Published in the Proceedings of 20th International Conference on Computer Design, ©2002 IEEE

Power-Performance Trade-offs for Energy-Efficient Architectures: A Quantitative Study*

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

† Electrical and Computer Engineering
University of Delaware
Newark, DE 19716
Delaware, U.S.A.
{hyang,ggao,theobald}@capsl.udel.edu

‡ Supercomputer Education & Research Centre
Dept. of Computer Science & Automation
Indian Institute of Science
Bangalore, 560 012, INDIA
govind@csa.iisc.ernet.in

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 power-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 that by introducing a few slow FUs in place of fast FUs in the baseline architecture, the total energy consumed by FUs can be considerably reduced. When 2 out of 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 with the baseline configuration, while the performance degradation caused by using slow FUs is small. If performance demand is less critical, then energy reduction of up to 40.3% compared with the baseline configuration can be achieved.*

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 [2]. 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 [12]. In such processors, some functional units (FUs) operate at full CPU speed ("fast" FUs) while others run at a slower speed ("slow" FUs), typically half of the full speed. Slow FUs can run at lower voltages, which lead to a better-than-linear energy reduction per operation.

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 [3] 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 [10, 21].

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 [11, 15], 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, 5, 6] and was successfully used in evaluating the software pipelining algorithm implemented in a production-quality compiler [17]. However in the context of low power, effective software pipelining algorithm

*This work is supported by DARPA through contract number AFRL F30602-00-2-0564.

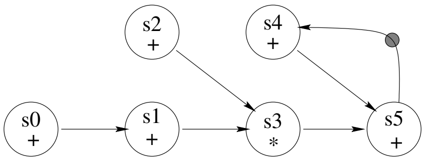


Figure 1. Motivating Example

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

Figure 2. A Schedule for Architecture a1

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

Figure 3. A Schedule for Architecture a2

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 which 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) [15, 11, 14] for the machine configuration $\mathcal{M}$. The term *minimal energy schedule* means that 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 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? Further, in such architectures, to what extent can compilers trade-off performance for energy reduction?

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. A rate-optimal schedule for this DDG with an II of 2 is shown in Figure 2. 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 still 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 II that we can achieve for this DDG would be 3. This example shows 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.

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.

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 elaborate our experimental framework. Section 4 deals with the experimental results. A discussion on related work and concluding remarks are provided respectively in Sections 5 and 6.

2. Integer Linear Programming Formulation

Let the number of nodes in the DDG be N and let II be the initiation interval. 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) [15, 11, 14]. The methods stops as soon as a schedule is found for the first II greater than or equal to MII.

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 an $N \times 1$ vector $\Gamma = [t_0, t_1, \cdots, t_{N-1}]^T$ to represent the schedule time of the operations in the first iteration. We use an $II \times N$ 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. It is easy to see that

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

We define $\kappa = [k_0, k_1, \cdots, k_{N-1}]^T$, where

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

Further Γ and A are related by the following equation.

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

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

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

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.

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

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.

Next we formulate the resource constraints for resources types which 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:

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

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

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

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

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 [19]. To reflect this, we add the condition that 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 $(t + 1)\ mod\ II$. This can be expressed in the form of a linear constraint as:

$$v_{t,i} + a_{t,i} - v_{(t \oplus 1),i} < 2. \tag{7}$$

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.

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:

$$f_i = \sum_{t=0}^{II-1} u_{t,i} \text{ for all } i \in [0, N-1] \tag{8}$$

Now, for each dependence arc (i, j) in the DDG, the precedence constraint between i and j is formulated as [16]:

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

where d_{ij} is the latency from instruction i to j and m_{ij} is the dependence distance [15, 11, 14]. 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:

$$t_j - t_i \ge f_i \cdot d_{ij,fast} + (1 - f_i) \cdot d_{ij,slow} - II \times m_{ij} \text{ for all } (i, j) \tag{9}$$

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. 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. 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:

$$\min \sum_{r} \sum_{i \in \zeta(r)} (C_{r,fast} \cdot f_i + C_{r,slow} * (1 - f_i)) \qquad (10)$$

The full ILP formulation has the objective function as Equation 10 under the constraints given by Equations 1 – 9.

