VLIW Instruction Scheduling for Minimal Power Variation

SHU XIAO
Singapore Polytechnic
and
EDMUND M-K. LAI
Massey University

The focus of this paper is on the minimization of the variation in power consumed by a VLIW processor during the execution of a target program through instruction scheduling. The problem is formulated as a mixed-integer program (MIP) and a problem-specific branch-and-bound algorithm has been developed to solve it more efficiently than generic MIP solvers. Simulation results based on the TMS320C6711 VLIW digital signal processor using benchmarks from Mediabench and Trimaran showed that over 40% average reduction in power variation can be achieved without sacrificing execution speed of these benchmarks. Computational requirements and convergence rates of our algorithm are also analyzed.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors-compilers, optimization

General Terms: Algorithms

Additional Key Words and Phrases: Instruction scheduling, power variation reduction, VLIW processors

1. INTRODUCTION

Very Long Instruction Word (VLIW) processors are designed for executing programs that exhibit a high degree of instruction-level parallelism (ILP) such as those for multimedia signal processing. They are able to execute several instructions in parallel on separate functional units. The number of instructions being executed

Authors' Addresses: The authors were with the School of Computer Engineering, Nanyang Technological University, Singapore.
S. Xiao is now with the Department of Mathematics & Science, Singapore Polytechnic, Singapore 139651. Email: xiaoshu@sp.edu.sg
E. M-K. Lai is now with the Institute of Information Sciences and Technology, Massey University, Wellington, New Zealand. Email: e.lai@massey.ac.nz
Extension of conference paper [Xiao and Lai 2004]. The mixed-integer program in this paper makes use of a substantially improved set of constraints. The lower bound estimation of the branch-and-bound algorithm has also been improved. The experimental results are therefore completely new.

simultaneously varies from clock cycle to clock cycle. Since the power consumed by the processor at any given time depends on the number as well as the type of instructions that are being executed, there are potentially large fluctuations in processor supply current [Sami et al. 2002]. Large fluctuations in supply current will lead to increase in power supply voltage noise, commonly known as the di/dt problem, which may cause timing and logic errors [Chang et al. 1997; Smith et al. 1999; Chandrakasan et al. 2000]. For high-performance processors, the problem is more pronounced because they generally have a larger number of gates, wider datapaths and operate at higher clock frequencies, leading to larger surge currents within a shorter time period. Furthermore, large current variation is usually correlated with large current spikes that can adversely affect chip temperature [Brooks and Martonosi 2001] to which chip reliability and sub-threshold leakage power are exponentially related [Dhodapkar et al. 2000]. Besides, maximum battery life is attained when the *variance* of the discharge current distribution is minimized [Pedram and Wu 2002]. Battery efficiency may change by as much as 25% depending on the discharge current profile given the same average current.

Published works on the control of power variation use either the hardware approach [Pant et al. 1999; 2000; Grochowski et al. 2002; Joseph et al. 2003; Powell and Vijaykumar 2003; 2004; El-Essawy and Albonesi 2004] or the hybrid hardware/software approach [Hazelwood and Brooks 2004]. For VLIW processors, since the instruction schedules produced by the compiler largely determines the power profile, power variation control can be achieved through instruction scheduling. Two methods have been proposed specifically for minimizing processor power variance. The first one extends the performance-oriented iterative modulo scheduling algorithm by adding power-aware heuristics [Yun and Kim 2001]. The alternative method is to formulate this scheduling problem as a mixed-integer program (MIP) [Yang et al. 2002; Xiao and Lai 2004]. The advantage of the later approach is that optimal solutions can be guaranteed. However, the computational complexity of algorithms for solving the MIP is generally much higher than that for heuristic ones, particularly if generic MIP solvers are used as in [Yang et al. 2002].

Furthermore, there are three main problems with the resource usage model in [Yang et al. 2002; Xiao and Lai 2004]. First, they assumed that the instructions scheduled onto each functional unit were executed on dedicated resources in a fully pipelined manner. In fact, instructions with multi-cycle functional unit latency will lock the target functional unit for a number of cycles. No new instruction can be dispatched to that functional unit during this locking period. Second, they did not consider the issue of register utilization. Rescheduling instructions can change variables' lifetimes, which may increase pressure on the registers. Without sufficient registers, the register allocator must insert spill and restore code into the schedule which will cause extra delay and increase the total energy consumption of the resulting schedule. Third, the sharing of resources, such as read/write ports and shared buses, between the functional units had not been taken into account. As a result, illegal schedules may result.

Another problem with [Yang et al. 2002] is that a rather unrealistic instruction-level power model has been used. It assumes that every pipeline stage in a given functional unit consumes the same amount of power. In fact, significant differences

exist between the power consumed at different stages of the pipeline which are reflected by experimentally verified VLIW instruction-level power models such as the one proposed in [Julien et al. 2003]. Optimal schedules obtained based on the simplistic power models may be far from optimal in reality.

In this paper, the problem of VLIW instruction scheduling for minimal power *variance* is solved using the MIP approach. We shall refer to it as the *power-balanced scheduling problem*. We make use of an accurate VLIW instruction-level power model which is a modified version of the one proposed in [Julien et al. 2003]. Our MIP formulation is based on a more complete resource usage model which takes into account constraints imposed by multi-cycle functional unit latency as well as limited shared resources such as registers, read/write ports and buses that have not been considered before. Appropriate data dependency constraints similar to those used in [Chang et al. 1997; Leupers and Marwedel 1997; Wilken et al. 2000] are included to ensure that optimal execution performance of the resulting schedule is maintained. The major contribution of this paper is the problem specific branch-and-bound algorithm. It solves the MIP much more efficiently than generic solvers, making the technique feasible for production compilers. In particular, the heuristics used to guide the branching and selection processes greatly cut down the search space. The lower bound estimation process is computationally efficient, which also accelerates the convergence of the branch-and-bound algorithm. Another advantage of the proposed branch-and-bound algorithm is that the peak power can also be flexibly bounded by the branching heuristics. Empirical results show that an average power variation reduction of over 40% can be achieved without degrading the execution performance of the target programs and with a relatively low computational cost.

The rest of this paper is organized as follows. The original VLIW power model and our modifications that allow it to be used for instruction scheduling are presented in Section 2. In Section 3, our MIP formulation of the problem is presented. Our efficient branch-and-bound algorithm is developed in Section 4. Section 5 shows the experimental results based on the C6711 VLIW digital signal processor, using benchmarks programs from Mediabench and Trimaran.

2. INSTRUCTION-LEVEL POWER MODEL

Two instruction-level power models specifically for VLIW architectures can be found in the literature. The first one, presented in [Sami et al. 2000; 2002; Bona et al. 14; Benini et al. 2002; Zaccaria et al. 2003], has been used to characterize a four-issue VLIW core with a six-stage pipeline. It takes into account factors such as instruction ordering and power consumption in the pipeline stage. Experimental results carried out on a set of embedded DSP benchmarks have demonstrated an average error of 4.8% compared to gate-level simulations. The main disadvantage of this model is its complexity, which requires a large number of parameters to be estimated. Since these parameters are normally obtained via power measurements, the number of measurements required is prohibitively large. Furthermore, it also reduces the computational efficiency of the instruction scheduling algorithm.

The power model proposed in [Julien et al. 2003], on the other hand, has lower complexity. Furthermore, the accuracy of this model is comparable to the first

one with a maximum error between estimation and measurement of 4% for the C62 and 6% for the C67 series of digital signal processors. Another advantage is that the modeling methodology which is based on algorithmic activities is generally applicable to any VLIW processor. However, some modifications to this model are needed in order to be suitable for our instruction scheduling problem. In Section 2.1, we shall briefly review the main components of this power model, followed by a description of our modifications in Section 2.2.

2.1 An Algorithmic Activity Based Power Model

The algorithmic activity based power modeling methodology is motivated by the observation that the architectural complexity of VLIW processors hides the details of many internal activities. Hence no significant power consumption difference can be observed between similar types of instructions. For instance, for the C6201 digital signal processor, an addition and a multiplication dissipate about the same amount of power. The same is true for data transfer instructions between on-chip memories. Modeling of a target VLIW processor involves grouping its architectural components into functional blocks based on a functional-level concurrent activity analysis. The activity rates of these functional blocks and their interactions are modeled by instruction-level algorithmic parameters. The power cost is estimated from the instruction-level algorithmic activities of the target program.

The instruction-level algorithmic activity based modelling method in [Julien et al. 2003] is illustrated with C62. The model described in this section is based on this processor.

2.1.1 *Functional Blocks with Concurrent Activities.* The architecture of the target VLIW processor is modeled as two functional blocks – the instructions management unit (IMU) and the processing unit (PU). They model the concurrent activities in the pipeline steps. The pipeline steps can be separated into three stages:

(1) The fetch stage. It includes program address generation (PG), program address send (PS), program access ready wait (PW) and program fetch packet receive (PR), which fetch the instructions from program memory.
(2) The decode stage. The instructions are first dispatched to the correct unit (DP) and then decoded by the processing unit (DC).
(3) The execution stage. The instruction is executed in a variable number of steps.

Figure 1 shows the two functional blocks together with the associated concurrent activities. The first five pipeline steps (PG to DP) are performed by the IMU and the rest by the PU.

