Optimizing Throughput and Latency under Given Power Budget for Network Packet Processing*

Jilong Kuang and Laxmi Bhuyan
Computer Science & Engineering Department
Unviersity of California, Riverside
900 University Ave, Riverside, CA 92521, USA
{jkuang,bhuyan}@cs.ucr.edu

***Abstract*—Current state-of-the-art task scheduling algorithms for network packet processing schedule the program into a parallel-pipeline topology on network processors to maximize the throughput. However, there has been no existing work targeting power budget for packet processing on off-the-shelf multicore architectures. As energy consumption, reliability and cooling cost for packet processing systems become increasingly important, it is necessary to integrate power-awareness into a scheduler to meet the power budget.**

In this paper, we propose a novel scheduling algorithm to optimize both throughput and latency given a power budget for network packet processing on multicore architectures. This algorithm addresses power-aware parallel-pipeline scheduling problem by applying per-core DVFS to optimally adjust frequency on each core. We implement our algorithm on an AMD machine with two Quad-Core Opteron 2350 processors and compare the results with existing algorithms given the same power budget. For six real packet processing applications, our algorithm improves throughput and reduces latency by an average of 64.6% and 25.2%, respectively.

I. Introduction

[1]The explosive growth of network bandwidth requires orders-of-magnitude increase in packet processing throughput. In addition, many applications (e.g., fast IP-lookup, real-time voice and video) demand not only high throughput but also low latency.

The advent of commodity multicore platforms in the market has opened a new era of computing for network applications. More and more network packet processing systems have been developed on such platforms based on general-purpose processors (e.g., Intel's Xeon [1] and AMD's Opteron [2]), network processors (e.g., Intel's IXP platform [3]) and programmable logic devices (e.g., NetFPGA [4]). These systems incorporate parallel scheduling to exploit packet and task-level parallelism for higher throughput and lower latency.

Besides parallelization, pipelining is another important technique to further improve throughput in packet processing on multicore architectures. The advantage of pipelining comes from overlapping periodic executions from different threads, which enhances the whole system throughput. Therefore, current task scheduling algorithms for packet processing usually combines parallelization and pipelining to maximize the throughput, such as Random [5] and Bipar [6].

However, there is no existing work targeting the power budget for parallel-pipeline scheduling. Previous power-aware algorithms either have not considered latency (e.g., [7] [8]), or have not explored the parallel-pipeline topology for task scheduling (e.g., [9] [10] [11]). It is both interesting and challenging to develop novel task scheduling algorithms that will increase throughput and reduce latency. As energy consumption, reliability and cooling cost for network packet processing systems become increasingly important, the scheduling algorithm must consider power constraint.

Traditionally, there are two popular approaches to deal with power-awareness. One is Power Management (PM), which saves power by dynamically turning on or off processors to adapt to the varying resource demand. The other is Dynamical Voltage and Frequency Scaling (DVFS), which trades power with computation time by adjusting processor voltage and frequency. Because PM can not be applied to task scheduling [8], we resort to DVFS to integrate power-awareness into parallel-pipeline scheduling. Widely-used DVFS schemes not only include chip-wide DVFS such as IBMs TPMD [13] and Intels Foxton technology [14], but also include per-core DVFS such as AMDs Quad-Core Opteron [15], which can perfectly support our algorithm.

To the best of our knowledge, we are the first to address the power budget issue for parallel-pipeline scheduling in network packet processing. Our algorithm works as follows. In the first step, we reduce power by lowering the frequency on parallel nodes without compromising throughput or latency. This goal can only be achieved for parallel-pipeline topology. If the resulting power consumption still exceeds the power budget, we go to step two. In the second step, we reduce the power with throughput unchanged and minimal latency increase by optimally adjusting the frequency on each core. If both step one and step two can not satisfy the power constraint, we go to step three. In the third step, we reduce the power with minimal throughput and latency performance loss adopting similar approach as in step two. Step three and step two are

[1]*This research was partly supported by NSF grants CNS 0509440, CCF 0811834 and CNS 0832108

recursively executed until the power budget is finally met.

It is also important to note that this algorithm is generally applicable to any type of multicore packet processing systems ranging from general-purpose processors to network processors and programmable logic devices as long as per-core DVFS is available. In addition, the scheduling granularity can also vary from fine-grain (e.g., basic block) to course-grain (e.g., loop, function, task) in practice. We implement our algorithm as well as five other conventional algorithms for six real packet processing applications chosen from NetBench [16] and PacketBench [17] on an AMD machine with two Quad-Core Opteron 2350 processors [2]. The five chosen algorithms are Clock Gating (CG) [8] in PM category, and S-SPM [9], PDP-SPM [10], G-SPM [11] and P-SPM [11] in DVFS category. Compared to existing algorithms given the same power budget, our algorithm exhibits substantially better throughput and latency by an average of 64.6% and 25.2%, respectively.

The rest of this paper is organized as follows. Section II introduces preliminaries, including the application model, the power model and the problem statement. Section III presents the optimization model and the three-step power-aware scheduling algorithm in detail. Section IV describes the experimental framework and shows the performance evaluation. Section V addresses the related work and Section VI concludes this paper.

II. Preliminaries

A. Application Model

We define a task graph as a weighted DAG by tuple G=(V, E, C, T), where V=$\{n_i$,i=1:$v\}$ is the set of nodes and v=$|V|$, E=$\{e_{i,j}$=$<n_i, n_j>\}$ is the set of communication edges and e=$|E|$. C is the set of edge communication times and T is the set of node computation times. $c_{i,j}$ is the communication time on edge $e_{i,j}$ and t_i is the computation time on node n_i. Figure 1 (a) gives a DAG example.

