trace. (7) PEAK: a more specialized version of PATTERN. It uses heuristics based on the expectation of narrow peaks. It expects rising *run_percents* to fall symmetrically back down and falling *run_percents* to continue falling.

Surprisingly, the simplest policy, FLAT, is optimal for low delay values, while LONG_SHORT, which is scarcely more complex, is optimal for the higher delay values. Of the most sophisticated predicting algorithms, PEAK does best, coming close to FLAT and LONG_SHORT in the medium-delay range. Several of the more complicated predictive algorithms performed poorly (AGED_AVERAGE, CYCLE, and PATTERN). We might then conclude that simple algorithms based on rational smoothing rather than complicated prediction schemes may be most effective. Nevertheless, further possibilities for prediction remain to be tried, like policies that might sort past information by process-type, or where applications could provide the system with useful information.

Schedulers for real-time systems

Interval based scheduling is simple and easy to implement, but it often incorrectly predicts future workloads and degrades the quality of service. In non-real-time systems, excess cycles left over from the previous interval might be spilled into later intervals

when speed is not fast enough to complete an interval's work. In a real-time system, tasks are specified by the task start time, the computational resources required and the task deadline. The voltage-clock scaling must be carried out under the constraint that no deadline is missed. An optimal voltage schedule is defined to be one for which all tasks complete on or before deadlines and the total energy consumed is minimized.

Two major scheduling techniques are offered for real time-systems: 1) *Inter-task* and 2) *Intra-task*. On one hand, *inter-task* schedules speed changes at each task boundary to meet a deadline associated with each task, while *intra-task* schedules speed changes within a single task. On the other hand, *inter-task* approaches make use of a prior knowledge of the application workloads and produce predictions for the application demands based on past history, while *intra-task* approaches try to take advantage of slack time, which results from the fact that within an individual task boundary the execution time may change significantly depending on the executed program path.

1) Inter-task schedulers:

Scheduling algorithms for real time systems, that minimize energy consumption while all tasks are guaranteed to complete on or before deadlines, were proposed in [12]. This technique is based on the assumption that the timing parameters of each job are known off-line. Two algorithms were given in the paper. The first one takes $O(N^2)$ time (where N is the number of jobs) to find the minimum constant speed needed to complete each job, since constant voltage tends to result in a low power consumption. The second algorithm, with $O(N^3)$ time complexity, build on the first one and give two results. First, the minimum constant voltage (or speed) needed to complete a set of jobs is obtained.

Secondly, a voltage schedule is produced, which is the set of critical intervals and their associated speed. This voltage schedule always saves more energy than the first algorithm, which applies the minimum constant speed when the processor is busy while shuts down the processor when it is idle. This approach to construct a low-energy voltage schedule is greedy since it tries to find the minimum constant speed during any critical interval. It guarantees to result the minimum peak power consumption. However it may not always produce the minimum-energy schedule.

In [13], more application specific DVS algorithms were proposed, targeting power consumption in MPEG decoding. The first algorithm is DVS-DM (*DVS with delay and drop rate minimizing algorithm*), which is a kind of interval-based DVS in a sense that it schedules voltage based on previous workload. This algorithm tries to scale the supply voltage according to the delay value and the drop rate. The second algorithm is DVS-PD (*DVS with decoding time prediction*), which determines the voltage not only by previous workload but also by predicted MPEG decoding time. The prediction, in this case, is based on frame size and frame type. From the simulation results in [13], it was found that DVS-PD shows the best performance with respect to energy consumption and DVS-DM is slightly better that the conventional shutdown algorithm. Outstanding energy saving with DVS-PD is due to higher prediction accuracy of future workload than other approaches. It's also found that energy saving is closely related with average decoding time and fluctuation. With DVS-DM, high fluctuation makes it difficult to predict future workload based on the previous workload only and it results in low efficiency. On the contrary, it's found that that DVS-PD is not much affected by the fluctuation. Instead, performance of DVS-PD in terms of energy consumption depends on the error rate of the

predictor, which implies that if decoding time is predicted more accurately, DVS algorithm can be more efficient. More details concerning MPEG decoding and DVS can be found in chapter III. Our proposed DVS prediction algorithms are presented in Chapter IV.

All the above *inter-task* scheduling techniques are applied *online*, during execution time. DVS is applied only on the task boundaries.

2) Intra-task Schedulers:

As opposed to the above *inter-task* scheduling techniques, which are applied *online*, during execution time, *intra-task* techniques are applied *offline*, during compile-time. They try to identify different possible paths within one task, and change the voltage accordingly to save power while meeting all the deadlines. Figure 3 shows the different paths one task can take during execution, mainly because of the conditional statements (if-then-else, while, etc…). Each node represents a basic block of this task, with the number of cycles required to execute it. Depending on the chosen path, the total number of cycles varies for the same task, which means a different execution time and therefore a possible frequency/voltage scaling to save power while still meeting the deadline. All the techniques proposed below try to take advantage of this *intra-task* slack time to reduce power consumption.

Intra-task DVS technique, using program checkpoints under compiler control, was introduced in [28]. Checkpoints indicate places in the code where the processor frequency and voltage should be re-calculated and scaled. They are generated at compile time. The program is profiled, using a representative input data set, and collect

minimum/maximum energy dissipated and cycle count for checkpoint transitions. A run-time voltage scheduler is created and follows, in an energy efficient way, the run-time power profile, which represents the available power budget, while simultaneously meeting the deadline.

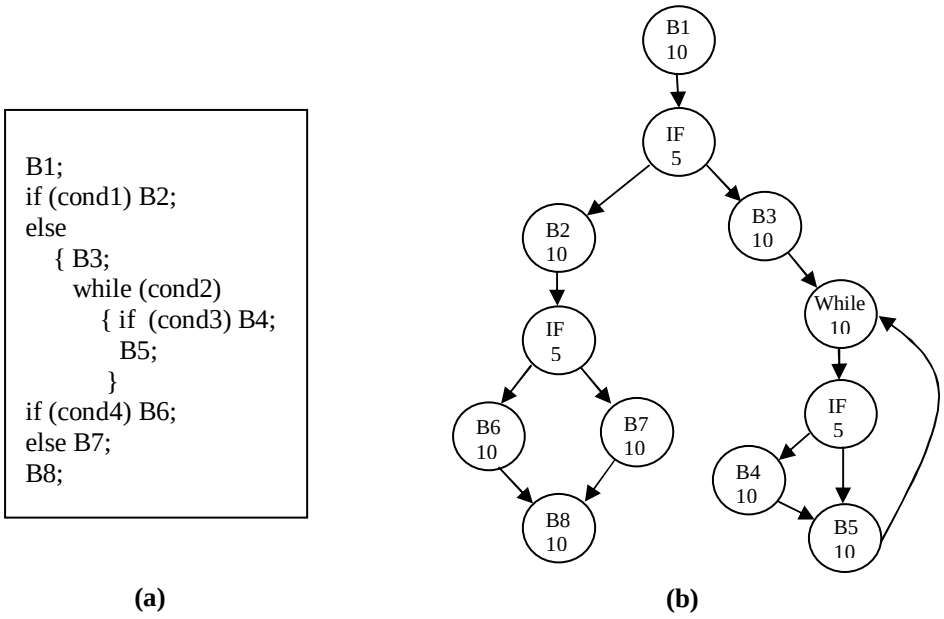


Figure 3: Intra-task paths. (a) Example program, and (b) its flow graph.

A similar approach was also presented in [29], where the compiler is used to annotate an application's source code with temporal information. This information captures the dynamic behavior of the application, which may vary by executing different paths with different execution times. During program execution, the operating system periodically adapts the processor's frequency and voltage based on this temporal information. The main contribution of this scheme is the collaborative compiler and operating system *intra-task* approach. It uses the strength of each of the compiler and OS to get fine-grained information about an application's execution, and then applies DVS.

The COPPER (Compiler-controlled continuous Power-Performance) framework was presented in [27]. COPPER uses a variety of architectural and compiler technologies to control the power profile of the application. It focuses on dynamic register file reconfiguration, frequency and voltage scaling. The power profile is controlled by creating multiple code versions that are selected by the runtime system. This helps achieving performance goals within energy constraints. The information computed by the compiler, such as time, energy profile and code characteristics, is carried down to the run-time system using tables and code annotations.

In [31], an Automatic Voltage Scaler (AVS) was proposed; it automates the development of real-time programs on a variable-voltage processor. Using AVS, DVS-unaware real-time programs can be converted to DVS-aware low-energy programs in a way completely transparent to software developers.

Finally, [32] explores the opportunities and limits of compile-time DVS scheduling. A detail analytical model was presented, that helps determine the achievable power savings in terms of simple program parameters, the memory speed, and the number of available voltage levels. This model helps point to scenarios, in terms of these parameters, for which we can expect to see significant energy savings, and scenarios for which we cannot. One important result of this modeling is that as the number of available voltage levels increase, the energy savings obtained decrease significantly. If we expect future processors to offer fine grain DVS settings, then compile-time intra-program DVS settings will not yield significant benefit and thus will not be worth it.

2.2.2 System-based DPM

There is little benefit in optimizing the microprocessor core if other elements dominate the energy consumption. Therefore, to effectively optimize system energy, it is necessary to consider all of the critical components.

A system-level power management technique, which targets saving the power of subsystems or devices, was presented in [15]. Examples of devices include hard disk drives, I/O controllers, displays and network interface cards. The most widely adopted system-level power management technique is shutting down hard drives and displays, after some time of idleness. Other unused I/O devices can be equally shut down to save energy, which was the purpose of the DPM techniques discussed in [15]. But, changing power states has both time and power overheads. Consequently, a device should sleep only if the saved energy justifies the overhead. Therefore, the main problem in successfully applying these techniques is to know when to shut down a unit and when to wake it up.

Power management policies can be classified into three categories based on the methods to predict whether a device can sleep long enough: (1) *Time-out policies*: assume that after a device is idle for a certain time-out value, it will remain idle for at least T_{be} (*break-even time*, the minimum length of an idle period after which shutting down the device will save power). The main drawback of these policies is the energy wasted during this time-out period. (2) *Predictive policies*: eliminate the time-out period by predicting the length of an idle period before it starts. When an idle period is predicted to be longer than the break-even time (T_{be}), the device sleeps right after it's idle. (3)

Stochastic policies: model the arrival of requests and device power-state changes as stochastic processes, such as Markov processes.

The policies mentioned above, were implemented using *filter driver*, which is a device driver inserted between the operating system kernel and another device driver. The filter driver intercepts communications between the two drivers and can pass, add, delete or change the exchanged messages.

Each policy was graded by six criteria: power, number of shutdowns, shutdown effectiveness, interactive performance, memory and computation requirements. No policy was found to have best grades for all criteria. When a policy saves power aggressively, it usually generates more shutdowns and degrades performance. On the other hand, if a policy is more conservative in power saving, it is likely to issue fewer shutdowns. While performance and accuracy improve, these policies consume more power. Finally, the resource requirements of a certain policy are also important. Even though providing excellent power savings, some policies become less appealing because they require a substantial amount of energy, resource generally scarce and expensive.

In [16], an OS-directed power management technique was proposed in order to improve the energy efficiency of sensor nodes using DPM. The basic idea is to shut down devices (CPU, memory, sensor, radio…) when not needed and wakes them up when necessary. A power-aware sensor node model essentially describes the power consumption in different levels of node sleep states. Every component in the node can have different power modes, but also has latency overhead associated with transitioning to that mode. Therefore each node sleep mode is characterized by power consumption

and latency overhead. In general a deeper sleep state consumes less power and has a longer wake-up time.

2.2.3 Cluster System-based DPM

Dynamic Power Management techniques can also be extended and applied to more than just one system at a time. In [17], DPM was used in server clusters, reducing the energy consumption of the whole cluster by coordinating and distributing the work between all available nodes.

Five policies for reducing the energy consumption of server clusters with varying degrees of implementation complexity were presented. The first policy, *Independent Voltage Scaling* (IVS), simply uses voltage scaled processors. Each node independently manages its own power consumption. The second policy also uses DVS but in a coordinated manner between nodes to reduce cluster power consumption. It's called *Coordinated Voltage Scaling* (CVS). The third policy, called vary-on/vary-off (VOVO), turns off server nodes so that only the minimum number of servers required to support the workload are kept alive. Nodes are brought online as and when required. The fourth policy, called *Combined Policy*, combines IVS and VOVO while the fifth uses a combination of CVS and VOVO and is called *Coordinated Policy*.

These policies were evaluated in terms of both their response time and energy savings. Combining DVS with VOVO offers the most energy savings with VOVO-IVS and VOVO-CVS. All five policies can be engineered to keep server response times within acceptable norms.

