University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Maximizing Pipelined Functional Units Usage for Minimum Power Software Pipelining

Hongbo Yang
R.Govindarajan♯
Guang R. Gao
George Cai†

CAPSL Technical Memo 41
Sep 27, 2001

♯Supercomputer Education & Research Centre, Dept. of Computer Science & Automation,
Indian Institute of Science
†Intel Corp

Abstract

This paper presents a new power-aware software pipelining method which can minimize power consumption of software pipelined loops on VLIW architecture without sacrificing performance. Our method is motivated by the following facts: (1) functional units in modern architectures are fully pipelined; (2) in a loop body, there exists instructions which are not on critical (recurrence) cycle(s). Traditional software pipelining approach schedules instructions as long as the required resources are available irrespective of whether the instructions are on or off critical cycle(s). However, intuitively from the angle of power reduction, if no performance penalty will be incurred, it may be reasonable to postpone the issue of certain non-critical instructions so they can be scheduled to the same functional unit of a prior instruction. The idea here is to reduce the number of functional units which are in use in each cycle, thereby reducing the power consumed by the processor.

In this paper we formulate the power consumption problem in software pipelined loops as an integer linear programming(ILP) problem. Within this model, the pipelined functional unit usage in each cycle are modeled precisely, and the power minimization acts as the objective function. The power-aware software pipelining approach for an Intel Itanium-like architecture is evaluated on loops extracted from *SPEC2000* integer benchmarks using the SGI Pro64 open source compiler. Our experimental results show that the our method can save power for more than 59% loops without any degradation in performance. For these cases, the average power saving on the functional units is 15.9%.

Contents

1 Introduction **1**

2 Motivation **3**

3 Integer Linear Programming Formulation **5**

4 Results **8**
4.1 Experimental Framework 8
4.2 Experimental Results 10

5 Related Work **11**

6 Conclusions **11**

List of Figures

1 Data Dependence Graph 3

List of Tables

1 Target Machine Configuration 3
2 A Feasible Schedule for DDG in Figure 1 4
3 The Energy Saving Schedule for DDG in Figure 1 5
4 Statistics on SPEC 2000 Benchmark Evaluated 9
5 Power Model 9
6 Power Saving for SPEC 2000 Benchmarks 10

1 Introduction

Power dissipation is becoming one of the major design issues of future high performance processor architectures and embedded systems. In this paper, we focus on the impact of *software pipelining* [21] on power consumption. Software pipelining is an important compilation technique applied on loops to exploit instruction level parallelism. In the past, resource constrained software pipelining has been studied extensively by several researchers and a number of *modulo scheduling* algorithms have been proposed in the literature [6, 16, 21, 31]. The objective of a software pipelining method is to construct a schedule that satisfies both the resource constraints of the architecture and the dependence constraints imposed by the program, such that the constructed schedule has a very low *initiation interval* (II). The schedule which achieves the lowest possible II for the given resource constraint is said to be a *rate-optimal* schedule. For a comprehensive survey of software pipelining methods the reader is referred to [30].

This paper presents a new power-aware software pipelining method for VLIW architectures, which can minimize the power consumption of software pipelined loops without sacrificing performance. This is possible due to the following facts: first, a large number of instructions have *scheduling slacks* — i.e., each of these instructions can be scheduled in one of many time steps without degrading the performance of the schedule. This is because either these instructions are off the critical (recurrence) cycles in the loop or they do not use the critical resource(s). Such slack has been profitably used to reduce the register pressure in some of the software pipelining methods [16, 25]. For instruction scheduling, it is found that a significant number of instructions have slacks beyond 1 cycle [4]. Second, in a modern VLIW architecture, functional units are usually fully pipelined and multiple instances of the same kind of functional units are provided to unleash instruction level parallelism [19]. For a fully pipelined functional unit, on each cycle, a new instruction can be issued to it. Thus a number of instructions can be present in different stages of the functional unit in a given time step. This increases the utilization of a functional unit and other functional units of the same kind can be released. This in turn leads to reduced power consumption in *all-or-nothing clock-gating* model [3].

The target processor we are modeling use all-or-nothing clock-gating to gate the clock of idle functional units. *Clock-gating* is a circuit technique to gate input clock of unused partitions thus disable needless toggling at each cycle. It is pervasively used in modern processor design to realize power saving. However, aggressive clock-gating cannot be used indiscriminately since it may generate glitches, cause clock skew and severely increase the complexity of timing verification at high frequency [13, 36, 14]. Thus in practical design, the granularity at which the clock-gating is applied differs from processor to processor. Clock-gating on functional unit level is a reasonable design choice and it is adopted in pratice. Alpha 21264 microprocessor [13] is a working example, its divider datapath is operated on one conditional clock. Since clock is the major contributor to CPU power [10], under all-or-nothing clock-gating model, it is reasonable to assume that the functional units bear only two states – active or inactive, and nothing in between. Correspondingly, throughout this paper we assume that for each functional unit, constant amount of power is consumed in active state, including dynamic power and leakage power; whereas in inactive state, only leakage power is consumed [9].