2.1.2 *Instruction-Level Algorithmic Parameters.* In the C6x family of VLIW processors [Texas Instruments 2000], eight instructions constitute a *fetch packet* and are fetched at the same time. The execution of the individual instructions in a fetch packet is partially controlled by a bit in each instruction which determines whether the instruction executes in parallel with another instruction. All instructions that are executed in parallel constitute an *execute packet*. The remaining instructions are executed in the cycles afterwards. The maximum size of an execute packet is

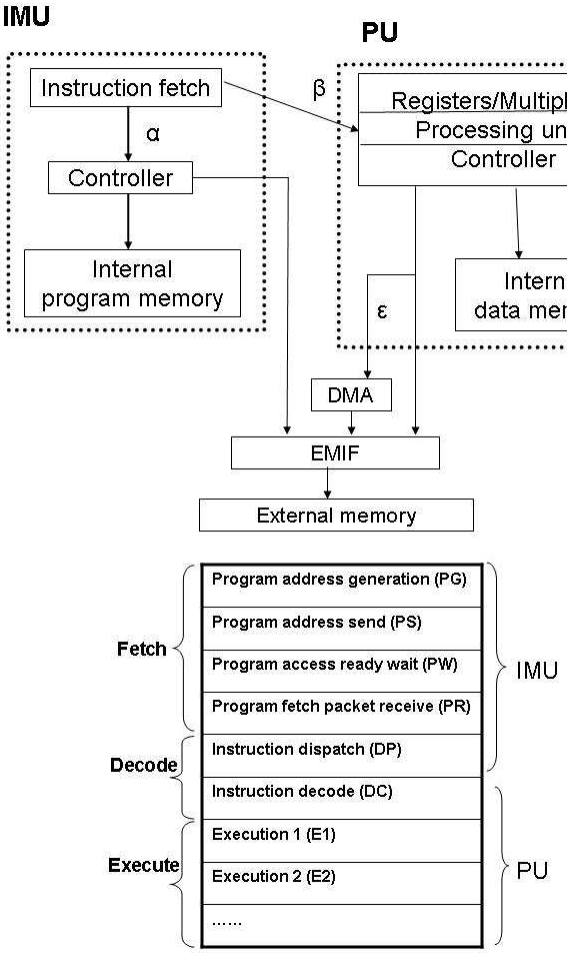

Fig. 1. Functional blocks with concurrent activities (from [Julien et. al. 2003]).

equal to the maximum issue width of the processor. Figure 2 shows an example of a fetch packet with three execute packets.

The activity rate of the functional blocks and their interactions are modeled by two algorithmic parameters α and β. They are obtained from the compiled code and have a significant impact on the final power consumption. The parallelism rate α indicates the average flow between the fetch stages and the program memory controller in the IMU. It can be defined in terms of the number of fetch packets NFP and execute packets NEP.

$$\alpha = NFP/NEP \tag{1}$$

Since $NFP \leq NEP$, $\alpha \leq 1$ with $\alpha = 1$ when parallelism is highest. In the example shown in Figure 2, $NFP = 1$, $NEP = 3$ and $\alpha = 1/3$.

The processing rate β between the IMU and the PU represents the utilization

```
Execute Packet 1 :
                ADD  .L1   A1, A2, A3
          ||    ADD  .L2   B1, B2, B3
Execute Packet 2 :
                LDW  .D2   *B5, B1
Execute Packet 3 :
                ADD  .L2   B2, B3, B4
          ||    MPY  .M1   A1, A2, A3
          ||    ADD  .L2   B1, B2, B3
          ||    NOP
          ||    ADD  .L1   A1, A2, A3
```

Fig. 2. Example of a fetch packet which consists of three execute packets.

rate of the processing units. It is given by

$$\beta = \frac{1}{NPU_{\max}} \left(\frac{NPU_{tot}}{NEP} \right) \tag{2}$$

where NPU_{max} is the number of processing units in the processor and NPU_{tot} is the total number of instructions which have been executed on the processing units. Thus NPU_{tot}/NEP indicates the average number of the processing units used per cycle. For the C62 processor, $NPU_{max} = 8$ and in the example shown in Figure 2, $NPU_{tot} = 7$ since one of the instruction is a NOP (no operation) which does not involve any execution. Thus, we have $\beta = 7/24$.

2.1.3 *Power Model Coefficients.* The total power I_{tot} consumed by the processor when there is no instruction or data cache misses is proportional to the power consumed by the IMU and the PU. Hence

$$I_{tot} = I_{IMU} + I_{PU} + e \tag{3}$$

where

$$I_{IMU} = a\alpha \tag{4}$$

$$I_{PU} = b\beta \tag{5}$$

and e is a constant which represents the idle power consumed by the processor when it is not fetching or executing any instruction.

The power consumption laws that form the power model are able to take into account all the power consumption sources including pipelines and internal memories. If pipeline stalls are considered, then α and β should be replaced by

$$\alpha' = \alpha(1 - PSR) \tag{6}$$

$$\beta' = \beta(1 - PSR) \tag{7}$$

respectively where PSR is the pipeline stall rate. If the power consumed by direct memory access (DMA) is included, then (3) becomes

$$I_{tot} = I_{IMU} + I_{PU} + I_{DMA} + e \tag{8}$$

with

$$I_{DMA} = f\varepsilon \tag{9}$$

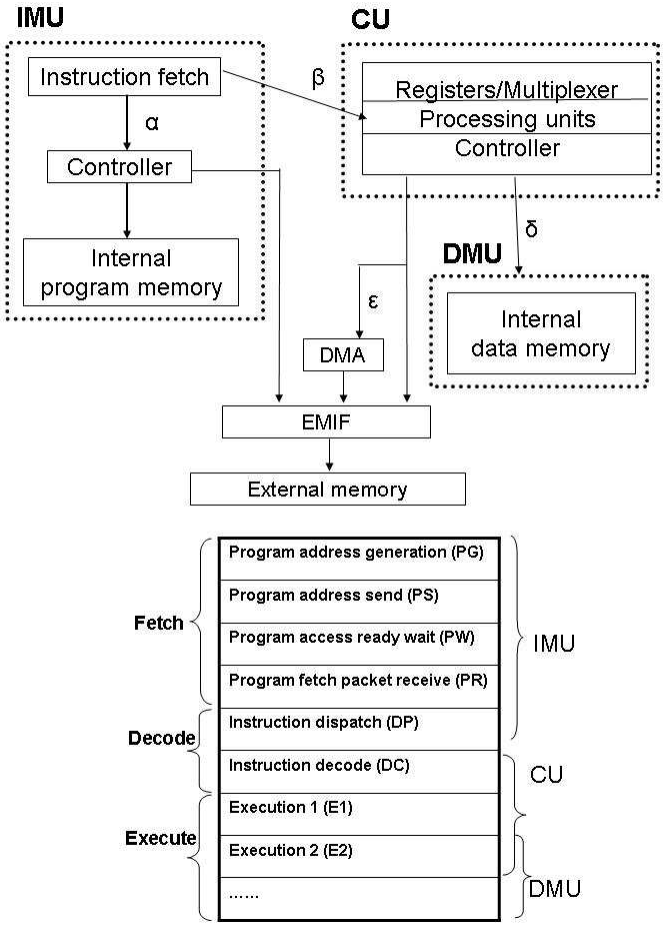

Fig. 3. The modified power model.

ε is the DMA utilization rate which represents the activity level of the DMA unit.

2.2 Model Modifications

In order to perform power-balanced instruction scheduling, run-time power profiling must be enabled by the power model. We are not able to produce such power profiles using the power model described in Section 2.1. Therefore we propose some modifications to it which are described in detail in Sections 2.2.1 to 2.2.4.

2.2.1 *Functional Blocks with Concurrent Activities.* Table I shows the average power cost of the instruction decode step DC and the various execution steps E1, E2, and E3 to E5 with respect to the parallelism rate α. It is obvious that the power cost of these steps are significantly different. More importantly, concurrent activities between internal memory and the processing units in the PU exist only when the executed instructions involve data memory access. In order to perform better run-

Table I. Power cost of each pipeline step (I_{step}) of a C6201 running at 200MHz (from [Julien et. al. 2003]).

α	$I_{step}(mA)$			
	Fetch and DP	DC and E1	E2	E3 to E5
0.125	197.4	60	4	95
0.25	111	64.75	4	95
0.5	69.6	62.5	NA	NA
1	48.125	63.31	NA	NA

time power profiling, fine-grained concurrent activity analysis is necessary. We propose that the PU be divided into two separate functional blocks. For those instructions that do not involve memory access, the decode stage can be grouped with the execute steps to form the computation unit (CU). Another functional block called the data memory unit (DMU) handles the execution steps that access internal data memory. The subdivided functional block grouping is shown in Figure 3.

2.2.2 *Instruction-Level Algorithmic Parameters.* The concepts of fetch and execute packets are unique to the C6x family of processors. In order to make our model applicable to more generic VLIW architectures, the definitions for α and β given by (1) and (2) have to be modified. Since the parallelism rate α is reflected by the average width K of the instruction words in IMU, it can be defined by

$$\alpha = K/K_{max} \tag{10}$$

where K_{max} is the issue width of the processor. The utilization rate β can be redefined as

$$\beta = NPU/NPU_{max} \tag{11}$$

where NPU is the average number of processing units used per cycle with the same physical meaning as the NPU_{tot}/NEP term in (2). For the C62 processor, $K_{max} = NPU_{max} = 8$. Applying (10) to the example in Figure 2, we have $K = 8/3$, and thus $\alpha = 1/3$ which is the same as that by (1). Similarly applying (11), we have $NPU = 7/3$ and $\beta = 7/24$.

With the subdivided functional blocks in Figure 3, an additional algorithmic parameter δ is needed to represent the activity rate of the DMU and its interactions with the CU. Parameter δ which represents the data memory access rate can be computed by

$$\delta = NMU/NMU_{max} \tag{12}$$

where NMU_{max} is the maximum number of internal data memory accesses that can be executed in a single instruction cycle and NMU is the average number of internal data memory accesses executed per cycle.

2.2.3 *Power Model Coefficients.* The expression for the total power I_{tot}, previously given by (3), now becomes

$$I_{tot} = I_{IMU} + I_{CU} + I_{DMU} + e \tag{13}$$

where I_{IMU}, I_{CU} and I_{DMU} are the power consumed by the IMU, CU and DMU respectively. Now,

$$I_{CU} = c\beta \tag{14}$$

```
LOOP :
        ADD  .L1  A1, A2, A3
  ||    ADD  .L2  B1, B2, B3
        ADD  .L1  A1, A2, A3
  ||    ADD  .L2  B1, B2, B3
        ...  ...
        ADD  .L1  A1, A2, A3
  ||    ADD  .L2  B1, B2, B3
        B    .S1   LOOP
```

Fig. 4. Example C6711 elementary program with $\alpha = 1/4$, $\beta = 1/4$ and $\delta = 0$.

```
LOOP :
        ADD  .L1  A1, A2, A3
        ADD  .L1  A1, A2, A3
        ...  ...
        ADD  .L1  A1, A2, A3
        B    .S1   LOOP
```

Fig. 5. Example C6711 elementary program with $\alpha = 1/8$ and $\beta = 1/8$ and $\delta = 0$.

$$I_{DMU} = d\delta \tag{15}$$

where c and d are the model coefficients for the CU and DMU respectively.

