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

A Quantitative Study on Performance-Power Impact of Dual-Speed Pipeline Architectures

Hongbo Yang
R. Govindarajan†
Guang R. Gao
Kevin B. Theobald

CAPSL Technical Memo 42
June 10, 2002

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

Abstract

The drastic increase in power consumption by modern processors emphasizes the need for power-performance trade-offs in architecture design space exploration and compiler optimizations. This paper reports a quantitative study on the power-performance trade-offs in software pipelined schedules for an Itanium-like EPIC architecture with *dual-speed* pipelines, in which functional units are partitioned into fast ones and slow ones. We have developed an integer linear programming formulation to capture the energy/performance tradeoffs for software pipelined loops. The proposed integer linear programming formulation and its solution method have been implemented and tested on a set of `SPEC2000` benchmarks. The results are compared with an Itanium-like architecture(baseline) in which there are four functional units(FUs) and all of them are fast units.

Our quantitative study reveals:

1. There is considerable energy saving by introducing a few slow FUs in place of fast FUs in the baseline architecture. When 2 out 4 FUs are set as slow, the total energy consumed by FUs is reduced by up to 31.1% (with an average reduction of 25.2%) compared to the baseline configuration.
2. Although using slow FUs may cause some performance degradation, our results show that such degradation is small in a large majority of cases. The average performance degradation for a configuration with 2 out of 4 FUs being slow is only 6.2%. Even when 3 out of the 4 FUs are slow units, the average performance degradation is within 15% compared to that achieved by the baseline configuration with all fast FUs.
3. If performance demand is less critical, then further energy reduction can be achieved by trading performance for energy. For example, if 30% decrease in the performance can be tolerated, then an energy saving of up to to 40.3% can be achieved, with an average of 34%.

Keywords: Low power micro-architecture, low power compilation techniques, integer linear programming, software pipelining.

Contents

List of Figures

List of Tables

1 Introduction

The past decade has seen a tremendous increase in the performance of general purpose microprocessors. However, this higher performance is often accompanied by an undesirable, sometime excessive, power consumption. This is predicted soon to become a limiting factor in high-performance processor design [3]. Consequently, architects must consider power-performance tradeoffs when exploring the processor design space.

This paper deals with low power architectures, compiler techniques, and their interplay. Recent research in low power design proposes architectures with components operating at different pipeline speeds [17]. In such processors, some functional units (FUs) operate at full CPU speed ("fast" FUs) while others run at a slower speed, typically half of the full speed ("slow" FUs). Slow FUs can run at lower voltages, which lead to a better-than-linear energy reduction per operation [10].

The rationale behind such a design is that schedule slack is present for some instructions, i.e., they are not on the critical path, so prolonging their execution time should not hurt performance. A limit study [5] demonstrates that more than 75% of execution cycles have at least one instruction that has slack. By issuing instructions on the critical path to fast pipelines and those which are off the critical path to slower functional units, significant power and energy reduction can be achieved [14, 26].

This paper reports a quantitative study of power-performance tradeoffs in the design space by exploring the interplay between low-power architecture features and compiler optimizations. Our study concentrates on software pipelining [15, 19], a compile-time instruction scheduling technique for loops. In our prior work, performance-oriented software pipelining under given timing and resource constraints has been formulated as an *integer linear programming* problem [1, 8, 9] and was successfully used in evaluating the software pipelining algorithm implemented in a production-quality compiler [22]. However in the context of low power, effective software pipelining algorithm which strives for both optimal performance and minimal energy is still unexplored. The problem that we are addressing in this paper is:

> *Given a loop $\mathcal{L}$ and a machine configuration $\mathcal{M}$ with slow and fast FUs, find a rate-optimal software pipelined schedule in which the schedule consumes minimal energy.*

We use the term*rate-optimal schedule* to imply that no other schedule for the loop $\mathcal{L}$ can have a lower *initiation interval*(II) for the machine configuration $\mathcal{M}$. And we use the term *minimal energy schedule* refers no other schedule for the loop $\mathcal{L}$ can have a lower energy consumption with given II for the given architecture $\mathcal{M}$. This paper formulates the above problem as an ILP problem and explores both the architecture and compiler side of design space using the ILP formulation and its solution method. In particular, this study explores:

[Architecture side:] In order to achieve significant power reduction without incurring performance degradation, how many of the FUs in an architecture can be slow FUs? By

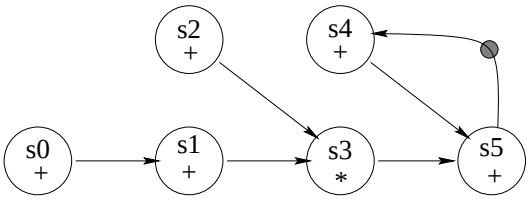

Figure 1: Motivating Example

increasing the number of slow FUs, how does the application performance degrade?

[Compiler side:] To what extent can the compiler exploit this knowledge of slow and fast FUs in the architecture, and schedule instructions in such a way to reduce energy without significant degradation in performance?

1.1 Example

We illustrate the benefits of slow FUs with a simple example. Figure 1 shows a data dependence graph (DDG) for a loop body, with a loop-carried dependence (from one iteration of the loop to the next) from s5 to s4. Suppose we want to run this loop on an architecture **a1** with 3 Integer Add units and 2 Integer Multiply units. All FUs are fast and fully pipelined, and their latencies are 1 cycle for Add and 3 for Mult. The cycle containing s4 and s5 imposes a lower bound on how frequently the loop body can be initiated. This is the Recurrence Minimum Initiation Interval (RecMII), given by

$$RecMII = \max_{\forall cycles\ C} \left\lceil \frac{d(C)}{m(C)} \right\rceil$$

where $d(C)$ is the sum of the latencies of the instructions in cycle C and $m(C)$ is the sum of the loop-carried dependences around cycle C [21]. In our example, there is only one cycle, and RecMII is $\lceil (1+1)/1 \rceil = 2$.

