Exploiting Schedule Slacks for Rate-Optimal Power-Minimum Software Pipelining*

Hongbo Yang† R. Govindarajan‡ Guang R. Gao† George Cai♯ Ziang Hu†

†Dept of ECE
University of Delaware
Newark, DE 19716, USA
{hyang,ggao,hu}@capsl.udel.edu

‡Dept of CSA
Indian Institute of Science
Bangalore, 560012, INDIA
govind@csa.iisc.ernet.in

♯Intel Corp
RA2-401,2501 N.W. 229th Ave
Hillsboro, OR 97124, USA
george.cai@intel.com

Abstract

Increasing power consumption in high performance processors and the proliferation of embedded systems demand new compiler techniques geared toward both high performance and low power. Software pipelining, an effective compiler optimization to exploit instruction level parallelism across loop iterations, has been studied extensively. However, previous software pipelining methods focus on performance only. This paper presents a software pipelining method that reduces power consumption while keeping performance optimality. This is accomplished as schedule slacks exist for non-critical instructions even in performance optimal schedules, and by exploiting the slack appropriately, it may be possible to reduce the number of functional units used in the schedule.

This paper formulates the problem of minimizing power consumption in software pipelined loops as an integer linear programming (ILP) problem and experimented this approach in a re-engineered MIPSpro compiler. By executing the program generated by the re-engineered compiler on Wattch power simulator, we observe that our approach can reduce dynamic energy consumption of a set of kernels from `SPEC` *benchmarks by up to 15.4% (8.5% on an average) in comparison with the existing scheduler of MIPSpro compiler, which strive only for high performance. The proposed approach is also helpful to reduce leakage power. With a leakage power reduction mechanism which applies power supply gating whenever possible, schedules generated by our approach consume up to 54.5% (or 31.8% on an average) less leakage power than that consumed by the MIPSpro compiler generated schedules.*

*This work is supported in part by DARPA, contract number 1120-24596, SGI, Delaware Research Partnership(DRP) program, grant number 002672, and Intel Corp, grant number 2002-0814.

1 Introduction

Power consumption is becoming one of the major design constraints in today's high performance processors and embedded systems. Excessive power consumption in high performance processors in small chip areas cause high chip temperature, affecting both hardware packaging cost and processor reliability adversely. For embedded systems, particularly hand-held devices, reducing power consumption also helps lengthening the battery life. Technology trend suggests that compiler can play an important role in reducing power consumption, calling for new compilation techniques which can achieve both high performance and low power or make judicious power-performance tradeoffs at compile-time.

Software pipelining [16] is an effective instruction scheduling technique to exploit instruction level parallelism across loop iterations. In a software pipelined schedule, the execution of instructions belonging to different iterations of a loop are overlapped. Successive instances of an instruction corresponding to different iterations are separated by II *(initiation interval)* cycles. The objective of a software pipelining method is to construct a schedule that has a high computation rate, or equivalently a low II. In the past, resource constrained software pipelining has been studied extensively by several researchers and a number of *modulo scheduling* algorithms [7, 13, 16, 22] have been proposed. For a comprehensive survey of software pipelining methods the reader is referred to [21].

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 two 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 path in the loop or they do not use the critical resource(s). Such slacks have been profitably used to reduce the register pressure in some of the software pipelining methods [13, 17]. Second, in a modern VLIW architecture, functional units are usually fully pipelined. For a fully pipelined functional unit, on each cycle, a new instruction can be issued to the functional unit for execution. Thus it is possible to delay the schedule time of certain non-critical instructions and obtain a schedule that uses fewer number of functional units. Such schedules often do not degrade the performance, but influence the power consumption in a favorable manner.

More specifically, a reduction in the number of simultaneously used functional units, allows more functional units to be in an idle or inactive state. On such inactive functional units, *clock-gating*, a circuit level design technique to achieve low power consumption by disabling toggling of the functional unit input clock, can be employed. Modern processors use clock gating extensively. The rationales for such a design are: (i) functional units are not fully utilized in general; (ii) a large proportion of processor power is dispensed to the clock network. Clock-gating can be applied either at functional unit level or at pipeline stage level. The disadvantage of applying clock-gating at pipeline stage level (fine-grain clock-gating) is that it may generate glitches, causing clock skew and a large increase in the complexity of timing verification, especially at high frequencies [11, 25, 12]. Thus, clock-gating applied at functional unit level is a reasonable design choice. Alpha 21264 microprocessor is an example of such a design [11], in which the divider datapath is operated on a single conditional clock. Clock-gating applied on functional unit level means that the input clock is gated only when none of the functional unit stages is active. In such a design, minimizing the number of functional units used effectively increases the number of functional units that will be clock-gated and hence minimize the overall dynamic power consumption of functional units.