The power model coefficients a, c, d and e in (4) and (13)-(15) can be estimated by measuring the supply current to the processor (I_{tot}) in a similar way to [Tiwari et al. 1994; Russell and Jacone 1998]. A set of elementary programs are designed to activate the IMU, CU and DMU together or separately. For each elementary program the values of α, β and δ could be computed as described in Section 2.2.2. Each elementary program is typically placed in an infinite loop in order to obtain a stable current reading. There are some constraints on the size of the loops. On one hand, we want to minimize the impact of the branch at the end of the loop by having more instructions within the loop. On the other hand, the loop size should not be so large that it causes cache misses which are undesirable.

Suppose N elementary program experiments are conducted. Then the power model coefficients a, c, d and e can be estimated by linear regression that minimizes the sum of squares of the residuals $\sum_{q=1}^{N}(a\alpha + c\beta + d\delta + e - I_{tot})^2$. In order to cope with the effect of nuisance factors and the imprecision inherent in measurements, a large N is required.

The physical meaning of the model coefficients can be illustrated through the example programs in Figures 4 and 5. The program in Figure 4 consumes more power than the one in Figure 5. The power difference, according to the equations (10)-(15), is given by

$$\frac{a}{K_{max}} + \frac{c}{NPU_{max}}$$

The (a/K_{max}) component is the additional power consumed by the functional block IMU because the issue width of the instruction word K in Figure 4 is one more than the other program. The additional power consumed by the first program is also due to the number of processing units used per cycle NPU. Since this difference is

one, the power difference is (c/NPU_{max}).

2.2.4 *Power Consumption As A Function Of Time.* Our scheduling problem requires the computation of power consumption as a function of time. Let the instruction schedule be $\langle W_1, W_2, ..., W_i, ..., W_t \rangle$ where $W_i = \left(w_i^1, w_i^2, ..., w_i^k\right)$ is the long instruction word fetched at the i-th time slot of this schedule and $w_i^j,\ 1 \leq j \leq k$ are the individual instructions that make up W_i. Suppose the i-th time slot of this schedule corresponds to the PG (program address generation) pipeline step when W_i is fetched. Then the average power in the i-th time slot of this schedule is the sum of the power of the IMU, the power of the CU and the power of the DMU in this time slot. It can be expressed mathematically as

$$I^{(i)} = I_{IMU}^{(i)} + I_{CU}^{(i)} + I_{DMU}^{(i)} + e \tag{16}$$

According to equations (4) and (10), the power of the IMU in the i-th time slot $I_{IMU}^{(i)}$ is proportional to the parallelism $\alpha^{(i)}$ activated in this time slot. As depicted in Figure 3, in the i-th time slot when the instruction word W_i is in its PG pipeline step, W_{i-1} is already in the PS step, W_{i-2} is in the PW step, and so on. Let K_i, K_{i-1}, K_{i-2}, K_{i-3} and K_{i-4} be the width of the instruction words in time slots i, $i-1$, $i-2$, $i-3$ and $i-4$ respectively. We have

$$I_{IMU}^{(i)} = a\alpha^{(i)} \tag{17}$$

$$\alpha^{(i)} = K^{(i)} \Big/ K_{\max} \tag{18}$$

$$K^{(i)} = \frac{1}{5} \sum_{m=0}^{4} K_{i-m} \tag{19}$$

where $K^{(i)}$ is the average width of the instruction words in IMU in the i-th time slot.

Similarly, in the i-th time slot, instruction word W_{i-5} is in the DC (decode) step in the functional block CU. At the same time, W_{i-6} is in its first execution (E1) step. Even if instructions with multiple execution pipeline steps exist, we do not need to add up those instruction words which are in their pipeline steps after E1. This is because, according to Table I, the execution steps after E1 in CU consume much less power compared to the steps before. For example, there is no significant power dissipation difference between an addition and a multiply instruction because the second step E2 which exists for multiply only, consumes only 4mA per execution against 60mA per execution in the DC or E1 step. According to (14) and (11), the power of the CU in the i-th time slot $I_{CU}^{(i)}$ is proportional to the utilization rate $\beta^{(i)}$ activated in this time slot. Hence,

$$I_{CU}^{(i)} = c\beta^{(i)} \tag{20}$$

$$\beta^{(i)} = NPU^{(i)} \Big/ NPU_{\max} \tag{21}$$

$$NPU^{(i)} = \frac{1}{2} \sum_{m=5}^{6} NPU_{i-m} \tag{22}$$

where NPU_{i-5} and NPU_{i-6} are the number of instructions at their pipeline step

DC and $E1$ respectively.

Finally, in the i-th time slot, if any one of W_{i-7}, W_{i-8} and W_{i-9} is involved in the three pipelines steps for data memory access in the functional block DMU (address send, access ready wait and data receive), we have

$$I_{DMU}^{(i)} = d\delta^{(i)} \tag{23}$$

$$\delta^{(i)} = NMU^{(i)} \Big/ NMU_{\max} \tag{24}$$

$$NMU^{(i)} = \frac{1}{3} \sum_{m=7}^{9} NMU_{i-m} \tag{25}$$

where NMU_{i-7}, NMU_{i-8} and NMU_{i-9} are the number of internal data memory accesses executed at time slots i, $i-1$ and $i-2$ respectively.

2.3 Model Accuracy

In [Julien et al. 2003] the instruction-level algorithmic activity based modeling method was applied to the C62 and the C67 VLIW digital signal processors. These two processors share the same functional block analysis but the model coefficient values are different. The models were validated using a number of classical digital signal processing algorithms: a finite impulse response (FIR) filter, a least-mean-square (LMS) filter, a discrete wavelet transform with two image sizes: 64×64-pixel (DWT1) and 512×512-pixel (DWT2), an enhanced full-rate (EFR) vocoder based on the global system for mobile communication (GSM) standard, and an MPEG application. The average error between the estimates and the physical measurements was 2.5%, with a maximum error of 4%. The maximum error between the estimates and the physical measurements of the C67 was 6% [Julien et al. 2003].

The idea behind this instruction-level algorithmic activity based power modeling method is that the power consumed by a processor is proportional to the power consumed by the functional blocks, which are grouped by concurrent activity analysis. The power by each functional unit is proportional to the degree of parallelism in it. The architectural complexity of VLIW processors hides the details of many internal activities. Our modifications do not downgrade the model accuracy, because only the model representation has been changed in order to better profile power consumption over time. The idea behind this model remains the same.

3. MIXED-INTEGER PROGRAM FORMULATION

The MIP for power balanced scheduling consists of two main parts – the objective function and the constraints. The objective function is to minimize the power variance for the program segment considered. Instruction scheduling is traditionally done using list scheduling on basic blocks or modulo scheduling on loops [Allen and Kennedy 2002]. But for VLIW processors techniques are used enlarge the instruction scheduling scope and "region" refers to each program segment considered [Faraboschi et al. 2001; Kathail et al. 2001].

There are three main types of constraints in this MIP. Due to data dependency and resource usage conflicts among the instructions, not all combinations of instructions are allowed in a single very long instruction word. All legal combinations must firstly satisfy the data dependency constraints imposed by the data flow require-

ments of the target program. At the same time, they must also satisfy the resource constraints which are imposed by the architecture of the VLIW processor. The third type of constraints ensures that the total execution time of the schedule will not exceed that of the initial schedule. Therefore if a speed-optimized instruction schedule is used as the initial schedule for the MIP, the power-balanced solution will also be speed-optimized.

The complete mixed-integer program is formally given by **P1** below. The total execution time is divided into time slots. The total number of time slots and instructions is given by t and n respectively. A complete instruction schedule X is composed of the set of non-zero binary decision variables x_i^k. x_i^k equals 1 if instruction k is allocated to time slot i; otherwise it is 0.

Each functional unit is capable of executing certain types of instructions. A functional unit is indexed by the unit type and the index of the particular unit within that type. *Functional unit latency* L^k is the number of cycles that the instruction engages the functional unit k. The number of *delay slots* D^l associated with an instruction l is the number of cycles required before the result of that instruction, once issued, is available.

For ease of reference, the notations used in the MIP are listed in Appendix A. Detailed discussions on the objective function and the constraints are given in subsequent sections.

P1: $\min\ P(X)$

subject to

$$\begin{aligned} X &= \bigcup_{i,k:x_i^k=1} \{x_i^k\} \qquad k = 1,...,n; i = 1,...,t \\ x_i^k &\in \{0,1\} \qquad\qquad k = 1,...,n; i = 1,...,t \end{aligned} \tag{26}$$

$$\sum_{i=1}^{t} i x_i^m - \sum_{i=1}^{t} i x_i^l \geq D^l \quad \forall < l, m > \in E \tag{27}$$

$$\begin{aligned} \sum_{k=1}^{n} a_{qj}^k x_i^k \leq\ & 1 - \sum_{k=1}^{n} \Lambda(i - \sum_{z=1}^{t} x_z^k)\Lambda(\sum_{z=1}^{t} x_z^k + L^k - i) a_{qj}^k \\ & j = 1,...,u;\ q = 1,...,c_j;\ i = 1,...,t \end{aligned} \tag{28}$$

$$\sum_{l=1}^{n} R_c^l \leq g,\ c = 1,\cdots,t \tag{29}$$

$$\sum_{k=1}^{n}\sum_{j=1}^{u}\sum_{q=1}^{c_j} b_{kqj}^l a_{qj}^k x_i^k \leq 1, \quad l = 1,...,s;\ i = 1,...,t \tag{30}$$

$$\sum_{k=1}^{n}\sum_{j=1}^{u}\sum_{q=1}^{c_j} d_{qj}^l a_{qj}^k x_i^k \leq 1, \quad l = 1,...,r;\ i = 1,...,t \tag{31}$$

$$\sum_{k=1}^{n}\sum_{j=1}^{u}\sum_{q=1}^{c_j} e_{qj}^{l} a_{qj}^{k} x_i^k \le 1, \quad l = 1, ..., w;\ i = 1, ..., t \tag{32}$$

$$\sum_{i=1}^{t} i x_i^k + D^k \le t, \quad k = 1, ..., n \tag{33}$$

$$\sum_{i=1}^{t} x_i^k = 1, \quad k = 1, ..., n \tag{34}$$

3.1 Objective Function

Power consumption in each of the t time slots is obtained through equations (16)-(23). The variables in these equations are related to the binary decision variables in the MIP in the following ways.