We can compute another lower bound for the initiation interval by considering available resources. The Resource Minimum Initiation Interval (ResMII) is given by

$$ResMII = \max_{r}\ (ResMII_r) = \max_{r}\ \left(\left\lceil \frac{d_{\max,r} * N_r}{F_r} \right\rceil \right)$$

where N_r is the number of instructions which are executed in FU type r, F_r is the number of FUs in type r, and $d_{\max,r}$ is the maximum number of cycles for which the FU is used by an instruction. If the FUs are pipelined, then $d_{\max,r}$ is 1. In our example, the individual ResMII for Add and Mult FUs are $\lceil (5/3) \rceil$ and $\lceil (1/2) \rceil$ respectively. Thus, the overall ResMII is 2. The minimum initiation interval (MII) is the maximum of ResMII and RecMII; MII=2 in our example. (Several papers on software pipelining [15, 18, 19] discuss computing RecMII, ResMII, and MII.)

Iteration	Time Steps										
	0	1	2	3	4	5	6	7	8	9	10
0	s0	s1,s2	s3		s4	s5					
1			s0	s1,s2	s3		s4	s5			
2					s0	s1,s2	s3		s4	s5	

Figure 2: A Feasible Schedule for the DDG for Architecture **a1**

Iteration	Time Steps										
	0	1	2	3	4	5	6	7	8	9	10
0	s0,*s2*	s1	*s3*					s4	s5		
1			s0,*s2*	s1	*s3*					s4	s5
2					s0,*s2*	s1	*s3*				
3							s0,*s2*	s1	*s3*		
4									s0,*s2*	s1	*s3*

Figure 3: A Feasible Schedule for the DDG for Architecture **a2**

Figure 2 shows one possible "rate-optimal" schedule with an initiation interval $II = 2$ for this DDG. The repetitive kernel in cycles 4 and 5 is shaded. We can see that there is some slack in the schedule, and not all FUs are fully utilized. If one Add FU and one Mult FU in **a1** were replaced with slow FUs operating at half the frequency, we could achieve the same II with this modified architecture **a2**, as shown in Figure 3. In this schedule, instructions s2 and s3 are scheduled in the slow Add and Mult FUs; these are shown in italics. Although the kernel appears later, i.e., the prologue is longer, the schedule is still optimal with $II = 2$. Architecture **a2** may save power relative to **a1**, without slowing down the loop.

On the other hand, if two of the Add units were slow (architecture **a3**), the schedule in Figure 3 would not work. Each loop iteration would require 5 additions, but each of the two slow units can perform only one addition every 2 cycles, so that all three Add units together can perform only 4 additions every 2 cycles. The best initiation interval that we can achieve for this DDG would be $II = 3$. One possible schedule is shown in Figure 4. This example shows

Iteration	Time Steps											
	0	1	2	3	4	5	6	7	8	9	10	11
0	*s0*	*s2*		s1	*s3*						s4	s5
1				*s0*	*s2*		s1	*s3*				
2							*s0*	*s2*		s1	*s3*	
3										*s0*	*s2*	

Figure 4: A Feasible Schedule for the DDG for Architecture **a3**

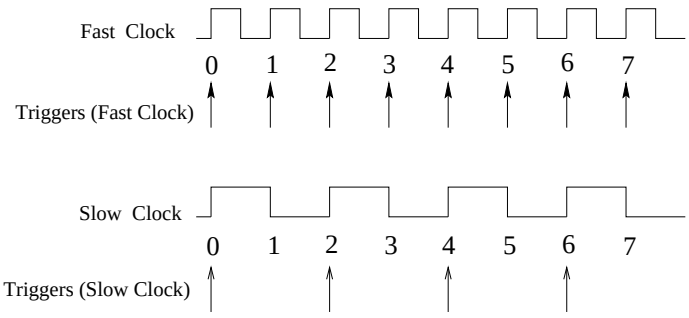

Figure 5: Rising Edges in Fast and Slow FUs

that by varying the number of fast and slow FUs, the architecture can be made more power efficient, but at some point this efficiency will lead to a performance penalty.

The difference in clock speeds, and therefore input rates, between fast and slow FUs present a few complications when scheduling instructions in a slow FU. If the slow FU is operating at half the clock frequency of the fast FU, then the slow FU will only have half the number of rising/falling edges on which activities can be triggered. Thus if an instruction is scheduled on a slow FU at an odd-numbered cycle, then the activities will not take place in the slow FU until the next even-numbered cycle on which the rising edge occurs (refer to Figure 5). Thus an instruction scheduled on a slow FU at an odd-numbered cycle will experience an additional cycle delay. Any other instruction dependent on this will have to be scheduled after $2\ell + 1$ cycles, where ℓ is the latency of the fast FU. Furthermore, if the II is odd, then each instruction will occur in an odd cycle in alternate iterations; when this occurs, the additional latency will be incurred. Thus, we assume that the latency of a slow FU is uniformly $2\ell + 1$. Fortunately, for the schedule shown in Figure 4 this is already satisfied. Lastly, as an instruction scheduled at odd cycle in a slow FU will have to wait for an extra cycle to get the rising/falling edge, we should also assume, conservatively, that a new instruction can be initiated on a slow FU only once every three cycles. Again, in the schedule shown in Figure 4, as instructions s0 and s2 are scheduled on different slow FUs, this constraint is also satisfied.