We assume the application is scheduled into a parallel-pipeline topology from the DAG based on a static scheduling policy. Suppose there are N nodes running on N processors with S pipeline stages and M_i parallel nodes in stage S_i. The stage time T_i is the maximal node computation time in S_i. Figure 1 (b) illustrates an example of such scheduling. In addition, we can label each node in the parallel pipeline scheduling as node $N_{i,j}$, where i refers to the stage number and j refers to the node order within a certain stage starting from the top.

We then define the two objective metrics in the paper: *throughput* and *latency*. In pipeline topology, *throughput* is calculated by the inverse of the longest stage time $\frac{1}{T_{max}}$, where $T_{max} = Max\{T_i\}$, and *latency* is computed as the sum of stage time $T_i, i = 1, 2, ..., S$. We ignore communication time in this paper without losing validation because it can be considered constant in DVFS scheme [11].

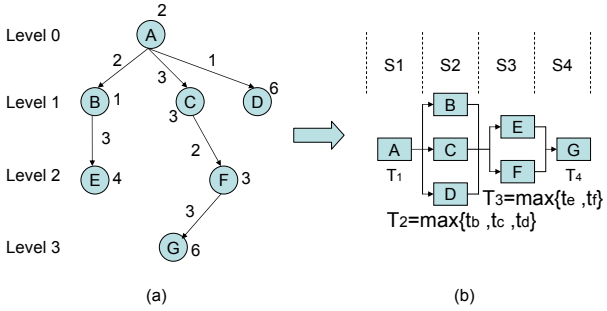

Fig. 1: A parallel-pipeline scheduling from DAG.

B. Power Model

Consider that task T consists of C clock cycles on processor P, which runs at voltage V and frequency f. We assume that C does not change with different V and f. For a given voltage V, processor P has an average power consumption Pow. It is known that processor power consumption is dominated by dynamic power dissipation given by: $Pow = K_a \cdot f \cdot V^2$, where K_a is a task/processor dependent factor determined by the switched capacitance.

The energy consumed by executing task T on processor P is computed as: $E = C \cdot \frac{Pow}{f}$. We can rewrite it as: $E = C \cdot E_{f,V} = C \cdot K_a \cdot V^2$, where $E_{f,V}$ is the average cycle energy. From this we can see that lowering the voltage would yield a drastic decrease in energy consumption. The frequency f is almost linearly related to the voltage: $f = K_b \cdot \frac{(V-V_T)^2}{V}$, where V_T is the threshold voltage and K_b is a constant. For a sufficiently small threshold voltage, the frequency is approximated to $K_b \cdot V$.

C. Problem Statement

Assume the initial parallel-pipeline scheduling with the highest frequency produces the best throughput and latency, which defines the upper bound for throughput and the lower bound for latency. Under such assumption, we give the problem statement as follows: *Given the parallel-pipeline scheduling and power budget, how to optimize the per-core frequency to maximize the throughput and minimize the latency for multicore architectures*.

The problem we want to optimize is: given a set of N cores $P_{1...N}$ that can each run at Q different frequency levels $F_{1...Q}$, find the best selection of frequency levels for all the cores that maximizes the throughput and minimize the latency, subject to the constraint that the total power is less than or equal to POW_{budget}.

We start with the objective functions for throughput and latency in Equation 1 and 2.

$$Maximize \ \ Th = \frac{1}{T_{max}} \quad (1)$$

$$Minimize \ \ L = T_1 + T_2 + ... + T_S \quad (2)$$

The execution time of each node t_i can be calculated as $t_i = \frac{C_i}{f_i}$, where C_i is the clock cycles of task n_i and f_i is the frequency on that core. $f_{1..N}$ are the set of frequency levels we are trying to find. Thus, throughput Th and latency L can be rewritten in Equation 3 and 4.

$$Th = \frac{1}{T_{max}} = \frac{1}{Max\{\frac{C_i}{f_i}\}} = Min\{\frac{f_i}{C_i}\} \quad (3)$$

$$L = T_1 + T_2 + ... + T_S = (\frac{C_1}{f_1} + \frac{C_2}{f_2} + ... + \frac{C_S}{f_S}) \quad (4)$$

Next, we define the constraint, which specifies that the total power is less than or equal to POW_{budget}. According to [12], we can linearly approximate the power equation as $p_i = b_i f_i + c_i$, where b_i and c_i can be obtained by the linear approximation of the power dependence on frequency [12]. The constraint equation can then be written in Equation 5, where all the $c_{1..N}$ constraints are folded into c:

$$b_1 f_1 + b_2 f_2 + ... + b_N f_N + c \leq POW_{budget} \quad (5)$$

However, combining Equation 3 and 4 with Equation 1 and 2, we can see that these objective functions are not linearly solvable by the conventional Linear Programming (LP) as in [12]. The two reasons are: 1) The total throughput in parallel-pipeline scheduling is not just a linear summation of the partial throughput from all processors. Instead, it only depends on the inverse of the longest stage time. 2) The latency from each stage is inversely proportional to the frequency, which is also non-linear. As a result, we form a new optimization model to this problem in Section III and propose a novel algorithm to address that model.

III. Power-Aware Scheduling Algorithm

In this section, we address the power-aware scheduling algorithm that is capable of optimizing both throughput and latency for parallel-pipeline topology. We first introduce the optimization model, followed by the three-step recursive algorithm in detail. Then, we address the practical issues with discrete frequency levels.

A. Optimization Model

As mentioned in Section II, we assume the initial parallel-pipeline scheduling with the highest frequency sets the upper bound for throughput and the lower bound for latency. Therefore, if the initial power consumption is already less than or equal to the power budget, the initial scheduling itself is acceptable.