K_i, K_{i-1}, K_{i-2}, K_{i-3} and K_{i-4} are the width of the instruction words in time slot i, $i-1$, $i-2$, $i-3$ and $i-4$ respectively. They can be computed by adding up the binary decision variables for corresponding time slots.

$$K_m = \sum_{k=1}^{n} x_m^k, \qquad m = i, i-1, \ldots, i-4 \tag{35}$$

NPU_{i-5} and NPU_{i-6} are the number of instructions decoded in time slots i and $i-1$ respectively. They are actually instructions already fetched in time slots $i-5$ and $i-6$ respectively. Therefore,

$$NPU_m = \sum_{k=1}^{n} x_m^k, \qquad m = i-5, i-6 \tag{36}$$

NMU_{i-7}, NMU_{i-8} and NMU_{i-9} are the number of internal data memory accesses executed at time slots i, $i-1$ and $i-2$ respectively. They relate to the instructions fetched in time slots $i-7$, $i-8$ and $i-9$ respectively. Hence,

$$NMU_m = \sum_{k=1}^{n} f^k x_m^k, \qquad m = i-7, i-8, i-9 \tag{37}$$

where $f^k = 1$ if instruction k involves internal data memory access. Otherwise it is zero.

Substituting (35)-(37) into (16)-(23), the total power consumption I_i in time slot i can be expressed in terms of the binary decision variables x_i^k.

$$\begin{aligned} I_i \;=\; & \tfrac{a}{K_{\max}} \cdot \tfrac{1}{5} \sum_{k=1}^{n} \left(x_i^k + x_{i-1}^k + x_{i-2}^k + x_{i-3}^k + x_{i-4}^k\right) \\ & + \tfrac{b}{NPU_{\max}} \cdot \tfrac{1}{2} \sum_{k=1}^{n} \left(x_{i-5}^k + x_{i-6}^k\right) \\ & + \tfrac{c}{NMU_{\max}} \cdot \tfrac{1}{3} \sum_{k=1}^{n} \left(f^k x_{i-7}^k + f^k x_{i-8}^k + f^k x_{i-9}^k\right) + e \end{aligned} \tag{38}$$

The average power over the duration of the whole schedule is

$$M = \frac{1}{t}\left(\sum_{i=1}^{t} I_i\right) \tag{39}$$

An appropriate objective function is therefore given by

$$P(X) = \sum_{i=1}^{t} (I_i - M)^2 \tag{40}$$

This objective function will be most appropriate for smoothing the discharge profile to improve battery efficiency. If it is desirable to bound the absolute peak power, the method discussed in Condition 1 of Section 4.2 could be used. However, it cannot guarantee a bound for the worst possible instantaneous current variation and so it cannot be used to solve the di/dt problem mentioned in Section 1.

3.2 Data Flow Dependency Constraints

Flow dependencies occur when one instruction m uses the result of another instruction l. The time slots where these instructions are scheduled are given by $\sum_{i=1}^{t} ix_i^l$ and $\sum_{i=1}^{t} ix_i^m$ respectively. The result of instruction l becomes available after D^l delay slots . Thus instruction m must be scheduled at least D^l time slots after instruction l. These constraints are specified by (27).

Register reuse can introduce additional flow dependencies. Anti-dependencies and output dependencies will not be considered, since they can be addressed by register renaming. Register pressure will be considered through the resource usage constraints.

3.3 Resource Usage Constraints

There are four types of resource constraints. They are the constraints on the functional units, registers, shared buses and read/write ports. We shall describe each of them in detail.

3.3.1 *Functional Unit Constraint.* There are two types of constraints on the functional units. The first one is simply that two instructions that use the same functional unit cannot be in the same instruction word. Since a_{qj}^k is the binary variable that indicates if the q-th functional unit of type j can execute instruction k, this type of constraint can be expressed mathematically as

$$\sum_{k=1}^{n} a_{qj}^k x_i^k \leq 1 \tag{41}$$

The second type of constraint on a functional unit is associated with instructions that has multi-cycle functional unit latency. These instructions occupy the functional unit for a number of cycles. So new instructions cannot be dispatched to that functional unit during this locking period. A functional unit q of type j is locked by an earlier instruction k in the current time slot i if the following three conditions are satisfied.

(1) Instruction k can be executed on this functional unit , i.e. $a_{qj}^k = 1$.

(2) The time slot when instruction k is issued, $\sum_{z=1}^{t} z x_z^k$, is earlier than i. This can be expressed as

$$\Lambda\left(i - \sum_{z=1}^{t} z x_z^k\right) = 1$$

where

$$\Lambda(x) = \begin{cases} 1, & x \geq 1 \\ 0, & otherwise \end{cases}$$

for integer values of x.

(3) Time slot i is in the lock-down period of instruction k, i.e. L^k time slots after k has been issued. Mathematically,

$$\Lambda\left(\sum_{z=1}^{t} z x_z^k + L^k - i\right) = 1$$

These three conditions can be combined into a single expression

$$\Lambda\left(i - \sum_{z=1}^{t} z x_z^k\right) \Lambda\left(\sum_{z=1}^{t} z x_z^k + L^k - i\right) a_{qj}^k = 1 \tag{42}$$

Check this against each of the n instructions gives us constraint (28).

3.3.2 *Register Constraints.* A rescheduling of instructions may lengthen a variable's lifetime, leading to increased pressure on registers. Therefore we need to analyze the lifetime of each register variable to ensure that there are sufficient registers for reallocation in each time slot. We shall assume that

(1) all registers are of the same type;
(2) every operand of an instruction occupies at most one register;
(3) the result of an instruction resides in the register used by later instructions.

These three assumptions describe the most common way by which instructions utilize the general registers. Some processors may have special-purpose instructions that make use of special registers. Extra register constraints can be included for the rescheduling of these special instructions. If an operand of certain instructions requires more than one register, the number of feasible time slots where this instruction can be reschedule into will be smaller since it takes up more registers. Furthermore, if data are allowed to be passed from one instruction to another directly, then these two instructions will have more feasible time slots to be rescheduled into.

The register usage chain can be derived from the dependence relations in the data flow graph. The lifetime of a register variable starts at the time when an instruction defines it until the time when it is last used by other instructions. Consider the data flow graph where only instruction m uses the register variable that instruction l has defined. Since $\sum_{i=1}^{c-D^l+1} x_i^l = 1$ if instruction l defined the variable no later than the time slot c and $\sum_{i=1}^{c-L^m+1} x_i^m = 0$ if instruction m uses it no earlier than c,

$$R_c^l = \sum_{i=1}^{c-D^l+1} x_i^l - \sum_{i=1}^{c-L^m+1} x_i^m \tag{43}$$

is equal to 1 if time slot c is in the lifetime of the defined register variable, where D^l is the number of delay slots of instruction l and L^m is the functional unit latency of instruction m.

Extending (43) to a set S of instructions that uses this register variable, we have

$$R_c^l = \Lambda \left(\sum_{\forall m \in S} \left[\sum_{i=1}^{c-D^l+1} x_i^l - \sum_{i=1}^{c-L^m+1} x_i^m \right] \right) \tag{44}$$

This is the basis of constraint (29), which ensures that the total number of live register variables in a given time slot does not exceed the total number of registers available.

3.3.3 *Shared Bus Constraints.* A shared bus l should not be used by more than one instruction in any time slot i. An instruction k would use shared bus l in time slot i if this instruction is allocated to time slot i, i.e. $x_i^k = 1$, and it can be executed on the q-th functional unit of type j that requires shared bus l, i.e. $a_{qj}^k b_{kqj}^l = 1$. Constraint (30) on shared bus usage is to ensure that the total number of instructions in a given time slot using the same shared bus must not exceed one.

3.3.4 *Read/Write Port Constraints.* If two functional units share a read register port that forbids parallel access for source operands, then instructions using these functional units cannot be scheduled to the same instruction word if they both require source operands. Similar restriction applies to the write register ports for the destination of the instructions. An instruction k would use read register port l in the time slot i if it is allocated to time slot i, i.e. $x_i^k = 1$, and it can be executed on the q-th functional unit of type j which shares read register port l, i.e. $a_{qj}^k d_{qj}^l = 1$. Thus we arrive at constraints (31) and (32).

3.4 Other Constraints

Constraint (33) guarantees that all execution deadlines must be met. In order to ensure that the execution performance of the target program is not compromised, we can set the total number of time slots as that in the optimal schedule obtained by a performance-oriented compiler.

Since the total execution time of the MIP solution does not exceed that of the initial schedule, the total power consumption of the whole program would remain unchanged. This is because equation (3) which is used for computing the total power is linear. Rescheduling instructions in this way will only result in a redistribution of power among the time slots available. While the power variation changes, the total power computed by (3) remains the same.

In addition, constraint (34) ensures that each instruction must be issued once and only once.

4. A BRANCH AND BOUND ALGORITHM

P1 can be solved using generic MIP solvers which are much more efficient than using exhaustive search. However, they do not have problem-specific knowledge to help reduce the search space and therefore are generally inferior to problem specific algorithms. In this section, we shall describe a branch-and-bound algorithm that

we developed for solving **P1** efficiently. One more advantage is that the peak power can also be flexibly bounded by one of the heuristics for branching.