1.2 Synopsis

Given an architecture with a fixed number of FUs in each resource class, it is interesting to find out how many of these FUs in each class could be operated at a slower clock rate, and hence at a reduced power consumption, and what is the degradation in performance, if any. In this paper, we propose an integer linear programming formulation for the above problem based on our earlier work. We apply our approach on `SPEC2000` integer benchmarks to evaluate different configurations (with different number of slow and fast FUs) of Itanium processor from both the performance and power angles. Also we study the additional power savings that can be obtained when the compiler trades performance for energy. Our results show that by introducing two slow FUs in the architecture, in the place of fast FUs, an average energy savings of 25.2% can

be obtained while the performance degradation is within 6.2% on the average.

Before proceeding to our study, a few remarks are in order. As the focus of this paper is on studying the architecture design space, admittedly, we use a somewhat approximate power model, and consequently, an approximate energy model. Also in this study we do not consider the power consumed by other components such as issue logic and caches. Finally, we ignore the power consumed by inactive functional units. Our experience obtained by using architectural-level power simulator [7] and Synopsys gate-level power simulator is that leakage power(power consumed by inactive functional units) is only a very small fraction of the total power. Although the case might change in the future, the increased leakage power problem can accordingly be addressed by related techniques like input vector control [26]. Our initial investigations show that it is possible to relax some of these assumptions and incorporate a more accurate power model in our integer linear program formulation, although at the expense of complicating the integer program formulation. We leave the details of these accurate models as well as making their formulation efficient for future work, as the emphasis of this paper is not on developing a power-aware software pipelining method.

The rest of the paper is organized as follows. The next section presents an elegant ILP formulation for power-aware software pipelining based on our previous work. In Section 3 we derive an analytical expression for energy consumption and validate it. Section 4 deals with the experimental evaluation and results. Related work is elaborated in Section 5. Lastly, we provide concluding remarks in Section 6.

2 Integer Linear Programming Formulation

In this section first we extend the integer linear programming formulation [8, 9] for software pipelining to handle dual speed pipelines. Subsequently in Section 2.2, we refine the formulation to more accurately model structural hazards in dual speed pipelines.

2.1 Initial Formulation

Let the number of nodes in the DDG be N and let II be the initiation interval [19, 15, 18]. In this paper we consider only repetitive schedules or modulo scheduling. The software pipelining methods attempts to find a schedule for each value of initiation interval II, starting from the minimum initiation interval (MII) [19, 15, 18]. The methods stops as soon as a schedule is found for the first II greater than or equal to the minimum initiation interval.

In modulo scheduling, the schedule time of operation i in the jth iteration is given by

$$t_{i,j} = t_i + j * II$$

where t_i is the schedule time of operation i in the first iteration. We use a $N \times 1$ vector Γ to represent the schedule time of the operations in the first iteration. The vector Γ is composed of variables $t_0, \cdots, t_{N-1}$. We use a $II \times N$ matrix A to represent the repetitive pattern of

instructions in the modulo schedule. The matrix $A = [a_{t,i}]$ where $a_{t,i} = 1$ if and only if node i is scheduled at time step t in the modulo schedule, where t is the range $[0..II-1]$. It is easy to see that

$$a_{t,i} = 1, \text{ if } t_i \bmod II = t.$$

Further Γ and A are related by the following equation.

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

The vector κ is an $N \times 1$ vector composed of $k_0, \cdots, k_{N-1}$. Equation 1 implicitly defines k_i as

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

Since each instruction i is scheduled in the repetitive pattern exactly once, the following equation must be satisfied for a legal schedule:

$$\sum_{t=0}^{II-1} a_{t,i} = 1 \qquad \text{for all } i \in [0, N-1] \tag{2}$$

For resources that operate on a single speed, that is those which are not partitioned to fast and slow ones, *e.g.,* the issue unit, the resource constraint can be specified using the following simple inequality.

$$\sum_{i \in \zeta(r)} a_{t,i} \leq R_r \qquad \text{for all } t \in [0, II-1] \tag{3}$$

where R_r is the number of resources of type r and $\zeta(r)$ represent the set of instructions in the loop that use resource type r. Note that resources like issue unit are used by all instructions, and hence $\zeta(Issue\ Unit)$ consists of all instructions; whereas in the floating point divide resource type, there may be only one function unit, and $\zeta(FP\ Divide)$ consists of only FP divide instructions, if any, in the loop.

Next we formulate the resource constraints for resources types which have slow and fast FUs. For example, in our motivating example, the Add and Multiply FUs have slow and fast FUs. For an instructions which goes to these resource types r, we also need to know whether the instruction is scheduled on slow or fast FU. We use two sets of variables $u_{t,i}$ and $v_{t,i}$ for this purpose: $u_{t,i} = 1$ if instruction i is scheduled on fast FU and $a_{t,i} = 1$; likewise $v_{t,i} = 1$ if instruction i is scheduled on slow FU and $a_{t,i} = 1$. Since the resource assignment for instruction i can be either fast FU or slow FU but not both, the following equation holds:

$$a_{t,i} = u_{t,i} + v_{t,i} \qquad \text{for all } t \in [0, II-1], \text{ and for all } i \in [0, N-1] \tag{4}$$

If there are $R_{r,fast}$ fast FUs and $R_{r,slow}$ slow FUs in resource type r, then the resource constraint for software pipelined schedule can be enforced using the inequality

$$\sum_{i \in \zeta(r)} u_{t,i} \leq R_{r,fast} \text{ for all } t \in [0, II-1] \text{ and for all } r \tag{5}$$

$$\sum_{i \in \zeta(r)} v_{t,i} \leq R_{r,slow} \text{ for all } t \in [0, II-1] \text{ and for all } r \tag{6}$$