Otherwise, we can reduce the frequency on each core to satisfy the power constraint. Hence, we express the optimization problem as follows: *given the initial throughput Th_0 and latency L_0 with the highest frequency and power budget, minimize the throughput and latency performance loss by optimally adjusting the frequency on each core.*

Equation 6 and 7 show the two new objective functions. They are inherently equivalent to the one introduced in Section II (Equation 1 and 2) but expressed from the complementary direction. We prioritize throughput to latency in our model because: 1) Throughput is still the most important metric for current network packet processing systems. 2) Latency is only required to meet the deadline requirement instead of minimization for most systems.

$$Minimize \ \Delta Th = Th_0 - Th \quad (6)$$

$$Minimize \ \Delta L = L - L_0 \quad (7)$$

B. A Three-Step Recursive Algorithm

Step One: In the first step, we reduce the power without compromising throughput or latency by keeping the pipeline stage time $T_i, i = 1, 2, ..., S$ unchanged. We define a critical node as the node in a pipeline stage that dominates the computation time. Therefore, the computation time of a critical node is equal to the pipeline stage time ($t_i = T_i$). For each stage S_i, we increase the computation time of non-critical nodes in that stage to the length of T_i. Since all stage times remain the same, the throughput and the latency will also keep unchanged during this step.

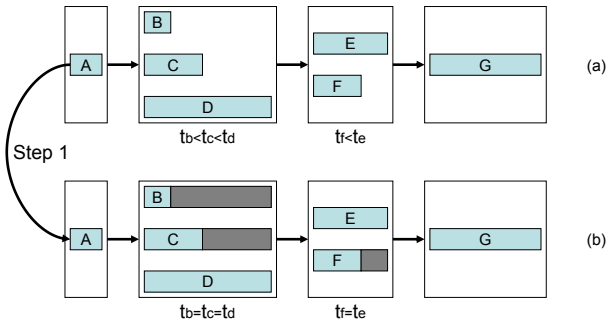

Fig. 2: Illustration of the first step of the algorithm. Grey area represents the extension of computation time. Power consumption is saved on nodes B, C and F.

Figure 2 illustrates the first step, where the length of a node represents its computation time. Figure 2 (a) is the same as Figure 1 (b). In Figure 2 (b), we can see that node B and node C are extended to the length of node D in that stage. Similarly, node F is extended to match node E in the next stage. Because increasing computation time and lowering frequency essentially refer to the same meaning [12], we use them interchangeably. As a result, power consumption is reduced by lowering the frequency on nodes B, C and F.

To quantify the power savings in the first step, we derive the formula in Equation 8 to calculate ΔP. For each node $N_{i,j}$, $\Delta t_{N_{i,j}}$ represents the difference of computation time.

$$\begin{aligned}\Delta P &= \Delta P_1 + \Delta P_2 + ... + \Delta P_N \\ &= \sum_{i=1}^{S}\sum_{j=1}^{M_i} \Delta P_{N_{i,j}} \\ &= \sum_{i=1}^{S}\sum_{j=1}^{M_i} (b_{N_{i,j}} \cdot (f - f_{new})) \\ &= \sum_{i=1}^{S}\sum_{j=1}^{M_i} (b_{N_{i,j}} \cdot (\frac{C_{N_{i,j}}}{t_{N_{i,j}}} - \frac{C_{N_{i,j}}}{T_i})) \\ &= \sum_{i=1}^{S}\sum_{j=1}^{M_i} (b_{N_{i,j}} \cdot C_{N_{i,j}} \cdot \frac{\Delta t_{N_{i,j}}}{T_i \cdot t_{N_{i,j}}}) \end{aligned} \quad (8)$$

If the resulting power consumption after step one is still larger than power budget, we proceed to step two.

Step Two: In the second step, we reduce the power with throughput unchanged and minimal latency increase. This is achieved by keeping the longest stage time T_{max} unchanged while we increase the stage time of other stages. We denote the stage with T_{max} as the bottleneck stage in the pipeline. Thus, all other stages are non-bottleneck stages.

We define ΔT as the shortest time period by which we can increase the latency. To minimize the latency increase, we iteratively increase the latency by ΔT until the power budget is satisfied or all the stages reach T_{max}. If the former comes true, the algorithm returns and the resulting scheduling guarantees the minimal latency increase, which will be proved shortly. Otherwise, if the latter comes true, we proceed to step three.

In each iteration, we optimally choose a non-bottleneck stage to increase its time from T_i to $T_i + \Delta T$. The candidate stage is chosen by comparing the potential power savings from all non-bottleneck stages. The stage with the largest power reduction will be selected. Because ΔT is the shortest time period that can be increased, and the corresponding power reduction is the largest during each iteration, The algorithm therefore guarantees optimality.

$$\begin{aligned}\Delta P_i &= \Delta P_1 + \Delta P_2 + ... + \Delta P_{M_i} \\ &= \sum_{i=1}^{M_i} (b_i \cdot (f - f_{new})) \\ &= \sum_{i=1}^{M_i} (b_i \cdot (\frac{C_i}{T_i} - \frac{C_i}{(T_i + \Delta T)})) \\ &= \sum_{i=1}^{M_i} (b_i \cdot C_i \cdot \frac{\Delta T}{T_i \cdot (T_i + \Delta T)}) \end{aligned} \quad (9)$$

Intuitively, a stage with more parallel nodes will be a good candidate because more power savings will be available in that stage. In fact, besides the degree of parallelism, other parameters also matter. For a given latency increase ΔT, the potential power savings of stage S_i can be obtained from Equation 9.

Figure 3 illustrates the process of this step. Figure 3 (a) comes from the end of step one as shown in Figure 2 (b). In Figure 3 (b), we extend nodes A, E and F to the length of 6, respectively, to further reduce the power consumption. Now we end up with a scheduling which consists of equal-length pipeline stages.

