Energy-Aware Scheduling for Streaming Applications on Chip Multiprocessors

Ruibin Xu, Rami Melhem, Daniel Mossé
Computer Science Department, University of Pittsburgh
{*xruibin,melhem,mosse*}@*cs.pitt.edu*

Abstract

Streaming applications have become increasingly important and widespread, and they will be running on soon-to-be-prevalent chip multiprocessors (CMPs). We address the problem of energy-aware scheduling of streaming applications, which are represented by task graphs, on CMPs using on/off and dynamic voltage scaling (DVS) on a per-processor basis. The goal is to minimize the energy consumption of streaming applications while satisfying two typical quality-of-service (QoS) requirements, namely, throughput and response time. To the best of our knowledge, this paper is the first *work to tackle this problem. We make the key observation that the trade-off between static power and dynamic power should play a critical role in both parallel processing and pipelining that are used to reduce energy consumption in the scheduling process. Based on this observation, we propose two scheduling algorithms, Scheduling1D and Scheduling2D, for linear and general task graphs, respectively. The proposed algorithms exploit the difference between the two QoS requirements and perform processor allocation, task mapping and task speed scheduling simultaneously. Experimental results show that the proposed algorithms can achieve significant energy savings (e.g., 24% on average for 70nm technology) over the baseline that only considers the response time requirement.*

1 Introduction

Streaming applications, which can be characterized as the ones that operate on a continuous stream of data, have become increasingly important and widespread. Examples of streaming applications include Internet audio and video streaming, automatic target recognition (ATR) found in radar digital signal processor (DSP) systems. Streaming applications are continuous in nature, and usually compute-intensive. This implies that they are energy-hungry, which will cause problems if they are running in energy-constrained systems, such as battery-operated embedded devices. Thus, there is a great need to optimize energy consumption for streaming applications, while satisfying their QoS requirements, which typically are *throughput* and *response time*.

We assume that a stream of data can be abstracted as a sequence of *requests* (e.g., a frame in video streaming is a request) and the streaming application is servicing the requests (e.g., decoding frames in video streaming) in succession. Therefore, a streaming application can be modeled as a periodic task in real-time systems. The throughput requirement, which is defined as the number of requests that are fed to the streaming application for service in one second, is equal to the reciprocal of task period. The response time requirement, which is defined as the maximum time allowed to service a single request, is equivalent to the end-to-end deadline constraint imposed on the task. For the rest of this paper, we use response time requirement and deadline interchangeably.

Many streaming applications are highly parallelizable, and thus very suitable to execute on multiprocessor systems, which will be the dominant computer architecture in the near future due to the emergence of chip multiprocessors (CMPs). By combining multiple small processor cores on a single chip, CMPs continue to push the processor performance growth beyond the clock rate limit. Several chip makers have released CMPs, such as IBM/Sony/Toshiba's 9-core CELL [1]. Intel has built a prototype of a CMP with 80 processor cores and announced that it will be commercially available in 2010 [2]. The trend is that more and more processor cores will be seen on a single chip.

As manufacturing process increases the number of processor cores on a chip, power density increases, making power management a major concern for CMPs. We assume that CMPs allow for two power management mechanisms for each processor core independently: (i) turning off (or putting into sleep mode) unused processor cores to decrease static power consumption resulting from leakage current that exists even in the absence of switching activities in circuits; and (ii) dynamic voltage scaling (DVS), which dynamically adjusts the voltage and frequency (speed) of a processor core to decrease dynamic power consumption re-

sulting from switching activities in circuits. There is a fundamental trade-off between static power consumption and dynamic power consumption for CMPs. Assuming perfect parallelism for a given workload subject to certain performance constraint, as the number of active cores on a CMP increases, the static power consumption of the CMP increases, while the dynamic power consumption decreases since the load on each core is smaller. As long as neither dynamic nor static power accounts for most of the total power, which is true for current technology, the two aforementioned power management mechanisms must be combined to optimize the power consumption that leads to minimum energy consumption for applications.

In this paper, we study the problem of scheduling streaming applications on a CMP with the goal of minimizing the energy consumption of the streaming applications while satisfying two QoS requirements, throughput and response time, for each application. We call this problem STREAM-CMP. We assume that the processor cores are architecturally simple and do not employ preemptive scheduling. This assumption is valid for CELL [1] and Intel's future 80-core CMP [2]. Thus, assuming that streaming applications do not share resources, the STREAM-CMP problem is reduced to scheduling each streaming application independently to minimize the energy consumption of each individual streaming application. A streaming application is represented by a task graph. The outcome of scheduling a streaming application are: (i) the number of active processor cores to execute the task graph; (ii) the mapping of tasks to active cores; and (iii) the execution speed of each task (also called *speed schedule*).

The STREAM-CMP problem is NP-hard because its special case, which assumes that the number of active cores is given and the deadline is equal to the period, is NP-hard [3]. The general STREAM-CMP problem deserves research attention for the following reasons.

First, the period is shorter than the deadline in many situations, one of them being when applying automatic target recognition (ATR) in unmanned autonomous vehicle (UAV). Straightforward application of scheduling algorithms that assume the deadline being equal to the period will force streaming applications into servicing each request within their periods, which means streaming applications service requests faster than required. This is in contrast with the common wisdom on DVS that the execution of a task should be slowed down for just-in-time completion. On the other hand, relaxing the deadline constraint for a streaming application can lead to reduction in energy consumption without affecting user satisfaction if the delay of processing the first request can be tolerated. This is true for video streaming applications.

Second, finding the appropriate number of active processor cores to execute an application is crucial for saving energy because the static power consumption accounts for a significant portion of the total power as feature sizes shrink to nanometer level [10]. In Section 4.1, we show that high static power may force streaming applications into servicing requests faster than required in order to save energy, which is counterintuitive.

Third, even when the optimal number of active cores is known, the task mapping and task speed scheduling (i.e., deciding task speed) pose new challenges for the general STREAM-CMP problem because of the interplay of the two QoS requirements under consideration.

The contribution of this paper is twofold. First, we propose the STREAM-CMP problem, of which we have already shown the importance and value. Second, we propose the *first* solution to the STREAM-CMP problem (see Section 2 for a comparison of related work). Specifically, we propose two scheduling algorithms, Scheduling1D and Scheduling2D, for linear and general task graphs, respectively. The proposed algorithms exploit the difference between the two QoS requirements and perform processor allocation, task mapping, and task speed scheduling simultaneously. Our algorithms use parallel processing and pipelining in the task mapping, as traditional algorithms for maximizing throughput or minimizing latency do. However, our algorithms focus on finding the appropriate number of processors and allotting the optimal amount of time to each pipeline stage and each task in order to save energy.

The remainder of this paper is organized as follows. We first review the closely related work in Section 2. Section 3 describes the streaming application model, system model, and power model. A fully polynomial time approximation scheme called Scheduling1D is proposed for scheduling linear task graphs in Section 4. In Section 5, we propose heuristics to reduce the complexity of scheduling general task graphs and extend Scheduling1D to a heuristic algorithm called Scheduling2D for scheduling general task graphs. The experimental results are reported in Section 6. Finally, we conclude the paper in Section 7.