Traditional software pipelining approaches schedule instructions in each cycle based on certain pri-

ority order, as long as there are available resources. This applies not only to *critical instructions*, those that are on critical recurrence cycle(s) or those that use critical resource(s), but also to all other instructions as well. In certain software pipelining methods, instructions are scheduled at the earliest possible time [31]. However, issuing instruction as early as possible may schedule non-critical instructions along with critical instructions at the same time step, requiring multiple instances of functional units to be active simultaneously. As explained earlier, since we assume a power model in which all or none of the stages of a functional unit is switched off, many instances of the functional units being active simultaneously increases power consumption. As opposed to this, if certain non-critical instructions are scheduled at later cycles to an already active functional unit, i.e., which have instructions in some of stages of the pipeline, some other functional units of the same kind can be completely powered down, resulting in reduced power consumption. Thus, from the angle of power reduction, it may be reasonable to delay the schedule of some of the non-critical instructions so that they can be issued in the available empty slot of an active functional unit at a later cycle. This could be done in such a way that there is no performance degradation be introduced.

One interesting question in software pipelining context therefore is: Is it possible to schedule instructions in such a way that rate-optimality, in terms of minimum initiation interval (II), is obtained while reducing the power consumption by reducing the total number of function units in use? To answer this question, we define the power aware software pipelining as below:

Problem Given a loop L and a machine architecture M, construct a schedule that achieves rate-optimality for L under the given resource constraints of M, and consumes the *minimal* power.

In this paper we formulate the power-aware software pipelining problem as an integer linear programming(ILP) problem. As illustrated in an earlier work [32], while an integer linear programming based method may not directly be used in a production compiler, it is still useful for the evaluation of (performance) bounds that can be achieved by any heuristic based method. Unlike some of the earlier (simpler) ILP formulation for pipelined functional units [11], the proposed power aware software pipelining ILP formulation models precisely the pipelined functional unit usage in each clock cycle. This in turn helps to model the power consumed accurately. In our ILP formulation, we use the minimal power consumption as the objective function.

The proposed ILP formulation is implemented on an Itanium-like architecture [18, 19] in SGI Pro64 compiler. We have tested our method on 1983 loops extracted from `SPEC2000` integer benchmarks. Our experimental results show that for 59% of the loops, the proposed power-aware software pipelining method can achieve better schedules in terms of power consumed than a performance-oriented power-unaware approach, within the same performance. For these cases, the average percentage of power saving on functional units is 15.9%. It should be noted here that our approach considers only the power savings in functional units, and hence the 15.9% reduction is only with regard to power consumed by functional units.

This paper is organized as follows. In section 2 we motivate our power-aware software pipelining method with the help of an example. Our integer linear programming formulation is described in Section 3. The experimental results on loops extracted from `SPEC2000` integer benchmark are reported in Section 4. A discussion on related work is presented in Section 5. Section 6 concludes our work.

2 Motivation

In this section, we motivate power-aware software pipelining with the help of a motivating example.

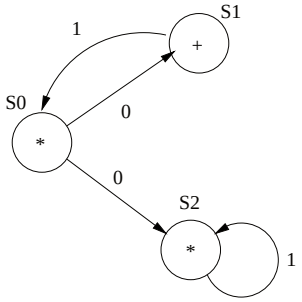

Figure 1: Data Dependence Graph

Consider a VLIW architecture with the machine configuration as shown in Table 1. As shown in the table, there are 1 adder and 2 multipliers in the architecture. The adder only takes one cycle (i.e., a trivial case of a fully pipelined unit). The two multipliers are fully pipelined with two cycle latency (2 stages). We use normalized data for power and assume that an active adder consumes 1 unit of power, while an inactive adder consumes 0.1 unit of power. The multiplier consumes 2.0 or 0.2 units of power depending on its state as in the Cai-Lim model [9]. In this paper, we focus on the power consumption of functional units while neglecting other components like issue logic, data cache and instruction cache, memory, etc.

Functional unit type	Adder	Multiplier
Numbers	1	2
Stages	1	2
Active power	1	2
Inactive power	0.1	0.2

Table 1: Target Machine Configuration

Consider the DDG shown in Figure 1, where there are three instructions (one add and two multiply instructions) in the loop. The directed arcs and the numbers adjacent to nodes in the DDG indicate, respectively, data dependences and dependence distance.

First, we calculate the lower bound on the initiation interval(II), which is the maximum of the bound imposed by the recurrences in the loop and the bound imposed by the resource constraints. They are referred to as RecMII and ResMII respectively [30]. RecMII is given by :

$$RecMII = \max\left\{ \left\lceil \frac{d(C)}{m(C)} \right\rceil \mid \forall C \in \text{ cycles of } G \right\}$$

where $d(C)$ is the sum of the latencies of the nodes in cycle C of the dependence graph, and $m(C)$ is the sum of the dependence distances around cycle C. In the given DDG, there is a critical cycle involving $S0$ and $S1$, which makes the RecMII to be 3.

