Power-Aware Modulo Scheduling for High-Performance VLIW Processors*

Han-Saem Yun
School of Computer Science and Engineering
Seoul National University
hsyun@davinci.snu.ac.kr

Jihong Kim
School of Computer Science and Engineering
Seoul National University
jihong@davinci.snu.ac.kr

ABSTRACT

For high-performance processors, the step power and peak power, which are closely related to the chip reliability, are important design constraints, often more than the average power. In VLIW processors where a single instruction may contain a variable number of operations, the step power and peak power vary significantly depending on the parallel schedule generated by a parallelizing compiler. In this paper, we propose a power-aware modulo scheduling algorithm for high-performance VLIW processors. The proposed algorithm reduces both the step power and peak power by producing a more balanced parallel schedule while not compromising performance. Experimental results show that the proposed scheduling technique significantly improves the power characteristics of high-performance processors over an existing power-unaware modulo scheduling technique.

1. INTRODUCTION

Power dissipation has become an important design constraint for high-performance processors such as modern superscalar processors and VLIW processors. Although performance is still the most important requirement for high-performance processors, increasing power dissipation is becoming a major obstacle to performance improvements in future microprocessors. In particular, the step power and peak power, which are closely related to the chip reliability, are important design issues, often more than the average power consumption, in high-performance processors.

1.1 Step Power and Peak Power

The *step power* [16], which is defined as the difference in the average power between consecutive clock cycles, represents the inductive noise Ldt/di at the microarchitectural level. Inductive noise, also known as ground bouncing, is a voltage glitch induced at power/ground buses due to switching currents passing through the wire inductance associated with power or ground rails. A large voltage surge due to the inductive noise may cause timing and logic errors, thus may reduce the chip reliability. For high-performance processors, the inductive noise problem is becoming more serious because the increasing clock frequency, the growing number of gates, and the wider datapath result in larger surge current to charge/discharge the power/ground buses in a shorter time leading to larger inductive noise. Even worse, with the growing usage of aggressive clock gating for reducing the average power consumption, the cycle-by-cycle current swing is getting larger.

The *peak power*, the maximum power dissipation during the execution of a given program, is closely related to the chip temperature [2], to which the chip reliability and sub-threshold leakage power are exponentially related [5]. Higher peak power leads to the device degradation, reducing the chip lifetime. As a result, complex cooling systems (which add significantly to the manufacturing cost) are used to avoid overheating and to ensure system reliability.

1.2 Related Work

Most previous work has been focused on hardware mechanisms to control the step power and peak power. Pant *et al.* [9, 10] proposed an improved version of clock gating to reduce the spike current by slowly turning on and off clock gated units at a modest cost in additional hardware and performance. Tang *el al.* [16] further enhanced this mechanism to reduce the performance loss. The work by Brooks *et al.* [2] controls the peak power in the context of dynamic thermal management.

There has been little published work that approached the problems of step-power reduction and peak-power reduction from the software perspective. The work by Toburen *et al.* [18] controls the peak power dissipation in VLIW processors by modifying the instruction scheduling algorithm of an optimizing compiler. Their power-aware scheduler places as many instructions as possible in a given VLIW instruction until the given power threshold is reached. However, their work is based on traditional list scheduling, which is not effective in exploring the high instruction-level parallelism available from modern multiple-issue processors such as VLIW processors. Therefore, their work is not applicable to most high-performance VLIW processors.

1.3 Contribution

In this paper, we propose a *power-aware* modulo scheduling algorithm for high-performance VLIW processors. In VLIW processors where a parallel instruction consists of several operations, the step power and peak power consumption varies significantly depending on how the parallel schedule is generated by a parallelizing compiler. The proposed algorithm reduces both the step power and peak power by constructing a more *balanced* parallel schedule while not compromising performance.

As will be shown later in the paper, the proposed algorithm is

*This work was supported in part by Korea Research Foundation Grant (KRF-2000-041-E00287).

ISLPED'01, August 6-7, 2001, Huntington Beach, California, USA.

quite effective in reducing the step power and peak power consumption in VLIW processors. This is because a parallelizing compiler can fully control the usage of all the functional units in a VLIW processor. On the other hand, the hardware-assisted techniques such as [9, 16] are often limited to the power reduction in a specific functional unit only without considering the processor-wide effect.