4.1 Preliminaries

We shall assume that an initial schedule is obtained by using a standard performance optimized compiler. Therefore the task is to reschedule instructions for minimal power variation. Starting with the initial schedule X_1, a branch-and-bound tree can be constructed using X_1 as the root node. Each node of the tree represents a feasible schedule. A branch of the tree connects a *parent* schedules X_r to a *child* schedule X_t if X_t is obtained from X_r by rescheduling one single instruction.

For every schedule X_r in the branch and bound tree, we define two associate sets of instructions. The first set, denoted U_r, consists of the instructions rescheduled along the path from X_1 to X_r. The second set, denoted V_r, consists of all the remaining instructions yet to be rescheduled. A schedule X_s is a *successor* of X_r if we can obtain X_s from X_r by rescheduling only the instructions in the set V_r.

Let I_i^u denote the total power consumption in time slot i due to the instructions in U_r. I_i^u can be computed in a similar way to (38), i.e.

$$\begin{aligned} I_i^u \;=\; & \tfrac{a}{K_{\max}} \cdot \tfrac{1}{5} \sum_{\forall k \in U_r} \left(x_i^k + x_{i-1}^k + x_{i-2}^k + x_{i-3}^k + x_{i-4}^k\right) + \\ & \tfrac{b}{NPU_{\max}} \cdot \tfrac{1}{2} \sum_{\forall k \in U_r} \left(x_{i-5}^k + x_{i-6}^k\right) + \\ & \tfrac{c}{NMU_{\max}} \cdot \tfrac{1}{3} \sum_{\forall k \in U_r} \left(f^k x_{i-7}^k + f^k x_{i-8}^k + f^k x_{i-9}^k\right) + e \end{aligned} \tag{45}$$

We shall now discuss the branching process, selection rules and lower bound estimation of our algorithm in detail.

4.2 Branching

Let X_{CBS} denote the current best schedule. For any schedule X_r, if the lower bound of the objective function values for all the successors of X_r is not less than the objective function value for X_{CBS} (i.e. $P(X_{CBS})$), then no better schedule exists among the successors of X_r. Therefore there is no need to branch from X_r and X_r can be removed from the leaf schedule pool. Otherwise we branch from X_r.

The branching process for a selected leaf schedule X_r is summarized in Algorithm 1. New schedules are generated from X_r by rescheduling an instruction in the set V_r. The higher priority is given to instructions with less data dependencies. Instructions with less data dependencies typically have more schedule slacks for power balancing. The convergence of the branch-and-bound algorithm should accelerate if these schedule slacks are explored earlier.

Suppose instruction k in V_r is to be rescheduled. First, we branch from X_r by rescheduling instruction k to all time slots. The child schedules are checked against the constraints (27)-(32). The objective function values of those feasible ones are then computed. For any feasible child schedule X_f, if $P(X_f) < P(X_{CBS})$, then X_f becomes the new X_{CBS}. Next, X_r is deleted from the leaf schedule pool and the generated feasible child schedules are examined to see which ones should be inserted into this pool. Let X_s be the new schedule after instruction k is rescheduled to time slot j. Then X_s can be inserted into this pool if it satisfies the following five conditions.

Algorithm 1: The branching process for a selected leaf schedule X_r. (The array "*RescheduleInstructionSequence*" descending stores all the instructions according to their priorities to be rescheduled.)

input : The selected leaf schedule X_r determied by the selection rules in Section 4.3.
output: The updated leaf schedule pool.

```
if the lower bound for X_r < the objective function value of the
CurrentBestSchedule then
    i ← the depth of X_r;
    Reschedule the instruction at RescheduleInstructionSequence[i+1] to all
    the time slots;
    if the objective function value of any feasible child schedule < the
    objective function value of the CurrentBestSchedule then
        Replace CurrentBestSchedule by this child schedule;
    end
    if the depth of X_r < n − 1 then
        /*Otherwise the branching process will reach the bottom of the
        branch and bound tree.*/;
        Add the feasible child schedules which satisfy the five conditions
        described in Section 4.2 to the leaf schedule pool;
    end
end
Delete X_r from the leaf schedule pool.
```

Condition 1. I_j^u should not be larger than the peak power of the current best schedule. That is,

$$I_j^u \leq P_{max}\left(X_{CBS}\right) \tag{46}$$

where $P_{max}\left(X_{CBS}\right)$ is the peak power of the current best schedule.

If it is desirable to limit the peak power of the resultant schedule, then we can replace $P_{max}\left(X_{CBS}\right)$ in (46) by a predetermined bound on the peak power. However, by doing this, potentially optimal solutions may be abandoned.

Condition 2. Data dependencies must be satisfied. These dependencies are expressed by the following inequality:

$$\sum_{i=1}^{t} ix_i^b - \sum_{i=1}^{t} ix_i^a \geq G_{ab} \tag{47}$$

for all instructions $a, b = 1, 2, ..., n$ and $G_{ab} > 0$. Here, $\sum_{i=1}^{t} ix_i^a$ and $\sum_{i=1}^{t} ix_i^b$ respectively obtain the time slots where instructions a and b are scheduled to, and

$$G_{ab} = \begin{cases} L_{ab}, & \text{if instruction } b \text{ directly or indirectly depends on instruction } a \\ -1, & \text{otherwise} \end{cases} \tag{48}$$

where L_{ab} denotes the length of the longest path from instruction a to instruction b in the data dependency graph.

Applying (47) to the new schedule X_s obtained by rescheduling instruction k to time slot j, we have

$$\begin{aligned} j - \sum_{i=1}^{t} i x_i^a &\geq G_{ak} \\ \sum_{i=1}^{t} i x_i^b - j &\geq G_{kb} \end{aligned} \tag{49}$$

where $\forall a, b \in U_r$ and $G_{ak}, G_{kb} \neq -1$.

Condition 3. The time slots where an instruction may be scheduled must satisfy (47) as well as the deadline constraints (33). For instruction k in the new schedule X_s, the range of possible time slots are given by

$$\sum_{j=1}^{G_{front}^k} x_j^k = 0, \quad \text{if} \quad G_{front}^k \geq 1$$

$$\sum_{j=G_{front}^k+1}^{t-G_{back}^k+1} x_j^k = 1$$

$$\sum_{j=t-G_{back}^k+2}^{t} x_j^k = 0, \quad \text{if} \quad G_{back}^k \geq 1$$

where

$$G_{front}^k = \max_h(G_{h,k}) \tag{50}$$

$$G_{k,F} = \max_l(G_{k,l}) \tag{51}$$

$$G_{back}^k = G_{k,F} + D^F \tag{52}$$

where D^F is the number of time slots needed to execute instruction F which is defined in (51).

Condition 4. The resource constraints (28-32) must be satisfied in time slot j.

Condition 5. The lower bound of the objective function values for the successors of X_s must be less than $P(X_{CBS})$.

4.3 Selection Rules

Given the current leaf schedule pool, the branch and bound algorithm uses the selection rules to choose the one for branching. The selection strategy is to give higher priority to the leaf schedules with less instructions in its U_r. Such a selection rule can ensure that the instructions with less data dependencies are rescheduled earlier. Instructions with less data dependency may have more schedule slacks for power balancing. Therefore, it would accelerate the convergence of the branch and bound algorithm if these schedule slacks are explored earlier.

4.4 Lower Bound Estimation

The efficiency and effectiveness of the branch-and-bound algorithm is highly dependent on how accurately the lower bound of the objective function, given a selected schedule X_r, can be estimated. There are two basic requirements for the lower bound estimation algorithm. First, the estimated lower bound should be tight, i.e. its value should not be too far off the optimal value for the successors of X_r. This helps to reduce the search space. Second, it should be computationally efficient.

A common approach is to relax the integer constraints on the non-branched decision variables in the MIP. The optimal solution of the relaxed program then provides a lower bound estimate. However, a direct application of this approach to **P1** is not computationally efficient enough since it still involves solving the original program albeit without the integer variables. More importantly, this approach does not provide us with a lower bound that is tight enough for our problem.

Our approach is to recognize that at this time we are only interested in obtaining a lower bound of the objective function values instead of an optimal instruction schedule. In any successor of X_r, the power consumption at time slot i, denoted by I_i, is a sum of the power consumption of instructions already rescheduled (I_i^u) plus the power consumption of the instructions from the set V_r. By redistributing the power consumption of instructions in V_r, we obtain the minimum of the objective function (40), which is a lower bound of the objective function values for the successors of X_r.

The total power consumption of each instruction in V_r is composed of power consumption in each function block which is spread out over a number of time slots. We shall denote the power consumed by an instruction *in a time slot* in function blocks IMU, CU and DMU as P_{IMU}, P_{CU} and P_{DMU}. For example, we have $P_{IMU} = \frac{1}{5}\frac{a}{K_{max}}$ according to (17) and each instruction stays in IMU for continous five time slots. Let y_i^{IMU}, y_i^{CU} and y_i^{DMU} denote the number of units of P_{IMU}, P_{CU} and P_{DMU} respectively in a particular time slot i. An alternative mixed-integer program, $\mathbf{P_{LB}}$, can be formulated based on these variables. The optimal solution of $\mathbf{P_{LB}}$ will provide us with a lower bound of the objective function for the successors of X_r. This alternative MIP is specified as follows.