CHAPTER III

MPEG DECODING AND DYNAMIC VOLTAGE SCALING (DVS)

3.1 MPEG Decoding

MPEG video compression is used in many current and emerging products. It is at the heart of digital television set-top boxes, DSS, HDTV decoders, DVD players, video conferencing, Internet video, handheld PCs, mobile phones and other applications. These applications benefit from video compression in the fact that they may require less storage space for archived video information, less bandwidth for the transmission of the video information from one point to another or a combination of both. Besides the fact that it works well in a wide variety of applications, a large part of its popularity is that it is defined in finalized international standards (MPEG 1, 2, 4, 7 and 21). In this thesis, MPEG-2 is used. The acronym MPEG stands for Moving Picture Expert Group [47], which worked to generate the specifications under ISO, the International Organization for Standardization [45] and IEC, the International Electrotechnical Commission [46].

In this section we describe the MPEG decoding characteristics and specifications. Section 3.1.1 explains the MPEG video layers. The MPEG format is presented in 3.1.2.

Section 3.1.3 overviews the MPEG encoding/decoding processes. Finally, section 3.1.4 illustrates the variability in the MPEG decoding process.

3.1.1 MPEG Video Layers

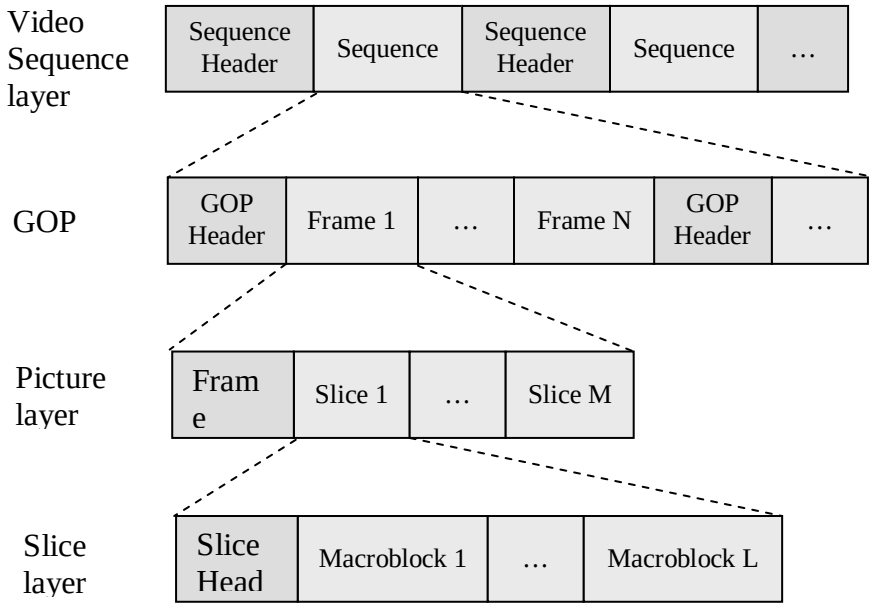


Figure 4: MPEG layers hierarchy.

MPEG video is broken up into a hierarchy of layers (Figure 4) to help with error handling, random search and editing, and synchronization with an audio bitstream. From the top level, the first layer is known as the video sequence layer, and is any self-contained bitstream, for example a coded movie or advertisement. The second layer down is the group of pictures (GOP), which is composed of one or more groups of intra (I) frames and/or non-intra (P and/or B) pictures that will be defined later. The third layer down is the picture layer itself, and the next layer beneath it is called the slice layer. Each slice is a contiguous sequence of raster ordered macroblocks, most often on a row basis in

typical video applications, but not limited to this by the specification. Finally, each slice consists of macroblocks, which are composed of arrays of luminance and chrominance pixels, or picture data elements.

3.1.2 MPEG Format

The MPEG video compression standard [41] defines a video stream as a sequence of still images or frames. A standard MPEG stream is composed of three types of compressed frames: *I*, *P* and *B*. *I* frames are only *intra-coded*, which refers to the fact that the various compression techniques are performed relative to information that is contained only within the current frame, and not relative to any other frame in the video sequence. In other words, only *spatial* processing is performed within the current picture or frame. The generation of *P* and *B* frames involves, in addition to intra-coding, the use of *motion prediction* and *interpolation techniques* in order to exploit the inherent *temporal*, or time-based, redundancies providing more efficient compression. As a result, *I* frames are, on the average, the largest in size, followed by *P* frames, and finally *B* frames.

After being decoded, video presentation units (i.e. frames) may be delayed in reorder buffers before being presented to the viewer. This is because, during encoding, MPEG transforms video frames into a sequence of *Intracoded (I), Predictive-coded (P), and Bidirectionally-coded (B)* frames, producing a sequence such as follows:

$$I_1\ B_2\ B_3\ B_4\ \mathbf{P_5}\ B_6\ B_7\ B_8\ \mathbf{P_9}\ B_{10}\ B_{11}\ B_{12}\ \mathbf{I_{13}} \quad (1)$$

The point to observe is that, a B frame is bidirectionally encoded from both its preceding I or P and its succeeding I or P frame; hence, at the time of decoding, the B frame would need, not only its preceding I or P frame, but also its succeeding I or P. Thus he MPEG

encoder places the succeeding I or P prior to the corresponding B frame. As a consequence, the above sequence would appear in the encoded stream as follows:

$$I_1 \ \mathbf{P_5} \ B_2 \ B_3 \ B_4 \ \mathbf{P_9} \ B_6 \ B_7 \ B_8 \ \mathbf{I_{13}} \ B_{10} \ B_{11} \ B_{12} \qquad (2)$$

During decoding, the P_5 is decoded before B_2, B_3 and B_4. P_9 is decoded before B_6, B_7 and B_8. I_{13} is decoded before B_{10}, B_{11} and B_{12}. These would have to be reordered back into the original sequence (1). This resequencing (2) is done in the display reorder buffers immediately after decoding. In particular, an I-picture or a P-picture decoded before one or more B-pictures must be delayed in the reorder buffer. It should be delayed until the next I-picture or P-picture is decoded. Thus, the decoding time and the presentation times differ by an integral of pictures for these reordered frames.

3.1.3 MPEG Encoding/Decoding

Figure 5 illustrates the MPEG video compression process. Video compression relies on the eye's inability to resolve High Frequency color changes, and the fact that there is a lot of redundancy within each frame and between frames. The encoder starts by converting the RGB signal (Red, Green, and Blue) into a YUV signal (Y represents the luminance signal, or how bright the picture is, and UV are two color difference signals). Then, the Discrete Cosine Transform is used, along with quantization and Huffman coding to predict a pixel value from all adjacent pixel values, removing the spatial redundancy: This generates the Intra-frames (I-frames). Prediction and motion compensation predicts the value of pixels in a frame from the information in adjacent frames, removing temporal redundancy: This generates P and B frames.

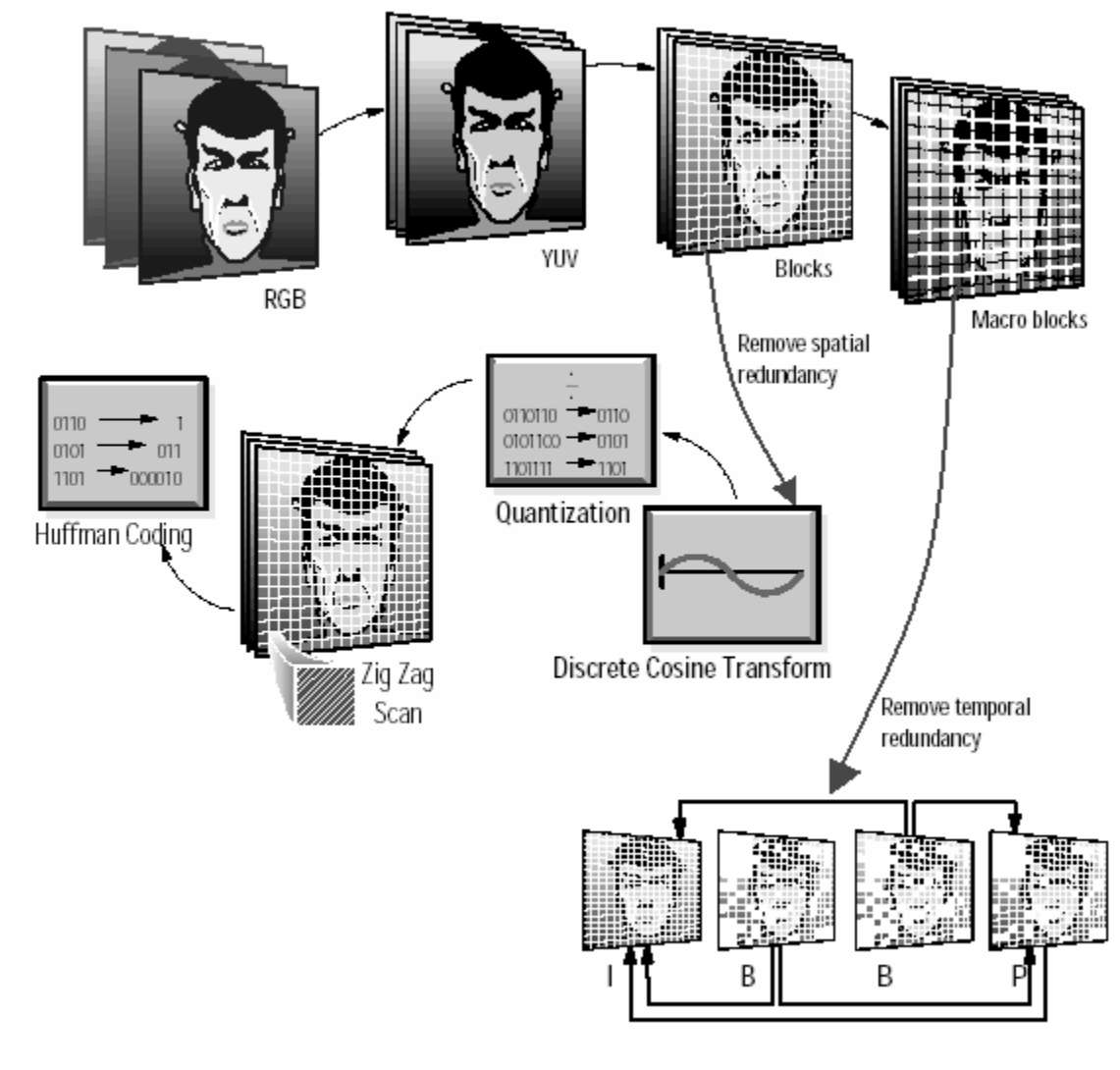


Figure 5: MPEG video compression (encoding) [44].

To decode a bitstream generated from the above encoder, it is necessary to reverse the order of the encoder processing. In this manner, an I frame decoder consists of an input bitstream buffer, a Variable Length Decoder (VLD, which restores the original lengths of the variable length codes produced by the encoder), an Inverse Quantizer (IQ), an Inverse Discrete Cosine Transform (IDCT), and an output interface to the required environment. For B and P frames, additional Motion Compensation (MC) and its

associated support structures are required. Figure 6 shows a block diagram of the MPEG decoder.

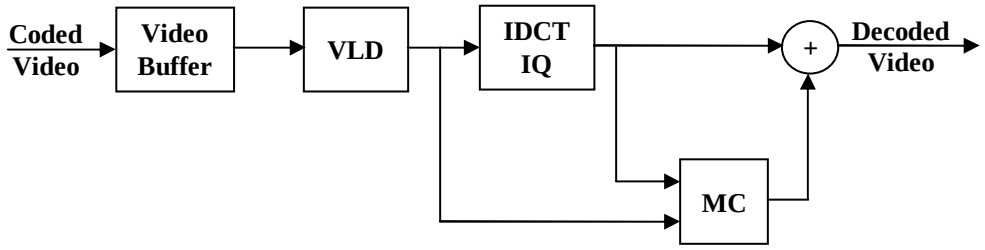


Figure 6: Block diagram of the MPEG decoder.

3.1.4 Variability in MPEG Decoding

We selected three movie clips for evaluating our algorithms with respect to energy consumption and drop rate. The general characteristics of each clip are shown in Table V below.

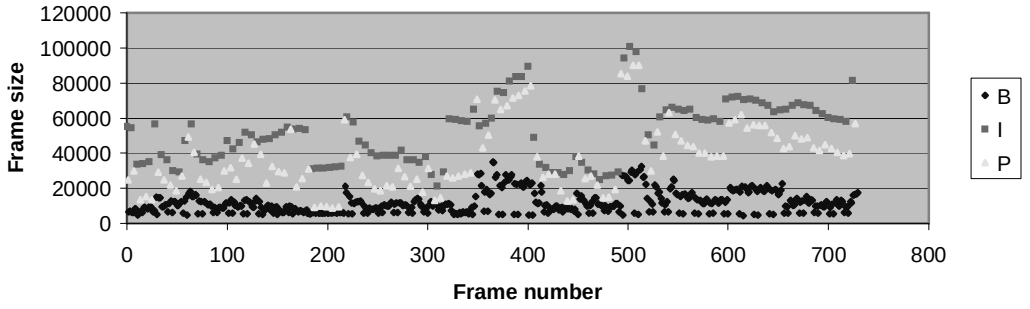
Table V: Movie clips characteristics.