The lower bound of II governed by the resource constraint is given by:

$$ResMII = \max\left\{ \left\lceil \frac{|\zeta(r)|}{R_r} \right\rceil \mid \quad \forall r \in [0, h-1] \right\}$$

where $\zeta(r)$ is the set of all instructions in the DDG G that use functional unit r for its execution, R_r is the number of functional units of type r defined by the micro-architecture, h is the number of different types of the functional units. For the given DDG, ResMII is 1.

The lower bound of II for the given DDG under the resource constraints of architecture M is thus $\max(3,1) = 3$. Table 2 gives a possible schedule with II $=$ 3. Thus we note that 3 is a feasible II for the given DDG.

Iteration	Time Steps						
	0	1	2	3	4	5	6
0	s0		s1	s2			
1				s0		s1	s2
2							s0

Table 2: A Feasible Schedule for DDG in Figure 1

Let us calculate the power consumption of the loop based on the activities of functional units in the repetitive pattern. Without loss of generality, we choose to study power consumption in the repetitive pattern from cycle 3 to cycle 5. At cycle 3, both multiply instructions($S0$ and $S2$) in the loop are issued, and they must be issued to two different multipliers to avoid structural hazard. At cycle 5, add instruction $S1$ is issued to the adder. Thus the two multipliers are busy for two consecutive cycles and idle for one cycle. The adder is busy for one cycle and idle for the remaining two cycles. During a single II, the energy consumed by the adder is

$$1 \times 1 + 2 \times 0.1 = 1.2$$

And the energy consumed by each of the two multipliers is

$$2 \times 2 + 1 \times 0.2 = 4.2$$

Thus the total energy consumed by all the functional units is

$$1.2 + 4.2 \times 2 = 9.6$$

An interesting question to ask is: does there exist any other rate-optimal schedule which consumes less energy? The answer is yes because the issue time of $S2$ can be delayed by one cycle without changing II, since $S2$ is not on the critical cycle, and the proposed delay should not increase II of the loop, hence its total execution time. Further, since the multiplier is fully pipelined, scheduling $S2$ at time step 4 in the linear schedule, i.e., time step 1 in the repetitive pattern does not conflict with schedule time of $S1$ which is in the critical cycle. Thus $S2$ can be scheduled at time step 4 without affecting the value of II. Applying this observation, we construct another schedule which is shown in Table 3.

Let us calculate the energy consumed by the new schedule. At cycle 3, $S0$ is issued, it goes to one of the two multipliers. At cycle 4, $S2$ is issued. Since the multiplier is fully pipelined, $S2$ can be issued

Iteration	Time Steps						
	0	1	2	3	4	5	6
0	s0		s1		s2		
1				s0		s1	
2							s0

Table 3: The Energy Saving Schedule for DDG in Figure 1

to the same multiplier as $S0$, causing higher utilization of one of the two multipliers and low (zero) utilization of the other. $S2$ completes its execution at the end of time step 5, thus, in this new schedule, one of the two multipliers is active for all 3 cycles (from time step 3 to 5) while the other one is always idle. Thus energy consumed by them during a single II is $2 \times 3 + 0.2 \times 3 = 6.6$. As for the adder is concerned, it consumes the same amount of energy as the previous schedule, which is 1.2. Taking all the functional units into account, power consumption of the new schedule is $6.6 + 1.2 = 7.8$. Compared to the prior one, the new schedule consumes 18.8% less of power.

One key observation that can be made from the two schedules is that the latter schedule makes a better use of the multiplier pipeline. Only one of the two multipliers is active while the other can be completely powered down, resulting in reduction in power/energy. As our motivating example shows, indeed, there exist considerable opportunity to reduce power/energy consumption of the software pipelined loop by taking advantage of the slack of non-critical instructions.

In the following section we formulate the power-aware software pipelining problem as an integer linear programming problem.

3 Integer Linear Programming Formulation

Let the number of nodes in the DDG be N. We formulate the problem of constructing a software pipelined schedule with a specific II as below. To achieve a rate-optimal schedule, we need to try successive values of II from MII (minimum initiation interval) until a schedule is found. Note that in our formulation N and II are constants. Further, in our formulation, we consider only modulo schedules which are periodic. In a periodic schedule, instruction s in the ith iteration is scheduled at time $II \cdot i + t_s$, where t_s is the schedule time of s in the first iteration, starting from zero. The schedule time of all instructions in the first iteration is represented by an N-element vector

$$\Gamma = [t_0, t_1, \cdots, t_{N-1}]^{Transpose}$$

Resource constraints can be checked in the repetitive pattern. The repetitive pattern is represented by an $II \times N$ matrix A. Matrix A is a 0-1 matrix with $a_{t,i} = 1$ if instruction i is issued at time step t in the repetitive pattern; otherwise $a_{t,i} = 0$. In our formulation, vector Γ is related to matrix A as

$$\Gamma = II \cdot \kappa + A^{Transpose} \times [0, 1, \cdots, (II-1)]^{Transpose}$$