In fact, Figure 3 shows the maximum power savings by step two, which results in maximal latency increase. Chances are that the actual latency would be lower than what we have seen here if the power budget is less stringent, in which case the algorithm would return earlier before every stage reaches T_{max}. According to Equation 8, we can also obtain the exact power savings in this step.

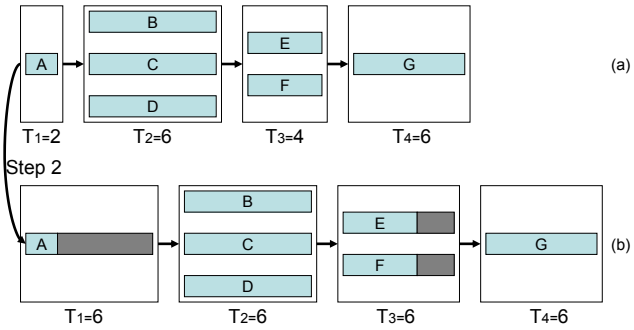

Fig. 3: Illustration of the second step of the algorithm. Grey area represents the extension of computation time. Power consumption is saved on nodes A, E and F.

Step Three: In the third step, we reduce the power by minimizing both the throughput and the latency performance loss. Remember that after step two, every stage has the same stage time T_{max}. Following the same rule of choosing a candidate stage in step two, we optimally choose a stage to further increase its stage time by ΔT. Since the original T_{max} is increased, the throughput is compromised accordingly. However, our algorithm is able to guarantee a minimal performance loss in this scenario.

To optimally choose the candidate stage, we follow the same formula in Equation 9. The only difference is that we need to substitute T_i with T_{max} in the equation. The proof of optimality is in line with that in step two, where the minimal time period increment guarantees that when we satisfy the power budget constraint, the performance loss is minimal.

Figure 4 demonstrates step three. Figure 4 (a) is the result of step two as shown in Figure 3 (b). Suppose the candidate stage at the moment is stage two. We then increase the stage time from 6 to $6 + \Delta T$ for that stage. Notice that all other stages remain unchanged as shown in Figure 4 (b).

After increasing the original T_{max} to $T_{max} + \Delta T$, we go back to step two with the updated T_{max} if further power reduction is needed. The algorithm then recursively executes step two and step three until the power budget is finally met as shown in Figure 5. Algorithm 1 gives the pseudocode for

the entire algorithm.

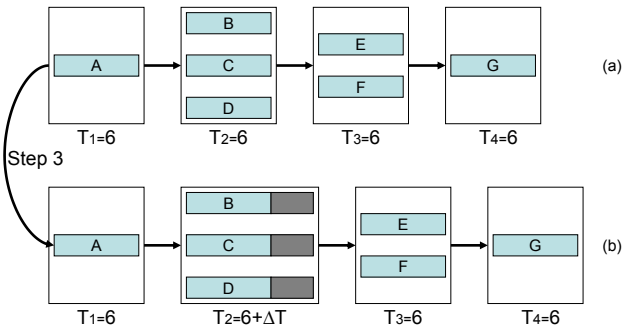

Fig. 4: Illustration of the third step of the algorithm. Grey area represents the extension of computation time. Power consumption is saved on nodes B, C and D.

With respect to the complexity, we conclude that step one has a complexity of $O(N)$, step two has a complexity of $O(\frac{T_{max}-T_{min}}{\Delta T} \cdot S^2 \cdot N)$ and step three has a complexity of $O(m \cdot \frac{T_{max}-T_{min}}{\Delta T} \cdot S^2 \cdot N)$. $\frac{T_{max}-T_{min}}{\Delta T}$ is the maximal number of iterations in step two and m is the maximal recursive times in step three. Because both ΔT and m depend on the number of discrete frequency levels in practice and are thus constant, the total complexity for our algorithm is $O(S^2 \cdot N)$. Therefore, our algorithm will terminate if 1) the power constraint is met or 2) the maximal number of updates is exceeded in either step two or step three. There will be no oscillations occurring in the updates within the while loop as shown in Algorithm 1, which guarantees the convergence of our algorithm.

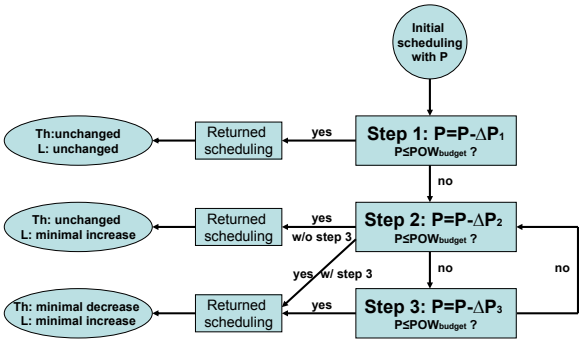

Fig. 5: The power-aware parallel-pipeline scheduling algorithm with the goals for throughput (Th) and latency (L).

C. Practical Issues with Discrete Frequency Levels

So far, we only address the ideal case where the frequency levels are continuous. However, in practice, those values are discrete, which requires some minor corrections in our algorithm.

The essential problem is about the shortest time period ΔT. In practice, the value of ΔT is determined by Equation 10, assuming f_1 and f_2 are two contiguous frequency levels and their difference is Δf. As a result, ΔT depends on the clock cycles C_i running on that processor and the current frequency on that processor.

$$\Delta T_i = \frac{C_i}{f_1} - \frac{C_i}{f_2} = C_i \cdot \frac{\Delta f}{f_1 \cdot f_2} \tag{10}$$

Thus, the following two changes are necessary in practice. First, in step one, we increase the computation time of non-critical nodes for each stage as much as possible without violating the initial stage time. We do not require that all non-critical nodes be increased to the same length of the critical node in that stage as in ideal case. Second, in step two and step three, we use the practical ΔT value obtained from Equation 10 when calculating the potential power savings for each stage.