The rest of the paper is organized as follows. In Section 2, we describe our target VLIW machine model and its power model. We briefly review modulo scheduling in Section 3. The proposed power-aware algorithm is explained in Section 4 while experimental results are presented in Section 5. We conclude with a summary and future work in Section 6.

2. TARGET VLIW MACHINE MODEL AND ITS POWER ESTIMATION METHOD

VLIW machines use long instruction words to execute multiple operations simultaneously. In this paper, we assume a VLIW machine model with a MIPS-like integer pipeline and an UltraSPARC-like floating-point (FP) unit pipeline. For an 8-issue VLIW model, we assume that there are one integer ALU (including a branch unit), two load/store units, one integer MPY/DIV, two FP ALUs and two FP MPY/DIVs. For a 16-issue VLIW model, we assume that the number of each functional unit is doubled over that of the 8-issue model. In a target VLIW model, each operation takes different cycles to execute. For example, load operations require one cycle in the execution stage while FP DIV operations spend 10 cycles in the execution stage.

For a given VLIW machine, we estimate the power consumption at the cycle level, taking the pipelined executions into account. For each operation op, we associate a power cost $p(op,i)$ that represents the power consumed by operation op at the i-th pipeline stage ($1 \le i \le n_s$) where n_s denotes the number of pipeline stages. In this paper, we assume that $p(op,i)$ values were given. They can be obtained from actual measurements of a target processor (e.g., [3]) or simulations of a detailed processor model. For experiments, we used $p(op,i)$ values extrapolated from information available in [14] and [17].

Given a program execution trace T, let T_i represent the set of operations in the VLIW instruction executed at the i-th cycle of T. Then power dissipation P_i at the i-th cycle is estimated as follows:

$$P_i = \sum_{j=1}^{n_s} \sum_{op \in T_{i-j+1}} p(op,j) \qquad (1)$$

Since our main goal is not to develop a cycle-accurate power model of VLIW processors, but to devise a power-aware scheduling algorithm for VLIW processors, admittedly, our power estimation method is rather simple and has many weaknesses. For example, in computing P_i, we do not consider inter-instruction effect or inter-operation effect as done in [18]. That is, we assume that each operation (instruction) contributes to the total power consumption, independently of other operations (instructions). However, the proposed algorithm can be easily extended to work with more accurate power estimation models, because the proposed algorithm does not depend on a particular power estimation technique.

3. MODULO SCHEDULING OVERVIEW

Software pipelining is an aggressive loop scheduling technique for VLIW processors. It transforms a sequential loop so that new iterations can start before preceding ones finish, thus overlapping the execution of multiple iterations in a pipelined fashion. *Modulo scheduling* [6, 13] is one of the scheduling algorithms for implementing software pipelining.[1] Since a large number of loops contain no conditionals, we concentrate on loops with no control flows in this paper. For loops with control flows, we assume a hardware mechanism that supports predicated execution. If-conversion [1, 11] can be performed to eliminate conditionals in loops with control flows under this hardware mechanism.

A loop $\mathcal{L}$ is modeled as a double-weighted directed graph $G = (V, E, \delta, d)$, called a data dependence graph (DDG)[2], where V is the set of operations in the loop, δ is a function from V to the positive integers representing the latency of each operation, E is the edge set of G and d is a function from E to non-negative integers. E and d specify dependences among operations. An edge $e = (v, v') \in E$ with a weight $d(e)$ states that for every iteration $i \ge d(e)$ of the loop, operation v' depends on the outcome of operation v in iteration $i - d(e)$ and cannot be initiated until the completion of v. Figure 1(b) shows the DDG of an example loop in Figure 1(a).

