Power Minimization under QoS Constraints

Jennifer L. Wong[†], Gang Qu[‡] and Miodrag Potkonjak[†]
[†]Computer Science Department, University of California, Los Angeles, CA 90095
[‡]Electrical and Computer Engineering Department,
University of Maryland, College Park, MD 20742

Abstract

QoS has been often addressed in multimedia, video, and networking research communities, but rarely in the design community. Our goal is to introduce the first system design technique for comprehensive quality-of-service (QoS) low power synthesis. Specifically, we study how to efficiently exploit the trade-off between the system cost and energy consumption in real-time systems that address packet-based multimedia transmission and processing. We first introduce a system of techniques that minimizes energy consumption of stream-oriented applications under two main QoS metrics: latency and synchronization. Specifically, we study how multiple voltages can be used to simultaneously satisfy hardware requirements and minimize power consumption, while preserving the requested level of QoS, in this case satisfying latency and synchronization requirements.

We have developed a provably optimal polynomial time off-line algorithm for multiple voltage scheduling of single and multiple processes. The off-line algorithm provides lower bounds on achievable power minimization and can be used as a starting point for the development and evaluation of an on-line approach. The effectiveness of the algorithm is demonstrated on a number of multimedia benchmarks.

1 Introduction

In the last decade, low power synthesis and optimization techniques have received a great deal of attention. A variety of techniques have been proposed for all steps of synthesis and compilation. The combination of new logic families and circuits, smaller feature sizes, more power efficient architectures, power-aware CAD tools, power-sensitive compilers and low power dynamic run-time policies resulted in a dramatic increase in energy efficiency. However, the power requirements of new product generations has been constantly challenging the limits of battery capacities. An illustrative example of the technology or application power trends is the evolution of the wireless phone. The first generation of wireless phones is analog, the second is digital, which is currently prevailing. A mezzanine generation of wireless phones with microbrowsers have just emerged. The imminently pending third generation includes powerful Internet access. After that the next generation will include numerous new features encompassing streaming media. Laptops with wireless modems already provide this type of service. The analysis of communication and digital signal processing requirements indicates that each wireless phone generation increases computational requirements by at least two orders of magnitude. Therefore, a need for new power minimization techniques has been constant in wireless communications.

The most popular mobile low power applications, such as audio and video, are stream-oriented. The nature of these applications imposes a need for addressing QoS requirements under energy constraints. Until now, this problem has not been addressed. Our goal is to develop a spectrum of

techniques and algorithms which minimize energy consumption under the most important streaming media QoS metrics, latency and synchronization. Specifically, we study how to use multiple voltage technologies to simultaneously satisfy hardware requirements (storage space and processing speed) and minimize the power consumption, while preserving the requested level of QoS in terms of latency and synchronization. The main result of the paper is a provably polynomial time off-line algorithms for multiple voltage scheduling of single and multiple processes. The algorithm orders and assigns packets of streaming media in such a way that energy consumption is minimized, while storage requirements are satisfied. The algorithm is dynamic programming-based and can be used for compile time scheduling of movies and audio or as starting point for the development of on-line algorithms for the same task.

2 Related Work

Our research results can be viewed in the context of two areas of related work: low power modeling and optimization, and quality of service.

Mainly due to a need for mobile applications, in the last decade low power research has attracted a great deal of attention. Both power modeling and optimization have been addressed on many levels of the synthesis process [12]. More recently, a number of researchers proposed the use of multiple voltages in order to reduce power consumption [5, 13, 16]. Furthermore, several variable voltage techniques have been reported [4, 8, 17]. Also, several industrial multiple voltage low power designs have been reported. The common denominator in all the efforts has been that the operations on the critical path are scheduled at higher voltage and therefore are executed faster and other operations are scheduled on a lower voltage and therefore the energy consumption is reduced. Another popular approach to power minimization is dynamic power management which aims to reduce the power consumption of electronic systems by selectively shutting down idle components [2, 11]. From one point of view our work can be interpreted as a combination of these two techniques, multiple voltages and system-level power management.