Algorithm 1 Power-aware parallel-pipeline scheduling

1: **if** $POW \leq POW_{budget}$ **then return**
2: **for** each stage S_i **do** /* Step 1 */
3: **for** each parallel task n_j **do**
4: $t_j \leftarrow T_i$
5: **if** $POW \leq POW_{budget}$ **then return**
6: **while** T_{max} unchanged **do** /* Step 2 */
7: **for** each stage S_i except T_{max} stage **do**
8: calculate ΔP_i according to Equation 9
9: choose stage S_i with Max$\{\Delta P_i\}$
10: **for** each parallel task n_j in S_i **do**
11: $t_j \leftarrow t_j + \Delta T$
12: update POW
13: **if** $POW \leq POW_{budget}$ **then return**
14: **while** $POW > POW_{budget}$ **do** /* Step 3 */
15: **for** each stage S_i **do**
16: calculate ΔP_i according to Equation 9
17: choose stage S_i with Max$\{\Delta P_i\}$
18: **for** each parallel task n_j in S_i **do**
19: $t_j \leftarrow t_j + \Delta T$
20: update POW and T_{max}
21: **if** $POW \leq POW_{budget}$ **then return else goto 6**

IV. Experiments and Evaluation

A. Experimental Framework

Figure 6 shows our experimental framework and flowchart, which consists of two steps. In the first step, we generate the program dependency graph (PDG) with profile information by SUIF/Machine SUIF compilers [18] [19], partition and map the application on an AMD machine with two Quad-Core Opteron 2350 processors [2].

More specifically, the original C program is first converted to the control flow graph (CFG). For the ease of dependency analysis, we include all the functions of an application into one single file. After that, we write a Machine SUIF pass to extract the PDG based on both control and data flow information [20].

Finally, by using the Halt library in Machine SUIF, we profile the program in the basic block level with continuous traffic traces to obtain the average execution time and execution frequency. At last, we schedule the program onto the real machine based on the application model presented in Section II.

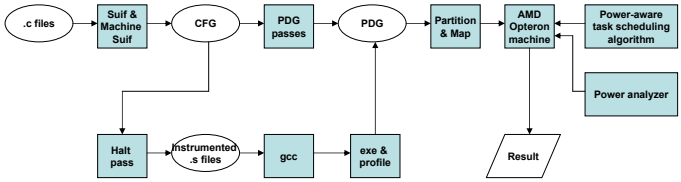

Fig. 6: Experiment framework and flowchart.

In the second step, we apply our power-aware task scheduling algorithm and five other existing algorithms on the initial parallel pipeline scheduling. The predefined frequency levels of the Opteron 2350 processor are shown in Table I. Meanwhile, the default power for each corresponding frequency is also listed in that table (default power refers to the power consumption when the system is not running our experimental applications). We use the EXTECH power analyzer (model 380801 [21]) to get the whole system power.

TABLE I: Frequency(GHz) and power(W) configuration.

Frequency Level	1.0	1.2	1.4	1.7	2.0
Default Power	133.0	134.5	137.0	141.2	142.5

The hardware configuration is set by default as an 8-core machine with the instruction cache size up to $4k$ instructions. Six network applications are chosen from NetBench [16] and PacketBench [17]. Their functionalities and code sizes are listed in Table II. For the code size, we measure them in terms of the number of instructions. The packet trace is from NetBench itself, which contains $10,000$ packets. The routing table used for IPv4-trie is MAE-WEST [17] and the routing table size for DRR, IPchains and Route is set to 128 by default. The input file for URL contains 100 lines of rules.

TABLE II: Packet processing applications.

Application	Functionality	Code Size (ins)
URL	URL-based switching	1428
Flow	Flow classification	3190
IPv4-trie	IPv4 routing based on trie	4596
Route	IPv4 routing based on radix	6600
DRR	Deficit-round robin scheduling	7633
IPchains	Firewall based on IP source	14735

B. Performance Evaluation

We compare our algorithm with five other conventional algorithms to show its performance advantage in optimizing throughput and latency given the same power budget. Latency refers to the average execution time of one packet in microseconds (usec). Throughput is measured by million packets per second (mpps). Power consumption is measured by watts (w) and we use the net power consumed exclusively by our experimental applications as the metric.

In the PM category, we choose Clock Gating (CG) [8] since it also addresses the energy reduction issue in packet processing for network processors. As regard to DVFS, we choose four different static power management schemes for comparison, namely S-SPM [9], PDP-SPM [10], G-SPM [11] and P-SPM [11]. The reason why we compare with them is that their power-aware schemes are all built on top of task scheduling, which are inline with our algorithm. We briefly introduce them as follows:

- CG (Clock Gating): reduces power consumption by turning off processors.
- S-SPM (Simple SPM): distributes global static slack proportionally to the length of the schedule.
- G-SPM (Greedy SPM): allocates global static slack to the first task on each processor.
- P-SPM (Parallel SPM): distributes global static slack according to the degree of parallelism.
- PDP-SPM (Proportional Distribution and Parallelism SPM): distributes global static slack according to the degree of parallelism and exploits the local slack.

C. Power Reduction in Step One

Figure 7 and Table III show the power reduction after step one for six applications compared with the initial power consumption. Four applications have lowered the power by an average of 10.5% and a maximum of 23.1% in Flow. These power savings come from the reduced frequency for non-critical parallel tasks as shown in Figure 2. URL and IPchains consume the same power because their scheduling can not benefit from step one. Notice that during this process, both throughput and latency keep unchanged, which means the power savings come at no cost.