Next we will express the precedence constraint. For this, we define a variable f_i for each instruction i to indicate whether instruction i is scheduled on fast FU (in which case $f_i = 1$) or on slow FU (in which case $f_i = 0$). The variable f_i is defined as:

$$f_i = \sum_{t=0}^{II-1} u_{t,i} \text{ for all } i \in [0, N-1] \tag{7}$$

Now, for each dependence arc (i, j) in the DDG, the precedence constraint between i and j is formulated as [21]:

$$t_j - t_i \geq d_{ij} - II \times m_{ij}$$

where d_{ij} is the latency from instruction i to j and m_{ij} is the dependence distance [19, 15, 18]. In the dual-speed architecture, the latency d_{ij} will be $d_{ij,fast}$ and $d_{ij,slow}$ for fast and slow FUs. The precedence constraints can be expressed as:

$$t_j - t_i \geq f_i \cdot d_{ij,fast} + (1 - f_i) \cdot d_{ij,slow} - II \times m_{ij} \text{ for all } (i, j) \tag{8}$$

A power/energy aware software pipelining method should try to construct a schedule that does not incur any performance degradation and also consumes the minimum energy. The performance a software pipelined schedule is related to the initiation interval (II) for which the integer programming method attempts to find a solution. A simple objective function to minimize the energy consumption for the same initiation interval II, is the one which minimizes the number of instructions issued to fast FUs. This objective function is based on the simple model that the energy consumed can be approximated to number of the instructions executed in that FU [16]. Assuming $C_{r,fast}$ and $C_{r,slow}$ are the weights for fast and slow functional units of type r, the objective function is given by:

$$\min \sum_{r} \sum_{i \in \zeta(r)} (C_{r,fast}.f_i + C_{r,slow} * (1 - f_i)) \tag{9}$$

The full ILP formulation has the objective function as Equation 9 under the constraints given by Equations 1 – 8. In the above ILP formulation, Equations 1 – 4 are same as in [8, 9]. However, Equations 5 – 8, which are used for modeling slow and fast FUs, are contributions of this paper.

2.2 Refined ILP Formulation

The formulation presented in the previous section is simple, but does not account the resource usage of slow FUs accurately. More specifically, when an instruction is scheduled on a slow FU, not only its latency (d_{ij}) is increased (to $d_{ij,slow}$), but also the slow FU has a slower throughput. As explained in Section 1.1, instructions can be initiated in the slow FU only on alternate cycles (of the fast clock). Since the time steps in our schedule is based on the fast clock, slow FUs (operating at half the clock frequency) have a structural hazard, and can have instructions scheduled on them only once every two cycles.

We will explain the above using an example. Assume instructions i and j are executed on the same resource type r, and both are scheduled on the slow FU. Further assume that there is only one slow FU of type r. Then if instruction i is scheduled at time t, then j cannot be scheduled at time $(t+1)$ mod II or vice-versa. In other words, instructions scheduled on a slow FU occupy the pipeline for more cycles (two in this case), even though the slow FU is also fully pipelined (on the slower clock). Thus our resource constraints for slow FUs (Equations 6) should be modified to reflect this. We do this by including the following constraint. If $(v_{t,i} = 1)$ and $(a_{t,i} = 1)$ then $(v_{(t \oplus 1),i})$ should also be 1, where $t \oplus 1$ represents $(t+1)$ *mod* II. Logically this is equivalent to

$$\neg \left((v_{t,i} = 1) \wedge (a_{t,i} = 1) \right) \vee (v_{(t \oplus 1),i} = 1)$$

which is equivalent to

$$(v_{t,i} = 0) \vee (a_{t,i} = 0) \vee (v_{(t \oplus 1),i} = 1).$$

The above can be expressed in the form of a linear constraint as:

$$v_{t,i} + a_{t,i} - v_{(t \oplus 1),i} < 2. \tag{10}$$

Note that if both $(v_{t,i} = 1)$ and $(a_{t,i} = 1)$ then $v_{(t \oplus 1),i}$ should be 1 in order to make the left hand side (strictly) less than 2. Note that in the above constraint we could have eliminated the term $(a_{t,i} = 1)$, as $(v_{t,i} = 1)$ already implies the former. However including the former, ensures that the inequality is not applied successively to $v_{(t \oplus 2),i}$, $v_{(t \oplus 3),i}$, *etc.*, and make them 1 too.

Lastly, to account for the additional cycle involved when instructions are scheduled at odd-numbered cycles on slow FUs (refer to Section 1.1), we consider (i) $d_{ij,slow} = 2 * d_{ij,fast} + 1$ and (ii) if $v_{t,i} = 1$, and then both $v_{t \oplus 1}$ and $v_{t \oplus 2}$ must be 1. Note that we apply conditions (i) and (ii) uniformly on all instructions (scheduled on slow FUs). This is because, even if an instruction is scheduled at even-numbered cycle, if the II is odd, the above conditions should hold. Now condition (i) can be easily incorporated in Equation 8. To incorporate condition (ii), we add the equation

$$v_{t,i} + a_{t,i} - v_{(t \oplus 2),i} < 2. \tag{11}$$

Including Equations 10 and 11 in the ILP formulation accounts for resource constraints accurately. However, a consequence of this — the fact that an instruction scheduled on a slow FU uses the FU exclusively for 3 cycles — disallows loops with initiation interval 1 or 2. This is in order to adhere to modulo scheduling constraint [18] which states that a functional unit cannot be used by an instruction in time steps that differ by a multiple of II. As a consequence, the lowest II for which a schedule can be found in the architecture involving slow and fast FU is 3.