The first QoS requirements, such as bounded delay, guaranteed resolution or synchronization have been conducted in the network and real-time operating systems (RTOS) communities. The most sound and practically relevant QoS model in the networking community was proposed by R. Cruz [7].The model assumes periodic segmentation of time. During each period each process receives a task of generally varying complexity. The cumulative sum of tasks for a process forms a demand curve imposed on the system. The system serves the task sequentially by allocating resources during each time period to one of the processes. The cumulative sum of the processed data forms a service curve. The main conceptual result in RTOS literature was presented by Rajkumar et al. [14]. They introduced an analytical approach for satisfying multiple QoS dimensions under a given set of resource constraints. They proved that the problem is NP-hard and developed an approximation polynomial algorithm for the problem by transforming it into a mixed integer programming problem [15]. Comprehensive survey of QoS research in these two areas is given in [1]. Recently, the first efforts in QoS, and in particular synchronization during the system design process has been reported in the design automation literature [10].

3 Preliminaries

In order to make the presentation self-sufficient, in this section we outline the used abstraction and models for power consumption, latency, synchronization and context switch overhead. Although we do not use the Demand-Supply model for modeling streaming processes due to the limitations of this model, [6] we provide in detail a summary of the model. There are two main reason for the detailed treatment of this model. The first reason is to provide a suitable framework for defining

key QoS metrics: latency and synchronization. The second is to discover the key trade-off in a very suitable framework.

One of the main components of power consumption is the switching power. The switching power can be modeled as $P = \alpha \cdot C_L \cdot V_{dd}^2 \cdot f$, where $\alpha \cdot C_L$ is the effective switching capacitance. Switching power is the dominating factor in power consumption. The greater throughput comes with the cost of higher voltage. The gate delay of the circuit is defined as $T = k(V_{dd}/(V_{dd} - V_t)^2)$ where k is a constant [3].

We assume the design operates using multiple voltage supplies and that the voltages change instantaneously with no overhead. These changes in voltages are assumed to happen only at the beginning or the end of a time unit. Furthermore, we assume that the voltage units are selected in such a way that the use of two consecutive voltages, $v_i \& v_i$, on two consecutive points is more effective than the use of voltages $v_i \& v_{i+1}$, due to the fact that power as a function of voltage is convex.

The Demand-Supply model for QoS was developed by Cruz et al. [7]. The model addresses the burstiness of QoS while handling resource allocation. This model assumes periodic segmentation of the time dimension. During each period each process receives a task of generally varying complexity. The cumulative sum of tasks for a process can be depicted as a demand curve imposed on the system. The system serves the task sequentially by allocating resources during each time period to one of the processes. The cumulative sum of the processed data forms a supply curve. While the DS-Curve explains many of the QoS metrics, such as latency, backlog, and synchronization, the key problem with this DS-Curve is that its representation of QoS for a small period can be large. The demand curve measures the burstiness of the service requirement. The service curve guides the resource allocation with QoS guarantees. Backlog is defined as the amount of demand that cannot be processed at that time point, and must then be carried over to the next time unit. The backlog in the QoS model is represented by the difference between the vertical positions of the demand curve and the service curve. Latency is the time between when the demand for a task arrives, and when it is processed. This is shown in the model by the horizontal difference between the demand curve and the service curve at any given vertical position.

A process is a program which assumes that it has independent use of the CPU. A process is long, in the sense that it consists of many tasks. These tasks are processed at periodic moments.With each task we associate a processing time and a storage requirement. Note that periodicity is not reducing the generality of our formulation because the tasks of the process could have a requirement of zero. Although in general there is no correlation between processing time and storage, a special case exists when there is a linear relationship. Since this important special case is often of interest, we treat it separately. Finally, note that satisfying all latency implicitly implies that the throughtput requirements are also satisfied.