Given a loop with the form of a DDG G and a specific resource constraint, the problem of software pipelining is defined as finding the minimal *initiation interval* (II) and constructing a schedule σ that is a function from $V \times N$ to N such that the following constraints are met ($\sigma(v,i)$ denotes the execution cycle in which the instance of operation v in iteration i is initiated):

Periodicity constraint: σ should be represented as a periodic form such that

$$\forall v \in V \;\; \forall i \in N, \;\; \sigma(v,i) = \sigma(v,0) + II \cdot i .$$

Dependence constraint: For every v and v' such that $e = (v, v') \in E$, the instance of operation v' in iteration $(i + d(e))$ should not be initiated until the completion of the instance of operation v in iteration i.

$$\forall e = (v, v') \in E \;\; \forall i \in N, \;\; \sigma(v', i + d(e)) \ge \sigma(v,i) + \delta(v)$$

Resource constraint: Let n_{R_k} represent the number of available R_k functional units and let $\tau(v)$ represent the functional unit in which v is executed. For each functional unit R_k, no more than n_{R_k} operations are initiated in the same cycle:[3]

$$\forall R_k \, \forall t \in N, \;\; |\{(v,i) | \tau(v) \equiv R_k \wedge \sigma(v,i) = t\}| \le n_{R_k}$$

From the periodicity constraint, it is sufficient to find II and $\sigma(v,0)$ for $v \in V$, called a *flat schedule*. The dependence constraint and resource constraint are translated as follows under the flat schedule:

$$\forall e = (v, v') \in E, \;\; \sigma(v', 0) \ge \sigma(v,0) + \delta(v) - II \cdot d(e)$$
$$\forall R_k, \; \forall \, 0 \le t < II,$$
$$|\{(v,0) | \tau(v) \equiv R_k \wedge \sigma(v,0) \equiv t \;\; (mod \;\; II)\}| \le n_{R_k}$$

Figure 1(c) shows a flat schedule with the initiation interval of 3 cycles for the DDG of Figure 1(b).

It is well-known that the problem of determining if a flat schedule exists for a given II is NP-hard and several heuristic approaches have been developed to find the minimal II and the flat schedule. The minimum initiation interval (MII) is a lower bound on the II

[1] Software pipelining is essentially equivalent to the *retiming* technique which is widely used in VLSI high level synthesis [4] and logic level synthesis [8].

[2] Such a graph is called a data flow graph (DFG) in the context of synchronous VLSI circuits.

[3] For simplicity, we assume that the functional units are fully pipelined. Complex resource constraints can be handled by *resource reservation table* [13].

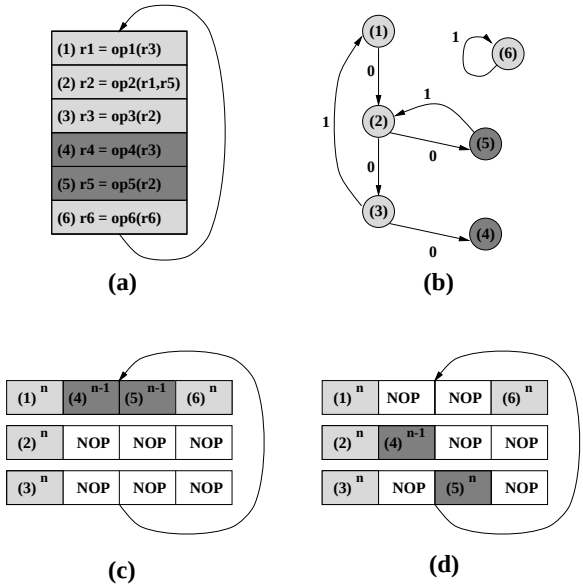

Figure 1: An example power-aware schedule: (a) An example loop, (b) its data dependence graph with $d(e)$ values (assuming operation latency = 1), (c) a performance-driven schedule, and (d) a power-aware schedule.

for which a flat schedule exists. The MII is obtained by separately considering the dependence constraint and the resource constraint, and taking the larger value between two [6]. The candidate II is initially set to the MII and increased until a legal modulo schedule is found.

4. POWER-AWARE MODULO SCHEDULING ALGORITHM

In this section, we present a power-aware modulo scheduling algorithm that reduces both the step power and peak power. Our main requirement in designing such algorithms is that performance must not be sacrificed for the reduced power consumption.