3 Experimental Framework

Using the ILP formulation proposed in the previous section, we explore how the design space of an Itanium-like architecture can be made power-efficient by introducing slow FUs. In this

study we focus on the performance of software pipelineable loops in the `SPEC2000` integer benchmarks. The following subsection briefly describes the Itanium architecture. Section 3.2 describes the analytical energy model used in evaluating energy-efficiency of different variants of Itanium architecture by introducing slow FUs. Our experimental methodology is detailed in Section 3.4.

3.1 Target Architecture

The target processor in our study is an Itanium-like processor [12, 13], the first microprocessor of the Intel Itanium Processor Family (IPF). Itanium is an *Explicitly Parallel Instruction Computing* (EPIC) processor, which can issue 6 instructions in a single cycle, and supports predicated execution, speculative execution, rotating registers, etc., to increase instruction-level parallelism. It has four types of functional units: *M-unit*, *I-unit*, *F-unit*, and *B-unit*, which work primarily for memory access, integer operations, floating-point operations and branch instructions respectively. Each instruction falls into one of the six categories: *A-type*, *I-type*, *M-type*, *B-type*, *F-type* and *X-type*. Since our power-aware software pipelining is applied only to loop bodies with single basic blocks, the instructions in the loop body are of *A-type*, *I-type*, and *M-type* only. There are no *F-type* instructions because we used integer benchmarks only.

I-type and M-type instructions are executed on I-units and M-units respectively, whereas A-type instructions can be executed on either type of FU. There are two of each functional unit type, and both are fully pipelined. One of the two FUs in each type can execute any of the instructions in that category while the other cannot execute a small subset of them. The difference is minor and the two instances of the FUs in each FU type are otherwise identical. Latency values used in our experiments are taken from the instruction timing of Itanium as defined in its micro-architecture manual [13].

3.2 Energy Model

We use a simple power model in our experiments. It is known that dynamic power dissipation P is given by $P = C \cdot V_{dd}^2 \cdot f$ where C is the effective switching capacitance, V_{dd} is the supply voltage and f is the clock frequency. In slow FUs, the clock speed is reduced and voltage can be accordingly reduced, leads to better-than-linear energy saving. Empirically, it was found that FUs working on half frequency consume about 40%(represented by C in the computation below) of the energy consumed by fast FUs. It is represented as follow:

$$E_{slow,i} \quad = \quad C * E_{fast,i} \tag{12}$$

Also we assume the total energy consumed by FUs incurred for the software pipelined schedule is the sum of the energy incurred for each instruction in the schedule. This is based on the same approximation that we used in our objective function in Section 2, namely, the total energy consumed is related to the number of instructions executed [16]. Admittedly, in this simple model we focus on FUs only and ignore the power used by other components such

as issue logic and caches, and we assume inactive functional units consume no energy due to clock gating. Thus, the total energy consumed in the dual speed pipeline architecture is:

$$E_{dual} = \sum_{i_{fast}} E_{fast,i} + \sum_{i_{slow}} E_{slow,i}$$

As dual speed pipelines are considered only for integer FUs, we assume the energy consumed by the *I-unit* and *M-unit* FUs are the same, i.e., $E_{fast,I-unit} = E_{fast,M-unit}$, and $E_{slow,I-unit} = E_{slow,M-unit}$.

Next, we compute the energy consumed by a single iteration of the schedule of a loop l as:

$$\mathcal{E}^1_{dual,\ell} = N_{fast,\ell} * E_{fast} + N_{slow,\ell} * E_{slow}$$

where $N_{fast,\ell}$ and $N_{slow,\ell}$ represent the number of instructions scheduled on fast and slow FUs in a single iteration of loop l, thus $N_{slow,\ell} + N_{fast,\ell} = N_\ell$. The above equation can be derived to:

$$\mathcal{E}^1_{dual,\ell} = N_{fast,\ell} * E_{fast} + N_{slow,\ell} * C * E_{fast} \tag{13}$$

Compared to Equation 13, the energy consumed by a single iteration of loop l in the baseline architecture is

$$\mathcal{E}^1_{base} = N_\ell * E_{fast} \tag{14}$$

From this, we can approximate the energy consumed by a loop l, whose trip count is T_ℓ, for the dual speed and baseline architectures as:

$$\mathcal{E}_{dual,\ell} = \mathcal{E}^1_{dual,\ell} * T_\ell = (N_{fast,\ell} + N_{slow,\ell} * C) * E_{fast} * T_\ell \tag{15}$$

$$\mathcal{E}_{base,\ell} = N_\ell * E_{fast} * T_\ell \tag{16}$$

Lastly, the energy consumed by all the software pipelineable loops in a benchmark for dual and baseline architectures are:

$$\mathcal{E}_{dual} = \sum_\ell \mathcal{E}_{dual,\ell} \tag{17}$$

$$\mathcal{E}_{base} = \sum_\ell \mathcal{E}_{fast,\ell} \tag{18}$$

Similarly the performances of the benchmark or the execution time spent in software pipelineable loops are given by:

$$\text{Exec. Time}_{dual} = \sum_\ell T_\ell * II_{dual,\ell} \tag{19}$$

$$\text{Exec. Time}_{base} = \sum_\ell T_\ell * II_{base,\ell} \tag{20}$$

where $II_{dual,\ell}$ and $II_{base,\ell}$ are the initiation intervals for loop l in the dual speed and base architectures.