Title	*Red's Nightmare*	*The Animatrix*	*UnderSiege*
Type	Animation (low motion)	Animation (High motion)	Movie (High motion)
Frame Rate	25 fps	23.976 fps	30 fps
Resolution	320x240	592x252	352x240
I frames	41	107	122
P frames	81	428	122
B frames	1088	1070	486

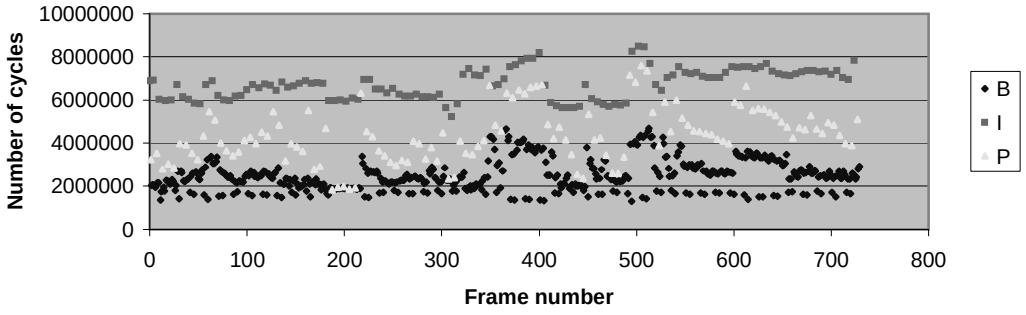
Figure 7 shows the frame size (a) and the number of cycles (b) for each frame in the *UnderSige* clip. We can notice the resemblance in the shape of the two graphs indicating the existence of a relationship between the two metrics, the frame size and the

number of cycles required to decode the frame. Figure 8 represents the required decoding clock according to the frame size. It can be simply analyzed to represent linear line shape.

However, it is unsuitable to decide the decoding clock only with frame size because as we can see, we have different shapes for different types and that's because I frame requires less decoding time per byte while B frames requires more decoding time per byte (see the slopes in Figure 8 and Table VI); If there are two frames with same size, one I frame and the other B frame, B frame has longer decoding time than I frame. Therefore, categorizing this data by frame type, we can get more accurate linear regression models (Table VI). Similar results were obtained for the other two movies. Moreover, this approach can be applied to different frame resolution because most MPEG stream represent similar regression model.



(a) Frame size.



(b) Number of cycles.

Figure 7: *UnderSiege*.

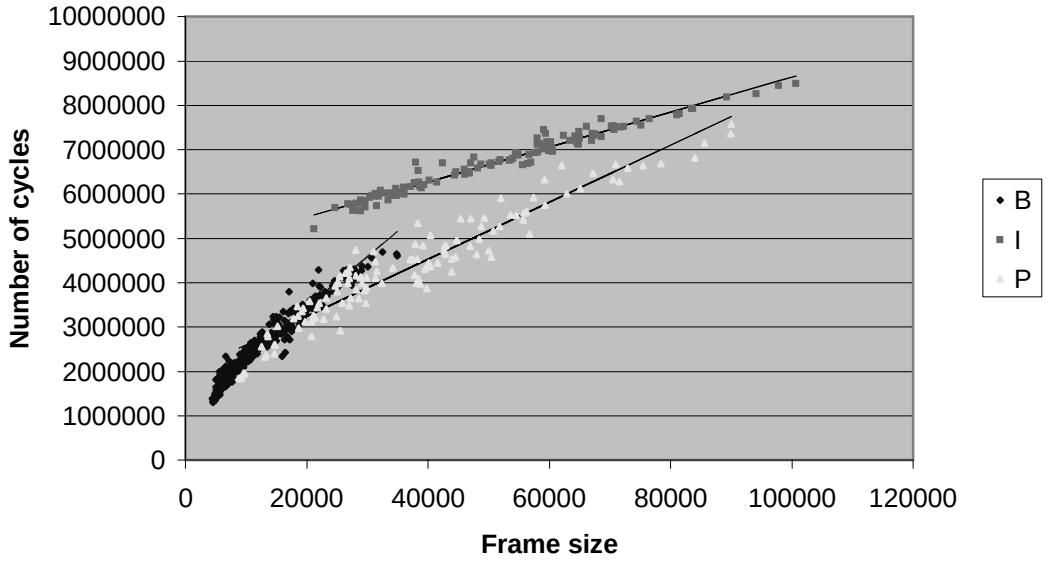


Figure 8: Number of cycles vs. Frame size (*UnderSiege*).

Table VI: Regression model for the expected decoding cycle.

	Regression model	R^2
I frame	39*Frame Size + 4.7*10^6	0.96
P frame	64*Frame Size + 1.9*10^6	0.92
B frame	115*Frame Size + 1.1*10^6	0.95

3.2 DVS for Low-Power MPEG Decoding

DVS has been proposed as a mean for a processor to deliver high performance when required, while significantly reducing power consumption during low workload periods. The advantage of DVS can be observed from the power consumption characteristics of digital static CMOS circuits and the clock frequency equation [37]:

$$E \propto C_{eff} V^2 f_{CLK} \tag{3}$$

$$delay \propto \frac{V}{(V - V_k)^\alpha} \text{ and } f_{CLK} \propto \frac{(V - V_k)^\alpha}{V} \tag{4}$$

where C_{eff} is the effective switching capacitance of the operation, V is the supply voltage,

and f_{CLK} is the clock frequency. α ranges from 1 to 2, and V_k depends on threshold voltage at which velocity saturation occurs [37].

Decreasing the power supply voltage would reduce power consumption quadratically as shown in (3). However, this would create higher propagation delay and thus force a reduction in clock frequency as in (4), leading to even lower power consumption. While it is generally desirable to have the frequency as high as possible for faster instruction execution, for some tasks where maximum execution speed is not required, the clock frequency and supply voltage can be reduced so that power can be saved. DVS takes advantage of this tradeoff between energy and speed. Since processor activity is variable, there are idle periods when no useful work is being performed, yet power is still consumed. DVS can be used to eliminate these power-wasting idle times by lowering the processor's voltage and frequency during low workload periods so that the processor will have meaningful work at all times, which leads to reduction in the overall power consumption.

However, the difficulty in applying DVS lies in the estimation of future workload. Different DVS prediction schemes were proposed in the literature, and some of these previous works were described in section 2.2.1. In this section, specially designed DVS algorithms for low-power MPEG decoding are described.

3.2.1 Example Study on DVS-based Energy Efficient MPEG Decoding

Figure 9 illustrates the advantage of DVS in video decoding as well as the design tradeoff between prediction accuracy, power savings, and deadline misses. The processor speed on the y-axes directly relates to voltage as discussed earlier, and reducing the speed

allows the reduction in supply voltage, which in turn results in power savings. The shaded area corresponds to the work required to decode the four frames and it is the same in all three cases.

Figure 9(a) shows the processor activity when the On/Off mechanism is used, which means that the processor runs at a constant speed when decoding and is shut down when idle. However, the corresponding power consumption is the largest because it uses the highest voltage/frequency setting and there is a quadratic relationship between the supply voltage and power consumption. Thus, power consumption for decoding frame1 using On/Off mechanism is: $23*1.8^2 = 74.5$ mW. Once a frame is decoded, the processor waits until its deadline (e.g., every 30 ms for the frame rate of 33 frames per second or fps), when the frame must be played out. During this waiting time the processor is idle and shut down. These idle periods are the target for exploitation by DVS. Figure 9(b) shows the ideal DVS case where the processor scales exactly to the voltage/frequency setting required for decoding the corresponding frame. Therefore, no idle time exists and power saving is maximized (power consumption for decoding frame1 using Ideal DVS would be: $30*1.5^2 = 67.5$ mW). Achieving this goal involves two important steps. First, the decoding time must be predicted. Second, the predicted decoding time is mapped to an appropriate voltage/frequency setting.

Inaccurate predictions in decoding time and/or insufficient number of voltage/frequency settings will introduce errors that lead to reduction in power saving and/or increase in missed deadlines as shown in Figure 9(c). In this figure, the decoding time for frame 1 is overestimated, resulting in more power consumption than required (power consumption for decoding frame1 using DVS with inaccurate predictions would

be: $27*1.6^2 = 69.5$ mW). On the other hand, the decoding times for frames 2 and 3 are underestimated, which leads to deadline misses that may degrade the video quality.

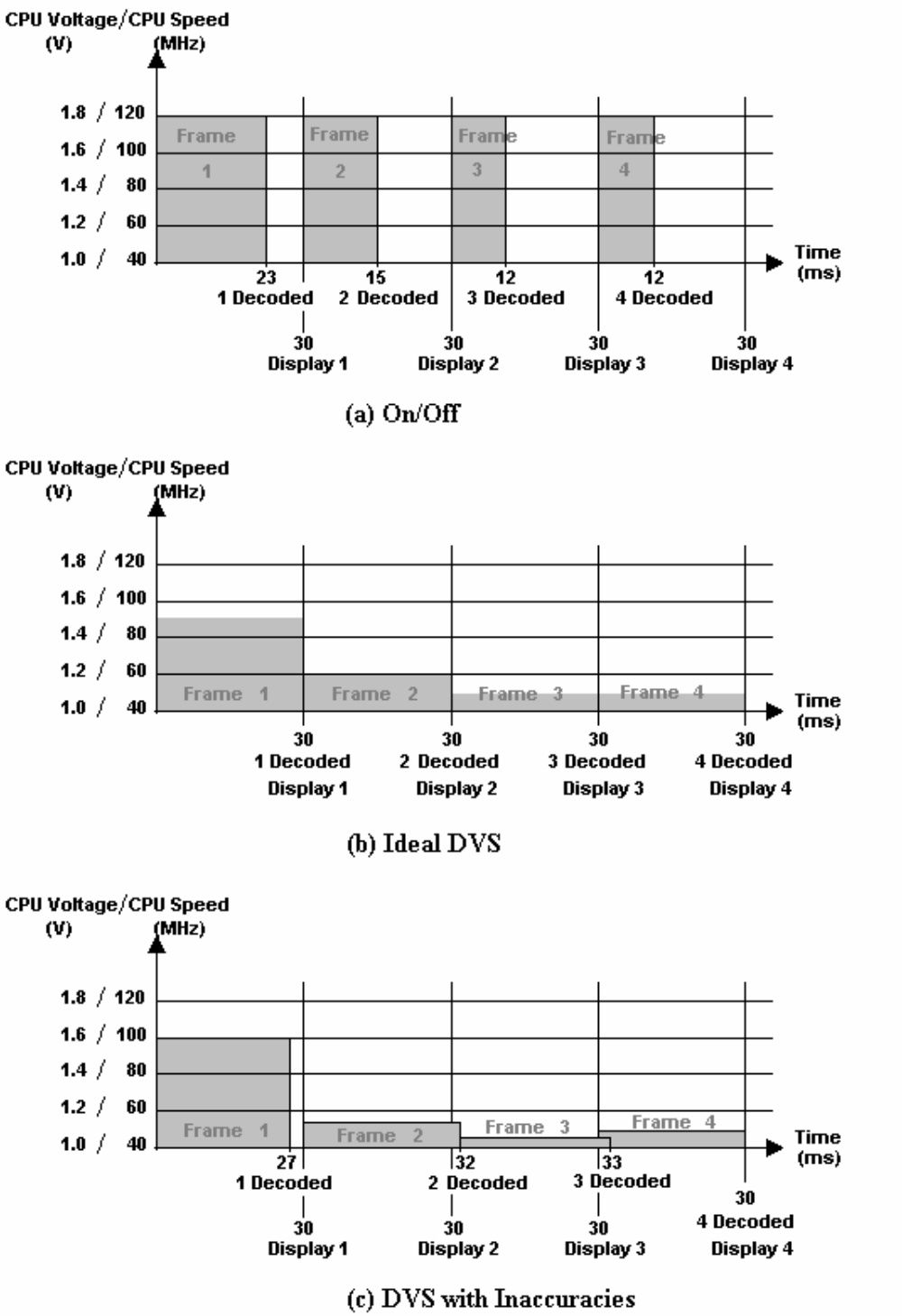


Figure 9: DVS for MPEG decoding.