4.1 Problem Formulation

Using the periodicity constraint, we can formulate the power consumption at each time slot of the software pipelined loop $\mathcal{L}^{\mathrm{SP}}$ of a sequential loop $\mathcal{L}$ as follows. (The superscript SP is used to distinguish the software pipelined loop from the original sequential loop.) Let $O_{\mathcal{L}^{\mathrm{SP}},i}$ be the set of operations (including NOPs) scheduled at the time slot i for $0 \le i < II$. Then the power consumed at time slot i, written by $P_{\mathcal{L}^{\mathrm{SP}},i}$, is estimated by

$$P_{\mathcal{L}^{\mathrm{SP}},i} = \sum_{j=1}^{n_s} \sum_{op \in O_{\mathcal{L}^{\mathrm{SP}},t \ \text{where } t=(i-j+1) \bmod II} p(op, j) \qquad (2)$$

If we consider the infinite execution of $\mathcal{L}^{\mathrm{SP}}$, which is represented by the execution trace $T_{\mathcal{L}^{\mathrm{SP}}}$, the peak power is equal to $\max_i\{P_{\mathcal{L}^{\mathrm{SP}},i} \mid 0 \le i < II\}$ and the step power at the cycle i is $|P_{\mathcal{L}^{\mathrm{SP}},i \bmod II} - P_{\mathcal{L}^{\mathrm{SP}},(i-1) \bmod II}|$.

A software pipelined loop $\mathcal{L}^{\mathrm{SP}}$ has the minimum peak power and minimum step power when all the $P_{\mathcal{L}^{\mathrm{SP}},i}$ values ($0 \le i < II$) are equal to $P_{\mathcal{L}^{\mathrm{SP}}(\mathrm{ideal})}$. $P_{\mathcal{L}^{\mathrm{SP}}(\mathrm{ideal})}$ is given by as follows:

$$P_{\mathcal{L}^{\mathrm{SP}}(\mathrm{ideal})} = \Big[\sum_{op \in \cup_{k=0}^{II-1} O_{\mathcal{L}^{\mathrm{SP}},k}} \sum_{j=1}^{n_s} p(op, j) \Big] \ / \ II \qquad (3)$$

For such an $\mathcal{L}^{\mathrm{SP}}$, the peak power is equal to $P_{\mathcal{L}^{\mathrm{SP}}(\mathrm{ideal})}$ and the step power is zero for the whole execution trace $T_{\mathcal{L}^{\mathrm{SP}}}$.[4] In order to reduce the step power and peak power, our basic approach is to schedule operations in such a way that the resulting $P_{\mathcal{L}^{\mathrm{SP}},i}$ function becomes as flat as possible. As a figure of merit for the flatness of $\mathcal{L}^{\mathrm{SP}}$, we use the following function $\mathcal{F}(\mathcal{L}^{\mathrm{SP}})$:

$$\mathcal{F}(\mathcal{L}^{\mathrm{SP}}) = \sum_{i=0}^{II-1} [P_{\mathcal{L}^{\mathrm{SP}},i} - P_{\mathcal{L}^{\mathrm{SP}}(\mathrm{ideal})}]^2 \qquad (4)$$

Our goal in power-aware modulo scheduling is to find $\mathcal{L}^{\mathrm{SP}}_{\mathrm{opt}}$ such that $\mathcal{F}(\mathcal{L}^{\mathrm{SP}}_{\mathrm{opt}}) \le \mathcal{F}(\mathcal{L}^{\mathrm{SP}}_{k})$ for all k, where $\mathcal{L}^{\mathrm{SP}}_{k}$ represents the k-th software pipelined loop schedule of the sequential loop $\mathcal{L}$. For such an $\mathcal{L}^{\mathrm{SP}}_{\mathrm{opt}}$, $P_{\mathcal{L}^{\mathrm{SP}}_{\mathrm{opt}},i}$ is as *balanced* as possible.

Consider an example loop shown in Figure 1, which has the MII of 3 cycles. Both schedules in Figures 1(c) and 1(d) are optimal in terms of performance achieving their IIs equal to the MII value. However, in terms of the step power and the peak power, the latter is better than the former. (The superscript n in the operation number indicates the n-th loop iteration.) For example, if we assume equal $p(op, j)$ values, the schedule in Figure 1(c) has the twice bigger peak power over the schedule in Figure 1(d). Similarly, for the step power, the schedule in Figure 1(d) has no power difference while the schedule in Figure 1(c) has the largest value between the first and second instructions.

This example clearly demonstrates that there exist large differences in power characteristics among the schedules with the equal execution times. Our goal is to find the power-efficient schedules among such performance-driven schedules without compromising performance.