2 Closely Related Work

There are 4 key elements in the STREAM-CMP problem: (i) streaming applications are represented by task graphs; (ii) multiprocessor; (iii) two different constraints, throughput and response time requirement; (iv) energy-aware scheduling using on/off and DVS. We will review related work that contains these elements.

Much research has been done on energy-aware scheduling of task graphs assuming the deadline being equal to the period using DVS. To name a few, Mishra et al. proposed several heuristics to obtain the execution speed of each task assuming the number of processors and task mapping are given [13]. Andrei et al. proposed a convex programming

based approach, again, assuming the number of processors and task mapping are given [3]. All these algorithms can be used as a component in our scheduling algorithm for general task graphs. Although previous work has not been explicitly extended to consider on/off, it is straightforward to do so by simply trying every possible number of processors. In Section 5.3, we propose a better approach based on hill-climbing.

Before DVS emerged as an important power management mechanism, various approaches [12, 16, 9] were proposed to use parallel processing and pipelining to maximize throughput or minimize number of processors in scheduling task graphs on multiprocessor systems. Because energy consumption was not under consideration, straightforward application of these approaches cannot fulfill the need for energy optimization. On one hand, scheduling approaches that maximize throughput tend to use more processors, which is not energy efficient for large static power, and do not guarantee to comply with the response time requirement. On the other hand, scheduling approaches that minimize the number of processors tend to use fewer processors, which is not good for large dynamic power. Our scheduling algorithms also apply parallel processing and pipelining. However, we use them as *energy reduction techniques* and focus on finding appropriate number of processors to embrace the trade-off between static and dynamic power, and allotting appropriate amount of time to each task to stretch its execution.

Combining on/off and DVS to exploit the trade-off between static power consumption and dynamic power consumption has been used in multiprocessor-like settings by a number of works. In [6], Elnozahy et al. proposed a power management policy to determine the optimal number of online servers and corresponding operating frequency to minimize the energy consumption of clusters. In [15], Xu et al. tackled a similar problem considering several practical issues. In [17], Anderson et al. studied energy-efficient synthesis of periodic task systems on multiprocessor platforms. However, all of the above research dealt with independent tasks and considered only deadline constraint. Thus, they cannot be applied straightforwardly to task graph scheduling.

More recently, Kim et al. [11] explored the effectiveness of the simultaneous application of pipelining and parallel processing as a total power reduction technique in uniprocessor design. Our work is different from theirs in several aspects: (i) They focused on uniprocessor that is under a single voltage domain while we focus on multiprocessor systems and processors can operate at different voltage and frequency and have the capability of turning on/off; (ii) They assumed idealized unlimited parallelism in instruction streams while we focus on task graph mapping; (iii) They only considered throughput constraint while we consider two QoS requirements; (iv) Their parallel processing width (instruction issue width) is fixed for all stages while we can use different number of processors at different stages. (v) Their goal was to minimize the power consumption while we are trying to minimize the energy consumption.

3 Models

3.1 Application Model

A streaming application is modeled as a task graph $G(V, E)$, which is a directed acyclic graph (DAG). There are n vertices in the task graph. The vertex $v_i \in V$ represents task τ_i and is associated with c_i, which is the worst-case number of cycles needed to execute τ_i. The directed edge e_{ij} represents dependency between task τ_i and τ_j, that is, τ_j is ready to begin execution only after τ_i finishes execution (τ_i is called the predecessor of τ_j and τ_j is called the successor of τ_i). A communication volume v'_{ij} is associated with edge e_{ij}, and determines the time and energy penalties when τ_i and τ_j are scheduled on two different processor cores. We assume that the communication cost is zero if the communicating tasks are scheduled on the same core. In the task graph, the source is the only vertex that has no predecessors and the sink is the only vertex that has no successors. In streaming applications, the source receives the requests and the sink emits the output of servicing the requests. The period is T (i.e., the streaming application must sustain a throughput of $\frac{1}{T}$), and the response time requirement (i.e., the deadline) is D.

3.2 System and Power Models

The system model under consideration is a typical homogeneous CMP architecture with distributed memory (Figure 1). There are a total of $\mathbb{N}$ processor cores available on the chip, each consisting of a processing unit, a local memory, and a switch. We assume that the program for each task is written in stream programming style [19], that is, the program is divided into three consecutive parts. The first part gathers data from the predecessors and puts data in the local memory. The second part does computation in processing unit on data in local memory. The third part sends data from local memory to communication networks.

As for power management, each processor core is capable of being turned on/off and changing its voltage and frequency dynamically. Each processor core provides M discrete frequencies, $f_1 < f_2 < \cdots < f_M$. When a processor core is off, its power consumption is zero. When a processor core is on, it is either (i) idle, consuming P_{idle} power or (ii) executing some task at some frequency f_i, consuming $P(f_i)$ power. Note that when a processor core

is on, P_{idle} represents the amount of the power that is always consumed. The static power of a processor core contributes the most to the idle power. Other contribution to the idle power depends on what processor does when idle (e.g., whether use clock-gating). Because the processing unit and local memory are in the same clock domain and all the memory accesses are to the local memory, the execution time of executing task τ_i at frequency f is $\frac{c_i}{f}$ and the corresponding energy is $P(f) \times \frac{c_i}{f}$. For communication cost, we adopt a linear model, that is, when transferring B bits of data, the communication delay is $t_p + \lambda B$ and the communication energy is γB where t_p is the propagation delay, λ is the reciprocal of the operating data rate of the interconnection network, and γ is the energy spent to transfer one bit of data. For this work, we assume fixed data rate, that is, fixed γ and λ. Under current technology, CMPs provide very high bandwidth for the interconnection network (e.g., CELL). Although the adopted linear model does not account for network contention, it was shown to work very well for high-bandwidth interconnection network [9].

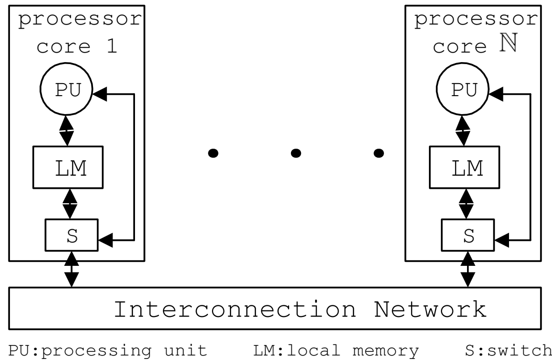


Figure 1. Chip multiprocessor model

4 Scheduling for Linear Task Graphs

In this section, we present the scheduling algorithm for a special type of task graphs, linear task graphs. Without loss of generality, we assume that task τ_1 is the source and τ_n is the sink. In a linear task graph, the only predecessor of task τ_i ($1 < i \leq n$) is task τ_{i-1}. The tasks can be arranged to form a straight line and a total order can be established on all tasks. Thus, there is only parallelism in time for linear task graphs. Linear task graph corresponds to the end-to-end task model in real-time system research [8].