A context switch is the time overhead which is incurred by a multitasking kernel when it decides to process different tasks. The amount of context switching time dramatically depends on the processor. For a DSP processor the context switch time is fairly low, around 10 cycles, while for a RISC processor it is much higher, approximately 100 cycles. For our off-line algorithm, we assume that there is no context switch time. If we considered the off-line algorithm with context switches it would be an NP-complete problem and therefore would not be possible to solve optimally.

Synchronization is the timing relationship between interacting media, which is one of the most important metrics for QoS, such as jitter and burstiness. The synchronization of two processes can be seen in the DS-Curve (Figure 1). The figure displays two tasks, τ_1 & τ_2, and their corresponding

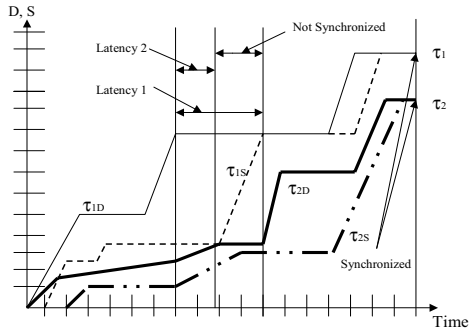

Figure 1: Latency and synchronization in the DS model.

service (dotted lines) and demand curves (solid lines) for the two tasks. We say that τ_1 & τ_2 are synchronized when both τ_1 & τ_2 are serviced at the same time. The latency is the time between when the demand arrives and when it is processed. The amount in which tasks τ_1 & τ_2 are not synchronized is the difference between the latencies.

Note that the DS-Curve model is actually not an accurate abstraction of the media data delivery process [6]. The major limitation is that during delivery the initial periodic nature of tasks can be made either highly dense or very sporadic in short time intervals. Nevertheless, there is an easy way to make the DS-Curve model adequate. Essentially all that is needed to consider all tasks which arrive during our time unit (eg. in the case of MPEG 40ms) as a single task is to create a new tasks which has processing time and storage requirements equal to the sum of the processing times and storage requirements of all tasks which have arrived.

4 Off-line Optimal Algorithm

In this section, we formulate the off-line QoS low power problem and present our optimal algorithm. We have one processor that can operate at multiple supply voltages. The goal is to service multiple processes with minimal energy consumption and the minimal amount of memory while meeting various QoS requirements.

4.1 Problem Formulation

A process consists of a seqence of tasks. With each task t_i, we associate

- a_i: The arrival time, the time when a task is generated from the process and makes the CPU request;
- p_i: The time needed to complete this task at the nominal voltage v_{ref};
- s_i: The storage demand which is the minimal amount of memory to store this task on its arrival.

Tasks may have QoS requirements such as latency and synchronization. Latency (or deadline) d_i is the time that task t_i has to be served after its arrival, that is, the actual finish time of task t_i must be earlier than $a_i + d_i$. Synchronization measures the interaction among tasks in different processes. We say that task t_i from one process and task t_j from another are k-synchronized if the difference of their finishing times is within k CPU units. We denote this by $syn(t_i, t_j) \leq k$.

The variable voltage processor has multiple supply voltages among which it can switch. The processor's processing speed varies as the voltage changes, so will be the actual execution time for a task to receive its required amount of service. Suppose a task needs one CPU unit at the nominal voltage v_{ref}, then the execution time to accumulate the same amount of processing at voltage v_{dd} is given by [3]:

$$\frac{(v_{ref} - v_t)^2}{v_{ref}} \cdot \frac{v_{dd}}{(v_{dd} - v_t)^2} \tag{1}$$

where v_t is the threshold voltage. We consider only the switching power which is proportional to the square of supply voltage.