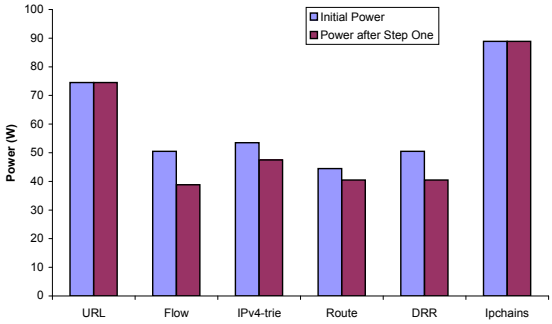

Fig. 7: Power consumption of six applications after step one.

D. Power and Latency Performance in Step Two

Figure 8, Figure 9 and Table III exhibit the power and latency after step two for six applications compared with

TABLE III: Power and latency performance of six applications compared to their initial state.

	URL	Flow	IPv4-trie	Route	DRR	Ipchains	Average
Power reduction percentage after step one	0.0%	23.1%	11.2%	9.0%	19.8%	0.0%	10.5%
Power reduction percentage after step two	9.8%	34.5%	28.0%	23.8%	31.7%	16.5%	24.1%
Latency increase percentage after step two	18.5%	28.9%	29.9%	19.0%	29.9%	20.9%	24.5%

the initial results. From Figure 8 we observe that all six applications have enjoyed power savings by an average of 24.1% through the frequency adjustment for tasks in non-bottleneck stages corresponding to Figure 3. More specifically, Flow achieves the maximum of 34.5% power reduction in this step due to its vastly differing stage times.

From Figure 9 we can see that the latency increase ranges from 18.5% in URL to 29.9% in DRR and IPv4-trie with an average of 24.5%. Although we trade latency with power on the same percentage scale, our algorithm guarantees the minimal latency increase while maintaining the throughput unchanged in this step. Moreover, because we demonstrate the maximal possible power reduction and latency increase in step two, chances are that actual latency performance would be even better if the real power budget is less than what we have achieved here.

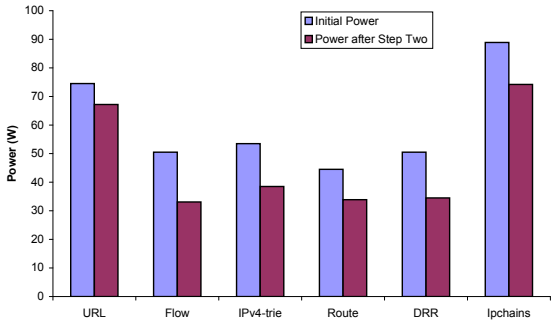

Fig. 8: Power consumption of six applications after step two.

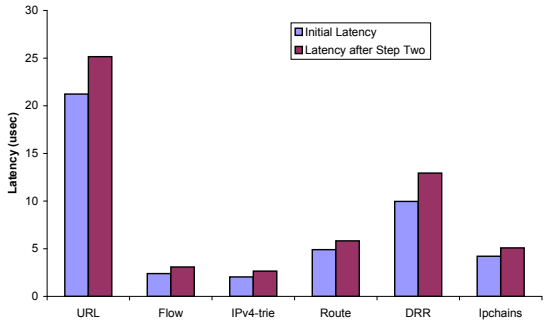

Fig. 9: Latency performance of six applications after step two.

E. Throughput and Latency Comparison with Five Other Algorithms in Step Three

We set the power budget to be 75% of the initial power consumption, so that all three steps in our algorithm will be required to satisfy the power budget constraint. The resulting throughput and latency of six different algorithms are shown in Figure 10 and Figure 11. With respect to throughput, our algorithm outperforms all others uniformly, with an average improvement of 64.6% compared to the lowest throughput for each application. The maximum increase appears in IPv4-trie where we observe a 100.6% improvement. The other five algorithms exhibit fluctuating performance for different applications as shown in Figure 10. This is because each of them has its own shortcoming. For CG, the reduction of active cores results in increased longest stage time, which adversely affects the throughput. For other SPM algorithms, they do not differentiate the bottleneck stage in parallel-pipeline scheduling. Therefore, they can not produce optimal throughput. Our algorithm is capable of achieving better throughput by optimally adjusting the frequency on each core during each step.

In terms of latency, we also observe the advantage of our algorithm from Figure 11. Compared with the highest latency for each application, our algorithm results an average of 25.2% latency reduction and the maximum reduction of 33.2% in Flow. PDP-SPM and P-SPM are the best among other SPM algorithms to produce low latency due to their consideration of parallelism. However, they still suffer from longer latency compared to our algorithm in most cases. Our strength lies in the fact that we iteratively apply step three and step two to guarantee the minimal latency increase, whereas PDP-SPM and P-SPM only greedily reduce frequency in parallel stages. On the other hand, we notice that CG performs better in terms of latency for four applications than our algorithm because it maintains the highest frequency all the time. However, its better latency performance actually comes at the cost of substantial throughput deterioration.

F. Power Budget Sensitivity Performance

Lastly, we analyze the effect of varying power budget on throughput and latency. We study a representative application IPchains for six algorithms by changing the power budget ratio from 1 to $4/8$ of the initial power consumption. From Figure 12 we can see that our algorithm always produces the best throughput. As the power budget decreases, G-SPM suffers most because it always reduces frequencies from the first stage, which happens to be the bottleneck stage in this application. The other four algorithms also follow the same decreasing trend. Although their throughput plummets with different speeds, all of them fall behind our algorithm. On

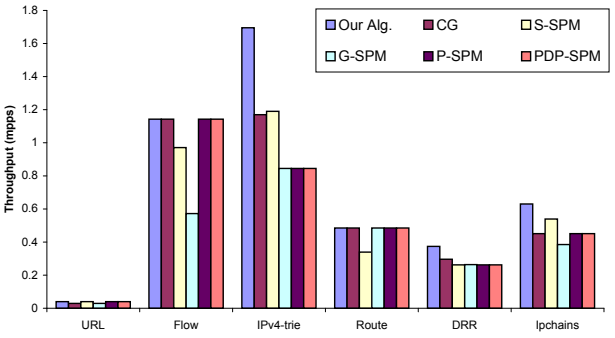

Fig. 10: Throughput performance comparison when power budget is 75% of the initial power consumption.