Although scheduling linear task graphs is NP-hard [3], it admits a fully polynomial time approximation scheme. That is, the solution returned by the scheduling algorithm is guaranteed to be within ϵ (ϵ is a user-defined parameter) of the optimal solution and the scheduling algorithm runs in time polynomial in $\frac{1}{\epsilon}$. Also, the scheduling algorithm for linear task graphs serves as the basis for our scheduling algorithm for general task graphs. Before we describe the scheduling algorithm for linear task graphs, we first simplify the problem to gain insight that will help us understand the problem.

4.1 Y-Oriented Load

We simplify the problem of scheduling a linear task graph by relaxing the application and power models. We relax the application model by assuming that the streaming application is represented by a single task τ, which is a *divisible load* [18] (Figure 2(a)) that can be arbitrarily partitioned into any number of load fractions that have precedence relations (i.e., all c cycles of τ can be only executed consecutively). We call this Y-oriented load (Figure 2(b)) because it is in the direction of Y-axis. There is also a notion of X-oriented load which will be described in Section 5.1. We relax the power model by assuming that the frequency can be changed continuously and that all communication cost is ignored. The power function of processor cores is

$$P(f) = C_0 + C_1 f^3 \tag{1}$$

where f is the operating frequency. The constant C_0 represents the static power and the term $C_1 f^3$ represents the dynamic power where C_1 is the effective switching capacitance. The form of the analytical power function is due to the fact that dynamic power consumption can be computed by $C_1 \times V_{dd}^2 \times f$ (V_{dd} is the supply voltage) and the frequency f is almost linearly related to the supply voltage [21]. Note that for many real-world processors, using curve-fitting for the actual power numbers will result in an analytical power function close to the one given by (1).

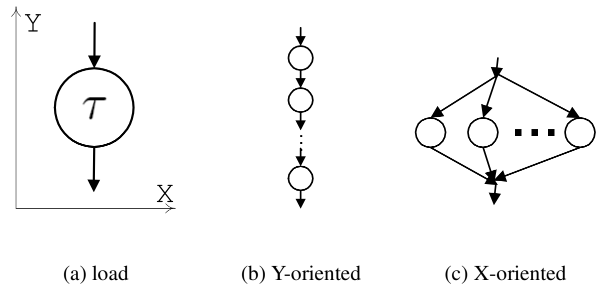


Figure 2. Divisible load

First, we ignore the deadline constraint and consider the problem of energy minimization of the load τ subject to only the throughput requirement, $\frac{1}{T}$. Pipelining is a natural approach to satisfying the throughput requirement and

load balancing is desired due to the convexity of the power function (1). Suppose that y ($y \leq \mathbb{N}$) processor cores are used to execute this load. Thus, each processor core (corresponding to a pipeline stage) is assigned $\frac{c}{y}$ cycles and the corresponding speed is $\frac{c/y}{T} = \frac{c}{yT}$. In servicing a single request, the static energy consumption is yC_0T and the dynamic energy consumption is $yC_1\left(\frac{c}{yT}\right)^3 T$. Therefore, the total energy consumption for servicing a request is

$$e_Y(y) \quad = \quad yC_0T + \frac{C_1c^3}{y^2T^2} \tag{2}$$

which is a unimodal function and has a global minimum. This shows that starting from a single stage, deepening the pipeline will reduce the energy consumption while satisfying the throughput requirement $\frac{1}{T}$, until the number of pipeline stages increases past a certain value y^*, which is the optimal number of pipeline stages for load τ. The optimal number of pipeline stages strikes a balance between static and dynamic power. In fact, by obtaining the first derivative of $e_Y(y)$ and equating it to zero, we have the optimal number of pipeline stages for executing τ, which is given by

$$y^* = \sqrt[3]{\frac{2C_1}{C_0}} \cdot \frac{c}{T} \tag{3}$$

Note that for the purpose of describing the basic idea succinctly, we allowed ourselves not to be rigorous, that is, we can use fractional number of cores and do not consider the boundary conditions.

Suppose that we now impose the deadline constraint D on τ. If $D = T$, we have to use only 1 pipeline stage; if $D = 2T$, we can use two pipeline stages and the energy consumption is reduced. This shows that the difference between T and D can have impact on energy reduction. We can continue relaxing the deadline constraint (i.e., increasing D) to reduce the energy consumption of τ until $D > y^*T$. In this case, the response time of τ is y^*T, which is less than the deadline constraint D. This shows that the static power affects the upper bound of the response time when the goal is to save energy, and sometimes the application needs to service requests faster than the response time requirement in order to save energy. This is counterintuitive because common wisdom on DVS scheduling says that the execution of tasks should be stretched as much as possible as long as the deadline constraint is not violated.

4.2 The Scheduling1D Algorithm

We now revert back to the models described in Section 3 to design scheduling algorithm to schedule a linear task graph. Since linear task graphs are analogous to Y-oriented load and there is only parallelism in time, we call our scheduling algorithm for linear task graphs Scheduling1D.

4.2.1 The Structure of the Optimal Solution

The basic scheduling strategy for linear task graphs is pipelining. There are three questions that need to be answered in order to schedule a linear task graph. First, how many pipeline stages does it need to execute this task graph? Each pipelining stage will correspond to a processor core. From the analysis in Section 4.1, we can see that the static power consumption will have a significant impact on the optimal number of pipeline stages. Second, how to map the tasks in the task graph to processor cores? Obviously, only consecutive tasks in the task graph will be mapped to a processor core (pipeline stage). Note that now the mapping granularity is tasks other than cycles as in Section 4.1. Due to the communication energy and delay, load balancing is not necessarily desired. Third, what speed is to be used for each processor core such that the delay of each stage is no more than the period and the total delay of all stages is no more than the deadline? Note that due to the convexity of the power function, all tasks on the same processor core will use the same speed.

The above three questions are correlated and should not be considered separately. Thus, the scheduling algorithm needs to perform finding the optimal number of stages, mapping, and speed scheduling simultaneously. We first present an optimal scheduling algorithm for linear task graphs. This optimal algorithm has worst-case exponential time complexity. Thereafter, we propose an approximation algorithm that is based on the optimal algorithm.

The optimal scheduling algorithm for linear task graphs is based on the recursive structure of the optimal solution. Let the vector-valued function $E_i(t) = [e, q, j, d]$ denote the optimal scheduling of the tasks τ_i through τ_n when the end-to-end delay from task τ_i to task τ_n is t. In this optimal scheduling, e denotes the minimum energy consumption executing the tasks τ_i through τ_n when servicing a single request, q denotes the optimal number of stages for the tasks τ_i through τ_n, j indicates that tasks $\tau_i, \tau_{i+1}, \ldots, \tau_j$ are mapped to the first stage of the q stages, and d is the time used for the first stage plus the communication delay from the first stage to the next stage. Suppose we are given the functions $E_{i+1}(\cdot)$ through $E_n(\cdot)$, we can compute $E_i(t)$ using the pseudo-code in Figure 3 (note that $v'_{n,n+1} = 0$, i.e., there is no communication after the sink).