4.2 The Base Algorithm : Iterative Modulo Scheduling (IMS)

We start with Rau's *iterative modulo scheduling* (IMS for short) as the base algorithm [13]. It outperforms the best-known modulo scheduling algorithm by Lam [6]. Figure 2 describes a simplified version of the IMS algorithm. (Rau's original algorithm includes a sophisticated backtracking procedure. For brevity, we do not include it in Figure 2.)

IMS calls FINDFLATSCHEDULE with successively larger values of II, starting with an initial value equal to the MII until the legal schedule is found. FINDFLATSCHEDULE repeats picking the highest priority operation and then selecting the best desirable time slot in which the operation is to be scheduled.

Rau used the priority function based on the length of the critical dependence cycle. Before selecting the best desirable time slot, COMPUTESLACK is called to compute the range of time slots (i.e., slack) where the operation may be placed without violating dependence constraints with the operations already scheduled.

Given a path $p = v_1 \to v_2 \to \cdots \to v_k$, let $\delta(p)$ and $d(p)$ represent $\sum_{i=1}^{k-1} \delta(v_i)$ and $\sum_{i=1}^{k-1} d((v_i, v_{i+1}))$, respectively. Assume that v has been already scheduled and v' is about to be scheduled. Adding dependence constraints along $v \overset{p_1}{\rightsquigarrow} v'$ and $v' \overset{p_2}{\rightsquigarrow} v$ yields:

$$\sigma(v) + \delta(p_1) - II \cdot d(p_1) \le \sigma(v') \le \sigma(v) - \delta(p_2) + II \cdot d(p_2)$$

Thus, the slack interval, [MinTime,MaxTime], of v' is computed

[4] Of course, an ideal $\mathcal{L}^{\mathrm{SP}}$ generally does not exist because of the dependence constraint and resource constraint.

as:

$$\text{MinTime} = \max_{v_s}[\sigma(v_s) + \delta(v_s \rightsquigarrow v') - II \cdot d(v_s \rightsquigarrow v')]$$

$$\text{MaxTime} = \max_{v_s}[\sigma(v_s) - \delta(v' \rightsquigarrow v_s) + II \cdot d(v' \rightsquigarrow v_s)]$$

where v_s is any scheduled operation.

From the computed slack, FINDTIMESLOT picks the best desirable time slot. Rau's original algorithm selects the earliest time slot in which resource conflict is not incurred. If no conflict-free slot is found, the operation cannot be scheduled unless some operations in the partial schedule are unscheduled. (The detailed description of unscheduling procedure can be found in [13].)

```
procedure IMS
  II := MII(); /* initialize the candidate II to MII */
  while (FINDFLATSCHEDULE(II) != SUCCESS)
    II := II + 1;
end procedure
```

```
function FINDFLATSCHEDULE(II)
  /* compute the priority of each operations */
  COMPUTEPRIORITY();

  /* repeat picking the highest priority operation and
     selecting the best desirable time slot at which the
     operation is to be scheduled until all operations
     have been scheduled */
  while (some operation is not scheduled)
    /* pick the highest priority operation */
    CurrOper := HIGHESTPRIORITYOPERATION();

    /* compute the time bounds in which the selected
       operation can be scheduled satisfying the
       dependence constraint */
    (MinTime, MaxTime) := COMPUTESLACK(CurrOper);

    /* select the best desirable time slot */
    TimeSlot := FINDTIMESLOT(
                    CurrOper, MinTime, MaxTime);

    /* schedule the operation at TimeSlot. */
    SCHEDULEOPERATION(CurrOper, TimeSlot);
  end while

  if (all operations are scheduled) return SUCCESS;
  else return FAIL;
end function
```

Figure 2: Original IMS algorithm.

4.3 Balanced IMS (BIMS) for Reduced Peak Power and Step Power Consumption

As explained in Section 4.1, our goal is to find schedule(s) with the most *balanced* power dissipation distribution while not sacrificing the performance that can be obtained from the original IMS.

IMS always places an operation as early as possible (ASAP) within the partial schedule constructed so far, resulting in a *skewed* schedule with an *unbalanced* power dissipation distribution. This ASAP policy has been widely used by software pipelining researchers but is somewhat a legacy from an over-reliance on the intuition underlying acyclic list scheduling.