Given n processes $\tau^1, \tau^2, \cdots, \tau^n$, each τ^k consists of a sequence of tasks $t_1^k, t_2^k, \cdots$. A *schedule* is a set consisting of the starting time, finishing time, and the voltage level for each task. A schedule is *feasible* if the processor starts each task after its arrival, finishes it before the latency constraint, and satisfies all the synchronization requirements. The quality of a schedule is measured by its energy consumption and the memory requirement. Since these two metrics are non-comparable to each other, we introduce the concept of competitiveness. We say two schedules are *competitive* if neither outperforms the other in both energy consumption and memory requirement. We formulate the problem as:

> *On a processor with multiple voltages, for a given set of processes, find all the feasible competitive schedules.*

We make the following assumptions:

1. Tasks in the same process have to be executed and completed in the FIFO fashion;
2. A task's processing demand p_i is proportional to its storage demand s_i;
3. The memory occupied by a task can be partially freed, but only at the end of a CPU unit[1];
4. The processor can instantaneously switch the supply voltage, but only at the beginning of each CPU unit.

We show how to find all feasible competitive solutions for a single process. Suppose the reference voltage $v_{ref} = 0.8v$, and there are two different voltage levels $v_{hi} = 3.3v$ and $v_{lo} = 1.8v$. From equation (1), we approximate the processing speeds are to be 3 and 10 at v_{lo} and v_{hi} respectively. Consider a process with six tasks, $t_0, t_1, \cdots, t_5$. For simplicity, we further assume that task t_i arrives at time i, and they don't have deadline constraints. Finally, we assume that the processing and memeory requirement are **4, 7, 12, 3, 5**, and **1** respectively for the six tasks.

We want to determine the voltage that we will use for each unit time, such that the energy consumption is minimized. We developed a dynamic programming based algorithm to achieve a polynomial run-time.

Table 1 shows the instant memory requirement at the end of each unit time, which is the minimal amount of memory required to store all the arrived but unfinished tasks. The table uses the time (in terms of CPU units) that the processor is operating at v_{lo} and v_{hi} to label the horizontal and vertical axises respectively. For example, entry (i, j) is the minimal memory requirement after running at v_{hi} for i CPU units and at v_{lo} for j units. Obviously this amount depends on when we use v_{lo} and when we raise to v_{hi}. Consider entry (1,1), whose content is the storage we need at the end of the second unit of time after we use v_{lo} for one unit of time and v_{hi} for the other. We can either apply v_{hi} in the first unit and v_{lo} in the second unit, or start at v_{lo} and switch to v_{hi} after one CPU unit. In the first case, since task t_0's processing demand is 4 and we are able to process 10 at v_{hi}, we will finish t_0, free the memory, and wait for t_1; then at v_{lo} in the next unit of time, we can finish 3 out of the 7 units of processing demand from t_1; now t_2 is coming, we need a total of $(7 - 3) + 12 = 16$ units of memory to store t_1 and t_2.

[1]Memory can be partially freed means that, for instance, if half of the processing demand is fulfilled at the end of one CPU unit, then we are able to free half of the space used to store this task. Our proposed algorithm can be easily modified when this is not allowed.

	0	1	2	3	4	5	6	7	8	9	10	11
0	4	8	17	17	19	17	14	11	8	5	2	0
1	7	12	12	14	10	7	4	1	0			
2	12	5	7	5	2	0						
3	5	5	1	0								
4	5	1	0									
5	1	0										
6	0											

Table 1: Memory occupied by the unfinished tasks.

In the second case, we can only finish 3 out of the 4 processing demand of task t_0 by the end of the first unit of time due to the slow processing speed at v_{lo}; however, after raising the voltage to v_{hi} during the second unit, we are able to finish both the remaining of t_0 and entire t_1; the storage for tasks t_0 and t_1 are freed and therefore when t_2 arrives, we only need 12 units of storage to store this new task. Thus, we fill entry (1,1) with 12, the smaller storage requirement of the two different strategies.