The computation of the energy for the first stage at Line 8 in Figure 3 needs further clarification. The first term $(P(f_s) - P_{idle})\frac{c_k}{f_s}$ is the dynamic energy consumption of the first stage for servicing a single request. The second term $P_{idle}T$ is the static energy consumption. Recall that P_{idle} is the amount of power that is always consumed in a

```
1.E_i(t).e = ∞
2.for j := i to n do //n is # of tasks
3.   c := Σ_{k=i}^{j} c_k
4.   for s := 1 to M do //M is # of frequencies
5.      d := c/f_s + t_p + λv'_{j,j+1}
6.      if d ≤ T then
7.         //e_1 is the energy for the 1^{st} stage
8.         e_1 := (P(f_s) − P_idle) c_k/f_s + P_idle T
9.         e := e_1 + γv'_{j,j+1} + E_{j+1}(t − d)
10.        if e < E_i(t).e then
11.           E_i(t).e = e
12.           E_i(t).q = E_{j+1}(t − d).q + 1
13.           E_i(t).j = j
14.           E_i(t).d = d
```

Figure 3. Computing $E_i(t)$

processor when it is on. Since the period is T, servicing each request will get the share of static power consumed for time T.

The optimal energy consumption and scheduling information of the whole linear task graph will be obtained from $E_1(D)$. We can see from Figure 3 that in order to compute $E_1(D)$, we need to compute $E_2(\cdot), E_3(\cdot), \ldots, E_n(\cdot)$. In general, $E_i(\cdot)$ depends on $E_k(\cdot)$ ($k = i+1, \ldots, n$). Thus, the base case is $E_n(\cdot)$ which denotes the scheduling information for a single task, τ_n. It is not difficult to see that the base case is a *step function* (piece-wise constant function) because there are only a limited set of discrete speeds available in processor cores. By induction, we can show that all functions $E_i(\cdot)$ are step functions. A step function can be represented by the end points of intervals in the function. Once all end points in a step function are identified, we can obtain any value of that step function. Because of the discrete nature and recursive structure of $E_i(\cdot)$, we can apply dynamic programming technique to compute these functions. We compute the functions $E_i(\cdot)$ ($i = 1, 2, \ldots, n$) in reverse order. That is, we first compute $E_n(\cdot)$, then compute $E_{n-1}(\cdot)$, ..., last compute $E_1(\cdot)$. Note that computing function $E_i(\cdot)$ is to identify all its end points, rather than compute a single value as in Figure 3. Because $E_i(\cdot)$ is a step function, we need to first describe the representation of step functions and the associated operations before we present the algorithm to compute $E_i(\cdot)$.

4.2.2 On Step Functions

We first formally define step function through the following two definitions.

Definition 1 *A point $\mathbb{P}$ is a 2-tuple (e, t), where e and t are nonnegative reals and denote energy and time respectively. We write the energy component as $\mathbb{P}.e$ and the time component as $\mathbb{P}.t$.*

Definition 2 *A step function (piece-wise constant function) $\mathbb{F}(\cdot)$ is represented by a point sequence $\mathbb{S} = [\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_m]$ where $\mathbb{P}_1.t < \mathbb{P}_2.t < \cdots < \mathbb{P}_m.t$. The step function $\mathbb{F}$ is defined on $t \geq \mathbb{P}_1.t$ such that $\mathbb{F}(t) = \mathbb{P}_i.e$ and $i = \max_{j=1,2,\ldots,m} \{j | t \geq \mathbb{P}_j.t\}$.*

Having formally defined step function, we will use $\mathbb{F}$ to denote a step function $\mathbb{F}(\cdot)$ unless confusion arises. Let $|\mathbb{F}|$ denote the number of points in $\mathbb{F}$. Obviously, computing $\mathbb{F}(t)$ can be done in time $O(\log |\mathbb{F}|)$ by using binary search.

We now look at three operations between a number and a step function.

Definition 3 *The operator $+_e$ is defined between a real x and a step function $\mathbb{F}$ such that $x +_e \mathbb{F} = \{(x+e, t) | (e, t) \in \mathbb{F}\}$ (i.e., the result is still a step function). Other operators, $\times_e$ and $+_t$ can be defined similarly.*

Obviously, the operators defined in Definition 3 can be performed in time $O(|\mathbb{F}|)$.

Finally, we describe two operations between step functions.

Definition 4 *The sum operator $+$ is defined between 2 step functions, $\mathbb{F}_1$ and $\mathbb{F}_2$, such that $\mathbb{F}_1 + \mathbb{F}_2 = \mathbb{F}$ and $\mathbb{F}(t) = \mathbb{F}_1(t) + \mathbb{F}_2(t)$. The merge operator $\cup$ is defined between 2 step functions, $\mathbb{F}_1$ and $\mathbb{F}_2$ such that $\mathbb{F}_1 \cup \mathbb{F}_2 = \mathbb{F}$ and $\mathbb{F}(t) = min(\mathbb{F}_1(t), \mathbb{F}_2(t))$.*

The resulting step function $\mathbb{F}$ by either the sum or the merge operators over m step functions $\mathbb{F}_i$ ($i = 1, 2, \ldots, m$) could have as many as $\sum_{i=1}^{m} |\mathbb{F}_i|$ points. The time component of each point in $\mathbb{F}$ comes from one of the $\mathbb{F}_i$'s. Because the points in $\mathbb{F}_i$ are already sorted, the time components of all points in $\mathbb{F}$ can be obtained by a procedure similar to merge sort in time $O((\sum_{i=1}^{m} |\mathbb{F}_i|) \log m)$. To compute the energy component of each point in $\mathbb{F}$, the sum operator takes constant time and the merge operator takes $O(\log m)$ time by using a priority queue. Thus, computing $\sum_{i=1}^{m} \mathbb{F}_i$ takes $O((\sum_{i=1}^{m} |\mathbb{F}_i|) \log m)$ time and computing $\cup_{i=1}^{m} \mathbb{F}_i$ takes $O((\sum_{i=1}^{m} |\mathbb{F}_i|) \log^2 m)$ time.

4.2.3 The Optimal Scheduling Algorithm

For succinct presentation, we do not show the computation of functions $E_i(\cdot).q$, $E_i(\cdot).j$ and $E_i(\cdot).d$ because they can be easily performed as a by-product of computing $E_i(\cdot).e$. We will also write $E_i(\cdot).e$ as $E_i(\cdot)$.