That is,

$$\begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{N-1} \end{bmatrix} = II \times \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_{N-1} \end{bmatrix} + \begin{bmatrix} a_{0,0} & \cdots & a_{II-1,0} \\ a_{0,1} & \cdots & a_{II-1,1} \\ \vdots & \vdots & \vdots \\ a_{0,N-1} & \cdots & a_{II-1,N-1} \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ \vdots \\ II-1 \end{bmatrix} \quad (1)$$

To help understanding what κ is, each element k_i of κ is associated with t_i in vector Γ as:

$$k_i = \left\lfloor \frac{t_i}{II} \right\rfloor$$

Intuitively $A^{Transpose} \times [0,1,\cdots,(II-1)]^{Transpose}$ is the offset of instructions in the repetitive pattern. The matrix Γ, κ and A for the schedule shown in Table 3 are:

$$\Gamma = \begin{bmatrix} 0 \\ 2 \\ 4 \end{bmatrix} \quad \kappa = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

The following equation guarantees that each instruction is scheduled exactly once in the repetitive kernel:

$$\sum_{t=0}^{II-1} a_{t,i} = 1, \quad \forall i \in [0, N-1] \quad (2)$$

If a dependence arc exists from node i to node j with dependence distance $m_{i,j}$ and the latency is d_{ij}, then Inequality 3 must hold.

$$t_j - t_i \geq d_{ij} - II \cdot m_{ij}, \quad \forall (i,j) \in E \quad (3)$$

Next we formulate resource constraints. Let $\zeta(r)$ denote the set of all instructions bound to functional units of type r. For a fully pipelined functional unit, it is exclusively occupied by some instruction only at the issue cycle of that instruction. The number of functional units of type r that are needed for the schedule at time step t is

$$\sum_{i \in \zeta(r)} a_{t,i}$$

Thus the following inequality enforces the resource constraint for all functional unit types and all time steps in the repetitive kernel.

$$\sum_{i \in \zeta(r)} a_{t,i} \leq R_r, \quad \forall t \in [0, II-1], \forall r \in [0, h-1] \quad (4)$$

where R_r is the number of available functional units of type r, h is the number of different types of functional units.

The resource and dependence constraints in our formulation are same as that in [11, 12]. Next we formulate the power consumption of the software pipelined schedule which is our contribution in this paper. To model the power consumption for each functional unit at each time step, we need to know the number of active functional units. For this purpose, besides the starting time of each instruction, we also need to know how long each instruction takes to execute in the pipeline. Furthermore, it is important to know exactly for how many time steps each functional unit is active, taking into consideration overlapped execution of different instructions on the same functional units. For example, in the schedule shown in Table 3 one of the two multipliers is busy for 3 cycles in the repetitive pattern, taking the overlapped execution of instructions $S0$ and $S2$.

An instruction i issued at time t in the repetitive pattern will be in stage s at time step $(t+s) \mod II$. To model the usage of the stages of different functional units, we introduce a 3-dimensional array $U = [u_{t,i,s}]$. In the array each element is defined as:

$$u_{t,i,s} = 1 \Longleftrightarrow \text{instruction } i \text{ is in stage } s \text{ of the pipeline at time step } t$$

Derived from above discussion, $u_{t,i,s}$ is defined by matrix A in Equation 5 and 6 as below. It should be noted here that different types of instructions may take different cycles on the functional unit, it is interpreted as that some instructions don't go through all the stages of the functional unit pipeline for their execution. That is why Equation 6 is needed.

$$\begin{aligned} u_{t,i,s} &= a_{(t-s) \bmod II,i} \\ \forall i \in [0, N-1], \forall t \in [0, II-1], \forall s \in [0, d_i - 1] \end{aligned} \tag{5}$$

$$\begin{aligned} u_{t,i,s} &= 0 \\ \forall i \in [0, N-1], \forall t \in [0, II-1], \forall s \in [d_i, L_r - 1] \end{aligned} \tag{6}$$

where d_i is the latency of instruction i and L_r is the maximum latency of all instructions that can be executed on functional unit of type r.

Theorem 3.1 *Two instructions i and j bound to functional unit of the same type can be issued to the same functional unit if and only if $u_{t,i,s} + u_{t,j,s} \leq 1$ for all $t \in [0, II-1]$ and $s \in [0, L_r - 1]$ in the software pipelined loop.*

Theorem 3.2 *In a software pipelined schedule, number of functional units of type r being used at time step t is*

$$F_{t,r} = max \left\{ \sum_{i \in \zeta(r)} u_{t,i,s} | s \in [0, L_r - 1] \right\}.$$

The above equation can be expressed in a linear form as:

$$F_{t,r} \geq \sum_{i \in \zeta(r)} u_{t,i,s} \quad \forall s \in [0, L_r - 1] \tag{7}$$

The above inequality must hold for all t and for all function unit types r.