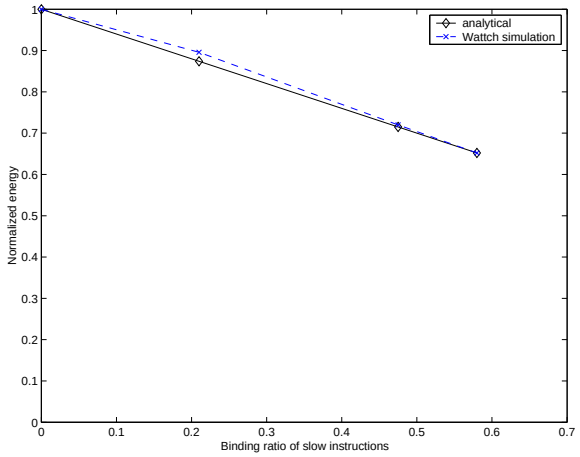

Figure 6: Comparing Our Analytical Energy Model with Simulation Results

3.3 Compare The Analytic Energy Model With Simulation Result

To estimate the accuracy of our energy model, we enhanced the Wattch power simulator [2] with dual-speed pipelines. We use the annotation bits of SimpleScalar [4] instruction word to mark whether instructions should be issued to fast or slow FUs. We "mimicked" the compiler behavior by manually adding annotation fields to the assembly program and running the executable program on Wattch with clock-gating [2] enabled. We performed these steps on a *matrix multiply* program and iterated several times to get programs where the number of instructions sent to slow FUs varied. We use the term *binding ratio* to refer to the ratio of number of instructions issued to the slow FUs to the total number of instructions. In this experiment, we considered instructions from all basic blocks, and simulated the energy consumed in executing them from the Wattch simulator. We compare this with the estimated energy using Equation 17 for the matrix multiply program in Figure 6. Comparison reveals that our analytical energy model and the results obtained from Wattch simulation are very close. This shows that our analytical energy model for FUs closely track the activity-based dynamic power simulation [2, 7].

3.4 Experimental Methodology

We implemented our ILP formulation on the Delaware Power-Aware Computing Testbed (DelPACT) platform [24]. The ILP formulation works with the Open64 compiler [23], which is an open-source compiler from SGI targeting Itanium processors. Open64 has a rich set of optimizations and we enabled them in our experiments by setting the highest optimization level. Our workload came from the `SPEC2000` integer benchmark suite. Benchmarks were compiled by Open64, executed on the "train" data sets to collect profiling information, then recompiled using this profiling information. DDGs were extracted and an integer linear programming problem was formed for each DDG. These were then passed to CPLEX [11], a commercial integer linear programming solver, which gave us the schedule, the initiation interval of the loop (denoted by

II_ℓ) and instruction partitioning (to fast and slow FUs) of each DDG.

For each of the benchmarks considered, we obtained the trip counts of all software pipelineable loops for the train input set by profiling. Using Equations 17 and 18 we computed the energy consumed by the software pipelineable loops in each benchmark. The execution time incurred on the software pipelineable loops are given by Equations 19 and 20. Since our work focuses only on software pipelining, we use the former as the energy measure for the benchmark and the latter as performance.

We report our results for only those benchmarks which were successfully compiled by Open64 compiler and produced correct results. These benchmarks, the number of software pipelineable loops in them, and characteristics of the DDGs associated with these loops are summarized in Table 1. In our experiment, each loop is given a fixed time limit for CPLEX to derive a solution, and within this time limit, 91% of the loops could be scheduled by CPLEX.

Benchmark	# of Loops	# of nodes		# of edges		Minimum II	
		(avg)	(max)	(avg)	(max)	(avg)	(max)
164.gzip	69	7.3	19	12.8	37	3	14
175.vpr	93	15.4	75	47.6	621	12	95
181.mcf	22	12.8	54	62.6	495	9	27
186.crafty	86	11.4	64	33.5	833	5	92
197.parser	116	7.4	29	16.1	69	5	18
256.bzip2	63	9.4	32	21.0	108	6	100
300.twolf	317	18.7	101	63.8	349	12	192

Table 1: Statistics on DDGs Extracted from SPEC 2000 Benchmarks

4 Experimental Results

The main observations from our experiments are summarized as follows:

1. Making some of the functional units as slow FUs can significantly reduce energy consumption for these benchmarks. Changing two fast functional units to slow ones can lower FU energy requirements up to 31.1% with an average reduction of 25.2% compared to the baseline configuration.

2. Despite the considerable energy saving, judicious use of slow functional units generally leads to a surprisingly small performance degradation. Even when 2 out of the 4 functional units are slow units, the average performance degradation experienced is only 6.2% compared to that achieved by configuration with all fast FUs.

3. If performance is less critical, more energy savings can be achieved by an energy-aware compiler. On a processor with 2 fast and 2 slow functional units, a maximum of 40.3%

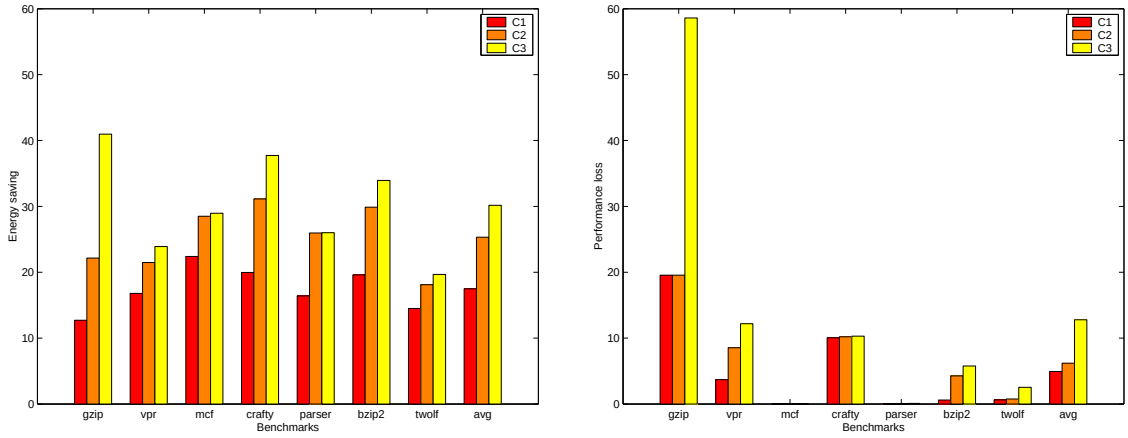

Figure 7: Energy Saving and Performance Penalty of Different Configurations

energy saving on FUs can be achieved over the baseline model if a maximum of 30% increase on the individual II values for the different loops is acceptable.