In Figure 3, we compute a single value of $E_i(\cdot)$. Now we make use of step functions to compute the whole function $E_i(\cdot)$ (i.e., identify all the end points in $E_i(\cdot)$). To do that, we first consider $(n - i + 1) \times M$ helper functions $\hat{E}_{i,j,s}$

($j = i, i+1, \ldots, n$ and $s = 1, 2, \ldots, M$), where M is the number of available discrete speeds. $\hat{E}_{i,j,s}$ denotes the energy function when tasks $\tau_i, \tau_{i+1}, \ldots, \tau_j$ are mapped to the first stage and f_s is the speed used in the first stage. A single value of $\hat{E}_{i,j,s}$ can be computed as

$$\begin{aligned}\hat{E}_{i,j,s}(t) &= (P(f_s) - P_{idle})\frac{\sum_{k=i}^{j} c_k}{f_s} + P_{idle}T \\ &+\gamma v'_{j,j+1} + E_{j+1}(t - \frac{\sum_{k=i}^{j} c_k}{f_s} - t_p - \lambda v'_{j,j+1})\end{aligned} \quad (4)$$

Computing the whole function $\hat{E}_{i,j,s}$ can be expressed using step functions described in Section 4.2.2 as Line 9 in Figure 4. The desired function E_i is obtained by merging the $(n-i+1) \times M$ $\hat{E}_{i,j,s}$ functions. During the merging process, the optimal values of $E_i(\cdot).q$, $E_i(\cdot).j$ and $E_i(\cdot).d$ corresponding to each point are also determined. The optimal scheduling algorithm for linear task graph is shown at Lines 1-12 in Figure 4. Line 13 is used for the approximation algorithm and will be explained Section 4.2.4.

We now analyze the time complexity and space complexity of computing E_i. Computing the helper function $\hat{E}_{i,j,s}$ takes time $O(|E_{j+1}|)$ and the number of points in $\hat{E}_{i,j,s}$ is also $O(|E_{j+1}|)$. The key operation in computing E_i is the merge operation over $(n-i+1) \times M$ step functions, each is of size $O(|E_{i+1}|)$. Thus, the time to compute E_i is $O(nM|E_{i+1}| \log^2 nM)$ and the number of points in E_i is $O(nM|E_{i+1}|)$. Since the base case is $|E_{n+1}| = 1$, we can obtain the closed forms of the time complexity and space complexity to be $O((nM)^{n-i+1} \log^2 nM)$ and $O((nM)^{n-i+1})$, respectively. Since the optimal solution is in $E_1(\cdot)$, the time complexity and space complexity of the optimal scheduling algorithm for linear tasks graphs are $O((nM)^n \log^2 nM)$ and $O((nM)^n)$, respectively.

PROCEDURE Scheduling1D(ϵ)
1. $E_{n+1} := \{(0,0)\}$
2. for $i := n$ downto 1 do
3. //compute E_i
5. for $j := i$ to n do
4. $c := \sum_{k=i}^{j} c_k$
6. for $s := 1$ to M do
7. $d := \frac{c}{f_s} + t_p + \lambda v'_{j,j+1}$
8. if $d \le T$ then
9. $\hat{E}_{i,j,s} := (P(f_s) - P_{idle})\frac{c}{f_s} + P_{idle}T$ $+\gamma v'_{j,j+1} +_e (d +_t E_{j+1})$
10. else
11. $\hat{E}_{i,j,s} := \phi$
12. $E_i := \bigcup_{j=i,\ldots,n,s=1,\ldots,M} \hat{E}_{i,j,s}$
13. $E_i := TRIM(E_i, (1+\epsilon)^{\frac{1}{n}} - 1)$

END

Figure 4. The Scheduling1D algorithm for linear task graphs

4.2.4 Approximation Algorithm

The time complexity of the optimal scheduling algorithm for linear task graphs depends greatly on the size of functions E_i. As we can see from the analysis of the optimal scheduling algorithm in Section 4.2.3, the size of E_i may grow exponentially as i goes from n to 1. Thus, we need to control the size of function E_i within some polynomial bound. To do that, we trim (i.e., remove some points) function E_i after it is computed at Line 12 in Figure 4. A trimming parameter δ ($0 < \delta < 1$) is used to direct the trimming. After function E_i is trimmed, it is only an approximation of the original function. Specifically, the energy components of any adjacent points (recall that the points are ordered on the time component) differ by at least a factor of δ. The choice of $\delta = (1+\epsilon)^{\frac{1}{n}} - 1$ (ϵ is a parameter of the scheduling algorithm) at Line 13 in Figure 4 will be clear at the end of this section. Figure 5 shows the trimming procedure.

PROCEDURE TRIM($\mathbb{F} = [\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_{|\mathbb{P}|}], \delta$)
1. $\hat{\mathbb{F}} := \{\mathbb{P}_1\}$
2. $l := \mathbb{P}_1$
3. for $i := 2$ to $|\mathbb{F}|$ do
4. if $l.e > (1+\delta)\mathbb{P}_i.e$ then
5. append $\mathbb{P}_i$ onto the end of $\hat{\mathbb{F}}$
6. $l := \mathbb{P}_i$
7. return $\hat{\mathbb{F}}$

END

Figure 5. The TRIM procedure

The function approximation achieved by the trimming procedure is inspired by [4] and was successfully used in [23]. Thus, we only sketch its analysis for the sake of completeness. Interested readers can refer to [4, 23] for more in-depth details.

Before computing the number of points in E_i after trimming, we prove an important lemma. Let E'_i ($i = 1, 2, \ldots, n$) be the step functions obtained if there is no trimming (i.e., Line 13 in Figure 4 is omitted). That is, E'_i is the functions returned by the optimal algorithm. By comparing E'_i and E_i, we can bound the amount of error introduced by the trimming procedure.

Lemma 1 *For every point $\mathbb{P}' \in E_i'$ where $1 \leq i \leq n+1$, there exists a point $\mathbb{P} \in E_i$ such that $\mathbb{P}'.e \leq \mathbb{P}.e \leq (1+\delta)^{n+1-i}\mathbb{P}'.e$ and $\mathbb{P}'.t \geq \mathbb{P}.t$.*

Proof 1 *This lemma is equivalent to $E_i(t) \leq (1+\delta)^{n+1-i}E_i'(t)$ for any value of t. The proof is by induction on i and the base case for $i = n+1$ obviously holds from Line 1 in Figure 4. In the induction step for E_i, we inspect Line 6 in Figure 4. From the hypothesis, $E_{i+1}(t)$ is within a factor of of $(1+\delta)^{n-i}$ of $E_{i+1}'(t)$. All the operations at Line 6 will preserve this property. After the trimming operation, the factor will be only increased by $(1+\delta)$, which will make $E_i(t)$ with a factor of $(1+\delta)^{n-i+1}$ of $E_i'(t)$.*

Using functions E_i' will lead to expected energy consumption of $E_1'(D)$ and using functions E_i will lead to expected energy consumption of $E_1(D)$. From Lemma 1, we have $E_1(D) \leq (1+\delta)^n E_1'(D)$. Since we choose δ to be $(1+\epsilon)^{\frac{1}{n}} - 1$, we have $E_1(D) \leq (1+\epsilon)E_1'(D)$.