Another major source of chip power consumption, other than the dynamic power, is due to leakage current. This component, referred to as leakage power, is gaining increasing importance especially with decreasing feature size [6]. Functional units always consume leakage power regardless of whether it is active or inactive. Proposed circuit level techniques to reduce leakage power, such as power supply gating [29], have the potential to reduce leakage power to zero. Power supply gating essentially turns off (shuts down) a functional unit when it is inactive. These techniques may require a significantly large number of cycles (in the order of a few hundreds or even thousands of cycles) for the transition between power-on and power-off modes. Thus power supply gating can be applied only on those functional units which are inactive for a sufficiently long period. Since our technique minimizes the number of functional units used in the software pipelined loop, the remaining inactive functional units can be shut down with power supply gating. Thus, our approach can achieve a larger leakage power reduction with the help of power supply gating.

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

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

In this paper we formulate the power-aware software pipelining problem as an integer linear programming (ILP) problem. ILP-based instruction scheduling approach is often acceptable in the context of compiling for embedded system, where the application is compiled once at the vendor site and run repeatedly at the customer site. Further, even in the context of compiling for general purpose processors, as argued in an earlier work [23], ILP based scheduling methods are still useful for the evaluation of (performance) bounds that can be achieved by any heuristic based method. This paper extends our previous work on rate-optimal software pipelining [9, 2] by modeling functional units usage and their power consumption on a cycle-by-cycle basis and trying to minimize the overall dynamic power consumption. The proposed approach is also helpful in enhancing the leakage power reduction achieved with power supply gating.

Our power-aware software pipelining method has been implemented in a re-engineered MIPSpro compiler targeting SimpleScalar [4], The energy consumption of the compiled code is obtained through Wattch simulator [3] with our enhancement to model power consumption of functional units more accurately. We compared our approach with the existing scheduling method in MIPSpro. Our experimental results with `SPEC NAS` floating point benchmarks reveal that the proposed approach can reduce dynamic power consumption in functional units by up to 15.4% (with an average of 8.5%). Initial experiments on leakage power reveal that our approach yields up to 51.5% (with an average of 31.8%) reduction in leakage power.

The rest of 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. We describe the experimental framework in Section 4. The experimental results are reported in Section 5. A discussion on related work is presented in Section 6. Section 7 concludes our work.

2 Motivating Example

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

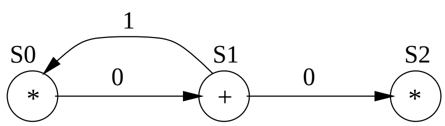

Figure 1: Data Dependence Graph

Consider a VLIW architecture with 1 adder and 2 multipliers. The adder is a single stage pipeline (i.e., a trivial case of a fully pipelined unit) and takes one cycle to complete an add operation. The two multipliers are 2-stage pipelines and take two cycle to complete a multiplication. We assume that an adder in active and inactive states consumes respectively 1 and 0.1 unit of power. Also we assume the multiplier consumes same amount of power as the adder. Table 1 summarizes the configuration of the underlying VLIW processor. Note that configuration shown in Table 1 is used just for illustration purposes and is only valid throughout this section. Our proposed formulation is flexible enough to handle any architectural setting.

	Adder	Multiplier
Number of Units	1	2
Number of Stages	1	2
Active power	1	1
Inactive power	0.1	0.1

Table 1: Target Machine Configuration

Consider the data dependence graph (DDG) shown in Figure 1, where there are three instructions (one add and two multiply instructions) in the loop. The directed arcs indicate data dependences. Each arc is associated with a number which represents the dependence distance. We illustrate how a rate-optimal software pipelined schedule can be altered to achieve reduced dynamic energy and leakage energy consumption without sacrificing performance.