Let us assume that the power consumed by functional unit of type r is P_r when it is active and $P_r/10$ when it is inactive. The energy consumed by the active functional units of type r during period of length II is:

$$\sum_{t=0}^{II-1} F_{t,r} \cdot P_r$$

The energy consumed by the inactive functional units is:

$$\sum_{t=0}^{II-1} (R_r - F_{t,r}) \cdot \frac{P_r}{10}$$

Thus, the total energy consumed by function units of all types during each period of II is:

$$\frac{9}{10} \cdot \sum_{r=0}^{h-1} \left(P_r \cdot \sum_{t=0}^{II-1} F_{t,r} \right) + \frac{T}{10} \cdot \sum_{r=0}^{h-1} (P_r \cdot R_r)$$

Since the second term in the above expression corresponds to the leakage power of the functional units, which is consumed irrespective of whether or not the functional unit is active, the power-aware software pipelining formulation only need to minimize the first term. To reduce the overall energy consumed by functional units of all types, the objective function should be:

$$\min \sum_{r=0}^{h-1} \left(P_r \cdot \sum_{t=0}^{II-1} F_{t,r} \right) \tag{8}$$

The above objective function should be minimized subject to Inequalities 1 – 7.

4 Results

4.1 Experimental Framework

We implemented our power-aware software pipelining approach in a framework that uses the SGI Pro64 compiler [33], which is an open source compiler targeted to Intel Itanium processor [18, 19]. The SGI Pro64 compiler is invoked with optimization level 3. At optimization level 3, extensive high level optimization including dead code elimination, copy propagation, common subexpression elimination, induction variable elimination and strength reduction are applied on the intermediate representation WHIRL. Then the WHIRL is lowered to another form of intermediate representation, which is called TOP. The transformations applied on TOP before the software pipelining are control flow optimization, block level optimization, if-conversion and critical path reduction. We extract data dependence graphs(DDG) at the beginning phase of software pipelining.

Using this compiler framework we apply our approach on loops extracted from `SPEC2000` integer benchmarks. The statistics on the data dependence graphs used in our experiments is summarized in Table 4. For these DDGs, using a simple program, we generate the integer linear programming formulation (as given in Section 3) and solve these ILP problem using a commercial ILP solver, *CPLEX* [17].

Benchmark	# of DDGs	avg # of nodes	avg # of edges	avg MII
gzip	69	7.3	12.8	3
vpr	93	15.4	47.6	12
gcc	415	11.2	26.4	8
mcf	22	12.8	62.6	9
crafty	86	11.4	33.5	5
parser	116	7.4	16.1	5
eon	101	7.7	19.9	4
perlbmk	182	9.7	25.7	7
gap	620	11.2	28.8	7
vortex	49	18.8	81.3	14
bzip2	63	9.4	21.0	6
twolf	317	18.7	63.8	12
total	2133	12.0	34.0	7.8

Table 4: Statistics on SPEC 2000 Benchmark Evaluated

The target processor for which our integer linear programming formulation is applied is an Itanium-like processor. We use the ISA defined by Intel IA-64 and operation latency defined by Itanium. The total number of instructions defined by the IA-64 architecture is 759. For the 759 different instructions, each of them falls into one of these six categories: *A-type instruction*, *I-type instruction*, *M-type instruction*, *B-type instruction*, *F-type instruction* and *X-type instruction*. The detailed description of each class is given in [18]. Since our power aware software pipelining method applies only to inner-most loops without conditionals, all the instructions in the loop body are non-control-flow instructions. Hence we only need to consider instructions of types *A, I, M* and *F*. In the micro-architecture defined by Itanium, the binding relation between instructions and functional units are as follows: *A-type instructions* can be executed by either I-unit(integer ALU) or M-unit(memory unit), *I-type instructions*, *M-type instructions* and *F-type instructions* are mapped to I-unit, M-unit and F-unit(floating-point unit) respectively. The complex binding of instruction types to functional units defined in [19] necessitates the ILP formulation given in Section 3 to be refined in order to accurately model the resource usage in the Itanium architecture. Interested reader is referred to [39] for the refined formulation.

Lastly, in this study we assume that the power consumption of each functional units as given in Table 5, based on available literature on the DEC Alpha [13] and Intel PentiumPro [26]. Power data in the table is normalized since we are not counting the absolute power, but evaluating the improvement can be made by our approach. We use the power model in Table 5 because power details of Itanium processor is not publicly available, however our ILP formulation can be smoothly re-targetted to Itanium architecture once we have more knowledge on Itanium low-level design details .

FU	I-unit	M-unit	F-unit
numbers	2	2	2
active power	1	1	1
inactive power	0.1	0.1	0.1

Table 5: Power Model

4.2 Experimental Results

We implemented our minimum power software pipelining formulation by integer linear programming using *CPLEX*. For comparison, also we implemented a power-unaware software pipelining formulation, which only use constraints 1 - 4 in Section 3. The power savings, in percentage, is given by:

$$R = \frac{P - P_{min}}{P} \times 100\%$$

where P and P_{min} are, respectively, the power consumed by the power-unaware and the power-aware software pipelining schedules.