3.2.2 Previous Low-Power MPEG Decoding Based on DVS Techniques

Based on the aforementioned discussion, DVS has a great potential in applications with high varying workload intensities such as video decoding, but accurate workload prediction is prerequisite in realizing the benefit of DVS.

Prediction algorithms employed in several DVS approaches differ mainly in the *prediction level* and *prediction mechanism*. *Prediction level* refers to the decoding level where predictions are made and processor settings are changed. The existing approaches use either *per-GOP* [13, 38] or *per-frame* [38-40] scaling. In *per-GOP* approaches, since the same voltage/frequency is used during decoding the whole GOP, we do not take full advantage of the high variability of decoding times among frames within a GOP.

Prediction mechanism refers to the way the decoding time of an incoming frame or GOP is estimated. Currently, all the approaches utilize some form of frame size and type vs. decoding time relationship. Some methods are based on a fixed relationship [13, 38, 40], while others use a dynamically changing relationship [38, 39]. In the fixed approach, a linear equation describing the relationship between frame sizes/frame type and frame decoding times is provided ahead of time. In the dynamic approach, the frame-size and decoding-time relationship is dynamically adjusted based on the actual frame sizes and decoding times of a video stream being played. The dynamic approach is better for high-motion videos where the workload variability is extremely high. In other cases, the fixed approach may perform better but its practical value is limited because the relationship is not usually available before actually decoding the stream.

In [13], two DVS algorithms were presented. One is *DVS with delay and drop rate minimizing algorithm* (DVS-DM) where voltage is determined based on previous

workload only. The second algorithm, *DVS with predicted decoding time* (DVS-DP), scales the supply voltage according to the predicted MPEG decoding time and previous workload. Simulation results show that DVS-PD improves energy efficiency without being affected by the fluctuation of the movie stream but it was related with the error rate of the predictor.

Three DVS techniques for video decoding and the corresponding prediction algorithms were discussed and compared in [38]: *GOP* is a per-GOP scaling approach that dynamically recalculates the slope of the frame-size/decode-time relationship based on the decoding times and sizes of past frames. In *Direct*, which is a per-frame and fixed approach, a linear model between frame sizes and decoding times is given, and then decoding time of a new frame is estimated. Finally, *Dynamic* is a per-frame scaling method that dynamically updates the frame-size/decoding-time model and the weighted average decoding time. Out of the three approaches simulated, *Dynamic* and *Direct* provided the most power savings. Among the two, *Dynamic* was much more practical, because of it is able to dynamically adapt to any video stream. The implementation of such an approach would not change any external behavior of the system. Thus, this approach is very suitable for portable multimedia devices, which require low-power consumption.

In the DVS algorithm presented in [39], the decoding time prediction is performed by maintaining a moving average of the decoding time for each frame type. Then, the decoding process is divided into two parts based on the required execution time and the expected energy consumption. One part captures the frame-dependant (FD) portion of the decoding process whereas the other part captures the frame-independent

(FI) portion. The FD part varies greatly according to the type of the incoming frame whereas the FI part remains constant regardless of the frame time. In the DVS scheme proposed in [40], the FI part is used to compensate for the prediction error that may occur during the FD part such that a significant amount of energy can be saved while meeting the frame rate requirements.

CHAPTER IV

PROPOSED DVS SCHEMES FOR POWER AWARE MPEG DECODING

This Chapter proposes three voltage estimation algorithms, which, based on the frame size, tries to estimate the voltage required to decode each frame just on time for display. These algorithms were introduced in section 4.1. In section 4.2, an additional voltage averaging technique was applied on top of the previous algorithms to allow more power saving. Finally, section 4.3 describes our proposed technique to obtain the frame size information, used by the voltage estimators.

4.1 Voltage Estimation

In this section we present our proposed voltage estimation algorithms: *regression*, *interval-avg*, and *interval-max*. All three algorithms try to estimate the number of clock cycles, required to decode each frame based on the frame size and type. The required voltage will be calculated based on the estimated decoding time. No ahead-of-time information is required for the estimation, as a dynamic approach is followed in all three

techniques. These algorithms are run separately for each frame type. Figure 10 shows the decoding time (or number of clock cycles) as a function of frame length for B frames of a sample movie clip.

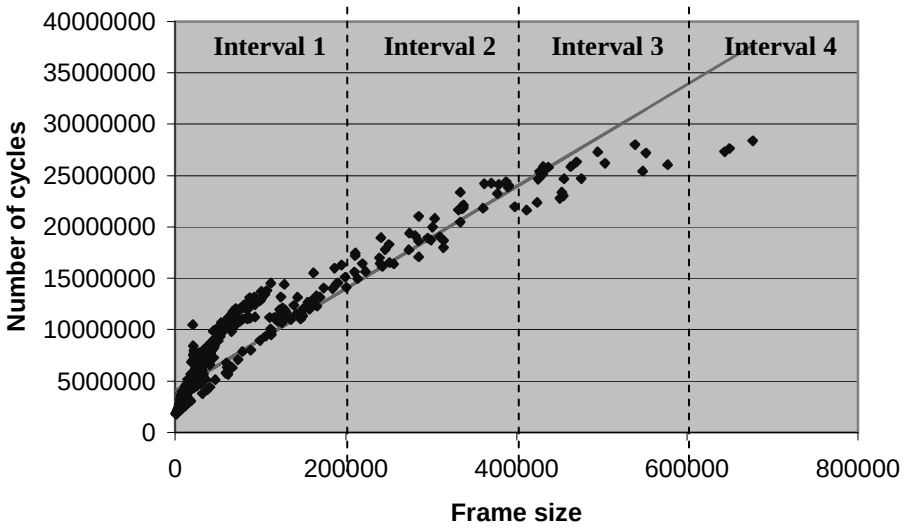


Figure 10: Decode time as a function of frame size.

Regression

This algorithm is based on the observation that the frame-size/decoding-time distribution follows a linear regression model [40]. In the example above (Figure 10) we can see the linear relationship (straight line), which will be used to estimate the decoding time based on the frame size: *number of cycles* = 50 * *frame size* + 4020185, with an R^2 coefficient of the linear regression through these sample points of 0.95, which looks very promising. But in order to construct this linear equation, ahead-of-time information on all these sample points is required. Our proposed *regression* algorithm bypasses this need of any a priori information by dynamically recalculating the slope of the frame-

size/decoding-time relationship based on the decoding times and sizes of past frames. The main difference between this algorithm and the other dynamic algorithms presented in [32, 33] is their prediction mechanism: while the other algorithms use a weighted average decoding time, *regression* estimates the decoding time using a linear regression equation, built from all the previous frames. This linear equation is revised each time a new frame is decoded, becoming more and more representative of the whole samples set. Figure 11 shows the algorithm for the *regression* approach.

For each frame *i* :

1- Get $frame\_size_i$ from frame header

2- $estimate\_decode\_time_i = actual\_linear\_regression\_equation(frame\_size_i)$

3- Get $real\_decode\_time_i$ after decoding the frame

4- Recalculate *actual_linear_regression_equation* based on all previous $frame\_size_{0...i}$ and $real\_decode\_time_{0...i}$ using the linear regression model

---

Figure 11: *regression* algorithm.

Depending on the frame-size/decoding-time distribution of each movie clip, this technique is expected to provide good results especially when R^2, the coefficient of linear regression, is closer to the unity. With *regression* we should get very accurate estimation of the decoding time, meaning close to ideal voltage scheduling, and therefore great power saving. But as far as QoS, the drop rate might be high because this algorithm is mainly based on averaging, and although the prediction is accurate, many frames will be slightly under-estimated but still be dropped.

Interval-avg

The main problem with the *regression* algorithm is that it is very computationally expensive. For each frame, heavy calculations must be performed in order to compute the linear regression equation. Therefore, a lighter version of the *regression* algorithm is proposed and presented in this section: *interval-avg*. This algorithm divides the frame-size/decoding-time distribution into *intervals* (as shown in Figure 10), making decisions locally within each interval based on simple average calculation. Depending on its size, every frame will belong to a certain interval. We assume that the interval size and/or the number of intervals are statically set and that each clip includes the information on the maximum and minimum frame size for the whole clip in the header of the first frame. *Interval-avg* estimates the decoding time of each frame based on the average decoding time for the interval the frames belong too.

This algorithm is expected to behave similarly to the *regression* algorithm by giving good estimates on the decoding time, bringing the power consumption down toward an ideal DVS behavior, but because of the averaging estimation effect, some number of frames will be under-estimated and dropped even if the error value is small. Figure 12 describes the *interval-avg* algorithm.

It is also noted that, for this algorithm, the interval size is an important design parameter: smaller intervals lead to fewer dropped frames because of a more representative averaging for each interval.

Initialization:

1- Get maximum and minimum frame size information from the first frame in the movie clip

2- Compute *interval_size*, and set *interval[1…n]*

For each frame i :

1- Get $frame\_size_i$ from frame header

2- Find *interval[k]* where $frame_i$ belongs according to its $frame\_size_i$

3- If *interval[k]* is not empty then $estimate\_decode\_time_i = Avg\_value\_of\_interval[k]$

4- else $estimate\_decode\_time_i = Avg\_value\_of\_interval[j]$ where *interval[j]* is the closest not empty interval to *interval[k]*

5- Get $real\_decode\_time_i$ after decoding the frame

6- Use $real\_decode\_time_{0...i}$ to recalculate Avg_*value_of_interval[k]*

Figure 12: *interval-avg* algorithm.

Interval-max

In order to bypass the averaging problem found in both previous algorithms, which caused a poor QoS, we designed this *interval-max* algorithm. *Interval-max* uses the same interval technique discussed in *interval-avg*, but instead of making the estimation based on the average value, *interval-max* uses the maximum value found in each interval.

This scheme presents a very easy to implement and yet, a very safe algorithm with improved QoS parameter. The overestimation of most frames will mainly result in fewer frames being dropped but more power consumed. Figure 12 describes the *interval-max* algorithm.

Again, we note that the interval size plays an important role balancing power consumption and QoS factor. In other words, having smaller intervals means closer upper bound values to the real values, and thus less power consumed at the cost of degraded QoS with more frames dropped.

Initialization:

1- Get maximum and minimum frame size information from the first frame in the movie clip

2- Compute *interval_size*, and set *interval[1…n]*

For each frame i :

1- Get $frame\_size_i$ from frame header

2- Find *interval[k]* where $frame_i$ belongs according to its $frame\_size_i$

3- If *interval[k]* is not empty then $estimate\_decode\_time_i = Max\_value\_in\_interval[k]$

4- else $estimate\_decode\_time_i = Max\_value\_in\_interval[j]$ where *interval[j]* is the closest not empty interval to *interval[k]*

5- Get $real\_decode\_time_i$ after decoding the frame

6- If $real\_decode\_time_i > Max\_value\_in\_interval[k]$ then $Max\_value\_in\_interval[k] = real\_decode\_time_i$

Figure 13: *interval-max* algorithm.

4.2 Voltage Averaging

In addition to the voltage estimation algorithms presented in the previous section, a voltage averaging technique can be used to provide more power saving. This technique is based on the observation that less power is consumed when using the average voltage rather than using different voltages during the same amount of time. Figure 14 below illustrates this observation. Figure 14(a) shows a regular DVS where power computation is 205.8 mW ($30*1.2^2 + 30*1.4^2 + 30*1.5^2 + 30*1.1^2$). Figure 14(b) shows the additional voltage averaging technique added to the regular DVS. Voltage averaging works as follow: after estimating the voltage required for each frame, a set of consecutive frames where the voltage required for each frame is higher than the previous one is created (Frames 1 to 3 in Figure 14). The new voltage for all the frames in the set is the average of their old voltages (in Figure 14(b), 1.36 = (1.2 + 1.4 + 1.5) / 3). As seen in Figure 14(b), the corresponding power consumption becomes 202.7 mW ($26*1.36^2 + 31*1.36^2 +$

$33*1.36^2 + 30*1.1^2$). The additional power saving seen in this example might not seem very significant, but when it's computed over thousands of frames with more voltage difference between them, the final result is expected to be of a more significant value.