Let $m(i,j)$ be the content of entry (i,j). We can reach this entry from entry $(i-1,j)$ by applying v_{hi} or from its left neighbor $(i,j-1)$ by applying v_{lo}, hence we have:

$$\begin{aligned} m(i,j) \quad = \min \quad & (s_{i+j} + \max(0, m(i-1,j) - sp_{hi}), \\ & s_{i+j} + \max(0, m(i,j-1) - sp_{lo})) \end{aligned} \tag{2}$$

where sp_{lo} and sp_{hi} are the processing speed at v_{lo} and v_{hi} respectively. The inner max is introduced to enforce that excess processing resource cannot be used for future work. We build Table 1 based on formula (2), where every row ends with an entry of 0 meaning that there are no tasks left.

While $m(i,j)$ gives the minimal storage requirement at the instant $i+j$, we may have used more storage already before this time. We further denote $M(i,j)$ as the minimal amount of storage that has been used up to time $i+j$ after running i units of time at v_{hi} and j at v_{lo}. Considering the voltage being used in the $(i+j)$-th unit, we observe that if we use v_{hi}, we can finish at most $\max(m(i-1,j), sp_{hi})$ and need a storage of $s_{i+j} + \max(0, m(i-1,j) - sp_{hi})$. Moreover, previously we have already required a storage at the amount of $M(i-1,j)$. This implies that

$$M(i,j) \geq \max(M(i-1,j), s_{i+j} + \max(0, m(i-1,j) - sp_{hi}) \tag{3}$$

Similar inequality holds if we use v_{lo}, therefore we have

$$M(i,j) = \min(\max(M(i-1,j), s_{i+j} + \max(0, m(i-1,j) - sp_{hi}), \tag{4}$$

$$\max(M(i,j-1), s_{i+j} + \max(0, m(i,j-1) - sp_{lo})) \tag{5}$$

Based on the recursive formulas (2) and (4), we calculate $M(i,j)$'s and store them in Table 2, where the last entry of the i-th row gives the minimal storage requirement to complete all the tasks by using v_{hi} for exactly i units.

The power consumption at $v_{hi} = 3.3v$ is 1, then the power consumption at $v_{lo} = 1.8v$ is 0.1 from our power model. Unlike the storage requirement, energy consumption is path independent. I.E., it depends on the total number of CPU units that we have used v_{lo} and v_{hi}, not the voltage at every individual time unit. For instance, if we have used v_{hi} for 2 units and v_{lo} for 4 units, then the total energy consumption is calculated as $1 \times 2 + 0.1 \times 4 = 2.4$.

Table 3 gives the memory requirements and total energy consumptions by different scheduling policies, where (i,j) in the first row indicates a schedule that uses v_{hi} for i units and v_{lo} for j units. Clearly from this table, we see that there exist three competitive optimal solutions, $(4,2)$,

	0	1	2	3	4	5	6	7	8	9	10	11
0	4	8	17	17	19	19	19	19	19	19	19	19
1	7	12	12	14	19	19	19	19	19			
2	12	12	12	14	14	14						
3	12	12	12	14								
4	12	12	12									
5	12	12										
6	12											

Table 2: Memory requirements by different schedules.

	(6,0)	(5,1)	(4,2)	(3,3)	(2,5)	(1,8)	(0,11)
M	12	12	12	14	14	19	19
E	6.0	5.1	4.2	3.3	2.5	1.8	1.1

Table 3: Memory and energy for different schedules

$(2,5)$, and $(0,11)$. They consume different amount of energy and require different amount of memory. We can then choose the one that fits our preference of memory and energy, and retrieve the actual schedule (i.e., the voltage for each CPU unit) by a simple backtracking approach.

4.2 Optimal Off-line Algorithm

Figure 2 shows the algorithm of finding all the competitive optimal solutions for multiple processes. A schedule in this case has to determine, for each CPU unit, which process to be executed and at which voltage level.