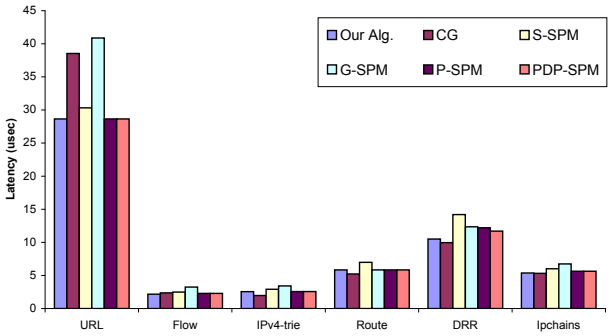

Fig. 11: Latency performance comparison when power budget is 75% of the initial power consumption.

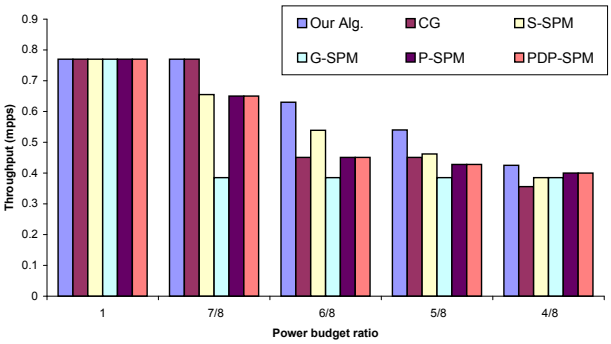

Fig. 12: Throughput performance comparison for IPchains application when power budget varies.

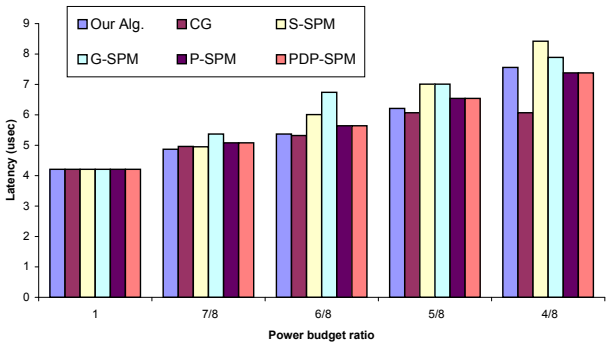

Fig. 13: Latency performance comparison for IPchains application when power budget varies.

an average, our algorithm exhibits 55.8% improvement on throughput compared to the worst algorithm for each application. The maximum improvement appears to be 100.0% over G-SPM when the power budget ratio is $7/8$.

Figure 13 illustrates the increasing trend in latency for six algorithms when the power budget decreases. Compared to the highest latency for each application, our algorithm shows an average of 13.0% reduction of latency with a maximum of 20.3% over G-SPM when the power budget ratio is $6/8$. We observe that P-SPM and PDP-SPM perform better than S-SPM and G-SPM in general, because both P-SPM and PDP-SPM take into consideration the degree of parallelism. However, these two still attain higher latency due to the lack of specialized optimization for parallel-pipeline topology. CG, on the other hand, has slightly lower latency in some scenarios, for it never reduces frequency. However, as shown in Figure 12, CG's throughput deteriorates substantially in those scenarios. Therefore, our algorithm once again proves its advantage in both throughput and latency due to its optimal adjustment of frequency on each core.

V. Related Work

We distinguish the vast literatures in this area by the following three criteria: 1) Whether PM or DVFS is used. 2) Whether task scheduling is considered. 3) Whether parallel-pipeline topology is used in the scheduling. We group related works into four categories according to their target domains and the above criteria.

The first group focuses specifically on packet processing for network processors. [8] develops a low power online technique to adjust the activities of PEs according to the varying traffic volume. It reduces energy consumption by turning off unnecessary PEs when traffic is light. [7] adapts the number of activated processors based on the queuing theory. It achieves minimized total energy consumption while maintaining a bounded delay. Both works belong to PM category without considering task scheduling.

The second group targets on CMP. [22] regulates concurrency and changes the processors/threads configuration as the program executes by hardware event-driven profiling. [23] optimizes a parallel workload by dynamically changing the number of active processors and the voltage/frequency levels. It applies chip-wide DVFS rather than per-core DVFS. [24] considers a 4-core CMP with core-level DVFS and examines different DVFS policies based on the exhaustive search of the solution space, which is not scalable to large systems. [12] proposes *LinOpt*, which uses linear programming (LP) to find the best voltage and frequency for each core. [25] proposes a chip-level power control algorithm that is systematically designed based on optimal control theory. Similar to the first group, none of these works considers task scheduling.

The third group combines traditional task scheduling with power. [26] is based on a list-scheduling heuristic with dynamic recalculation of priorities. It minimizes the energy by choosing the best combination of voltages for each task. [27] applies genetic list scheduling algorithms (GLSA) to schedule and map tasks. Besides simply exploiting available slack time, it also considers the PE power profile during a refined voltage section. [28] presents CASPER (Combined Assignment, Scheduling, and PowER-management) for task mapping and scheduling using a genetic algorithm. It employs two power management techniques (PDP-SPM for homogeneous system and PV-DVS for heterogeneous one) in the fitness function of the genetic algorithm. Although these works address task scheduling, they do not deal with parallel-pipeline topology.