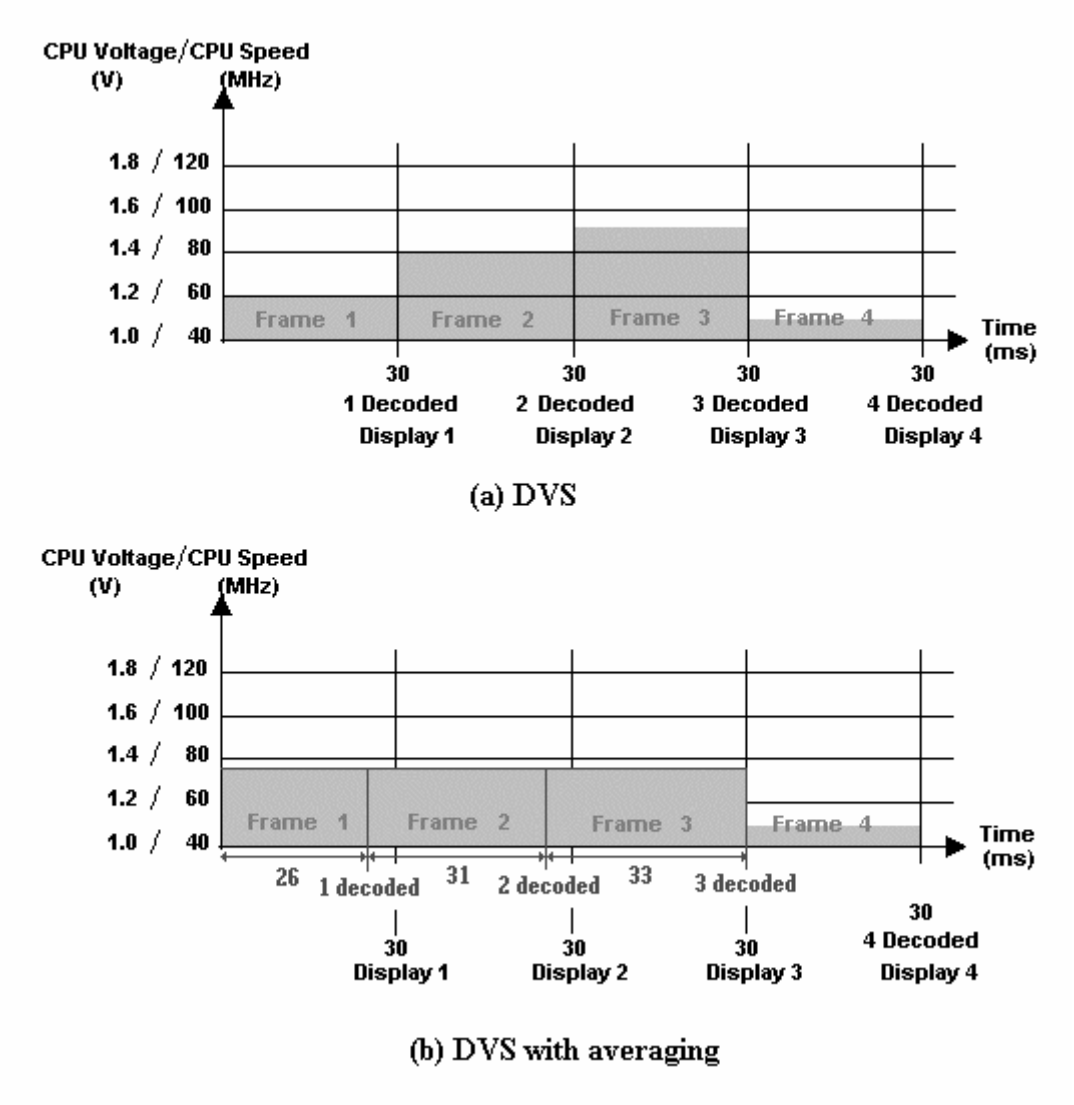


Figure 14: Voltage averaging.

Note that averaging is only applicable when a smaller voltage is followed by one or more higher voltage. And the reason is that when using the average voltage, the frame with originally smaller voltage (frame 1 in Figure 14) will finish earlier than its deadline

leaving some extra time for the following frame (frames 2 and 3) to use in order to decode on time. The frame with originally higher voltage than the average (frames 2 and 3) will always require, after averaging, more time decoding and the extra time is only useful when it comes from a previous frame.

From this observation we can predict that this averaging scheme will also affect the QoS as well. When decoding an MPEG stream, B frames, with usually smaller decoding time, will have there voltage averaged up while the following P or I frame will be averaged down. The extra time provided by the early decoding of the B frame will be used for decoding the next P or I frame, creating a sort of a dependency between the frames. The illustration in Figure 14 assumes no inaccuracy in predicting the decoding time and applying DVS, which is why all the frames are shown to finish on or before display time. But in a real situation, inaccuracies will occur, and error in estimation will cause frames to miss their deadline. In addition to that, and because of the dependency created by voltage averaging, any miss-prediction of the B frame will automatically affect the decoding of the following P or I frame. An underestimation on decoding the B frame will be spelled out as less time to be used by the next P or I frame, while an overestimation will provide even more time for the next frame to use. Therefore, this voltage averaging technique is expected to have a good effect on B frames decoding (less be frames dropped), and a mixed effect on P and I frames (almost the same frame drop is expected).

In order to apply voltage averaging, continuous decoding must be used, i.e. when a frame is decoded, the decoding of the second frame should start right away rather than waiting for the first one to be displayed. The first frame is buffered, waiting for its

display time. The MPEG player should be modified in order to support this feature. In addition to that, knowledge of the frame size and type of the following frames is also required: in order to average the voltage over consecutive frames (frames 1, 2 and 3 of Figure 14), the required voltage for each one of them should be known by the time we start decoding the first frame in the set (frame 1). In our case, we assumed knowledge of all the frames in the whole clip, and applied averaging accordingly. In practice, this information could be included in the header of the first frame of each GOP, so the voltage estimation then averaging for all the frames in the GOP is computed before starting the decoding of the first frame.

4.3 Implementation of the Proposed Algorithms

For all the algorithms proposed in this thesis, the frame size is the most important factor that decides the clock cycle of decoding a frame. A frame size can be estimated by parsing operation before decoding that frame [40], but this will add more estimation errors and delays leading to additional frame drop and power consumption. In this thesis we introduce a more accurate technique providing the exact frame size to the decoder and therefore a more accurate estimation can be made. The size information is not included in the standard MPEG-2 file specification (ISO/IEC 13818-2, [47]), but this standard offers unused fields in the '*picture header*' field (frame header) where the frame size information can be inserted by the encoder without changing the standard specification. The structure for the '*Picture Header*' field is as follows:

byte 0	**byte 1**	**byte 2**	**byte 3**
0000 0000 0000 0000 0000 0001 Start code prefix			Stream ID 0x00

byte 4								byte 5								byte 6								byte 7							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Temperal sequence number								frame type 1=I, 2=P 3=B, 4=D								VBV delay								---							

Additional fields are appended beginning at byte 7 bit 2:

If frame type = 2 (P) or 3 (B) the following 4 bits are appended to the header:

3				2	1	0
Full_pel_forward_vector				forward_f_code		

Plus, if frame type = 3 (B) the following 4 bits are appended to the header:

3				2	1	0
Full_pel_backward_vector				backward_f_code		

Additionally if the next bit is "1" (*extra_bit_picture*=1) it is followed by 8 bits of "extra" data or *extra_information_picture* (discarded by decoders). This continues until a "0" (*extra_bit_picture*=0) bit is encountered.

8	7	6	5	4	3	2	1	0
Extra_bit_picture=1	Extra_information_picture							

This means that we could insert the frame size information, in the *extra_information_picture* field. This field will be discarded by standard MPEG decoders but will be used by Power Aware MPEG decoders using the frame size to predict the decoding time of each frame.

CHAPTER V

PERFORMANCE EVALUATION

This chapter describes the experimental frameworks of our study and presents the simulation results. In section 5.1 we briefly overview the implementation setup by describing the system framework based on the *SimpleScalar* performance simulator, our MPEG power estimator and MPEG QoS estimator, and the modified MPEG decoder. Section 5.2 shows the simulation results using the MPEG video streams in section 3.1.4, comparing the different DVS schemes introduced in Chapter IV.

5.1 System Framework

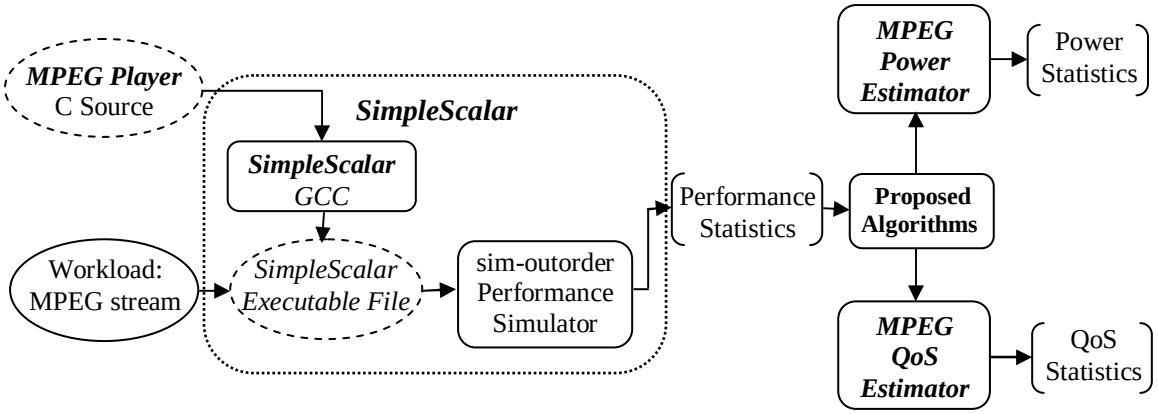


Figure 15: Experimental framework.

Figure 15 shows our simulation environment, which consists of modified *SimpleScalar* [18], our *MPEG power estimator* and *MPEG QoS estimator,* and the modified *Berkley MPEG Player* [48].

SimpleScalar

SimpleScalar is a tool set that performs fast, flexible, and accurate simulation of modern processors [18]. It consists of compiler, assembler, linker, simulation and visualization tools for the *SimpleScalar* architecture, which uses a close derivative of the MIPS Instruction Set.

There are six simulators, which differ in their speed and the details of the results: *sim-fast* and *sim-safe* are the two fastest simulators and the two simplest. They only do functional simulation. *Sim-safe* is a safer version of *sim-fast*, which also checks for correct alignment and access permissions for memory references. *Sim-cache* and *sim-cheetah* allow functional cache simulation with fully configurable caches (instruction and data, first and second level, associativity, etc.). They mainly differ in their cache simulation engines. These simulators are ideal for fast simulation of caches if the effect of cache performance on execution time is not needed. *Sim-profile* is a functional simulator able to produce varied profile information. It can generate detailed profiles on instruction classes and addresses, text symbols, memory accesses, branches and data segment symbols.

In this thesis, we use *sim-outorder*, which is the most complicated and detailed simulator supporting out-of-order issue and execution. *Sim-outorder* can, for example, generate cycle-by-cycle pipeline trace. It also allows a detailed configuration including

the number of ALUs (integer and floating point), RUU (Register Update Unit) capacity, cache and memory latency, memory bus width, branch predictor model (taken, not taken, perfect, bimodal, 2-level adaptive), etc. It implements a five-stage pipeline architecture which consists of the fetch stage IF, the instruction decode stage ID, the execution stage EXE, the memory access stage MEM, and the write back stage WB. Using the *SimpleScalar* GCC compiler included in the toolset we can test any source code. The compiler option can be used for software optimization. The source file to the compiler is a regular C program (e.g. the MPEG player C source code in our case). The compiled file is a *SimpleScalar* executable program that could run on the *sim-outorder* simulator. By the end of the simulation we can obtain many performance statistics of the program executed, e.g. number of instruction, IPC, cache miss rate, etc.

For our research, the simulator proxy system call handler was modified and two system calls for handling each frame decoding time were added: *start_decode* and *finish_decode* (Figure 16(b)). Therefore, before processing a frame, the MPEG player generates the *start_decode* system call, prompting the simulator to save relevant information from the current frame like frame number, frame type and decoding start time. When the decoding process completes, *finish_decode* system call is made, allowing the simulator to compute the decoding time, or more precisely the number of cycles required to decode the current frame. This process is repeated for every frame in the clip, providing, by the end of the simulation, a frame-by-frame detailed statistics. This information is used, later on, to check whether the frame missed its playout time due to miss-prediction, as well as to estimate the power consumption for each stream.

MPEG Player

For video decoding, the Berkeley MPEG decoder [48] was modified to generate the system calls described earlier (Figure 16(a)). Through these calls, the decoder communicates frame-related information to the simulator, which, by matching it to its system-related knowledge, will create a detailed and accurate profiling of the whole decoding process.

Some assumptions regarding this process were made. First, we suppose a continuous decoding is used: when a frame is overestimated and decoded before its display time, it will be buffered and the decoding of the following frame starts immediately. To avoid memory exhaustion, only one frame is buffered at a time. Another assumption is that the overhead of processor scaling is negligible. In practice this scaling overhead is significantly smaller than the granularity in which the DVS calls are made; they have negligible effect on the overall results.

a) MPEG decoder
1- New frame
2- Get frame *number* and *type*
3- Generate *start_decode(number, type)* system call
4- Decode frame
5- Generate *finish_decode* system call

b) Simulator
1- New system call
2- if system call = *start_decode(number, type)*
   then save frame *number*, *type* and *start_cycle* = *current_execution_cycle_number*
3- if system call = *finish_decode*
   then *decoding_time_in_cycles* = *current_execution_cycle_number – start_cycle*

Figure 16: System calls (a) generation (decoder), and (b) handling (simulator).