Out of the 2133 loops in Table 4, 146 of them is too large for the ILP solver and cannot be solved in a reasonable amount of time. The results for all the other loops are tabulated in Table 6. For each benchmark, we report the number of loops in which (i) no power saving is obtained, (ii) power saving is obtained by applying our power-aware software pipelining approach. For each case, we report the average II. Also for the second case, we report the average power saving in percentage. We observe that in 1174 loops – about 59% cases – the power aware software pipelining approach results in power saving. The overall average power saving in the 1174 loops is 15.9%. For the remaining 813 loops there is no power saving. The reason could be that: (i) the instructions in these DDGs doesn't have slack and thus cannot be moved around to effect power savings; and (ii) even in cases where instructions have slack, the slack may not be sufficient enough to move instructions to reduce the number of active functional units.

Benchmark	DDGs no saving		DDGs with saving		
	#	avg II	#	avg II	avg saving
gzip	36	1.8	33	4.8	19.1%
vpr	25	9.3	60	15.2	9.2%
gcc	184	6.5	224	8.7	16.3%
mcf	5	2.2	16	10.2	15.5%
crafty	45	2.8	40	6.8	22.0%
parser	64	3.3	52	6.8	15.6%
eon	39	2.7	58	5.4	22.9%
perlbmk	82	5.2	97	8.8	16.5%
gap	206	5.6	347	8.0	16.4%
vortex	14	13.6	30	17.1	12.4%
bzip2	33	3.6	28	7.1	18.4%
twolf	80	11.3	189	12.7	12.5%
total	813	5.8	1174	9.2	15.9%

Table 6: Power Saving for SPEC 2000 Benchmarks

Our studies reveal that the average value of II in the power saving loops is typically greater that that for the no-power saving loops in each benchmark. This is not surprising, as higher the value of II, higher is slack for different instructions and there is more scope for the power aware scheduling method to schedule instructions to achieve power reduction.

5 Related Work

Software pipelining has been studied extensively in the literature. Several modulo scheduling methods [6, 16, 21, 31] have been proposed. An extensive survey of these work have been presented in [30]. While many of these work concentrate on getting a rate-optimal schedule, other equally important issues to achieve high performance including register allocation and spill code generation [40, 24], prefetching in both numerical and non-numerical programs [28, 34] have been getting recent attention. Integer linear programming formulation is widely used to derive rate-optimal schedules [11, 12, 1]. Comparison between the rate-optimal scheduling formulation and the software pipelining in MIPSpro, which is a production quality compiler has been made in [32]. A lot of efforts have been put on applying ILP on instruction scheduling [38], register allocation [2] and software pipelining [7] while reducing its time-complexity thus applicability of ILP approach can be widened.

Applying compilation techniques to reduce power consumption is a relatively new topic. Power consumption on a per-instruction basis is analyzed in [35, 29]. Power-aware instruction scheduling [22] and register renaming [27] methods are studied to reduce Hamming distance of adjacent instruction words thus minimizing switching activities on the instruction cache data bus.

There are a lot of ongoing research in micro-architectural level power analysis [9, 3, 37] and reduction [8, 41, 20]. Synergy between compilation techniques and micro-architectural level power-reducing mechanisms is a must to achieve significant power saving. In particular, compiler researchers studied the interaction between program transformation and frequency/voltage scaling [15, 5, 23]. Exploiting schedule slacks to reduce power consumed by execution unit is attracting increasing attention [41, 20]. In contrast to aforementioned works, this paper is focused on current micro-architecture, without the need of introducing frequency and voltage scaling.

6 Conclusions

In this paper we address the problem of generating a software pipelined schedule for loop body that is optimal in terms of the power consumed by the functional units during the execution. This problem is motivated by two observations: (1) functional units in modern processor are fully pipelined; (2) there are instructions in the loop that are not on the critical cycle. By exploiting slacks in the rate-optimal schedule, we can come up with a schedule that consumes less power. This problem is formulated as an integer linear programming(ILP) problem and solved using a commercial solver. We evaluated this approach on `SPEC2000` benchmark and our experimental results show that, out of the 1987 loops tested, our power aware software pipelining approach produces schedules which consume less power in 59% cases. In these cases, the average power saving is 15.9%.

References

[1] Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. of the ACM SIGPLAN '95 Conf. on Program-*

ming Language Design and Implementation, La Jolla, Calif., Jun. 18–21, 1995. *SIGPLAN Notices*, 30(6), Jun. 1995.

[2] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *Proc. of the ACM SIGPLAN '01 Conf. on Programming Language Design and Implementation*, Snowbird, Utah, Jun. 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Ann. Intl. Symp. on Computer Architecture*, pages 83–94, Vancouver, Brit. Col., Jun. 12–14, 2000.

[4] Jason Casmira and Dirk Grunwald. Dynamic instruction scheduling slack. In *Proceedings of the 2000 KoolChips workshop*, Monterey, California, Dec 10th 2000.