Definitions: Assuming that there are m processes and k different voltages, we have $m \cdot k$ choices: running the ith process at voltage v_j $(1 \le i \le m, 1 \le j \le k)$. A *state* $S = (e_1, \cdots, e_m; u_1, \cdots, u_k)$ means that the ith process has been allocated e_i CPU units, and the processor has been working at voltage v_j for u_j CPU units. Notice that $\sum_{i=1}^{m} e_i \le \sum_{j=1}^{k} u_j$ and the equality holds if and only if at any time, there exists unfinished process(es). We say state $S = (e_1, \cdots, e_m; u_1, \cdots, u_k)$ *precedes* $S' = (e'_1, \cdots, e'_m; u'_1, \cdots, u'_k)$ if $i) e'_i \ge e_i, ii) u'_j \ge u_j, iii) \sum_{i=1}^{m} e'_i - e_i \le 1$, & $iv) \sum_{j=1}^{k} u'_j - u_j = 1$. If S precedes S', we say that S' *follows* S. Define $Prev(S) = \{S' : S' precedes\ S\}$, and $Next(S) = \{S' : S \in Prev(S')\}$. A state S is *reachable* if $Prev(S) \ne \phi$. A *final* state is a state when all processes' requests aresatisfied. A schedule is a sequence of states $\{S_1, S_2, \cdots\}$ such that $S_l \in Prev(S_{l+1})$ and all processes' processing loads are satisfied at the final state.

The off-line optimal algorithm consists of three phases. First, we build an $m \times k$ dimensional table which stores the minimal memory requirements for different schedules. For example, when there is only one process and two voltages, then we will have a table like Table 2. Step 1 computes the *Next* set for the initial state S_0, if all m processes require CPU at the beginning, then this set will have $m \times k$ elements. Steps 2-4 makes all the states in $Next(S_0)$ reachable, since they are all one move away from the initial state S_0. We denote the set of reachable states by $\mathcal{S}$. Steps 5-18 build the table recursively until there is no reachable state. We keep all the reachable states in a queue, we calculate the *Next* set for the head of the queue (state S) in Step 15, delete S from the queue and put all elements of $Next(S)$ into the queue in Step 16. When we compute $Next(S)$, we consider all the timing requirements, for example, if process i has a deadline at the end of next CPU unit and its remaining process requirement can be fulfilled only when we use the highest voltage, then $Next(S)$ will contain only one state, which assigns the current CPU unit to process i and apply the highest voltage. Because all other schedules will fail to meet process i's deadline. The memory requirement for each state is calculated using formulas similar to (2) and (4).

Input: m processes with their arrival time, processing load, and other timing requirement (deadline, synchronization, etc.); k different supply voltages.
Output: All competitive pairs of memory requirement and energy consumption, and one schedule for each such pair.

Algorithm:
Phase I: Configuration for all states.
1. Compute $Next(S_0)$ for the initial state S_0;
2. for each $S \in Next(S_0)$
3. { $Prev(S) = S_0$;
4. $\mathcal{S} = \mathcal{S} \cup S$;}
5. while ($\mathcal{S} \neq \phi$)
6. { for each $S \in \mathcal{S}$
7. { current_max_memory for state $S = \infty$;
8. for each $S' \in Prev(S)$
9. { calculate the max_memory requirement if S follows S';
10. if (max_memory $\leq$ current_max_memory)
11. { current_max_memory = max_memory;
12. current_previous_state = S'; }
}
13. max_memory for state S = current_max_memory;
14. previous_state for S = current_previous_state;
}
15. Compute $Next(S)$;
16. $\mathcal{S} = \mathcal{S} \cup Next(S) - S$;
17. for each $S' \in Next(S)$
18. $Prev(S') = Prev(S') \cup S$; }

Phase II: calculation for energy consumption.
19. for each final state S
20. calculate the energy_consumption for S;
21. compute all the competitive final states $\mathcal{F}$;

Phase III: determine one schedule for each competitive state.
22. for each competitive state $S = (e_1, \cdots, e_m; u_1, \cdots, u_k) \in \mathcal{F}$
23. { index $= l = \sum_{j=1}^{k} u_j$;
24. $S_{index} = S$;
25. while (index $\neq 0$)
26. { S' = previous_state for S_{index};
27. index = index - 1;
28. $S_{index} = S'$; }
29. report the schedule $(S_0, S_1, \cdots, S_l)$ for S; }

Figure 2: Algorithm for all off-line competitive schedules.