In contrast, our algorithm tries to build *balanced* schedules using a heuristic guided by the $\mathcal{F}(\mathcal{L}^{\text{SP}})$ cost function which indicates how much a power dissipation distribution is balanced over the entire program execution. In order to make the original IMS algorithm to be power-aware, we make the following two modifications:

- **Priority Function Modification**

  Instead of the length of the critical dependence cycle, we use the reciprocal of the slack width as the priority function for an unscheduled operation. Therefore, we need to recompute the priority values at each iteration of the **while** loop in FINDFLATSCHEDULE. With the modified priority function, operations with smaller slack widths are scheduled early so that performance is not sacrificed. If these operations are scheduled later, it is more likely that, for the current II value, no legal schedule can be found. Although the re-computation of the priority at each iteration of the **while** loop may incur additional compilation time, we believe that the dynamic priority adjustment provides more accurate critical-path information, thus guaranteeing that the resulting schedule does not suffer any performance loss.

- **Time Slot Selection Modification**

  When selecting the best desirable time slot for the current operation to be scheduled, the *balancing* cost function is used so that the partial schedule is constructed as *balanced* as possible. That is, the operation is positioned into the time slot at which incurs the least increase of the flatness measure $\mathcal{F}(\mathcal{L}^{\text{SP}})$ among all the conflict-free time slots in the slack. When $P_{\mathcal{L}(\text{ideal})}$ is computed, only the operations in the partial schedule are considered. Ties are broken by the ASAP policy as in the original IMS to assist the critical-path consideration.

In short, two major IMS decisions are modified so that both performance and power dissipation distribution are simultaneously considered.

5. EXPERIMENTAL RESULTS

In order to evaluate the power reduction effect of the BIMS algorithm over the original IMS algorithm, we implemented the BIMS algorithm as well as the original IMS algorithm on a SPARC-based VLIW testbed environment [12]. We incorporated our power model into the scheduling modules as well as the simulator, so that the detailed instruction-level power statistics are collected. We experimented with two machine configurations, an 8-issue VLIW machine and 16-issue VLIW machine, as described in Section 2. As test programs, SPEC95 FP benchmark programs were used.

Since the performance of parallel schedules produced by the BIMS algorithm should be as good as that of the schedules by the IMS algorithm, we first compared the execution cycles of the schedules from two algorithms. In all the benchmark programs, there was negligible performance impact ($< 0.5\%$), which was due to the code differences in prolog and epilog code sections, which are outside software-pipelined loop bodies.

We also evaluated the performance of the schedules by the IMS algorithm. If the performance is close to optimum, it implies that the schedule is tight and there is not much freedom in scheduling each operation, resulting in small exploration space for the BIMS. The quality of the IMS was evaluated by comparing the actual II values with the theoretical MII values. 99.2% of the loops tested achieved their IIs equal to the MII values, indicating that the IMS algorithm finds high-quality schedules in terms of the performance. Even under this restrictive search space for power-aware schedules, the BIMS algorithm is quite effective in reducing the peak power and step power consumption as shown in the next subsections.

5.1 Impact on Peak Power Consumption

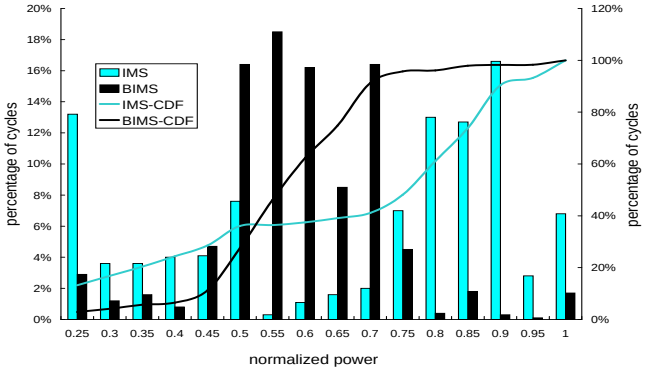

Figure 3: Normalized power distribution for an 8-issue VLIW machine.