To compute the upper bound of the number of points in E_i, we note that after the trimming procedure, the energy components of any two adjacent points differ by at least a factor of δ. Let the leftmost point in E_i be denoted by $\mathbb{P}_l$ and the rightmost point in E_i be denoted by $\mathbb{P}_r$. Thus, we have

$$\mathbb{P}_l.e > (1+\delta)^{|E_i|-1}\mathbb{P}_r.e$$

By plugging in $\delta = (1+\epsilon)^{\frac{1}{n}} - 1$ and some algebraic manipulations, we will obtain $|E_i| = O(\frac{n \log \lambda}{\epsilon})$ where $\lambda = \frac{\mathbb{P}_l.e}{\mathbb{P}_r.e}$. Thus, the number of points in E_i is upper bounded by a polynomial in $\frac{1}{\epsilon}$.

5 Scheduling for General Task Graph

In this section, we present the scheduling algorithm for general task graphs. For general task graphs, there exists not only parallelism in time (Y-oriented load), but also parallelism in space (which we call X-oriented load). Thus, we first study X-oriented load by relaxing the application and power models, as in Section 4.1, to gain insight into the problem. Then we provide two heuristics to reduce the complexity of scheduling general task graphs. Finally, we present the scheduling algorithm.

5.1 X-Oriented Load

We relax the application and power models in the same way as in Section 4.1: We look at the problem of energy minimization of the load τ subject to the deadline constraint, D, and assume that all c cycles of τ can be executed in parallel (*X-oriented load* (Figure 2(c)). Parallel processing is a natural approach to satisfying the deadline constraint and load balancing is desired due to the convexity of the power function. Suppose that x ($x \leq \mathbb{N}$) processor cores are used to execute this load. Thus, each processor core is assigned $\frac{c}{x}$ cycles and its corresponding speed is $\frac{c/x}{D} = \frac{c}{xD}$. The static energy consumption is xC_0D and the dynamic energy consumption is $xC_1\left(\frac{c}{xD}\right)^3 D$. Therefore, the total energy consumption is $e_X(x) = xC_0D + \frac{C_1c^3}{x^2D^2}$, which is very similar to (2). The optimal number of active cores for executing τ is

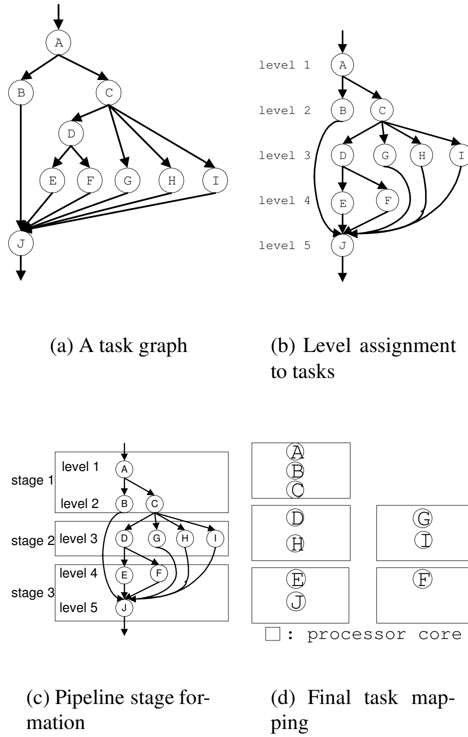
$$x^* = \sqrt[3]{\frac{2C_1}{C_0}} \cdot \frac{c}{D} \quad (5)$$

Thus, we can obtain very similar result, that is, starting from uniprocessor, increasing the degree of parallel processing can initially reduce the energy consumption by reducing the dynamic energy of the cores while satisfying the deadline constraint D. However, the energy consumption will start to increase after the degree of parallelism increases past certain value due to the static energy used by too many cores. The optimal degree of parallelism strikes a balance between static and dynamic power.

5.2 Scheduling Heuristics

A general task graph can be roughly regarded as a mixture of X-oriented load and Y-oriented load. For X-oriented load, we use parallel processing to reduce energy consumption while satisfying the deadline constraint; for Y-oriented load, we use pipelining to reduce energy consumption while satisfying the throughput requirement. Thus, the first step of our scheduling algorithm is to identify the X-oriented load and Y-oriented load of the task graph. To this end, our *first heuristic* is to use the classical topological sort to assign a level to each node in the task graph. The level of a node (task) is equal to 1 plus the length of the longest path from the source to this node. By assigning a level to each node in the task graph, we essentially morph the task graph into a two-dimensional structure. Figure 6(a) and 6(b) show an example of task graph morphing. The tasks on the same level represent the X-oriented load and the tasks across different levels represent the Y-oriented load.

To match the two-dimensional structure of the morphed task graph, we conceptually consider the processor cores to form a two-dimensional logical structure (Figure 6(d)). In mapping the task graph onto the processor cores, the X-dimension is used to map the X-oriented load and Y-dimension is used to map the Y-oriented load, while considering energy consumption in both dimensions. The logical arrangement of processor cores makes the underlying logical tiled structure the same as the structure of the task graph, which will make the mapping process computationally tractable. Our *second heuristic* is to let a row of processor cores in the logical tiled structure correspond to a



(a) A task graph

(b) Level assignment to tasks

(c) Pipeline stage formation

(d) Final task mapping

Figure 6. An example of scheduling for general task graphs

pipeline stage in the mapping process, and we only allow contiguous levels of the morphed task graph to be mapped to the same pipeline stage. Figure 6(c) shows a possible mapping from levels to pipeline stages, and Figure 6(d) shows possible mapping from tasks to processor cores on each pipeline stage.

5.3 The Scheduling2D Algorithm

Unlike previous work on energy-aware task graph scheduling, which separated task mapping and speed scheduling, we interweave task mapping and speed scheduling because they are closely correlated. There are two types of mappings, each corresponding to a dimension. The first type of mapping is called *Y-mapping*, which is performed along the Y-dimension (pipelining dimension). Performing Y-mapping includes: (i) determining the optimal number of pipeline stages; (ii) allotting time to each stage while guaranteeing that the allotted time for each stage is no greater than T and that the sum of the allotted times for all stages is no greater than D; (iii) mapping levels to pipeline stages. The second type of mapping is called *X-mapping*, which is performed along the X-dimension (parallel processing dimension) for each stage. Performing X-mapping for each stage includes: (i) determining the optimal number of active cores for the stage; (ii) mapping tasks to active cores; (iii) deciding execution speed for each task while guaranteeing that all tasks finish executing and transferring data to their successors within the allotted time for the stage. Note that task speed scheduling occurs during X-mapping. Because of the mapping along two dimensions, we call the scheduling algorithm Scheduling2D.