The last group is the real-time community. Most of these works also couple traditional task scheduling algorithms with power reduction schemes. They propose different strategies based on static power management (SPM). [9] uses simple static power management (S-SPM), which distributes global static slack proportionally to the length of the schedule. [11] proposes both greedy static power management (G-SPM), where the entire global static slack is allocated to the first task on each processor, and static power management for parallelism (P-SPM), where the degree of parallelism is taken into considered. [10] introduces the static power management with proportional distribution and parallelism (PDP-SPM) which exploits both the global and local slack. This scheme is so far the best among other SPM mechanisms. However, no parallel-pipeline topology is considered in this group, either.

VI. Conclusion

In this paper, we proposed a novel algorithm to optimize both throughput and latency given a power budget for network packet processing on multicore architectures. This algorithm addresses power-aware parallel-pipeline scheduling problem by applying per-core DVFS to optimally adjust frequency on each core. We implemented our algorithm in addition to five other conventional algorithms on an AMD machine with two Quad-Core Opteron 2350 processors. Compared to existing algorithms given the same power budget for six real packet processing applications, our algorithm exhibits substantially better throughput and latency by an average of 64.6% and 25.2%, respectively.

References

[1] "Intel Xeon Machine," http://www.Intel.com/Xeon.

[2] "AMD Opteron Machine," http://www.amd.com/opteron.

[3] "Intel IXP2XXX Product Line of Network Processors," http://intel.com/design/network/products/npfamily/index.htm.

[4] "NetFPGA," http://www.netfpga.org/.

[5] N. Weng and T. Wolf, "Pipelining vs multiprocessors-choosing the right network processor system topology," in *Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR '04)*, 2004.

[6] J. Yu, J. Yao, L. Bhuyan, and J. Yang, "Program mapping onto network processors by recursive bipartitioning and refining," in *Proc. of IEEE/ACM Design Automation Conference (DAC '07)*, San Diego, CA, June 2007.

[7] R. Kokku, U. B. Shevade, N. S. Shah, M. Dahlin, and H. M. Vin, "Energy-efficient pakcet processing," *University of Texas at Austin Technical Report TR04-04*, 2004.

[8] Y. Luo, J. Yu, J. Yang, and L. Bhuyan, "Low power network processor design using clock gating," in *Proc. of Design Automation Conference (DAC '05)*, June 2005.

[9] F. Gruian, "System-level design methods for low-energy architectures containing variable voltage processors," in *Proc. of the First International Workshop on Power-Aware Computer Systems*, 2000.

[10] S. Hua and G. Qu, "Power minimization techniques on distributed real-time systems by global and local slack management," in *Proc. of Asia South Pacific Design Automation Conference (ASP-DAC '05)*, Jan. 2005.

[11] R. Mishra, N. Rastogi, and D. Zhu, "Energy aware scheduling for distributed real-time systems," in *Proc. of the Intl. Parallel and Distributed Processing Symposium*, Apr. 2003.

[12] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," in *Proc. of the 35th International Symposium on Computer Architecture (ISCA '08)*, 2008, pp. 363–374.

[13] M. S. Floyd, S. Ghiasi, T. W. Keller, K. Rajamani, F. L. Rawson, J. C. Rubio, and M. S. Ware, "System power management support in the ibm power6 microprocessor," *IBM Journal of Research and Development*, no. 6, 2007.

[14] R. McGowen, C. A. Poirier, C. Bostak, J. Ignowski, M. Millican, W. H. Parks, and S. Naffziger, "Power and temperature control on a 90-nm itanium family processor," *IEEE Journal of Solid-State Circuits*, no. 1, Jan. 2006.

[15] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, "An integrated quad-core opteron processor," in *Proc. of International Solid State Circuits Conference*, Feb. 2007.

[16] G. Memik, W. H. Mangione-Smith, and W. Hu, "Netbench: A benchmarking suite for network processors," in *Proc. of ICCAD'01*, 2001.

[17] R. Ramaswamy and T. Wolf, "Packetbench: A tool for workload characterization of network processing," in *Proc. of WWWC-6 '03*, 2003.

[18] "Machine-SUIF, Harvard University," http://eecs.harvard.edu/hube/software/nci/overview.html.

[19] "SUIF Compiler System," http://suif.stanford.edu/.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, no. 3, pp. 319–349, 1987.

[21] "EXTECH Power Analyzer," http://extech.com/instruments/.

[22] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proc. of International Conference on Supercomputing (ICS '06)*, June 2006.

[23] J. Li and J. F. Martinez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *Proc. of International Symposium on High-Performance Computer Architecture (HPCA '06)*, Feb. 2006.

[24] C. Isci, A. Buyuktosunoglu, C. Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. of International Symposium on Microarchitecture*, Dec. 2006.

[25] Y. Wang, K. Ma, and X. Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation," in *Proc. of the 36th International Symposium on Computer Architecture (ISCA '09)*, June 2009.

[26] F. Gruian and K. Kuchcinski, "Lenes: Task scheduling for low-energy systems using variable supply voltage processors," in *Proc. of Asia South Pacific Design Automation Conference (ASP-DAC '01)*, Jan. 2001.

[27] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles, "Iterative schedule optimization for voltage scalable distributed embedded systems," *ACM Transactions on Embedded Computing Systems*, no. 1, Feb. 2004.

[28] V. Kianzad, S. S. Bhattacharyya, and G. Qu, "Casper: An integrated energy-driven approach for task graph scheduling on distributed embedded systems," in *Proc. of IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP '05)*, July 2005.