3. Experimental Framework

Using the ILP formulation proposed in the previous section, we explore how the design space of an Itanium-like architecture [8, 9] can be made power-efficient by introducing slow FUs. Itanium is an *Explicitly Parallel Instruction Computing* (EPIC) processor that can issue 6 instructions in a single cycle. 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 [9, 8]. In our work, we concentrate on *M-unit* and *I-unit* for power/energy reduction.

We use a simple power model in our experiments. Empirically, it was found that FUs working on half frequency consume about 40% (represented by C in following equation) of the energy consumed by fast FUs [21]. That is,

$$E_{slow} = C * E_{fast}.$$

Also, since we consider only slow *M-unit* and *I-unit* FUs in our experiments, we assume that the energy consumed by the two types of FUs are same. Thus, Further, 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. Ignoring the energy consumed by inactive functional units and components such as issue logic and caches, the energy consumed by a single iteration of the schedule of a loop l as:

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

where $N_{fast,\ell}$ and $N_{slow,\ell}$ represent the number of instructions scheduled on fast and slow FUs. Using the trip counts of individual loops, the total energy consumed by all software pipelineable loops as well as their total execution time were calculated. For details on the energy model, we readers are referred to [19].

In our implementation, we used the Open64 compiler [18], with the highest optimization level enabled, to extract the DDGs for software pipelineable loops from the `SPEC2000` integer benchmark suite. For the extracted DDGs, our implementation generates the ILP formulation which is solved by a commercial integer linear programming solver, CPLEX [7]. The CPLEX solver gave us the schedule, the initiation interval of the loop, and instruction partitioning (to fast and slow FUs) of each DDG. By taking trip count of each DDG (which we obtained by profiling using the "train" data sets) into account, we get an overall estimate of the program performance and energy consumption. Table 1 summarizes the benchmarks used in our experiments, the number of software pipelineable loops in them, and the characteristics of the DDGs associated with these loops.

Benchmark	No. 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 Used

4. Experimental Results

4.1 Impact of Slow FUs

In all our experiments, we considered fast and slow FUs for the *I-unit* FUs and *M-unit* FUs only, together referred to as integer FUs. Further, in our experiments we kept the total number of integer FUs as 4, while the number of slow/fast FUs is varied among the configurations. We consider the following four configurations:

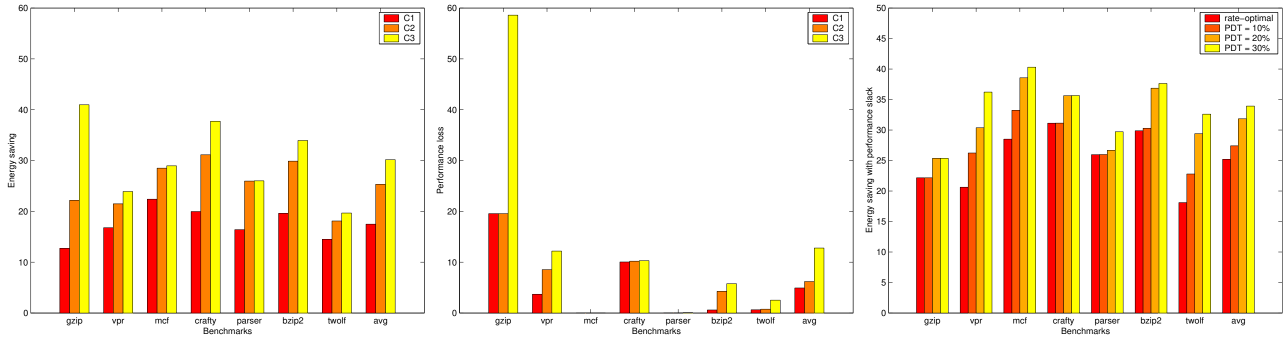
C0 2 fast *I-unit* and 2 fast *M-unit* FUs

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

C2 1 fast and 1 slow in each integer FU type

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

Figure 4(a) and (b) show 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 compromising 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



(a) Energy Gain under Different Configurations

(b) Performance Penalty under Different Configurations

(c) Energy Gain Trading Performance

Figure 4. Energy-Performance Tradeoff

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`.