4.1 Energy Gain and Performance Degradation by Slow Functional Units

The performance degradation and energy gain in introducing some slow FUs in the place of fast FUs is depicted in Figure 7. In this experiment, we considered fast and slow FUs for the *I-unit* FUs and *M-unit* FUs only. Together we refer to these as integer FUs. In all experiments, there are 4 integer FUs, while the number of slow and fast FUs varies among the configurations. We consider the following four configurations:

C0 2 fast *I-unit* and 2 fast *M-unit* FUs — this is the baseline architecture

C1 1 fast *I-unit*, 1 slow *I-unit*, and 2 fast *M-unit* FUs

C2 1 fast *I-unit*, 1 slow *I-unit*, 1 fast *M-unit* and 1 slow *M-unit* FUs

C3 2 slow *I-unit*, 1 slow *M-unit*, and 1 fast *M-unit* FUs

Figure 7 shows the energy gain and performance loss of configurations **C1**–**C3** on all benchmarks, compared against the energy and performance of the baseline configuration **C0**. The last set of bars indicate the average energy reduction and performance degradation taken across all benchmarks. For configuration **C1**, we observe an energy gain of 12% or more in all the applications. With configurations **C2**, the energy savings increase to 18% and more. Lastly, for configuration **C3**, the energy gain is higher, ranging from 20% to 41%. It is important to note that the above energy savings are obtained only by trading off a little or no performance in many cases. For configuration **C1**, except for benchmarks `gzip` and `crafty` which show relatively large performance degradation, 19.6% and 10% respectively, all other benchmarks incur very little performance degradation, often less than 1%. With configurations **C2**, the performance degrades, by 5% – 20%, in 4 of the benchmarks, while for other three (`mcf`,

`parser` and `twolf`) the performance degradation is negligible. Lastly, for configuration **C3**, both the performance degradation and the energy gain are higher. It should be noted here that even with configurations **C2** and **C3** (with, respectively, 2 and 3 slow FUs), the degradation in performance is within 15% for all the benchmarks, except `gzip`.

It is important to note here that our method obtains *rate-optimal* schedules where the optimality is for a given configuration (**C1**, **C2**, or **C3**); i.e., there can be no other schedule that has a lower initiation interval for the given configuration. Among the rate-optimal schedules for a given configuration, our approach obtains one that consumes the minimum energy (as calculated by our energy model). In this sense, the schedules obtained by our approach are optimal in terms of both performance and energy for the given configuration.

Lastly, we observe that `gzip` suffers relatively large performance loss when slow functional units are introduced. This observation agrees with that of related work by Seng et al [14]. Interestingly, two benchmarks, namely `mcf` and `parser`, incur no performance degradation even when 3 of the 4 integer units act as slow FUs. To summarize, for the benchmarks we studied, configurations **C1**, **C2** and **C3** consume, respectively, 17.5%, 25.3%, and 30.2% less energy, on the average, compared to the **baseline** architecture.

4.2 Trading Performance for Energy

Next we address the question that whether further energy savings can be obtained in dual speed architectures if the compiler is allowed to trade performance. To answer this question, we conduct a set of experiments on configuration **C2** (with 2 fast and 2 slow FUs). Here, for each software pipelineable loop, we set the optimal II obtained for configuration **C2** as the base II value. We allow our compiler to choose an II value which is within a performance degradation threshold (PDT) from the base II value, and try to obtain energy-efficient schedules. We chose PDTs as 10%, 20%, 30% for each application.

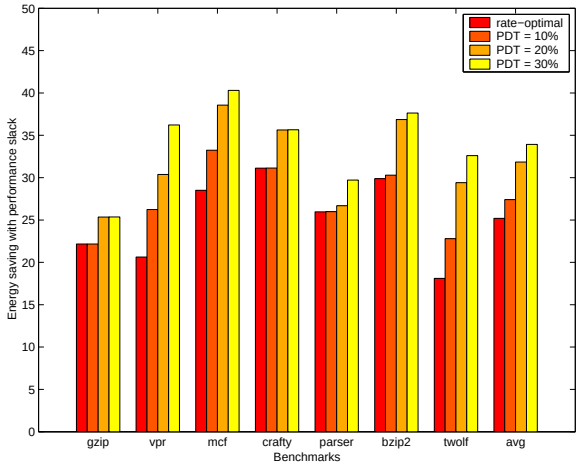

Figure 8: Energy saving for Three Different Slowdown Factors

In Figure 8 we plot the energy savings compared to the baseline configuration (C0) for various PDT values. The bar corresponding to rate-optimal refers to a PDT value of 0%, which necessarily means that the compiler does not trade performance for power, and tries to obtain the optimal schedule for configuration **C2**. As shown in the figure, energy gains in three benchmarks, namely `vpr`, `mcf`, and `twolf`, increase with increased PDT value. However, in the other benchmarks, the energy savings appear only when the PDT is increased from 10% to 20%. The insignificant benefit in performance when the PDT is increased from 0% to 10% can be explained by the fact that these benchmarks have a lower average Minimum II(MII) than the three benchmarks, as shown in Table 1. For those DDGs with small minII, 10% PDT leaves no space for them to increase its II. Hence their schedules are not changed for a large fraction of loops compared with the rate-optimal schedules obtained under configuration **C2**. The actual performance degradation for these benchmarks is less than 1% when 10% PDT is applied. When PDT of 20% is given, energy gain is more apparent for all benchmarks. The average energy saving for PDTs of 20% and 30% is 31.8% and 34% respectively.