MPEG QoS Estimator

The main Quality of Service (QoS) metric for the video decoding process is the frames miss rate. When a frame misses its display time, it is dropped and the perceptual quality of the video will be reduced. We developed a simple *MPEG QoS estimator*, which is based on the *SimpleScalar* performance statistics and computes the number of frames dropped due to miss-prediction. . This estimator is simple, relatively accurate and very fast; we only need to run the time-consuming *SimpleScalar* simulator once for each movie and obtain the actual number of cycles required for each frame (*cycle_actual*). The *MPEG QoS estimator* applies the proposed prediction algorithms to process this information and estimates the number of cycles (*cycle_estimate*) needed to decode each frame based on the frame size and type. It will produce the miss rate by using the following simple rule: for any frame *i*, if

$$cycle\_estimate_i < cycle\_actual_i$$

then it is underestimated and will be dropped, else the frame will be displayed. This is because the frequency ($frequency\_estimate_i$) assigned to decode $cycle\_estimate_i$ cycles in the limited fixed time (e.g 30ms) will be lower than the actual frequency needed and thus the time required to decode $cycle\_actual_i$ cycles using $frequency\_estimate_i$ will be greater than the allowed time and the frame will miss its deadline and will be dropped.

MPEG Power Estimator

We developed a simple and fast power estimator in order to obtain enough accuracy to compare the power consumption of the different algorithms studied in Chapter IV. Figure 17 details the *MPEG Power estimator* algorithm.

For each frame *i,*

1- Get $cycle\_estimate_i$, $frequency\_estimate_i$ and $voltage\_estimate_i$.

2- Get $cycle\_actual_i$

3- Compute $actual\_time_i = cycle\_actual_i * frequency\_estimate_i$

4- if $actual\_time_i > deadline\_time$ then $actual\_time_i = deadline\_time$

5- Compute $Power_i = actual\_time_i * voltage\_estimate_i^2$

When all frames are decoded, compute overall estimated power: $Power = \sum power_i$

Figure 17: MPEG power estimator algorithm.

From the estimation algorithms in Chapter IV, we know for each frame *i* the estimated number of cycles ($cycle\_estimate_i$), the estimated frequency ($frequency\_estimate_i$) and voltage ($voltage\_estimate_i$); and from the *SimpleScalar* performance statistics we know the actual number of cycles ($cycle\_actual_i$). The actual time spent decoding the frame ($actual\_time_i$) is the product of the used estimated frequency and the actual number of cycles. The estimated power ($power_i$) for each frame is proportional to the estimated voltage squared times the actual decoding time.

For the power estimation above we assume that the voltage is exactly proportional to the operating frequency. This is a reasonable assumption except when the voltage approaches the threshold value. The second assumption is that voltage and frequency can take any real value, which is not true for real-life systems where only a finite discrete number of levels are available. The effect of the discrete levels will be an increase in the power consumption and a decrease in the drop rate. Finally, we assume that for an underestimated frame, i.e. when the decoding time exceeds the allowed deadline time (*deadline_time*, e.g. 30 ms), the frame will be dropped just before reaching the display

time, and therefore the whole frame will not be decoded. This assumption is illustrated in Figure 16, line 4.

5.2 Simulation Results

In this section, we present the experimental results. We first show the benefit of employing these algorithms in terms of power consumption (Section 5.2.1), and then present the QoS performance (Section 5.2.2).

5.2.1 Power Consumption

Figure 18 shows the relative power consumption of the proposed schemes in comparison with the shutdown (On/Off) algorithm. Note that the *ideal DVS* shown in this figure is the normal DVS algorithm with an ideal estimator where all the frames are decoded just on time for display: neither over- nor under-estimation in decoding, providing an ideal QoS performance.

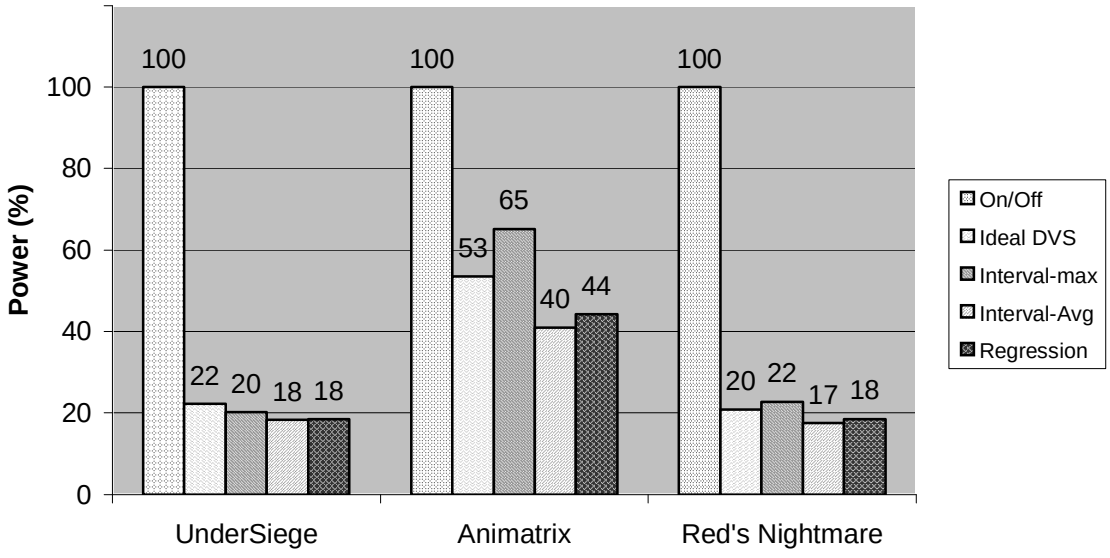


Figure 18: Power consumption.

All three algorithms performed much better than the shutdown algorithm and saved additional 35% to 83%. As expected, the *interval-avg* based algorithm consumes the least energy among all the others, followed very closely by the *regression* algorithm, while the *interval-max* consumed the most energy; and that's because the first two algorithms try to get an accurate estimation of the ideal voltage while the last algorithm only tries to find an upper-bound of the required voltage in each interval. Note also that in most cases the proposed algorithms performed even better than the *ideal DVS* and the main reason behind that is the voltage averaging technique, used on top of the other estimation techniques (but not in the *ideal DVS* case), which provides additional power savings. Figure 19 shows the extra power saving due to the averaging technique as compared to using only the estimation techniques. (*Interval-max* was used as the estimation technique in Figure 19. Similar results were obtained when using the two other techniques.)

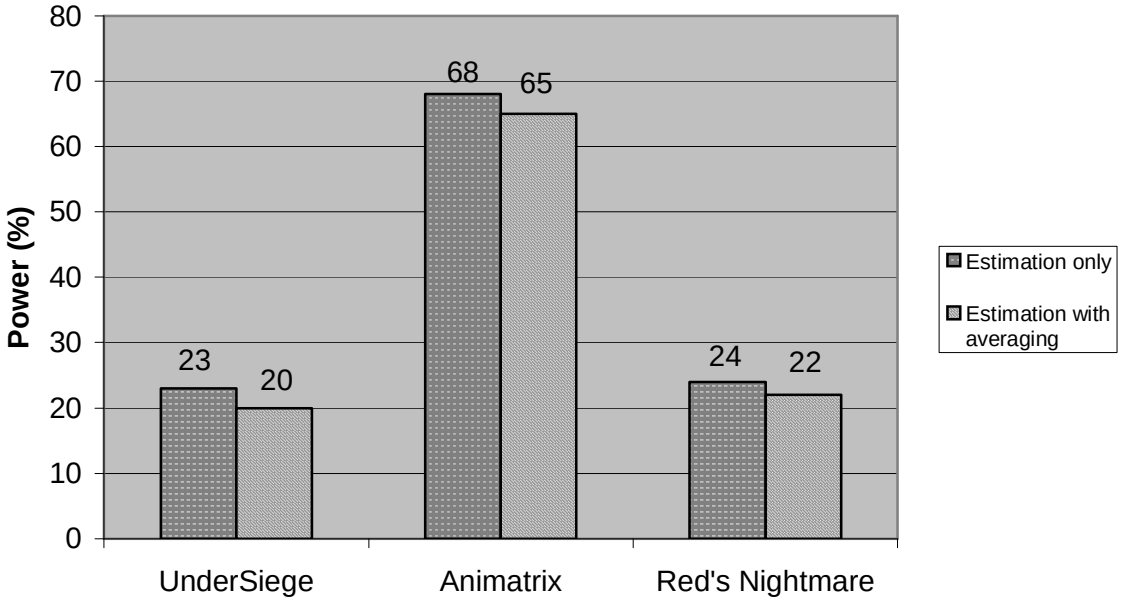
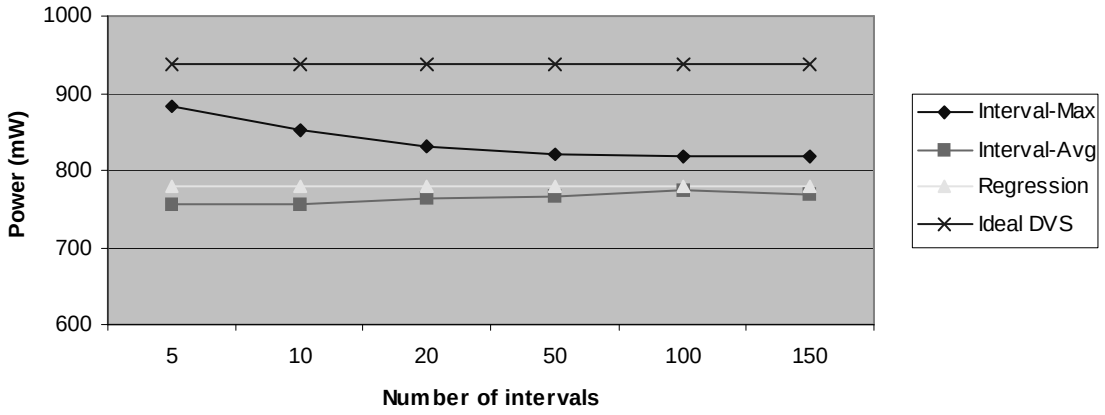


Figure 19: Voltage averaging effect on Power.

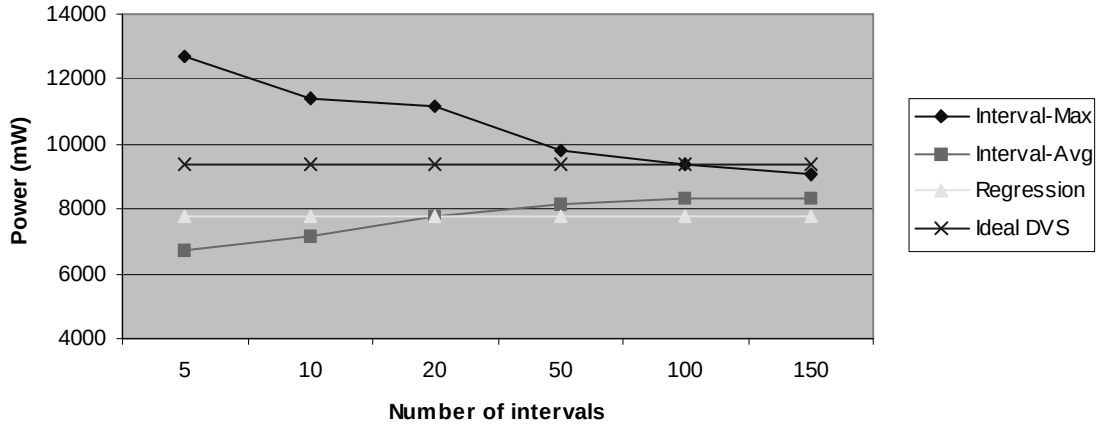
Finally Figure 20 shows the effect on the power consumption when varying the number of interval or resolution for these two protocols, *interval-max* and *interval-avg*. As expected, for *interval-max*, the more intervals we have the more power saving we can achieve; the reason is that more intervals means smaller ones and therefore the upper-bound of each interval, which is used by this algorithm, will be closer and closer to the average, leading to less power consumption. Different results were obtained for *interval-avg*. With more intervals, more power levels will be used and therefore the effect of the voltage averaging technique, mainly based on the level differences, will be masked leading to slight increase in power consumption. This 'power levels' effect was more clear for the *interval-avg* technique and almost negligible for *interval-max*, because in the later algorithm the 'decreasing upper-bound' effect was more significant and produced more power saving.