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 [10]. 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. In Figure 4(c) we plot the energy savings compared to the baseline configuration (C0) for various PDT values. 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. When a PDT value of 30%, is used, the energy gain is more apparent for all benchmarks. The average energy savings for PDTs of 20% and 30% are 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

A limit study on available schedule slack was performed by Casmira et al [3] which report that in more than 75% of the execution cycles there exists at least one instruction that has slack. Pyreddy et al [12] 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 runtime profiling to guide instruction issue to these two sets of functional units. 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 [10]. The mechanism they adopt to predict instruction criticality is the critical path predictor buffer [4]. Although interesting, the buffer consumes additional power which may offset the power gain in functional units.

The work by Zhang et al [21] 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. A relevant work by Yun et al [20] targets power-aware software pipelining. They proposed a heuristic algorithm which extended iterative modulo scheduling [13], tries to minimize *step power* for a software pipelined loop on a cycle-by-cycle basis.

6. Conclusions

This paper reports a quantitative study of power-performance trade-offs 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.

We have proposed an elegant integer linear programming formulation for rate-optimal software pipelining on architectures involving dual-speed pipelines. We explore the design space of Itanium-like architectures for energy-performance tradeoffs. Our experimental results show that considerable energy saving can be achieved by introducing a few small functional units, with very small performance degradation. We also demonstrated that further energy gains can be obtained by trading performance.

Acknowledgment

The authors would like to acknowledge George Cai from Intel for his support on building our power-aware computing testbed. We wish to thank Roy Ju from Intel, Chenyong Wu from Open Research Compiler (ORC) team of Chinese Academy of Sciences, and Peng Zhao from University of Alberta, for their help and comments on the initial drafts of this paper. Last, we acknowledge the research fundings from DOE and NSF for building our compiler infrastructure.

References

[1] E. R. Altman, R. Govindarajan, and G. 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.

[2] D. Brooks, P. Bose, S. Schuster, H.Jacobson, P. K. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. Cook. Power-aware microarchtiecture: Design and modelling challenges for next-generation microprocessors. *IEEE Micro*, 20(6), Nov 2000.

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

[4] 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.

[5] R. Govindarajan, E. R. Altman, and G. 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.

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

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

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

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

[10] J.Seng, E.Tune, and D.Tullsen. Reducing power with dynamic critical path information. In *Proc. of MICRO-34*, pages 114–123, Austin, TX, Dec 2001.

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

[12] R. Pyreddy and G. Tyson. Evaluating design tradeoffs in dual speed pipelines. In *Workshop on Complexity-Effective Design*, Goteborg, Sweden, Jun 2001.

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

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

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

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

[17] J. 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.

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

[19] H. Yang, R. Govindarajan, G. R. Gao, and K. B. Theobald. A quantitative study on performance-power impact of dual-speed pipeli ne architectures. CAPSL Tech. Memo 42, Dept. of Elec. and Computer Eng., U. of Delaware, Newark, Del., Jun 2002. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[20] H.-S. Yun and J. 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.

[21] W. Zhang, N.Vijaykrishnan, M.Kandemir, M.Irwin, D.Duarte, and Y. Tsai. Exploiting VLIW schedule slacks for dynamic and leakage energy reduction. In *Proc. of MICRO-34*, pages 102–113, Austin, TX, Dec 2001.