$\mathbf{P_{LB}}$:
$$\min\ P(X) = \sum_{i=1}^{t} (I_i - M)^2$$

subject to

$$\begin{aligned} y_i^{IMU} &\geq 0 \\ y_i^{CU} &\geq 0 \\ y_i^{DMU} &\geq 0 \end{aligned} \tag{53}$$

$$I_i^v = y_i^{IMU} P_{IMU} + y_i^{CU} P_{CU} + y_i^{DMU} P_{DMU}, \quad i = 1, ..., t \tag{54}$$

$$\sum_{i=1}^{t} I_i^v = I_{total} - \sum_{i=1}^{t} I_i^u \tag{55}$$

$$I_i = I_i^u + I_i^v, \quad i = 1, ..., t \tag{56}$$

where

$$I_{total} = tM \tag{57}$$

$\mathbf{P_{LB}}$ has the same objective function as **P1**. However, the decision variables, y_i^{IMU}, y_i^{CU} and y_i^{DMU}, are the real-valued instead of binary (x_i^k). I_i^u represents the total power of the instructions that has already been rescheduled. I_i^v represents the power of the instructions that has yet to be rescheduled. In solving $\mathbf{P_{LB}}$, we are essentially redistributing the various portions of power of those instructions in V_r, instead of the whole instructions, to the available time slots. Note that no actual instruction schedules are explicitly obtained.

The advantage of using $\mathbf{P_{LB}}$ is that it implicitly ensures that instructions will not occupy fractions of a time slot by sub-dividing the power of a single instruction in a functional block into integer number of time slots. In contrast, if we simply relax the constraints on the binary decision variables x_i^k of **P1** in a conventional way, some instructions will not start at the beginning of a time slot. This leads to a solution with power variation much less than that obtained through $\mathbf{P_{LB}}$. Thus $\mathbf{P_{LB}}$ is able to provide us with a much more accurate lower bound. This is stated formally by the following theorem.

THEOREM 4.1. *For a given schedule X_r, the optimal value of $P(X)$ obtained by the integer program $\mathbf{P_{LB}}$ is a lower bound of the objective function $P(X)$ of the mixed-integer program* **P1** *for all successors of X_r.*

PROOF. For a given schedule X_r, let s_{LB} denote the optimal value of $P(X)$ for the integer program $\mathbf{P_{LB}}$. We need to prove that for any successor X_s of X_r, $s_{LB} \leq P(X_s)$.

Let $\mathbf{P1'}$ be the subproblem of **P1** with the partial schedule X_r. It is an MIP with binary decision variables x_i^k for $k \in V_r$ and $i = 1, ..., t$. Let $s_{P1'}$ denote the optimal objective function value of $\mathbf{P1'}$. Then for any successor X_s of X_r,

$$s_{P1'} \leq P(X_s) \tag{58}$$

Now compare $\mathbf{P_{LB}}$ with $\mathbf{P1'}$ for X_r. The two formulations have the same objective function. But in $\mathbf{P_{LB}}$, the data dependency and resource constraints on the instructions in V_r have been removed. Since $\mathbf{P_{LB}}$ is actually the same problem as $\mathbf{P1'}$ but with less constraints, the optimal value of $\mathbf{P_{LB}}$ must be less than the one obtained by $\mathbf{P1'}$. That is, for X_r,

$$s_{LB} \leq s_{P1'} \tag{59}$$

Therefore, based on (58) and (59), $s_{LB} \leq P(X_s)$ for any successor X_s of X_r. □

Furthermore, $\mathbf{P_{LB}}$ is a very simple integer program which can be solved very efficiently. It does not have any data dependency and resource constraints as in

input : The schedule X_r.
output: The lower bound of the objective function values for all the successor generated from X_r.

```
bound = 0;
calculate each I_i^u according to (45); initiate each I_i with its I_i^u: I_i = I_i^u ;
tempCount ← number of units of P_DMU consumed by the instructions in V_r;
for i ← 1 to tempCount do
    Suppose the value in time slot W is the smallest among the current I_i (i = 1 to t), then I_W = I_W + P_DMU;
end
tempCount ← the number of units of P_IMU consumed by the instructions in V_r;
for i ← 1 to tempCount do
    Suppose the value in time slot W is the smallest among the current I_i (i = 1 to t), then I_W = I_W + P_IMU;
end
tempCount ← the number of units of P_CU consumed by the instructions in V_r;
for i ← 1 to tempCount do
    Suppose the value in time slot W is the smallest among the current I_i (i = 1 to t), then I_W = I_W + P_CU;
end
bound ← the objective function value computed from the obtained I_i(i = 1, ..., t);
Return bound;
```

Algorithm 2: A water-filling algorithm to solve the integer program $\mathbf{P_{LB}}$.

P1. Hence it can be solved by using a simple water-filling algorithm as shown in Algorithm 2. This algorithm starts with the schedule where each time slot already has power consumption given by I_i^u. The power units of the instructions in V_r are placed into the time slots of this schedule by filling those time slots with lowest power first. The power units which are larger are chosen first because it is easier to use the smaller portions to "fill in the gaps" later so that power variation is minimized.

5. PERFORMANCE EVALUATION

Texas Instruments' C6711 digital signal processor is used as the target VLIW processor for our experiments. Its detailed resource constraints and instruction set information can be found in [Texas Instruments 2000]. Figure 6 illustrates its internal organization. The algorithmic activity based power model for the C6711 described in Section 2.2 is employed.

The Mediabench [Lee et al. 1997] and the Trimaran benchmarks [Chakrapani et al. 2005] are used. The benchmark programs are compiled using the compiler in Code Composer Studio with optimization options "-o3" (optimization enabled at

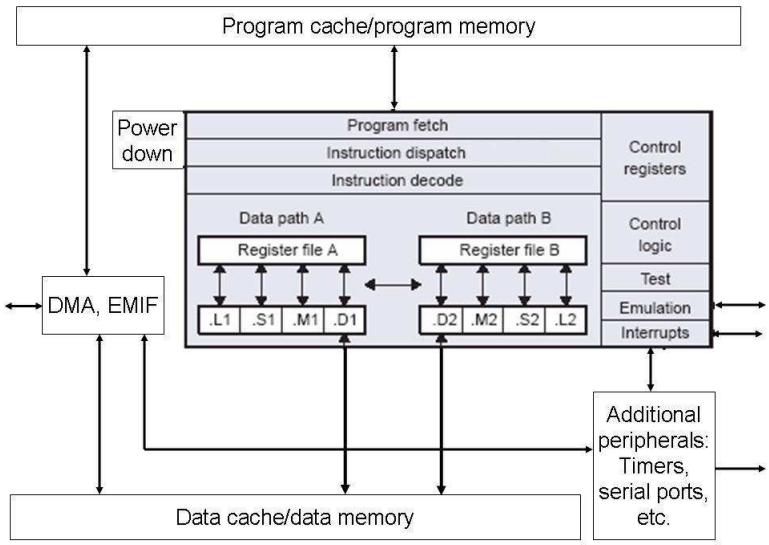

Fig. 6. Functional Organization of TMS320C6711 (from [TI 2000]).

file level) and "-ms0" (speed first, size second). The instruction schedules obtained are used as the initial feasible solution for **P1**. By setting the total number of available time slots as these speed optimized ones, the solution provided by **P1** will have the same speed performance but with power variations minimized. In this way, the proposed branch-and-bound algorithm is used an additional back-end phase in the non-power-aware compiler.

All our computational experiments were conducted on an Intel Pentium 4 personal computer running at 2.8 GHz with 512MB RAM under Microsoft Windows 2000. The numerical results are shown in Table II. The major findings are summarized as follows.

(1) *Reduction in power variation*: Overall, our algorithm produces schedules with an average improvement of 46.85% for the Trimaran benchmarks and 41.76% for the MediaBench benchmarks while maintaining the same speed performance as the non-power-balanced schedules (refer to Columns "Fn", "F" and "Imv"). Comparison between the values under Columns "Fn" and "F" for Trimaran and Mediabench benchmarks is also respectively highlighted in Figure 7.

(2) *Reduction in maximum power deviation*: The reductions on average in terms of maximum deviation from the mean are 24.10% for the Trimaran benchmarks and 17.31% for the MediaBench benchmarks (refer to Columns "fn", "f" and "Imf").

Figure 8 respectively shows the percentage of rescheduled instructions and the percentage of time slot with rescheduled instructions for each Trimaran benchmark. These percentages are not exactly consistent with those improvement percentages under Column "Imv" in Table II. The reason is that the optimization achievement also depends on the power consumption and execution clock cycles of the

Table II. Power variation reduction, computational efficiency and largest power deviation reduction of Trimaran benchmarks and Mediabench Benchmarks.

Trimaran	Fn ($\times 10^6$)	F ($\times 10^6$)	T (sec.)	fn	f	Imv (%)	ImvT ($\times 10^6$)	Imf (%)
bmm	12.23	5.98	1.86	400	324	51.09	3.36	19.05
dag	1.55	0.953	0.046	290	290	38.57	13.01	0
eight	0.96	0.63	0.031	227	138	34.22	10.56	39.16
fact2	0.65	0.38	0.03	162	115	41.53	8.69	29.45
fib	0.753	0.347	0.046	162	94	53.85	8.81	41.98
fib-mem	1.07	0.504	0.11	213	147	52.87	5.19	30.86
fir	9.28	6.14	1.185	468	423	33.81	2.65	9.66
hyper	0.69	0.374	0.031	156	99	45.68	10.14	36.68
ifthen	1.59	0.988	0.062	290	290	37.83	9.70	0
mm	4.04	2.076	0.498	399	310	48.67	3.95	22.15
mm-double	5.56	3.08	0.72	390	308	44.61	3.45	20.98
mm-dyn	4.14	1.74	0.688	399	229	58.15	3.49	42.55
mm-int	3.42	1.64	0.39	420	287	52.21	4.58	31.56
nested	2.12	0.76	0.16	414	195	64.20	8.71	52.85
sqrt	3.94	2.60	0.69	365	365	33.88	1.95	0
strcpy	1.16	0.61	0.093	224	224	47.31	5.90	0
wave	2.16	0.91	0.20	302	203	57.95	6.21	32.85
Average						46.85	6.49	24.10

Mediabench	Fn ($\times 10^6$)	F ($\times 10^6$)	T (sec.)	fn	f	Imv (%)	ImvT ($\times 10^6$)	Imf (%)
adpcm	3.74	1.84	8.97	384	267	50.85	0.212	30.38
epic	240.99	134.74	188.72	821	718	44.09	0.56	12.49
g721	65.78	35.01	23.47	524	496	46.77	1.31	5.43
gsm	362.23	265.77	191.71	796	631	26.63	0.50	20.68
jpeg	1416.44	781.59	1497.38	883	741	44.82	0.42	16.10
mpeg2	981.26	614.46	808.78	955	775	37.38	0.45	18.80
Average						41.76	0.58	17.31

Fn: power variation defined by (40) of schedules produced by Code Composer
F: power variation defined by (40) of schedules produced by the branch-and-bound algorithm
T: the computation time of the branch-and-bound algorithm
fn: maximum power deviation from the mean for schedules produced by Code Composer
f: maximum power deviation from the mean for schedules produced by the branch-and-bound algorithm
Imv: percentage improvement of "F" over "Fn"
ImvT: the gain per unit time $ImvT = \frac{Fn-F}{T}$
Imf: percentage improvement of "f" over "fn"

rescheduled instructions.