5.2.2 QoS

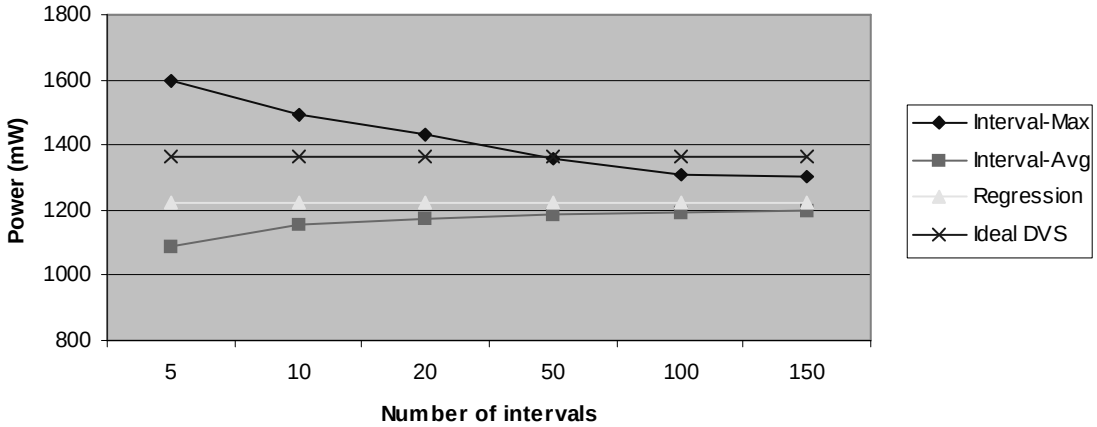
While all three algorithms have almost the same improvement on energy efficiency of MPEG decoding, they have different impact on the quality of service (QoS) in playing a real-time MPEG stream. Figure 21 shows the experimental results of the number of dropped frames to the total number of frames, which represents the QoS of MPEG decoding. As we can see, the *interval-avg* technique performed the worst, followed by *regression*. *Interval-max* provided, by far, the best QoS. The reason behind these results is the inherent tendency in the last algorithm to overestimate the voltage requirement, causing more energy to be spent but save more frames from being dropped, while the first two algorithms tend to average their estimation, causing less power to be



(a) *UnderSiege* clip



(b) *Animatrix* clip



(c) *Red's Nightmare* clip

Figure 20: Interval effect on power.

consumed but more frames to be dropped. The difference seen between different movie clips is mainly due to the difference in their number of cycle vs. frame size distribution and linear model representation.

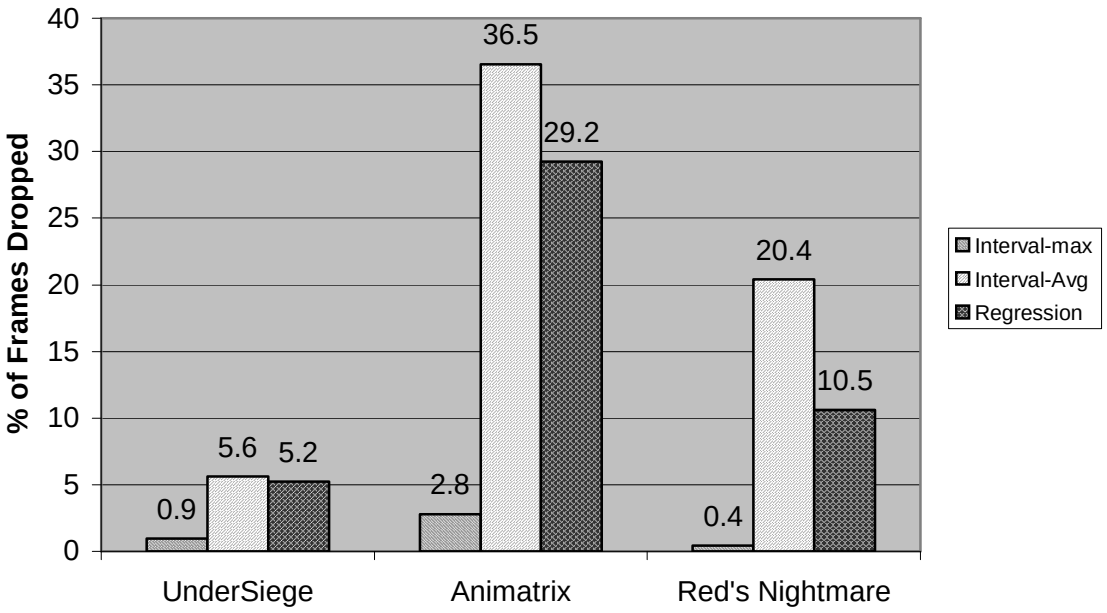


Figure 21: QoS or ratio of dropped frames to the total number of frames.

In the next Figure 22, we show the effect of the averaging technique on the QoS, or the number of frames dropped. As predicted in Chapter IV, the averaging effect on QoS is as follows: less B frames are dropped and almost the same drop rate for P and I frames. And the reason behind that is the functioning of the averaging algorithm that tend to average the power between consecutive frames giving that the following frame requires more voltage than the previous one, which is most of the time the case with B frames followed by a P or I frame. By averaging the voltage, the B frame would receive more power than required and will finish way before the display time, which will save it from being dropped even if it was underestimated by the estimation algorithm.

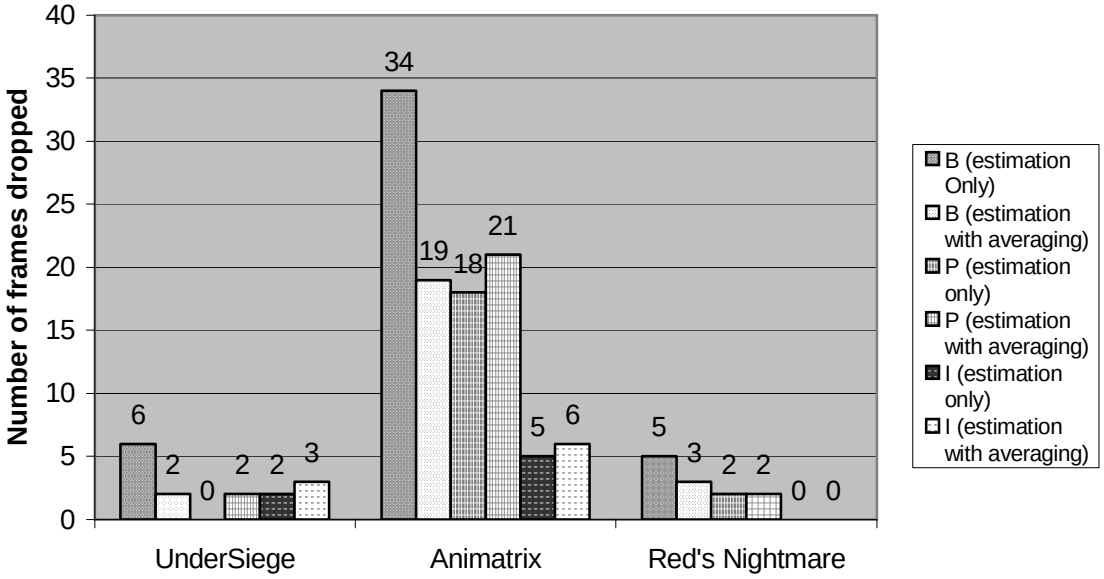


Figure 22: Voltage averaging effect on QoS.

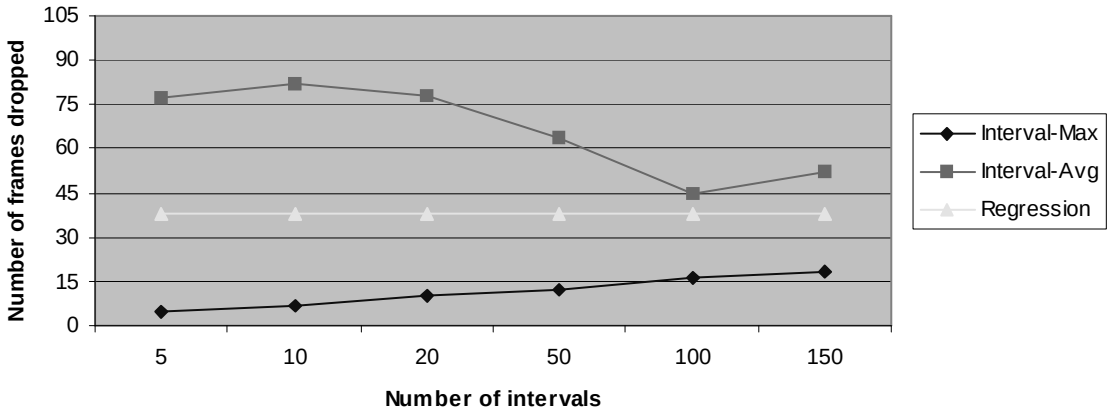
The case is different for the averaged P or I frames which will receive less power than normally needed but will also receive the extra time left from the previous B frame which will lead to a more complicated and unpredictable situation depending on different factors: the over/under estimation of both the previous B frame and/or the current P or I frames, which will cause the last frame to be dropped sometimes and be saved some other times, leading to an almost unchanged drop rate for P and I frames. Overall, we can conclude that the used voltage averaging technique will help improve the QoS by reducing the drop rate. (*interval-max* was used as the estimation technique in Figure 22, similar results were obtained when using the two other techniques)

Finally Figure 23 shows the effect on the QoS performance when varying the number of interval. As expected, for *interval-max*, the more intervals we have more frames will be dropped; the reason is that more intervals means smaller ones and therefore the upper-bound of each interval, which is used by this algorithm, will be

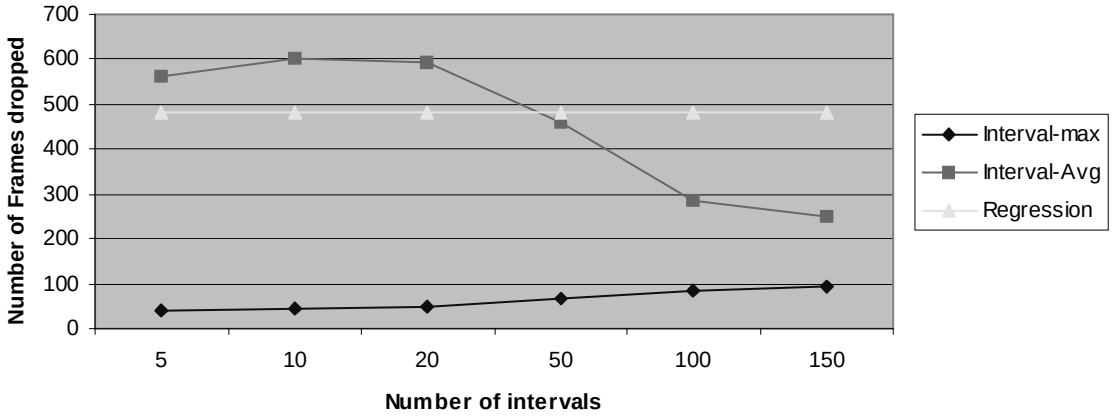
smaller, and thus fewer frames will be in each interval causing more frames to be dropped before finding the upper limit because of the iterative approach of the *interval-max* algorithm. Different results were obtained for *interval-avg*. As the number of intervals increased, their size decreased, and the average of each interval became more representative for the interval causing fewer frames to be dropped because better estimated. Although the QoS evolvement for different resolutions is more fluctuating than in the *interval-max* case, the general trend for the *interval-avg* technique shows a QoS improvement when more intervals are used.

These final observations along with results from the previous section (5.2.1) show that using different number of intervals makes it possible to respond to different constraints: an 'optimum' resolution could be used to provide a certain minimum allowable QoS and/or to consume a maximum available power. This 'optimum' resolution is computed dynamically, based on the available constraints, and will be the study of future works. As for this study, only one constraint could be satisfied at a time by choosing the appropriate resolution once and for all to maximize the QoS or minimize power consumption.