We first explain X-mapping since it is less involved. X-mapping is mostly the classical multiprocessor scheduling problem (which is NP-hard) because if the number of active cores for a stage is known, we can apply the classical list scheduling algorithm to approximate the load-balancing mapping, and then apply the techniques[1] in [7, 13, 3] to obtain the execution speed for each task. As to the problem of determining the optimal number of active cores to be used in a pipeline stage, it is straightforward to use a brute-force approach to check every possible number of active cores and find out the number of cores resulting in the minimum energy consumption. However, this approach is not suitable for large number of tasks. We propose to apply hill-climbing method to search for the best solution using Formula (5) as the starting estimation on the optimal number of active cores. This approach has much lower time complexity than the brute-force approach and our experiments show that the solutions obtained by this approach are very close to those obtained by the brute-force approach.

Y mapping is very similar to scheduling linear task graphs if we treat each level in general task graphs as a task in linear task graphs. However, Y-mapping cannot be performed alone because it requires knowledge from performing X-mapping. Next, we will describe the details of Y-mapping.

Suppose that the total number of levels is L. Let the vector-valued function $E_i(t) = [e, q, j, d]$ ($i \leq j \leq n$ and $0 < d \leq t$) denote the scheduling of the tasks on level i to level n given an allotted time t (end-to-end delay from level i to level L). In this scheduling, e denotes the energy consumption of the tasks on level i to level L, q denotes the number of stages for the tasks on level i to level L, j indicates that level $i, i+1, \ldots, j$ are mapped to the first stage of the q stages, and d is the time allotted to that stage (including the delay resulting from communication between that stage and the next stage), while level $j+1$ to level L are mapped to subsequent $q-1$ stages and the mapping information can be recursively obtained from $E_{j+1}(t-d)$. We can see that the definition of $E_i(\cdot)$ is very similar to that in Section 4.2.3 if we associated levels for general task graphs with tasks for linear task graphs. The scheduling algorithm, Scheduling2D, for general task graphs is shown

[1] All these techniques take into consideration communication among tasks

in Figure 7. Scheduling2D can be regarded as an extension to Scheduling1D (Figure 4). The difference between these two algorithms is the scheduling for the first stage of the q stages.

ALGORITHM Scheduling2D(ϵ)
1.use topological sort to assign a level to each task
2.//L is the total number of levels
3. $E_{L+1} := \{(0,0)\}$
4. for $i := L$ downto 1 do
5. //compute E_i
6. for $j := i$ to L do
7. compute c as the sum of cycles of the tasks on level i through level j
8. compute m as the maximum number of tasks on level i through level j
9. $I := \frac{c}{mf_M}$
10. for $k := 1$ to $\lfloor \frac{T}{I} \rfloor$ do
11. $d := kI$
12. $\hat{E}_{i,j,d}$:=XMAP$(i,j,d) +_e (d +_t E_{j+1})$
13. $E_i := \bigcup_{j=i,\ldots,L,d=1,\ldots,T} \hat{E}_{i,j,d}$
14. $E_i := TRIM(E_i, (1+\epsilon)^{\frac{1}{L}} - 1)$
END

PROCEDURE XMAP(i,j,d) //perform X-mapping
15.use formula (5) to estimate number of cores
16.use an algorithm from [7, 13, 3] to perform mapping and speed scheduling for the tasks on level i to level j subject to real-time constraint d
17.use hill-climbing to search for better solution
18.return the minimum energy consumption
END

Figure 7. The Scheduling2D algorithm for general task graphs

For linear task graphs, because a stage corresponds to a single processor and all tasks in the same stage have the same speed, the time allotted to the first stage only has M possibilities, each corresponding to one of the available M discrete speeds. However, for general task graphs, the scheduling for the first stage is a case of X-mapping in which multiple processors may be used and different tasks may have different speeds. Enumerating every possible number of processors and possible speed for each task will result in exponential number of schedules for the first stage. Note that not all of such schedules are useful because if a schedule consumes more energy and uses more time than another schedule, then the former is useless.

Our approach is to use the alloted time for the first stage directly to decide the schedule. For any given allotted time, X-mapping will attempt to find the schedule with minimum energy consumption. We need to discretize the allotted time in order to make the scheduling for the stage tractable. We use a heuristic to choose the discretization interval. For a given stage, let c be the sum of the cycles of all tasks in this stage and m be the maximum degree of parallelism for this stage (equal to the maximum number of tasks on any level that is mapped to this stage). We use the discretization interval of $\frac{c}{mf_M}$, which can be regarded as the minimum possible allotted time for this stage.

6 Experimental Results

In this section, we evaluate our proposed scheduling algorithms through simulations. Due to space limit, we only show the evaluation results for the Scheduling2D algorithm. The purpose of the evaluation is to quantify the gains of our algorithms over the previously existing algorithms for the problems similar to STREAM-CMP since no other work has proposed a solution to STREAM-CMP. Multiple task graphs and different power models were employed in the evaluation, as follows.

Task graphs: Both synthetic and real-world task graphs were used in the experiments. The synthetic task graphs were generated by TGFF v3.0 [14] using the sample input files that come with the software package. A real-world task graph is obtained from automatic target recognition (ATR), which is a streaming application that does pattern matching of targets in images. A typical platform for ATR is unmanned autonomous vehicle (UAV). ATR must sustain a required incoming rate of images and process each image within a required amount of time to meet UAV mission requirement[2].

The task graph of ATR is different for different number of target detections in an image. We chose the one corresponding to 3 target detections to be used in our experiments, as shown in Figure 8. Each of the three paths, $B \rightarrow E \rightarrow H \rightarrow K$, $C \rightarrow F \rightarrow I \rightarrow L$, and $D \rightarrow G \rightarrow J \rightarrow M$, corresponds to the processing for one target detection. This is because most of the images in the data set has 3 target detections. While this paper focuses on static scheduling on fixed task graphs, the case for task graphs varying from request to request is left for future work.

Power models: For the processor model in the experiments, we used Intel XScale [22], which provides 5 discrete frequencies: 150 MHz, 400 MHz, 600 MHz, 800 MHz, and 1 GHz. The static power of the processor was varied to evaluate the effect of static power on the scheduling.

[2]The documentation of ATR explicitly specifies these two QoS requirements.

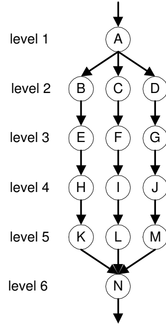


Figure 8. Task graph of ATR

Different values of static power result in different processor power model. For communication cost, we used a transmission rate of 20 Gbytes/s and the transmission power is set to 20% of maximum processor power when the communication link is fully utilized [20].

Methodology: The X-mapping in our Scheduling2D algorithm is based on the S-SPM algorithm [13] because of its low time complexity and reasonably good performance. Also, we set the parameter ϵ to 0.05. We chose the latest work [3] on a subproblem of STREAM-CMP, which assumes that the period is equal to the deadline, to be the baseline against which Scheduling2D compared. A convex programming based approach[3] was used in [3] to obtain the execution speed for each task given the task mapping. Since it does not consider turning processor core on/off, we enhanced it by trying all possible number of processor cores to find the minimum energy consumption. We used the classical earliest task first (ETF) list scheduling heuristic [24] to perform the task mapping.