2.1 Dynamic Energy Reduction

The lower bound on the initiation interval (II) is the maximum of the II bound imposed by the recurrences in the loop and that imposed by the resource constraints. These are referred to as RecMII and ResMII respectively [21]. RecMII is given by :

$$RecMII = \max \left\{ \left\lceil \frac{d(C)}{m(C)} \right\rceil \mid C \text{ is a cycle in } G \right\}$$

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 cycle involving $S0$ and $S1$, which makes the RecMII 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 that use functional unit of type r for its execution, R_r is the number of functional units of type r, and h is the number of functional unit types. For the DDG shown in Figure 1, ResMII is 1.

The lower bound on II, Minimum Initiation Interval (MII), is the the maximum of *RecMII* and *ResMII*. For our example DDG, *MII* is $\max(3, 1) = 3$. Table 2 gives a possible schedule for this DDG with II $=$ 3. This is a rate-optimal schedule since *II* equals *MII*.

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 dynamic 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 scheduled, and they must be issued to two different multipliers to avoid structural hazard. Hence, the two multipliers are in use until the end of of cycle 4. At cycle 5, add instruction

$S1$ is issued to the adder, and the adder is used for one cycle. Thus the two multipliers are busy for two consecutive cycles and idle for one cycle, consuming a total energy of 4 units. The adder is busy for one cycle and idle for the remaining two cycles, consuming total energy of 1. Thus the total energy consumed by all the functional units is 5.

An interesting question to ask is: does there exist any other rate-optimal schedule which consumes less dynamic energy? The answer is yes and such a schedule is shown in Table 3. Let us calculate the dynamic energy consumed by the new schedule. At cycle 3, $S0$ is scheduled and it goes to one of the two multipliers. At cycle 4, $S2$ is scheduled. Since the multiplier is fully pipelined, $S2$ can be issued 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 cycle 5, thus, in this new schedule, one of the two multipliers is active for all 3 cycles (from time step 3 to 5) consuming an energy of 3 units while the other one is always idle, consuming no dynamic energy. Thus the dynamic energy consumed by these two multipliers during a time interval of length II is 3. As for the adder is concerned, it consumes the same amount of energy as in the previous schedule, which is 1 unit. Taking all the functional units into account, the dynamic energy consumption of the new schedule is 4. Compared to the earlier schedule, the new schedule consumes 20% less dynamic energy.

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

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

2.2 Leakage Energy Reduction

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 is clock-gated, resulting in reduction in dynamic energy (or average dynamic power). Next, let us discuss how the same method can also help to reduce leakage energy. Assume that the above software pipelined schedule is iterated 1000 times. Since II is 3, it takes 3000 cycles for completion of the loop (ignoring prologue and epilogue). Totally we have 3 functional units. Without any leakage energy reduction mechanism, in each cycle, the leakage power consumed by these functional units is 0.3 unit. Thus, the leakage energy consumed in the repetitive pattern (with a length of 3 cycles) is 0.9 unit . The schedule shown in Table 2 consumes $0.9 \times 1000 = 900$ units of leakage energy. Since a functional unit is inactive only for at most 2 consecutive cycles, which is much less than the transition time between power-on and power-off modes, none of the functional units in this schedule can be power-supply-gated. Thus, the functional units in this schedule continue dissipating leakage power during the inactive time steps.

However, for the schedule shown in Table 3, the second multiplier is inactive for all 3000 cycles and can be powered down during the loop execution. Assume 500 cycles for the transition from power-on to power-off state (referred to as initiation latency) and another 500 cycles for the transition from power-off to power-on modes (referred to as recovery latency) [29]. For the schedule shown in Table 3, we can eliminate the leakage current in the second multiplier for 2000 cycles. Note that the functional unit still consumes leakage energy during the transition period. Thus, the schedule shown in Table 3 reduces the leakage energy by $0.1 \times 2000 = 200$ units, resulting in 22.2% reduction in total leakage energy.

As our motivating example shows, indeed, there exist considerable opportunities to reduce both dynamic and leakage 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 iteration 0. The schedule time of all instructions in iteration 0 is represented by an N-element vector

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

The repetitive pattern is represented by an $II \times N$ matrix A. Each element of matrix A is either 0 or 1; $a_{t,i} = 1$

if instruction i is issued at time step t in the repetitive pattern and 0 otherwise. 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,

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

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 the resource constraints. Let $\zeta(r)$ denote the set of all instructions that are executed in functional units of type r. A fully pipelined functional unit is exclusively occupied by an 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.