Lastly, we note that a PDT value refers to an increase in the individual II. Hence a PDT of, say 20%, does not necessarily mean a 20% degradation on the execution time. The average performance degradation for the benchmarks is only 15.4% when the PDT is 20%.

5 Related Work

Exploiting instruction schedule slack to reduce power consumption with minimal performance degradation is extensively studied in literature. A limit study on available schedule slack was performed by Casmira et al [5]. Their result shows that in the course of program execution, for more than 75% of the execution cycles there exists at least one instruction that has slack. However they didn't mention any work on techniques to exploit such instruction scheduling slack for energy saving.

Pyreddy et al [17] extend this idea by partitioning functional units to fast ones and slow ones, operating the slow ones at half the rate of fast ones, and using run-time profiling to guide instruction issue to these two sets of functional units. It is their assumption that profiling needs to be done in a separate pass other than the actual run. In contrast, Seng et al proposed to analyze instruction criticality dynamically and issue critical instructions to fast functional units while non-critical instructions to slow ones [14]. The mechanism they adopt to predict instruction criticality is the critical path predictor buffer [6]. Although interesting, the buffer consumes additional power which may offset the power gain in functional units.

The work by Zhang et al [26] uses compilation techniques to determine the fast or slow functional units where each instruction should be issued to. They designed a energy-oriented heuristic algorithm to re-order a schedule made by some performance-oriented scheduling algorithm to achieve energy saving with minimal performance degradation. Our work complements theirs in the following aspects. First, our study focuses on loop software pipelining while their study is on global scheduling with acyclic dependence graphs. Second, we use an integer linear

programming based approach to evaluate the limits of performance energy tradeoffs an "optimal scheduler" can do. Our method can be used to evaluate heuristic, more practical, algorithms and potentials of their improvements [22].

A relevant work by Yun et al [25] targets power-aware software pipelining. They proposed a heuristic algorithm which extended iterative modulo scheduling [20], tries to minimize *step power* for a software pipelined loop on a cycle-by-cycle basis. Their objectives are to derive a schedule under which "power consumption are better balanced" under a VLIW architecture where the speed of function units are fixed.

6 Conclusions

This paper reports a quantitative study of power-performance tradeoffs in the design space of energy efficient architectures. It studies the interplay between low-power architecture features and compiler optimization techniques, specifically software pipelining. It explores both the architecture and the compiler side of design space. An Itanium-like architecture model involving dual speed pipelines (slow and fast FUs) is used in this study.

This design space exploration is performed under the Delaware Power-Aware Computing Testbed (Del-PACT) platform on SPEC2000 integer benchmark programs. We have proposed an elegant integer linear programming formulation for rate-optimal software pipelining on architectures involving dual-speed pipelines. Using the integer linear programming approach we explore the design space for Itanium-like architectures, assuming that some of the FUs are slow FUs.

The main results and observations of this paper are:

1. Energy gain by introducing a few slow FUs in the place of fast FUs is considerable. The energy consumed by all FUs is reduced by up to 31.1% (25.2% on average) when 2 out of 4 FUs are set as slow.

2. Performance degradation caused by slowing down some of the FUs is small in a large majority of the cases. The average performance degradation for a moderate setting, i.e., 2 out of 4 FUs are set as slow, is only 6.2%.

3. Further energy gains can be achieved by the compiler if performance slack is available. When the performance degradation threshold is set at 30%, energy savings rise to 40.3% in the best cases (34% on average).

Further research directions include making comparison with heuristic approaches for energy-optimal scheduling approaches, reducing leakage power and incorporating run-time information in software pipelining algorithm to achieve even larger energy saving.

Acknowledgment

The authors would like to express especial thanks to George Cai from Intel for his support on building our power-aware computing testbed, to Roy Ju from Intel who carefully reviewed this paper and gave helpful suggestions. Our thanks also goes to Chenyong Wu from Open Research Compiler(ORC) team of Chinese Academy of Sciences and Peng Zhao from University of Alberta, who has helped us a lot in compiling and profiling SPEC 2000 benchmarks. Besides, support from DARPA, DOE and NSF is crucially important in building our compiler infrastructure.

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 Programming Language Design and Implementation*, pages 139–150, La Jolla, Calif., Jun. 1995.

[2] 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. 2000.

[3] D.M. Brooks, P. Bose, S.E. Schuster, H.Jacobson, P .K. Kudva, A. Buyuktosunoglu, J-D. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. Power-aware microarchtiecture: Design and modelling challenges for next-generation microprocessors. *IEEE Micro*, 20(6), Nov 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] E.Tune, D.Liang, D.Tullsen, and B.Calder. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, Feb 2001.

[7] 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.

[8] 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.–Dec. 1994.

[9] 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.

[10] 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.

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

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

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

[14] 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.

[15] 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. 1988.

[16] 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.

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

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

[19] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Ann. Microprogramming Work.*, pages 183–198, Chatham, Mass., Oct. 1981.

[20] 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.–Dec. 1994.

[21] Raymond Reiter. Scheduling parallel computations. *J. of the ACM*, 15(4):590–599, Oct. 1968.

[22] 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 1996.

[23] SGI. Open64 open source compiler. http://open64.sourceforge.net.

[24] Hongbo Yang, Guang R. Gao, Andres Marquez, George Cai, and Ziang Hu. Power and energy impact by loop transformations. In *Workshop on Compilers and Operating Systems for Low Power 2000 (COLP'01)*, Spain, 2001.

[25] 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.

[26] 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.