Figure 9 shows the convergence behavior for a program block with 60 instructions and 39 time slots. As a result of the proposed branching rules and selection rules, the branch-and-bound algorithm is able to reach a solution with an objective function value within 4.4% of the global optimum after only 380 leaf schedules have been visited.

For complex programs, the time taken to reach the optimal solution may be unacceptably long. Given the convergence behavior of the branch-and-bound al-

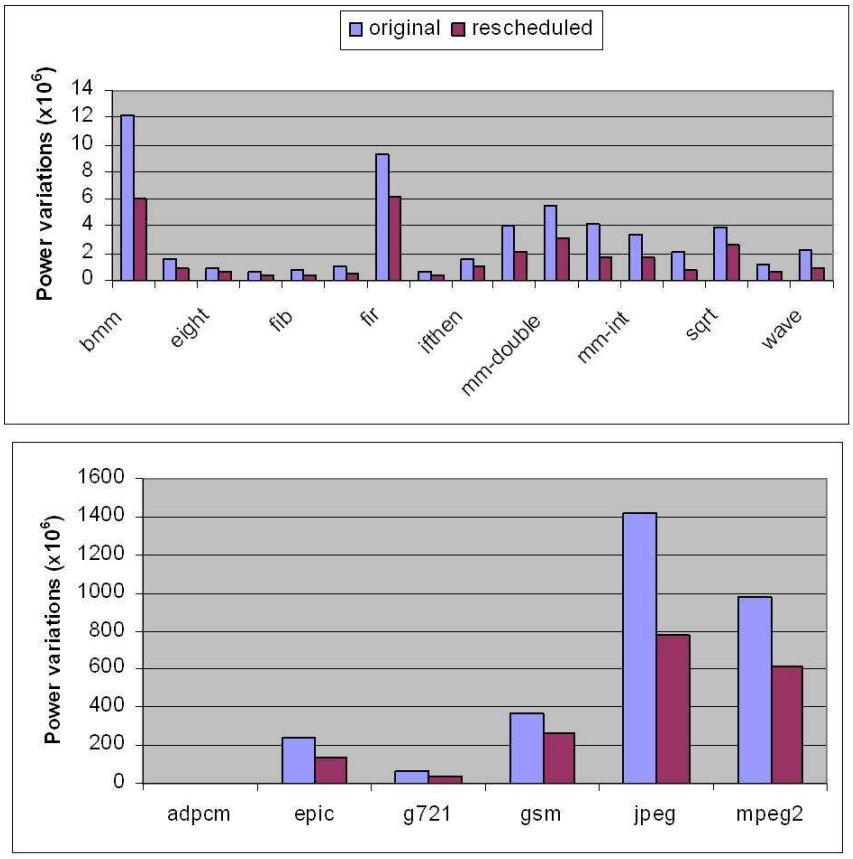

Fig. 7. Comparison of power variation between the original schedules and the rescheduled for Trimaran benchmarks and Mediabench Benchmarks.

gorithm, it may be sensible to obtain a sub-optimal solution within a reasonable time instead. Using the most time consuming mediabench benchmarks, "jpeg" and "mpeg2", we set a maximum limit of 300 nodes per instruction block. The results in Table III shows that an average saving of 70.39% in computation time can be achieved at the cost of a reduction in the improvement of power variation by 9.68% (refer to Columns "dImv" and "uT"). This tradeoff result between computation time and power variation reduction percentage is also highlighted in Figure 10 for jpeg and mpeg2 respectively.

6. CONCLUSIONS

Although VLIW is an energy-efficient architecture and VLIW instruction scheduling techniques for performance optimizations are adaptable to total energy optimization, instruction schedules that are optimized for speed often exhibit large variation in processor power consumption during the execution of the target program. This paper focusses on minimizing the power variance by rescheduling instructions without compromising on the execution speed. The problem is formulated as a mixed-

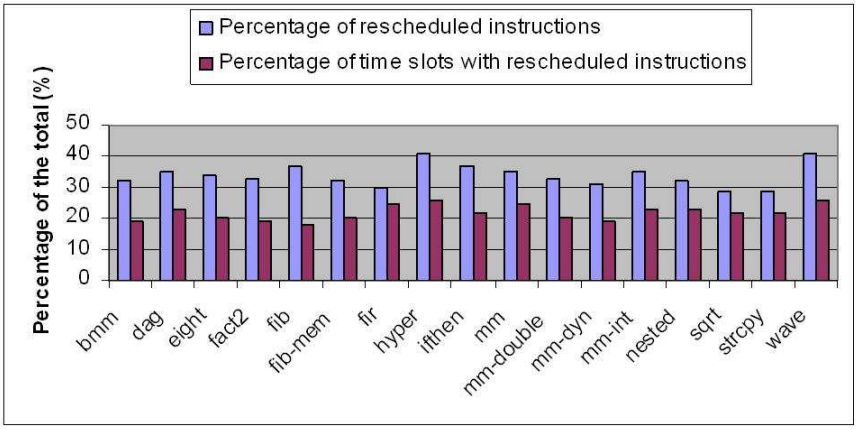

Fig. 8. Percentages of rescheduled instructions and time slots with rescheduled instructions in the power-balanced schedules for Trimaran benchmarks.

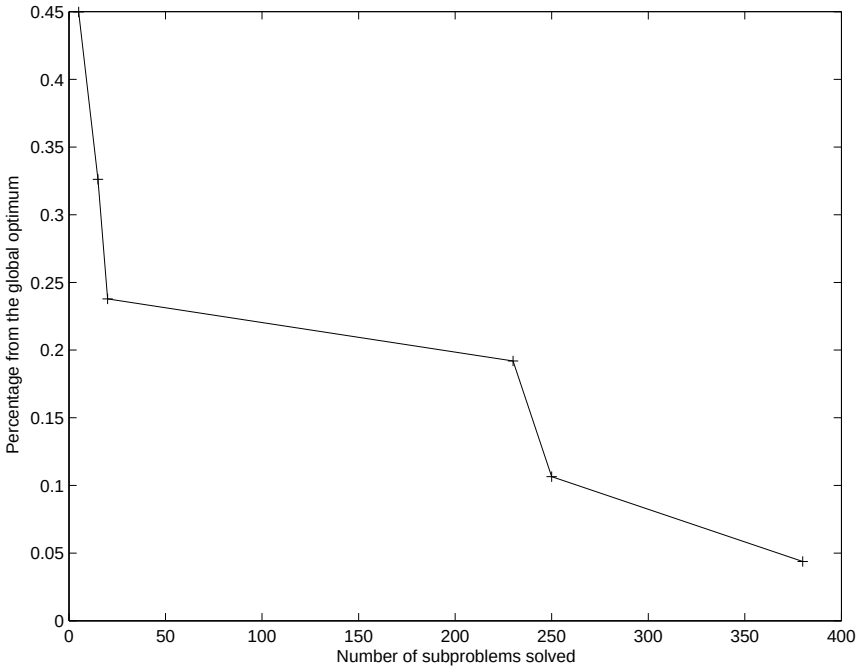

Fig. 9. Convergence behavior of the branch-and-bound algorithm for a problem size of 60 instructions and 39 time slots.

Table III. Tradeoff between computation time and power variation reduction with maximum subproblem 300.

Program	Fn ($\times 10^6$)	F_{300} ($\times 10^6$)	T_{300} (sec.)	Imv_{300} (%)	$ImvT_{300}$ ($\times 10^6$)	dImv (%)	uT (%)
jpeg	1416	909	490	35.83	1.035	8.99	67.25
mpeg2	981	716	214	26.99	1.237	10.39	73.53
Average						9.68	70.39

Fn: power variation defined by (40) of the schedules produced by Code Composer
F_{300}: power variation defined by (40) of schedules produced by the branch-and-bound algorithm with maximum branching limit of 300
T_{300}: the computation time for F_{300}
Imv_{300}: percentage improvements of "F_{300}" over "Fn"
$ImvT_{300}$: the gain per unit time $ImvT_{300} = \frac{Fn - F_{300}}{T_{300}}$
dImv: the degraded best objective function values computed by $dImv = Imv - Imv_{300}$
uT: computation time saving percentage by comparing "T_{300}" and "T" in Table II

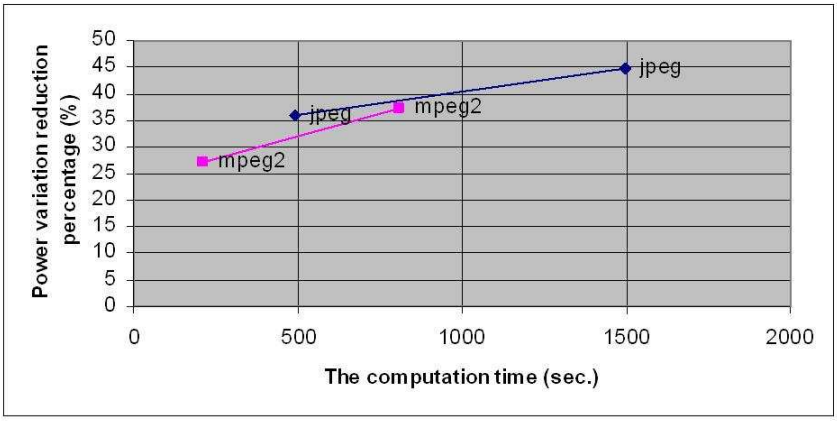

Fig. 10. Tradeoff between computation time and power variation reduction percentage with maximum subproblem 300.

integer program. The major contribution of this paper is a branch-and-bound algorithm that can solve this MIP much more efficiently than generic solvers, making the technique more attractive for use in practical compilers. In particular, the heuristics used to guide the branching and selection processes are able to reduce the search space substantially. The lower bound estimation process is effective and computationally efficient, which also accelerates the convergence of the branch-and-bound algorithm. Furthermore, the peak power can also be flexibly bounded by the heuristics for branching. The results of simulation experiments based on the C6711 VLIW digital signal processor using benchmarks programs from Mediabench and Trimaran confirmed the effectiveness and efficiency of our method.