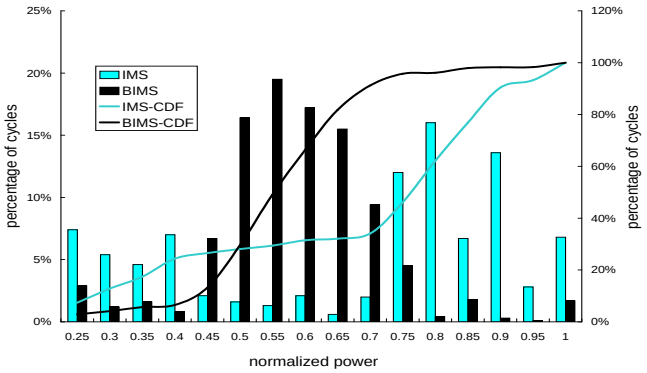

Figure 4: Normalized power distribution for a 16-issue VLIW machine.

Figures 3 and 4 show the impact on the peak power consumption of the proposed BIMS for the 8-issue and 16-issue VLIW machine configurations, respectively. The *x* axis of the graphs shows the normalized power consumption values, 1 being the maximum power dissipated under the given machine configuration. The *y* axis indicates the relative ratio (in percentage) of the cycles that consumed the corresponding normalized power during the benchmark executions. The curves in the graphs represent the cumulative distribution functions (CDFs) of the normalized power consumption.

As shown in Figures 3 and 4, the average power of the schedules generated from the BIMS algorithm are clustered around 0.57. On the other hand, under the IMS, the benchmark executions spend the large number of cycles at high power consumption levels. For example, for the 8-issue VLIW machine, the IMS-generated schedules spend 58.9% of their execution cycles consuming more than 70% of the maximum power. However, under the BIMS, only 8.8% of program executions were spent consuming more than 70% of the maximum power.

Figures 3 and 4 also illustrate the balancing effect of the BIMS algorithm on the $P_{L^{SP},i}$ distribution. For example, in Figure 3, the standard deviation of the normalized power distribution under the IMS is 0.31 while it is reduced to 0.08 for the BIMS. As the number of resources increases, the search space to find alternative schedules becomes bigger, thus the BIMS algorithm finds better schedules for machine configurations with longer instruction words.

5.2 Impact on Step Power Consumption

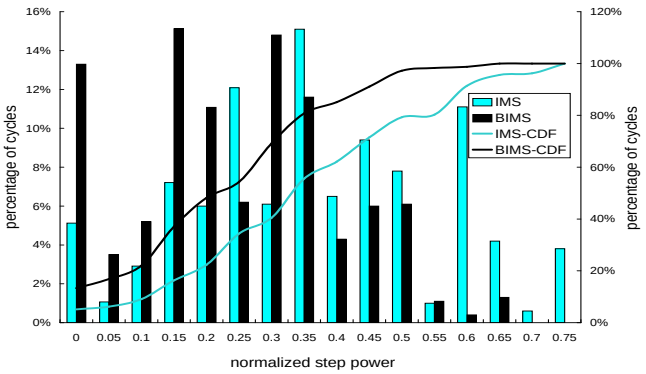

Figure 5: Normalized step power distribution for an 8-issue VLIW machine.

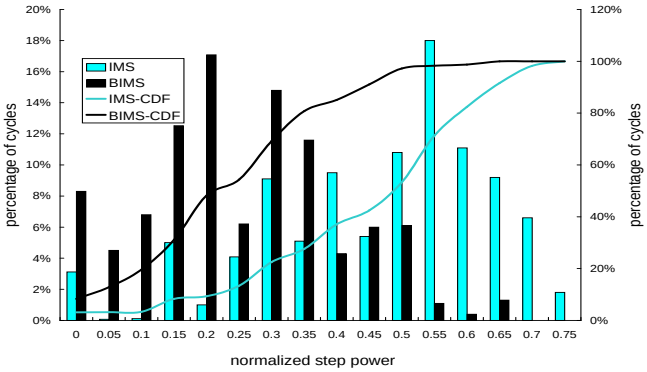

Figure 6: Normalized step power distribution for a 16-issue VLIW machine.

Figures 5 and 6 illustrate the impact on the step power consumption of the BIMS for the 8-issue and 16-issue VLIW machine configurations, respectively. The *x* axis of the graphs shows the normalized step power consumption values, 1 being the maximum step power value under the given machine configuration. As with the peak power experiment, the schedules from the BIMS algorithm has smaller step power values over ones from the IMS algorithm. For example, for the 16-issue VLIW machine, the geometric mean of step power values when the IMS is used is 0.41 while it is reduced to 0.24 when the BIMS is used.