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

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

The resource and dependence constraints in our formulation are same as that in [9, 10]. Next we formulate the power consumption of the software pipelined schedule which is our contribution in this paper. To model the power consumption at each time step, we need to know the number of active functional units of each functional unit type. 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 $U = [u_{t,i,s}]$, defined as:

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

From above discussion, $u_{t,i,s}$ can be defined using matrix A as below:

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

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

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. Since different types of instructions (e.g., floating point divide and square root) which execute in the same functional unit may have different execution latencies, we need Equation 6 which appropriately defines $u_{t,i,s}$ for instruction type which has shorter execution latency.

Claim 3.1 *Two instructions i and j bound to a 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 schedule.*

Claim 3.2 *In a software pipelined schedule, the number of functional units of type r used at time step t is given by:*

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

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 functional 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 a period of length II is:

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

The energy consumed by the inactive functional units of type r 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 a period of II cycles 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 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[1]. Thus 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 Experimental Framework

We implemented our power-aware software pipelining approach in the MIPSpro compiler, a production-quality compiler from SGI targeting MIPS processors. For this work, the most relevant components of MIPSpro compiler are software pipelining [23] and integrated global and local scheduling(IGLS) [18]. The compiler constructs the data dependence graph for each basic block. A data dependence graph is scheduled by the software pipelining method, if the compiler deems the loop enclosing the basic block as software-pipelineable, or by the IGLS method otherwise. We plugged in our ILP approach to both these components (namely software pipelining and IGLS scheduling). The ILP formulation discussed in the previous section is easily adapted for optimal instruction scheduling for basic blocks. Both software pipelining and IGLS in the MIPSpro compiler are performance-oriented but power-unaware.

Since most power simulation tools [3, 26, 8] are built on top of SimpleScalar [4], to facilitate power simulation, we re-engineered the MIPSpro compiler to generate SimpleScalar assembly code. Also the machine description of MIPSpro compiler is modified to reflect our target machine configuration which is used in the simulation. We chose Wattch [3] for power simulation. Wattch is an activity-based power simulator based on SimpleScalar. However, the current implementation of Wattch simulates functional unit power by assuming that each instruction access the functional unit only for a single cycle. It does not model the pipeline stages of functional units from power perspective. By keeping track of the instructions executed in the functional unit pipelines, our enhanced Wattch simulator counts number of functional units that are active at every cycle. Power is then calculated based on functional unit activities.

Another important modification required is that SimpleScalar simulates superscalar processors while we want to study the effect of our software pipelining method on VLIW architectures, where the run-time instruction issue behavior strictly conforms to the schedule generated by the compiler. We used the annotation bits of SimpleScalar instruction word for this purpose. Our re-engineered MIPSpro compiler annotates assembly instructions according to the schedule constructed. During the simulation, only instructions with same annotation fields can be issued in the same cycle.

In our experiments we assumed a VLIW architecture which can issue up to 6 instructions in every cycle. The latency and the number of functional units available in the architecture are summarized in Table 4. The **ALU** units carry out simple arithmetic calculations like `add` and `sub` and the **MULT** function units is for more complicated arithmetic operations like `mul` and `div`.

We use the `NASA` kernels from `SPEC` floating point bench-

[1] As we shall see later, our formulation only attempts to reduce the dynamic power/energy. However the same objective function also helps to reduce the leakage energy.

FU type	numbers of FUs	latency (in cycles)
Int ALU	3	1
Int MULT	3	2
FP ALU	3	2
FP MULT	3	4

Table 4: Processor Configuration

mark suite to evaluate effectiveness of our approach.[2] We report performance results for 5 of the 7 kernels which can be compiled by our re-engineered MIPSpro compiler and run correctly on the simulator. The benchmarks used are `mxm` (matrix multiply), `cfft2d` (complex 2D-FFT), `cholsky` (Cholsky solver for systems of linear equations), `btrix` (vectorized block tridiagonal solver) and `vpenta` (inverts 3 pentadiagonal matrices simultaneously). In these benchmarks, floating point units are used much more extensively than integer units. Thus, in the following discussion, we focus on the power consumed by **FP_ALU** and **FP_MULT** units of the architecture.

The benchmarks are compiled at the highest optimization level `-O3` of the MIPSpro compiler. At this level, optimization modules such as alias analysis, strength reduction, common subexpression elimination, global and local register allocation, loop unrolling and integrated global and local scheduling, software pipelining are invoked. Besides, precise data dependence analysis is also performed [1] which eliminates ambiguous dependences in the data dependence graph, giving more freedom to block-level instruction scheduling and software pipelining methods.