[5] C.Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of International Symposium on Low Power Electronics and Design(ISLPED) 2001*, 2001.

[6] James C. Dehnert and Ross A. Towle. Compiling for Cydra 5. *Journal of Supercomputing*, 7:181–227, May 1993.

[7] Alexandre E. Eichenberger and Edward S. Davidson. Efficient formulation for optimal modulo schedulers. In *Proc. of the ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 194–205, Las Vegas, Nev., Jun. 15–18, 1997. *SIGPLAN Notices*, 32(6), Jun. 1997.

[8] Daniele Folegnani and Antonio González. Energy-effective issue logic. In *Proc. of the 28th Ann. Intl. Symp. on Computer Architecture*, pages 230–239, Göteborg, Sweden, Jun. 30–Jul. 4, 2001. IEEE Comp. Soc. and ACM SIGARCH. *Computer Arch. News*, 29(2), May 2001.

[9] G.Cai and C.H.Lim. Architectural level power/performance optimization and dynamic power estimation. Cool Chips Tutorial, in conjunction with 32nd Annual International Symposium on Microarchitecture. Haifa, Israel, Nov 1999.

[10] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9), 1996.

[11] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 85–94, San Jose, Calif., Nov. 30–Dec.2, 1994.

[12] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1133–1149, November 1996.

[13] Michael K. Gowan, Larry L. Biro, and Daniel B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *35th Design Automation Conference Proceedings 1998*, San Francisco, CA, June 15-19 1998.

[14] Stephen H. Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, Feb 2001.

[15] C-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems (PACS'00)*, Cambridge, MA, Nov 2000.

[16] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, Albuquerque, N. Mex., Jun. 23–25, 1993. *SIGPLAN Notices*, 28(6), Jun. 1993.

[17] ILOG. CPLEX mixed integer solver. http://www.cplex.com.

[18] Intel. *Intel IA-64 Architecture Software Developer's Manual*, Aug 2000.

[19] Intel. *Itanium Processor Microarchitecture Reference for Software Optimization*, Aug 2000.

[20] J.Seng, E.Tune, and D.Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th Annual International Symposium on Micro-architecture*, Austin, TX, Dec 2001.

[21] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, Geor., Jun. 22–24, 1988. *SIGPLAN Notices*, 23(7), Jul. 1988.

[22] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang. Compiler optimization on instruction scheduling for low power. In *Proceedings of 13th International Symposium on System Synthesis*, Madrid, Spain, Sept 20-22 2000.

[23] Tao Li and Chen Ding. Instruction balance and its relation to program energy consumption. In *Proceedings of International Workshop on Languages and Compilers for Parallel Computing*, Kentucky, Aug 2001.

[24] Josep Llosa, Eduard Ayguadé, Antonio Gonzalez, Mateo Valero, and Jason Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Transactions on Computers*, 50(3):234–249, Mar 2001.

[25] Josep Llosa, Mateo Valero, Eduard Ayguadé, and Antonio González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 350–360, Ann Arbor, Mich., Nov. 29–Dec.1, 1995.

[26] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3 of *ACM Computer Architecture News*, pages 132–141, New York, June 27–July 1 1998. ACM Press.

[27] M.Kandemir, N. Vijaykrishnan, M. J. Irwin, W. Ye, and I. Demirkiran. Register relabeling: A post-compilation technique for energy reduction. In *Workshop on Compilers and Operating Systems for Low Power 2000 (COLP'00)*, 2000.

[28] T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

[29] M.T.-C.Lee, V.Tiwari, S.Malik, and M.Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans on Very Large Scale Integration(VLSI) Systems*, 5(1), Mar 1997.

[30] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Journal of Supercomputing*, 7:9–50, May 1993.

[31] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 63–74, San Jose, Calif., Nov. 30–Dec.2, 1994.

[32] John Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 1–11, Philadelphia, May 22–24, 1996. *SIGPLAN Notices*, 31(6), Jun. 1996.

[33] SGI. Pro64 compiler. http://open64.sourceforge.net.

[34] Artour Stoutchinin, José Nelson Amaral, Guang R. Gao, Jim Dehnert, Suneel Jain, and Alban Douillet. Speculative prefetching of induction pointers. In *Proceedings of The International Conference on Compiler Construction*, Lecture Notes in Computer Science. Springer-Verlag, April 2001.

[35] Vivek Tiwari, AHARAD MALIK, and Andrew Wolfe. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 1996.

[36] Vivek Tiwari, Deo Singh, Suresh Rajgopal, Gaurav Mehta, Rakesh Patel, and Franklin Baez. Reducing power in high-performance microprocessors. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 732–737, Los Alamitos, CA, June 15–19 1998. ACM/IEEE.

[37] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. of the 27th Ann. Intl. Symp. on Computer Architecture*, pages 95–106, Vancouver, Brit. Col., Jun. 12–14, 2000. IEEE Comp. Soc. and ACM SIGARCH. *Computer Arch. News*, 28(2), May 2000.