6. CONCLUSION AND FUTURE WORK

We have described a power-aware modulo scheduling algorithm, Balanced IMS (BIMS), for high-performance VLIW processors. The BIMS algorithm reduces the step power and peak power consumption (which affect the processor reliability) from performance-critical loop bodies. The main characteristics of the schedules from the BIMS algorithm is that their power consumption distributions are better balanced over power-unaware modulo scheduling algorithms. Experimental results using SPEC95 FP benchmark programs show that the proposed algorithm is effective in reducing both the step power and peak power; In the case of step power consumption, the BIMS reduces on average 37.1% over the original IMS algorithm.

The current version of the BIMS algorithm can be further enhanced in several directions. For example, the BIMS algorithm can be integrated with post-pass low-power scheduling techniques such as [15, 7]. Although post-pass techniques work independently of the BIMS algorithm, it will be interesting to evaluate quantitatively the combined effect on the power consumption. We also plan to investigate how the BIMS affects the energy efficiency of aggressively clock-gated processors. Since the BIMS tries to spread the operation distribution evenly while clock gating prefers skewed executions, it will be an interesting future work to extend the BIMS algorithm for such processors.

7. REFERENCES

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of Control Dependence to Data Dependence. In *Proc. of the 1983 Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[2] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture (HPCA-7)*, 2001.

[3] N. Chang, K. Kim, and H. Lee. Cycle-Accurate Energy Consumption Measurement and Analysis: Case Study of ARM7TDMI. In *Proc. of International Symposium On Low Power Electronics and Design*, pages 185–190, 2000.

[4] L.-F. Chao and E. Sha. Scheduling Data-Flow Graphs via Retiming and Unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8(12):1259–1267, 1997.

[5] A. Dhodapkar, C. H. Lim, and G. Cai. TEM2P2EST : A Thermal Enabled Multi-Model Power/Performance ESTimator. In *Proc. of Workshop on Power-Aware Computer Systems*, 2000.

[6] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[7] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai. Compiler Optimization on Instruction Scheduling for Low Power. In *Proc. of the 13th International Symposium on System Synthesis*, pages 55–60, 2000.

[8] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6:5–35, 1991.

[9] M. Pant, P. Pant, D. Wills, and V. Tiwari. An Architectural Solution for the Inductive Noise Problem Due to Clock-gating. In *Proc. of International Symposium On Low Power Electronics and Design*, pages 255–257, 1999.

[10] M. Pant, P. Pant, D. Wills, and V. Tiwari. Inductive Noise Reduction at the Architectural Level. In *Proc. of International Conference on VLSI Design*, pages 162–167, 2000.

[11] J. C. H. Park and M. S. Schlansker. On Predicated Execution. Technical Report HPL-91-58, Hewlett Packard Laboratories, 1991.

[12] S. Park, S. Shim, and S.-M. Moon. Evaluation of Scheduling Techniques on a SPARC-Based VLIW Testbed. In *Proc. of the 30th Annual International Symposium on Microarchitecture (Micro-30)*, pages 104–113, 1997.

[13] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture (Micro-27)*, pages 63–74, 1994.

[14] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria. Instruction-Level Power Estimation for Embedded VLIW Cores. In *Proc. of the 8th International Workshop on Hardware/Software Codesign (CODES2000)*, pages 34–38, 2000.

[15] D. Shin and J. Kim. An Operation Rearrangement Technique for Low Power VLIW Instruction Fetch. In *Proc. of Workshop on Complexity-Effective Design*, 2000.

[16] Z. Tang, N. Chang, S. Lin, W. Xie, S. Nakagawa, and L. He. Ramp Up/Down Floating Point Unit to Reduce Inductive Noise. In *Proc. of Workshop on Power Aware Computer Systems*, 2000.

[17] V. Tiwari, S. Malik, and A. Wolfe. Power Analysis of Embedded Software: A First Step Towards Software Power Minimization. *IEEE Transactions on VLSI Systems*, 2(4):437–445, 1994.

[18] M. C. Toburen, T. Conte, and M. Reilly. Instruction Scheduling for Low Power Dissipation in High Performance Microprocessors. In *Proc. of the Power Driven Microarchitecture Workshop*, 1998.