For our ILP implementation, the DDGs of the basic blocks are extracted by the MIPSpro compiler. An ILP problem is generated for each DDG. It should be noted here that in the MIPSpro compiler, if the DDG is not software-pipelineable, it will be scheduled by IGLS more than once during the compilation, each time with a different objective. Our ILP approach is applied at the last invocation of IGLS (i.e., after global and local register allocation), since at this pass the instructions best resemble the final assembly code.

Lastly, the generated ILP problem is solved using CPLEX [14], a commercial ILP solver. The resulting schedule is fed back into the compiler, and the compiler completes the remaining steps of compilation to generate assembly code.

[2] Our approach is not profitable in integer benchmarks since most integer operations execute for only one cycle, while our software pipelining method attempts to increase the utilization of the same FUs one of whose stage is already active.

5 Experimental Results

Our major results are summarized as follows:

- Our approach reduces dynamic energy consumed by floating point functional units by up to 15.4% (or 8.5% on an average) for `NASA` kernels, without any performance degradation.
- Assuming a perfect power supply gating mechanism, our ILP generated schedules achieves up to 51.5% (with an average of 31.8%) reduction in leakage energy than the MIPSpro generated schedules.

5.1 Reduction of Dynamic Energy

We compared the dynamic energy consumed by our power-aware schedulers and that produced by the power-unaware MIPSpro schedulers. Table 5 shows the reduction in dynamic energy achieved by our approach on the two kinds of floating point functional units, namely **FP_ALU** and **FP_MULT**. The table also reports the overall reduction in dynamic energy for the various benchmark programs. As shown in the table, `vpenta` benefits most from our approach. The overall energy reduction is 15.4%. The FU energy consumption of `btrix` does not reduce significantly, even with the power-aware scheduler. Further study on the benchmark reveals that it has a large basic block with a large II = 96, our software pipelining method generates a large ILP problem and the ILP solver fails to construct optimal schedule for this within a given time limit (3 minutes). We also observe a moderate reduction in the energy consumed by either the **FP_ALU** or the **FP_MULT** unit in each of the other benchmarks.

Benchmark	**FP_ALU**	**FP_MULT**	Total
`mxm`	12.5%	0%	6.7%
`fft`	4%	15.2%	10%
`cholsky`	3%	13%	9.3%
`btrix`	2%	1%	1.1%
`vpenta`	7.4%	19%	15.4%
average	5.8%	9.6%	8.5%

Table 5: Saving on Dynamic Energy Consumed by FUs

Since dynamic power is directly derived from functional unit activities [19], power savings shown in Table 5 is achieved through reduced functional unit activities. To illustrate the reduction in functional unit activities, we define $T(x)$ which represents the number of cycles at which the number of active functional units used is greater than or equal to x. Figure 2 shows $T(x)$ for the schedules generated by our ILP approach and by the MIPSpro compiler for the mxm program, where x ranges from 1 to 5. Note that floating point units do not consume any dynamic power for $T(0)$ cycles. In no time step, all 6 FP units were used simultaneously in mxm.

As shown in the figure, T(5) and T(4) are reduced by 100% and 50% respectively. $T(x)$ for lower values of x ($x = 2$ and $x = 3$) also reduces considerably. Our schedules have a higher $T(1)$ value compared to that of MIPSpro schedules. This seems to be counter-intuitive to our claim that our approach is reducing number of functional units being used. However, actually this is not the case and the reasoning is as follows. Our approach schedules instructions in such a way to use only a subset of the available functional units extensively while releasing other FUs. Therefore those FUs being used must be used for longer time than the original schedule (although total execution cycle for the program is not adversely affected), which makes $T(x)$ larger for some small x. To illustrate this further, consider the example