From the table built in the first phase, we can easily see the total memory requirement for each schedule, which is the value at its corresponding final state. In Phase II, we calculate their energy consumption. Recall that the energy consumption is path-independent. Let P_j be the power for voltage v_j, then for final state $S = (e_1, \cdots, e_m; u_1, \cdots, u_k)$, all schedulers with this final state will consume energy in the amount of $E = \sum_{j=1}^{k} P_j \cdot u_j$. So for each final state, we associate with the pair (M, E), the memory requirement and the energy consumption. Recall also that two final states S and S' are *competitive* if i) $M \leq M'$ and $E \geq E'$, or ii) $M \geq M'$ and $E \leq E'$.

In the third phase, we find a schedule for each competitive final state. We achieve this by using backtracking as shown in Steps 23-29. The existence of state S' in Step 26 is guaranteed by the way in which we build the memory requirement table in Phase I. Therefore, we have:

Theorem 4.1: The algorithm in Figure 2 finds all the feasible competitive schedules.

We analyze the complexity of the algorithm, for a fixed processor that has k supply voltages to execute m processes, in terms of the total processing demands. Suppose that we need X CPU

tasks	1-voltage	2-voltages	3-voltages	4-voltages
1	1	35.3%	45.6%	51.9%
2	1	41.8%	49.4%	54.3%
3	1	47.6%	53.0%	56.6%
5	1	52.9%	56.3%	58.7%
6	1	55.7%	58.5%	60.2%
8	1	57.2%	59.9%	60.7%
10	1	57.9%	60.8%	61.1%
average	N/A	49.77%	54.79%	57.64%
median	N/A	52.9%	56.3%	58.7%

Table 4: Energy savings by off-line algorithm using multiple supply voltage assuming $3.3V$ nominal voltage with the same amount of energy.

units to service all the processes at the reference voltage. In Phase I, we essentially fill in the entries of an $m \times k$ dimensional table (Table 2 is an example with $m = 1$ and $k = 2$), where entries along dimension $< i, j >$ represent the storage requirements when the i-th process is processed with the j-th voltage. If the i-th process needs $X_{i,j}$ units under the j-th voltage, when all the other $m \times k - 1$ dimensions are fixed, the number of entries we need to fill along this dimension will be $X_{i,j}$, which is in the order of $O(X)$. Since the table is $m \times k$ dimensional, and the cost for computing each entry is constant, the run-time in Phase I will be $O(X^{mk})$.

The calculation of energy consumption in Phase II takes constant time for each final states. According to how many units we have run at the reference voltage, it is clear that we will have at most X different final states (c.f. Table 2 for an example). Thus, the cost here is $O(X)$. In the last phase, we determine a feasible schedule for each competitive final states by backtracking. In step 27, we move one entry closer to the starting point, and the total number of steps we need is also in the order of $O(X)$. Therefore, we have:

Theorem 4.2: If we need X CPU units to service all the processes at the reference voltage, the run-time of the proposed algorithm is $O(X^{mk})$.

5 Experimental Results

We used six streaming applications [9] to establish the effectiveness of the approach: IJG JPEG encoder and decoder, MSG MPEG encoder and decoder, CCITT G.721 encoder, and PGP encryption and description module.