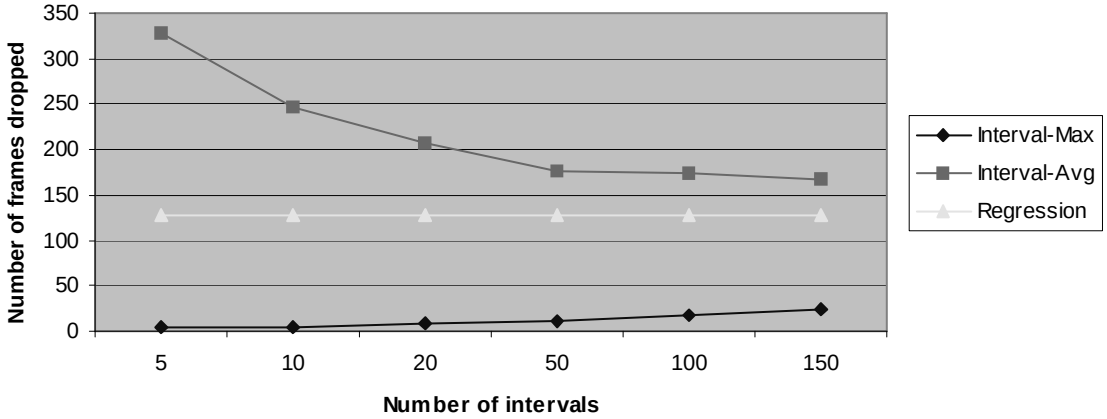
From the power and QoS analysis presented in this section we can conclude that the *interval-max* algorithm, although the simplest to implement, looks to be the most promising: it provides the best QoS by far with only small increase in power consumption as compared to the other algorithms, plus it is so predictable when using different intervals to respond to different constraints making the best candidate for any future work researching the 'optimum' resolution which will dynamically adjust to any outside variable constraint as explained in the previous paragraph.



(a) *UnderSiege* clip.



(b) *Animatrix* clip.



(c) *Red's Nightmare* clip

Figure 23: Interval effect on QoS.

CHAPTER VI

CONCLUSION

Due to the variability of the workload of real-time applications such as MPEG decoding, applying dynamic voltage scaling techniques is not straightforward. The main problem is the miss-prediction of the next workload, causing a frame to be dropped or sub-optimum power to be consumed.

In this thesis, we proposed and compared three DVS estimation techniques for power aware video decoding: *regression* (using linear regression models for estimation), *interval-max* and *interval-avg* (dividing the decoding-time/frame-size distribution into intervals and making estimation decision based on the interval maximum decoding time, for the first algorithm, or on the average decoding time of each interval, for the second one). In addition to these estimation algorithms, a *voltage averaging* technique was also proposed and applied on top of the previous ones, aiming mainly to improve the power consumption. We compared these algorithms using three sample streams and found that, with respect to energy consumption, all three of them performed much better compared to the conventional shutdown technique (35% to 85% additional power saving). And,

mainly because of the *voltage averaging* technique, the power consumption was most of the times better than even that of the ideal DVS, with no estimation inaccuracies and no voltage averaging. *Regression* and *interval-avg*, and due to their relatively accurate estimation, had the best power saving. On the other hand, and with respect to the QoS or the frame drop rate, *interval-max* performed the best (0.4% to 2.8% of frames dropped), mainly because of its not so accurate but "safe" estimation technique.

In addition to that, we also studied the impact of varying the interval-size parameter in both *interval-max* and *interval-avg* algorithms and found that decreasing the interval size also decreased the power consumption but increased the drop rate for *interval-max*, while an opposite effect was seen for *interval-avg*: more power consumption and less frames dropped. This effect was found to be clearer for *interval-max* and somewhat fluctuating for *interval-avg*, making the first algorithm more suitable for further investigations in this direction. For this study, the interval size was static and determined once for the whole duration of the decoding process, so as a future work, it would be interesting to research how this parameter could be changed, dynamically, in order to respond to any real-time constraints like serving a better QoS or a lower power consumption, or finding an optimum solution for any given situation.

Finding more accurate prediction mechanisms and new ways to exploit DVS for low power video decoding is always critical, and opportunities for improvement still exists, especially in the less studied area where DVS could be also used on other parts of the system, such as the memory or network interface. In addition, more future work would be in investigating the usage of DVS systems on streaming video, especially for mobile devices where additional constraints like packets drop and delay must be taken

into consideration, requiring a more complicated study of all the factors, and maybe a totally different approach to the subject. It might also be interesting to find a way to combine these DVS techniques with power aware routing protocols and study the resulting benefits and drawbacks on processing streaming video in a mobile environment.

BIBLIOGRAPHY

[1] D.Brooks, P.Bose, S.Schuster, H.Jacobson, P.Kudva, A.Buyuktosunoglu, J.Wellman, V.Zyuban, M.Gupta and P.Cook, "**Power Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors**," *IEEE Micron*, pp. 26-44, December 2000.

[2] D.Brooks, V.Tiwari, and M.Martonosi, "**Wattch: A Framework for Architectural-Level Power Analysis and Optimizations**," *In Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[3] W.Ye, N.Vijaykrishan, M.Kandemir, and M.J.Irwin, "**The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool**," *In Proceedings of the Design Automation Conference*, June 2000.

[4] V.Tiwari, S.Malik, A.Wolfe, and M.T.Lee "**Instruction Level Power Analysis and Optimization of Software**," *J. VLSI Signal Processing*, August 1996.

[5] O.A.Patino and M.Jimenez, "**Instruction Level Power Profile for the PowerPC Microprocessor**," *Computing Research Conference 2003, UPR Mayagüez*, pp. 120-123, April 2003

[6] T.Cignetti, K.Komarov and C.Ellis, "**Energy Estimation Tools for the Palm™**," *In Proceedings of the ACM MSWiM'2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.

[7] J.Flinn and M.Satyanarayanan. "**PowerScope: A tool for Profiling the Energy Usage of Mobile Applications,**" *In Proceedings of the Workshop on Mobile Computing Systems and Applications (WMCSA)*, February 1999.

[8] F.Chang, K.Farkas and P.Ranganathan, **"Energy-driven Statistical Profiling: Detecting Software Hotspots,"** *In Proceedings of the Workshop on Power Aware Computing Systems*, February 2002.

[9] S.Gurumurthi, A.Sivasubramaniam, M.J.Irwin, N.Vijaykrishnan, and M.Kandemir, "**Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach**," *In the Proceedings of the International Symposium on High Performance Computer Architecture (HPCA-8)*, February 2002.

[10] M.Weiser, B.Welch, A.Demers and S.Shenker, "**Scheduling for Reduced CPU Energy**," *Usenix Association*, November 1994.

[11] K.Govil, E.Chan and H.Wasserman, "**Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU**," *MobiCom* 1995.

[12] G.Quan and X.Hu, "**Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors**," *Design Automation Conference*, 2001.

[13] D.Son, C.Yu and H.Kim, "**Dynamic Voltage Scaling on MPEG Decoding**," *International Conference on Parallel and Distributed Systems (ICPADS)*, 2001.

[14] T.Pering, T.Burd and R.Brodersen, "**Dynamic Voltage Scaling and the Design of a Low-Power Microprossor System**," *Workshop on Low-Power Microprocessors*, 1998.

[15] Y.Lu and G.De Micheli, "**Comparing System-Level Power Management Policies**," *IEEE Design & Test of Computers*, pp. 10-19, March 2001.

[16] A.Sinha and A.Chandrakasan, "**Dynamic Power Management in Wireless Sensor Networks**," *IEEE Design & Test of Computers*, pp. 62-74, March 2001.

[17] E.N.Elnozahy, M.Kistler and R.Rajamony, **"Energy-Efficient Server Clusters,"** *In Proceedings of the Second Workshop on Power Aware Computing Systems*,Feb 2002

[18] D.Burger and T.Austin, "**The SimpleScalar Tool Set, Version 2**," *Tech. Report No. 1342, Computer Sciences Dept. Univ. of Wisconsin*, June1997.

[19] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and W. Ye, "**Influence of Compiler Optimizations on System Power**," *In Design Automation Conference*, pp. 304-307, 2000.

[20] H.S. Kim, M.J. Irwin, N. Vijaykrishnan, and M. Kandemir, "**Effect of Compiler Optimizations on Memory Energy**," *In IEEE Workshop on Signal Processing Systems*, pp. 663-672, 2000.

[21] M. Kandemir, N. Vijaykhrishnan, M. Irwin, "**Compiler Optimizations for Low Power Systems**," *In Power Aware Computing textbook*, pp. 191-210, 2002.

[22] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura, "**Instruction Scheduling for Power Reduction in Processor-Based System Design**," *In Proc. of Design Automation and Test in Europe (DATE98)*, pp. 855-860, 1998.

[23] C.L. Su, C.Y. Tsui, and A.M. Despain, "**Low Power Architecture Design and Compilation Techniques for High-Performance Processors**," *COMPCON'94*, 1994.

[24] C.H. Gebotys, "**Low Energy Memory and Register Allocation Using Network Flow**," *DAC'97*, pp. 435-440, 1997.

[25] S. Steinke, R. Schwarz, L. Wehmeyer, and P. Marwedel, "**Low Power Code Generation for a RISC Processor by Register Pipelining**," *Technical Report No. 754, University of Dortmund, Dept. of CS XII*, 2001.

[26] J.L. Ayala and A. Veidenbaum, "**Reducing Register File Energy Consumption using Compiler Support**," *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, The Hague (Netherlands), 2003.

[27] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum, "**Architectural and Compiler Strategies for Dynamic Power Management in the COPPER Project**," *IWIA 2001 International Workshop on Innovative Architecture*, Maui, Hawaii, 2001.

[28] A. Azevedo, R. Cornea, I. Issenin, R. Gupta, N. Dutt, A. Nicolau, and A. Veidenbaum, "**Profile-based Dynamic Voltage Scheduling using Program Checkpoints**," *In Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, 2002.

[29] N. AbouGHazaleh, D. Mosse, B. Childers, R. Melhem, and M. Craven, "**Collaborative Operating System and Compiler Power Management for Real-Time Applications**," *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[30] C.H. Hsu and U. Kremer, "**Compiler-directed Dynamic Voltage Scaling for Memory-Bound Applications**," *Technical Report DCS-TR498, Department of Computer Science, Rutgers University*, 2002.

[31] D. Shin and J. Kim, "**Intra-Task Voltage Scheduling for Low-Energy Hard Real-Time Applications**," *IEEE Design & Test of Computers*, vol. 18, pp. 20-30, 2001.

[32] F. Xie, M. Martonosi, and S. Malik, "**Compile-time Dynamic Voltage Scaling Settings: opportunities and Limits**," *In Asia South Pacific Design Automation Conference (ASP-DAC'01)*, 2001.

[33] T. Burd and R. Brodersen, "**Energy Efficient CMOS Microprocessor Design**," *28th Hawaii Int'l Conf. on System Science*, Vol. 1, pp. 288-297, Jan. 1995.

[34] T. Burd and R. Brodersen, "**Design Issues for Dynamic Voltage Scaling**," *Int'l Symp. on Low power Electronics and Design*, pp. 9-14, 2000.

[35] W. Namgoong, M. Yu, and T. Meng, "**A High-Efficiency Variable-Voltage CMOS Dynamic DC-DC Switching Regulator**," *IEEE Int'l Solid-State Circuits Conf.*, pp. 380-381, 1997.

[36] K. Suzuki, et al., "**A 300 MIPS/W RISC Core Processor with Variable Supply-Voltage Scheme in Variable Threshold-Voltage CMOS**," *IEEE Custom Integrated Circuits Conf.*, pp. 587-590, 1997.

[37] K. Govil, E. Chan, and H. Wasserman, "**Comparing Algorithms for Dynamic Speed-Setting of a Low-Power CPU**," *Technical Report TR-95-017, ICSI Berkeley*, April 1995.

[38] E. Nurvitadhi, B. Lee, C. Yu and M. Kim, "**A comparative Study of Dynamic Voltage Scaling Techniques for Low-Power Video Decoding**," *The International Symposium on Low Power Electronics and Design (ISLPED)*, 2003.

[39] K. Choi, W. Cheng and M. Pedram, "**Frame-Based Dynamic Voltage and Frequency Scaling for an MPEG2 Player**," *Proceedings of International Conference on Computer Aided Design*, 2002.

[40] A. Bavier, A. Montz and L. Peterson, "**Predicting MPEG Execution Times**," *Proceedings of SIGMETRICS '98/PERFORMANCE '98*, 1998.

[41] J. Mitchell, W.Pennebaker, "**MPEG Video Compression Standard**," *Chapman and Hall*, 1996.

[42] "**Mobile Intel PentiumIII Processor**," *in BGA2 and Micro-PGA2 Packages Datasheet, Intel Corporation.*

[43] M. Fleischmann, "**LongRun Power Management**," *Transmeta Corporation*, 2001.

[44] "**MPEG-2: The Basics of how it Works**," *Hewlett Packard Lab.*

[45] "**ISO: International Organization for Standardization**," http://www.iso.ch

[46] "**IEC: International Electrotechnical Commission**," http://www.iec.ch

[47] "**MPEG: Moving Picture Expert Group**," http://mpeg.telecomitalialab.com

[48] Berkley MPEG Tools. http://bmrc.berkley.edu/frame/research/mpeg/