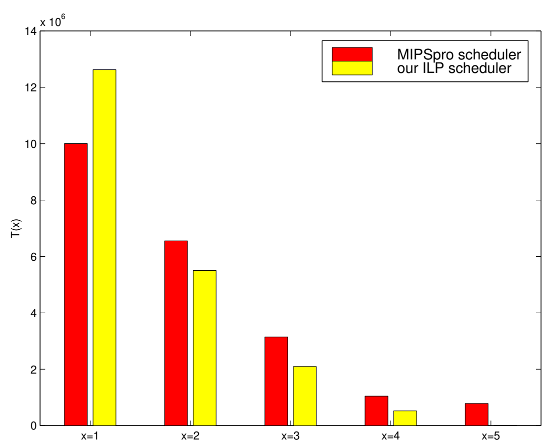

Figure 2: Reduction of FU Dynamic Activities by Our ILP Approach

discussed in Section 2. For the purpose of this discussion we will restrict our attention to only **FP_MULT** functional units. One of the multipliers is active for all 3 cycles in the schedule shown in Table 3, while the other is inactive throughout the loop. In this example, $T(1) = 3$, and $T(2) = 0$. Whereas for the schedule shown in Table 2, $T(1) = 2$ and $T(2) = 2$. Thus even though $T(1)$ is larger for the schedule shown in Table 3, it consumes less dynamic power (as discussed in Section 2) than the other schedule (shown in Table 2). Therefore, in the mxm program, our net gain on dynamic energy is still positive even though $T(1)$ of our schedule is larger than the MIPSpro generated one.

5.2 Leakage Power Reduction

Leakage power constitutes an increasingly large proportion of power budget, especially with decreasing feature size. This section studies how our approach helps to reduce the leakage power.

Proposed circuit level techniques to reduce leakage power such as power supply gating have the potential to reduce leakage power to zero. As discussed in Section 2.2, since the transition time between normal leakage power and zero leakage power is in the order of hundreds or even thousands of cycles, power supply gating is effective only when the functional units are inactive for long durations. As illustrated in Section 2, our approach reduces the number of simultaneously active functional units. This, in turn, may release a subset of available functional units for long time, facilitating power supply gating to be applied. Thus our approach constructs schedules that have more leakage reduction potential, which can be exploited with some architecture support.

In an attempt to evaluate the reduction in leakage energy reduction, we analyzed the execution trace of the schedule generated by our ILP approach and the MIPSpro compiler. In both cases, we assumed a perfect leakage power reduction mechanism. We assume that the leakage power of a functional unit is 0 at time step $(t+x)$ if it has been inactive since time t, where x is the initiation latency for power supply gating. The leakage power of the functional unit remains 0 until time step $(t' - y)$, where t' is the time step in the dynamic trace at which the function unit is active again and y is the recovery time for power supply gating. Note that in order to implement the power supply gating at $(t + x)$ and to start the recovery at time step $(t' - y)$, additional hardware support, in the form of instruction set architecture extension, and compiler support, in the form of power-down and power-on hints, may be required. In our experiment, we set $x = 500$ and $y = 500$. Further, we used normalized values for leakage energy and computed the reduction in leakage energy based on the execution trace obtained from the simulator. Note that, in our experiments, a functional unit still consumes leakage power during initiation and recovery times.

Our experimental results on leakage energy reduction are tabulated in Table 6. As seen from the table, with a perfect power supply gating, our approach can reduce the leakage energy by up to 51.5% (for cholsky). The average leakage energy reduction is 31.8%.

Benchmark	**FP_ALU**	**FP_MULT**	Total
`mxm`	34.8%	34.8%	34.8%
`fft`	0%	11.7%	7%
`cholsky`	45.7%	56.8%	51.5%
`btrix`	8%	28.6%	18.5%
`vpenta`	66.1%	23.7%	47.3%
average	31%	31.1%	31.8%

Table 6: Saving on Leakage Energy Consumed by FUs

6 Related Work

Exploiting instruction schedule slack to reduce power consumption by functional units with minimal performance degradation has drawn a lot of attention. A limit study on available schedule slack was reported by Casmira et al [5]. Their result shows that during the course of program execution, for more than 75% of the execution cycles, there exists at least one instruction that has schedule slack. However they didn't propose any technique to exploit such instruction scheduling slack for energy saving. Pyreddy et al [20] extend this idea and use run-time profiling to guide instruction issue to two sets of functional units, fast and slow functional units. The slow functional units are run at a lower frequency, typically half that of the fast functional units. The work of Seng et al [15] and that of Zhang et al [29] studied the performance and power implications of architecture of this kind by applying micro-architectural and compilation techniques to harness functional units with different speed to achieve maximal energy saving with minimal performance degradation. The above work [20, 15, 29] assume hardware support in the form of frequency and voltage scaling. Our work is different from their approach, in that it constructs rate-optimal power-minimum software pipelining schedule using integer linear programm formulation. Our work does not require such hardware support for frequency and voltage scaling.