The results of the off-line algorithm for energy saving is displayed in Table 4. We normalized the results to the case of 1-voltage at 3.3V, and use the same memory requirement from these results as the basis for the other tests. We performed test on the cases with 1, 2, 3, and 5 tasks. In the case of 2-voltages we used 3.3V and 1.8V, while for the case of 3-voltages we applied. 3.3V, 1.8V, and 1.0V. For the final case, 4-voltages, we applied 3.3V, 2.4V, 1.8V, and 1.0V for testing. In column 1, we have the number of tasks and the average values. The remaining columns show the percentage of energy savings for 2, 3, and 4-voltage. Our results show a 47.6% energy saving over the 1-voltage when using 2-voltages and the same amount of memory and 3 tasks. On average the energy savings increases with the number of voltages, but at a low rate.

6 Conclusion

We have developed an optimal polynomial-time algorithm for power minimization of streaming media applications under QoS requirements and hardware constraints using multiple voltages. The algorithm is flexible in the sense that it can address a variety of dual-primal QoS problem formulations as well as a variety of QoS dimensions, including latency, throughput and synchro-

nization. In addition, algorithm can be generalized to solve the initial problem under assumption that a user specified drop rate is not exceeded. The algorithm is practically fast in the sense that large instances of the problems can be solved rapidly.

References

[1] C. Aurrecoechea, et al. A survey of QoS architectures. Multimedia Systems, vol.6, no.3, pp. 138-151, 1998.

[2] L. Benini, A. Bogliolo, G.A. Paleologo, G. De Micheli. Policy optimization for dynamic power management. IEEE Transactions on CAD, vol.18, no.6 , pp. 813-833, 1999.

[3] A.P. Chandrakasan, S. Sheng, R.W. Brodersen. Low-power CMOS digital design. IEEE Journal of Solid-State Circuits, vol.27, no.4, pp. 473-484, 1992.

[4] A.P. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: an approach for energy efficient computing. ISLPED, pp. 344-352, 1996.

[5] J. Chang, M. Pedram. Energy minimization using multiple supply voltages. Int'l Symposium on Low Power Electronics and Design, pp. 157-162, 1996.

[6] E. Chang and H. Garcia-Molina. Medic: Memory and disk cache for media servers. IEEE Int'l Conference on Multimedia Computing and Systems, pp. 493-499, 1999.

[7] R.L. Cruz. Quality of Service Guarantees in Virtual Circuit Switched Networks. IEEE Journal on Selected Areas in Communications, vol.13, no.6, pp. 1048-1056, 1995.

[8] K. Govil, E. Chan, H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. ACM Mobile Computing and Networking, pp. 13-25, 1995.

[9] C. Lee, et al. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. International Symposium on Microarchitecture, pp. 330-335, 1997.

[10] G. Qu, M. Mesarina, M. Potkonjak. System Synthesis of Synchronous Multimedia Applications. Intl. Symposium on System Synthesis, pp. 128-133, 1999.

[11] Q. Qiu, M. Pedram. Dynamic power management based on continuous-time Markov decision processes. Design Automation Conference, pp. 555-561, 1999.

[12] J. M. Rabaey, M. Pedram. Low power design methodologies Kluwer Academic Publishers, 1996.

[13] S. Raje, M. Sarrafzadeh Scheduling with multiple voltages. Integration, The VLSI Journal, vol.23, no.1, pp. 37-59, 1997.

[14] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. Proceedings. IEEE Real-Time Systems Symposium, pp. 298-307, 1997.

[15] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical Solutions for QoS-based Resource Allocation Problems. Proceedings. The 19th IEEE Real-Time Systems Symposium, pp. 296-306, 1998.

[16] K. Usami, M. Igarashi, F. Minami, T. Ishikawa, et. al. Automated low-power technique exploiting multiple supply voltages applied to a media processor. IEEE Journal of Solid-State Circuits, vol.33, no.3, pp. 463-472, 1998.

[17] M. Weiser, B. Welch, A. Demers, S. Shenker. Scheduling for reduced CPU energy. USENIX Operating Systems Design and Implementation (OSDI), pp. 13-23, 1994.