We tested the two algorithms, baseline and Scheduling2D, for different power models, different deadline constraints, and different throughput requirements. For the power model, we varied the static power to reflect the percentages of the static power in total power being 22%, 44%, and 67% for the 70nm, 50nm, and 35nm technologies, respectively [5]. For the period T, we chose 10 values evenly distributed between the shortest possible execution time (time to execute the critical path of the task graph using the maximum speed) and half of the time to execute the critical path of the task graph using the minimum speed. For each value of T, we set the deadline D to $2T, 3T, \ldots$, until the longest possible execution time. Note that the baseline algorithm takes T as its deadline constraint because $T < D$.

Results: Table 1 shows the energy savings for all experiments, which show that Scheduling2D achieves significant energy savings over the baseline. The average energy savings are 24%, 14%, 11% for 70nm, 50nm, 35nm technologies, respectively. In general, it can be observed that as the static power increases, the energy saving obtained by Scheduling2D decreases. This is because high static power will force both algorithms to use fewer number of processor cores, and thus the optimization space is reduced. The average energy savings in Table 1 are conservative in the sense that their values are lowered by some cases that cannot be optimized by Scheduling2D. We analyzed the experimental results and observed that for a given task graph, as the period increases, the energy saving decreases. This is because when the period is large, there is no need to use pipelining and Scheduling2D will essentially act as the baseline.

[3] In some of the experiments, the convex program solver (called *ipopt*) that we used could not produce a solution due to its iteration limit. When that happened, we simply used S-SPM [13] in its place.

7 Conclusions and Future Work

In this paper, we study the problem of minimizing energy consumption of streaming applications running on CMPs while satisfying throughput and response time requirements. We propose scheduling algorithms that use parallel processing and pipelining to reduce energy consumption while meeting the two QoS requirements. Our work shows that simultaneously exploiting two QoS requirements saves energy. The experimental results show that our proposed algorithms can achieve significant energy savings for streaming applications running on CMPs.

References

[1] Cell broadband engine architecture documentation, 2005. http://www.ibm.com/developerworks/power/cell.

[2] Intel developer forum, 2006. http://www.intel.com/pressroom/kits/events /idf-fall_2006/pdf/idf 09-26-06 paul otellini keynote transcript.pdf.

[3] A. Andrei and M. Schmitz and P. Eles and Z. Peng and B. Al-Hashimi. Simultaneous communication and processor voltage scaling for dynamic and leakge energy reduction in time-constrained systems. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, 2004.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, 2001.

[5] D. Duarte and N. Vijaykrishnan and M. J. Irwin and H-S Kim and G. McFarland. Impact of scaling on the effectiveness of dynamic power reduction schemes. In *Proc. International Conference on Computer Design (ICCD)*, 2002.

[6] E.N. Elnozahy and M. Kistler and R. Rajamony. Energy-efficient server clusters. In *Proc. Workshop on Power-Aware Computer Systems (PACS'02)*, 2002.

[7] F. Gruian and K. Kuchcinski. Lenes: Task scheduling for low-energy systems using variable supply voltage processors. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2001.

Table 1. Energy savings of Scheduling2D over baseline

Task graph	# of tasks	# of edges	70 nm		50 nm		35 nm	
			avg.	max	avg.	max	avg.	max
creds1	9 - 13	10 - 18	27.65%	38.54%	18.74%	26.48%	16.97%	29.29%
kbasic_tables	19	21	22.38%	39.17%	12.58%	21.37%	7.98%	11.74%
kbasic_task	18 - 64	18 - 79	27.37%	43.21%	16.15%	31.97%	9.49%	20.44%
kseries_parallel_xover	21 - 38	24 - 41	22.53%	45.18%	14.39%	34.25%	9.34%	26.71%
kseries_parallel	20 - 62	19 - 61	23.41%	39.61%	13.76%	27.02%	13.01%	27.56%
packets	6 - 8	5 - 8	26.79%	42.32%	15.51%	28.95%	12.85%	31.69%
simple	11 - 24	12 - 32	24.66%	46.69%	13.04%	34.53%	11.01%	34.32%
ATR	14	15	18.56%	39.14%	7.51%	15.92%	5.76%	10.38%

[8] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[9] M.T. Yang and R. Kasturi and A. Sivasubramaniam. A pipeline-based approach for scheduling video processing algorithms on now. *IEEE Trans. Parallel and Distributed Systems*, 14(2), 2003.

[10] N. S. Kim and T. Austin and D. Blaauw and T. Mudge and K. Flautner and J. Hu and M. Irwin and M. Kandemir and N. Vijaykrishnan. Leakage current: Moore's law meets static power. *Computer*, 36(12):65–77, 2003.

[11] N. S. Kim and T. Kgil and K. Bowman and V. De and T. Mudge. Towards power-optimal pipelining and parallel processing under process variations in nanometer technology. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD)*, 2005.

[12] P. D. Hoang and Jan M. Rabaey. Scheduling of dsp programs onto multiprocessors for maximum throughput. *IEEE Trans. Signal Processing*, 41(6), 1993.

[13] R. Mishra and N. Rastogi and D. Zhu and D. Mossé and R. Melhem. Energy aware scheduling for distributed real-time systems. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[14] R. P. Dick and D. L. Rhodes and W. Wolf. Tgff: Task graphs for free. In *Proc. Sixth International Workshop on Hardware/Software Codesign*, Mar. 1998.

[15] R. Xu and D. Zhu and C. Rusu and R. Melhem and D. Mossé. Energy-efficient policies for embedded clusters. In *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 2005.

[16] S. Banerjee and T. Hamada and P. M. Chau and R. D. Fellman. Macro pipelining based scheduling on high-performance heterogeneous multiprocessor systems. *IEEE Trans. Signal Processing*, 43(6), 1995.

[17] S. Baruah and J. Anderson. Energy-efficient synthesis of periodic task systems upon indentical multiprocessor platforms. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, pages 428–435, Tokyo, Japan, Mar. 2004.

[18] V. Bharadwaj and D. Ghose and T.G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6:7–18, Jan. 2003.

[19] W. Thies and M. Karczmarek and S. Amarasinghe. Streamit: A language for streaming applications. In *Proc. International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[20] H. S. Wang, X. Zhu, L. S. Seh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *Proc. International Symposium on Microarchitecture*, 2002.

[21] N. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, Reading, MA, 1993.

[22] Intel XScale Microarchitecture, 2005. http://developer.intel.com/design/intelxscale/benchmarks.htm.

[23] R. Xu, C. Xi, R. Melhem, and D. Mossé. Practical PACE for Embedded Systems. In *Proc. ACM International Conference on Embedded Software (EMSOFT)*, Pisa, Italy, September 2004.

[24] Y. Chow and F. Anger and C. lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Computers*, 18(2), 1989.