A relevant work in power-aware software pipelining is by Yun et al [28]. This work introduces certain modifications to the iterative modulo scheduling algorithm [22] to minimize *step power* — the difference in the power consumed between two consecutive time steps — for a software pipelined loop. Their objective is to derive a schedule under which "power consumption is better balanced" under a VLIW architecture. This is different from our objective which achieves rate-optimal power-minimum software pipelined schedule.

Leakage power reduction is receiving increasing attention. Rele et al's work [24] and a recent work by You et al [27] analyzed functional unit usage throughout program exeuction and applied power supply gating on unused functional units. Our work is complementary to their work since the schedule generated by our approach reduces the number of active functional units. This will facilitate their approaches to have even larger gain on leakage power reduction.

7 Conclusions

In this paper we address the problem of generating a software pipelined schedule for loop body that is optimal in terms of both performance and the power consumed by the functional units. By exploiting schedule slacks in the rate-optimal schedule, we can come up with a schedule that consumes less dynamic power. This problem is formulated as an ILP problem. The objective function of the ILP formulation minimizes the dynamic energy (due to toggles/activities). Interestingly, the proposed formulation for reducing dynamic energy also helps to reduce energy due to leakage current.

We plugged in our ILP approach in the MIPSpro compiler and evaluated the power consumption for the generated code using an enhanced Wattch simulator. Our approach is evaluated using `NASA` kernels from `SPEC` benchmarks and compared with the original scheduling method in the MIPSpro compiler which is performance-oriented but power-unaware. Our comparison shows that the power-aware software pipelining approach reduces energy consumption in functional units by up to 15.4% (8.5% on an average) with no degradation in performance. Further, our experiments show that the proposed approach reduces leakage energy by up to 51.5% (with an average of 31.8%).

References

[1] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[2] Erik R. Altman, R. Govindarajan, and Guang R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. of SIGPLAN PLDI '95*, pages 139–150, La Jolla, Calif., Jun. 1995.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proc. of ISCA-27*, pages 83–94, Vancouver, Brit. Col., Jun. 2000.

[4] Doug Burger and Todd Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, Univ of Wisconsin, 1997.

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

[6] Vivek De and Shekhar Borkar. Technology and design challenges for low power and high performance. In *Proceedings of the 1999 International Symposium on Low Power Electronics and Design(ISLPED'99)*, San Diego, CA, Aug 1999.

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

[8] 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.

[9] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of MICRO-27*, pages 85–94, San Jose, Calif., Nov.–Dec. 1994.

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

[11] 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.

[12] 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.

[13] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of SIGPLAN PLDI '93*, pages 258–267, Albuquerque, N. Mex., Jun. 1993.

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

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

[16] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of SIGPLAN PLDI '88*, pages 318–328, Atlanta, Geor., Jun. 1988.

[17] Josep Llosa, Mateo Valero, Eduard Ayguadé, and Antonio González. Hypernode reduction modulo scheduling. In *Proc. of MICRO-28*, pages 350–360, Ann Arbor, Mich., Nov.–Dec. 1995.

[18] Srinivas Mantripragada, Suneel Jain, and Jim Dehnert. A new framework for integrated global local scheduling. In *Proc. of the 1998 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 167–174, Paris, Oct. 1998.

[19] Trevor Mudge. Power: A first-class architectural design constraint. *COMPUTER Magazine*, 34(4), Apr 2001.

[20] Ramu Pyreddy and Gary Tyson. Evaluating design tradeoffs in dual speed pipelines. In *Workshop on Complexity-Effective Design*, Goteborg, Sweden, Jun 30 2001.

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

[22] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of MICRO-27*, pages 63–74, San Jose, Calif., Nov.–Dec. 1994.

[23] John Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proc. of SIGPLAN PLDI '96*, pages 1–11, Philadelphia, May 1996.

[24] S.Rele, S.Pande, S. Onder, and R.Gupta. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction(CC'2002)*, Grenoble, France, Apr 2002.

[25] 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.

[26] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. of ISCA-27*, pages 95–106, Vancouver, Brit. Col., Jun. 2000.

[27] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing(LCPC'02)*, College Park, MD, July 2002.

[28] Han-Saem Yun and Jihong Kim. Power-aware modulo scheduling for high-performance vliw processors. In *Proceedings of the International Symposium on Low Power Electronics and Design 2001*, Aug 2001.

[29] 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.