[38] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *Proc. of the ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation*, pages 121–133, Vancouver, Brit. Col., Jun. 18–21, 2000. *SIGPLAN Notices*, 35(5), May 2000.

[39] Hongbo Yang, R.Govindarajan, Guang R. Gao, and George Cai. Maximizing pipelined functional units usage for minimum power software pipelining. Technical Report 41, Computer Architecture and Parallel Systems Laboratory, University of Delaware, 2001. ftp://ftp.capsl.udel.edu/pub/doc/memos/memo041.ps.gz.

[40] Javier Zalamea, Josep Llosa, Eduard Ayguade, and Mateo Valero. Improved spill code generation for software pipelined loops. In *Proceedings of ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver B.C., Canada, June 18-21 2000.

[41] W. Zhang, N.Vijaykrishnan, M.Kandemir, M.Irwin, D.Duarte, and Y. Tsai. Exploiting vliw schedule slacks for dynamic and leakage energy reduction. In *Proceedings of the 34th Annual International Symposium on Micro-architecture*, Austin, TX, Dec 2001.

On the Itanium Architecture, three types of functional units, namely integer ALU, memory unit, and floating-point ALU There are two instances for each type of functional units, and are referred to as I0, I1, M0, M1, F0 and F1. The instructions in Itanium are classified as of *A-unit*, *I-unit*, *M-unit* and *F-unit* type instructions. The binding relation between the instruction types and functional units is as follows: *A-unit* instruction can be executed on either integer ALUs or memory units, i.e., on I0, I1, M0 or M1. *I-unit* instructions are bound only to integer ALU (i.e., I0 or I1), M-unit instructions to memory unit (i.e., M0 or M1), and *F-unit* instructions to floating point ALU (i.e., F0 or F1). The two instances of the functional units in each type are slightly different. While a large majority of *M-unit*, *I-unit*, *F-unit* instructions can be executed on any of the two instances, a small portion of instruction can only be bound to one of them, namely M_0, I_0 or F_0.

Hence, the ILP formulation in Section 3 should be refined to reflect the flexibility in scheduling *A-unit* instructions to either Integer ALUs or Memory units, as well as to account the asymmetry between two instances in each functional unit type. We define the functional unit set as:

$$R = \{I_0, I_1, M_0, M_1, F_0, F_1\}$$

For an *A-unit* instruction i, the following equation relates $a_{t,i}$ and the specific functional unit (one of I_0, I_1, M_0 and M_1) to which it is mapped to.

$$a_{t,i} = I0_{t,i} + I1_{t,i} + M0_{t,i} + M1_{t,i} \qquad \forall i \in A \tag{9}$$

where the right hand side variables are all 0-1 integer variables. $I0_{t,i}$ is true if and only if instruction i is mapped to I_0 and is scheduled at time step t; similarly for variables $I1_{t,i}, M0_{t,i}$, and $M1_{t,i}$. Likewise, we can define similar variable to indicate the mapping of *I-unit*, *M-unit*, and *F-unit* instructions. The respective equations are:

$$a_{t,i} = M0_{t,i} + M1_{t,i} \qquad \forall i \in M \tag{10}$$

$$a_{t,i} = I0_{t,i} + I1_{t,i} \qquad \forall i \in I \tag{11}$$

$$a_{t,i} = F0_{t,i} + F1_{t,i} \qquad \forall i \in F \tag{12}$$

For specific instructions which can only go to I0, M0 or F0 functional units we have:

$$a_{t,i} = I0_{t,i} \qquad \forall i \in I_0 \tag{13}$$

$$a_{t,i} = F0_{t,i} \qquad \forall i \in F_0 \tag{14}$$

$$a_{t,i} = M0_{t,i} \qquad \forall i \in M_0 \tag{15}$$

The resource constraints on the functional units F is formulated as:

$$\sum_{i \in \zeta(I0)} I0_{t,i} \leq 1, \qquad \forall t \in [0, II-1] \tag{16}$$

$$\sum_{i \in \zeta(I1)} I1_{t,i} \leq 1, \qquad \forall t \in [0, II-1] \tag{17}$$

$$\sum_{i \in \zeta(M0)} M0_{t,i} \leq 1, \qquad \forall t \in [0, II-1] \tag{18}$$

$$\sum_{i \in \zeta(M1)} M1_{t,i} \leq 1, \qquad \forall t \in [0, II-1] \tag{19}$$

$$\sum_{i \in \zeta(F0)} F0_{t,i} \leq 1, \qquad \forall t \in [0, II-1] \tag{20}$$

$$\sum_{i \in \zeta(F1)} F1_{t,i} \leq 1, \qquad \forall t \in [0, II-1] \tag{21}$$

Lastly we note that $u_{t,i,s}$ can still defined in terms of $a_{t,i}$ as in Equations 5 and 6. Similarly, $F_{t,r}$ in Equation 7 is same as before, except that the definition of $i \in \zeta(r)$ changed appropriately as $i \in \zeta(I0)$, etc. formulation in Section 3 to the variables $I0_{t,i}$, $I1_{t,i}$, etc.