7. FUTURE WORK

The techniques proposed in this paper are for VLIW instruction scheduling at compile time. During the execution of the program, power variation would be affected by instruction and data cache misses. These factors are not considered in this paper. An extension of the current work is to develop techniques that can be used at the compile time to estimate and handle cache misses.

Only simulated results are presented. The problem would be treated more thoroughly if direct physical measurements of runtime power variation on a real system are performed. This requires measuring the instantaneous current from a program execution trace for any benchmark program at least the processor clock frequency. The existing setup uses sampled multimeter data for measuring runtime power consumption [Isci and Martonosi 2003]. But if this setup is used for our purpose, some dynamics measurement techniques have to be refined. There is need for synchronization between the software application execution, the processor's clock, and the oscilloscope [Muresan and Gebotys 2001], since a well and true power vs. time waveform is required in order to determine the position of an instruction or of a set of instructions in the waveform and then give some insight into the possible defect of our algorithm and the improvement to be made. The work next is to develop measurement schemes appropriate for our purpose.

A. NOTATIONS OF THE MIP

. n is the total number of instructions in the given schedule.

. t is the total number of time slots available.

. $x_i^k = 1$ if instruction k is allocated to time slot i. Otherwise, $x_i^k = 0$.

. X is the set of n variables x_i^k which equal to 1. Thus X specifies a schedule.

. u is the total number of functional unit types in the target VLIW processor.

. c_j is the total number of functional units of type j.

. g is the total number of physical registers.

. s is the total number of shared buses.

. r is the total number of shared read register ports which forbid parallel access.

. w is the total number of shared write register ports which forbid parallel access.

. $a_{qj}^k = 1$ if the q-th functional unit of type j can execute instruction k. It is zero otherwise.

. $b_{kqj}^l = 1$ if instruction k allocated to the q-th functional unit within type j uses shared bus l. It is zero otherwise.

. $d_{qj}^l = 1$ if the q-th functional unit of type j shares read register port l. It is zero otherwise.

. $e_{qj}^l = 1$ if the q-th functional unit of type j shares write register port l. It is zero otherwise.

. E is the set of all flow-dependent pairs $< l, m >$, where instruction m depends on instruction l.

. $R_c^l = 1$ if instruction l defines a register variable and it is live at time slot c. Otherwise, $R_c^l = 0$.

. D^k is the number of delay slots of instruction k.

. L^k is the functional unit latency of instruction k.

. $f^k = 1$ if instruction k involves internal data memory access. It is zero otherwise.

. I_i is the average power in the time slot i.

. M is the average power over all t time slots.

. A function $\Lambda(x)$ is defined by

$$\Lambda(x) = \begin{cases} 1, & x \geq 1 \\ 0, & otherwise \end{cases}$$

where x is an integer.

REFERENCES

Allen, R. and Kennedy, K. 2002. *Optimizing compiler for modern architectures: A dependence-based approach*. Morgan Kaufmann, San Francisco.

Benini, L., Bruni, D., Chinosi, M., Silvano, C., Zaccaria, V., and Zafalon, R. 2002. A framework for modeling and estimating the energy dissipation of VLIW-based embedded systems. *Design Automation for Embedded Systems 7*, 183–203.

Bona, A., Sami, M., Sciutos, D., Silvano, C., Zaccaria, V., and R.Zafalon. 2002 10–14,. Energy estimation and optimization of embedded VLIW processors based on instruction clustering. In *Proc. 39th Design Automation Conference*. New Orleans, USA, 886–891.

Brooks, D. and Martonosi, M. 2001. Dynamic thermal management for high-performance microprocessors. In *Proc. Int. Symp. on High-Performance Computer Architecture*. 171–182.

Chakrapani, L. N., Gyllenhaal, J., mei W. Hwu, W., Mahlke, S. A., Palem, K. V., and Rabbah, R. M. 2005. Trimaran: An infrastructure for research in instruction-level parallelism. In *Languages and Compilers for High Performance Computing: 17th International Workshop: Revised Selected Papers*, R. Eigenmann, Z. Li, and S. P. Midkiff, Eds. Lecture Notes in Computer Science, vol. 3602. Springer Verlag, 32–41.

Chandrakasan, A. P., Browhill, W. J., and Fox, F. 2000. *Design of high-performance microprocessor circuits*. Wiley-IEEE Press.

Chang, C., Chen, C., and King, C. 1997. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. *Computers Math. Applic. 34,* 9, 1–14.

Chang, Y. S., Gupta, S. K., and Breuer, M. A. 1997. Analysis of ground bounce in deep sub-micron circuits. In *VLSI Test Symposium, IEEE*. 110–116.

Dhodapkar, A., Lim, C. H., and Cai, G. 2000. Tem2p2est : A thermal enabled multi-model power/performance estimator. In *Proc. of Workshop on Power-Aware Computer Systems*. 112–125.

El-Essawy, W. and Albonesi, D. 2004. Mitigating inductive noise in smt processors. In *Proc. Int. Symp. on Low Power Electronics and Design*.

Faraboschi, P., Fisher, J. A., and Young, C. 2001. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE 89,* 11 (Nov.), 1638–1659.

Grochowski, E., Ayers, D., and Tiwari, V. 2002. Microarchitectural simulation and control of *di/dt*-induced power supply voltage variation. In *Proc. Int. Symp. on High-Performance Computer Architecture*. 7–16.

Hazelwood, K. and Brooks, D. 2004. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. In *Proc. Int. Symp. on Low Power Electronics and Design*. 326–331.

Isci, C. and Martonosi, M. 2003. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proc. Int. Symp. on Microarchitecture*. 93–104.

Joseph, R., Brooks, D., and Martonosi, M. 2003. Control techniques to eliminate voltage emergencies in high performance processors. In *Proc. Int. Symp. on High-Performance Computer Architecture*. 79–90.

Julien, N., Laurent, J., Senn, E., and Martin, E. 2003. Power consumption modeling and characterization of the TI C6201. *IEEE Micro 23,* 5 (Sep.-Oct.), 40–49.

Kathail, V., Schlansker, M. S., and Rau, B. R. 2001. Compiling for EPIC architectures. *Proceedings of the IEEE 89,* 11 (Nov.), 1676–1693.

Lee, C., Potkonjak, M., and Mangione-Smith, W. H. 1997. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proc. Int. Symp. on Microarchitecture*. 330–335.

Leupers, R. and Marwedel, P. 1997. Time-constrained code compaction for DSP's. *IEEE Trans. on Very Large Scale Integration Systems 5*, 112C122.

Muresan, R. and Gebotys, C. 2001. Current consumption dynamics at instruction and program level for a VLIW DSP processor. In *Proc. Int. Symp. on System Synthesis*. 130 – 135.

Pant, M., Pant, P., Wills, D., and Tiwari., V. 1999. An architectural solution for the inductive noise problem due to clock-gating. In *Proc. Int. Symp. on Low Power Electronics and Design*. 255–257.

Pant, M., Pant, P., Wills, D., and Tiwari., V. 2000. Inductive noise reduction at the architectural level. In *Proc. Int. Conf. on VLSI Design*. 162–167.

Pedram, M. and Wu, Q. 2002. Battery-powered digital CMOS design. *IEEE Trans. on Very Large Scale Integration Systems 10,* 5 (Oct.), 601–607.

Powell, M. D. and Vijaykumar, T. N. 2003. Pipeline damping: A microarchitectural technique to reduce inductive noise in supply voltage. In *Proc. Int. Symp. on Computer Arch.* 72–83.

Powell, M. D. and Vijaykumar, T. N. 2004. Exploiting resonant behavior to reduce inductive noise. In *Proc. Int. Symp. on Computer Arch.* 288–299.

Russell, J. T. and Jacone, M. 1998. Software power estimation and optimisation for high performance, 32-bit embedded processors. In *Proc. Int. Conf. on Computer Design: VLSI in Computers & Processors*. 328–333.

Sami, M., Sciuto, D., Silvano, C., and Zaccaria, V. 2000. Power exploration for embedded VLIW architectures. In *Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*. 498–503.

Sami, M., Sciuto, D., Silvano, C., and Zaccaria, V. 2002. An instruction-level energy model for embedded VLIW architectures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 21,* 9 (Sept.), 998–1010.

Smith, L., Anderson, R., Forehand, D., Pelc, T., and Roy, T. 1999. Power distribution system design methodology and capacitor selection for modern CMOS technology. *IEEE Transactions on Advanced Packaging 22,* 3 (Aug.), 284–291.

Texas Instruments. 2000. TMS320C6000 CPU and instruction set reference guide. Reference Guide SPRS088E, Texas Instruments Inc. Oct.

Tiwari, V., Malik, S., and Wolfe, A. 1994. Power analysis of embedded software a first step toward software power minimization. *IEEE Trans. on Very Large Scale Integration Systems 2,* 4 (Dec.), 437–445.

Wilken, K., Liu, J., and Heffernan, M. 2000. Optimal instruction scheduling using integer programming. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.* Vancouver, Canada, 121–133.

Xiao, S. and Lai, E. M.-K. 2004. A branch and bound algorithm for power-aware instruction scheduling of VLIW architecture. In *Proc. Workshop on Compilers and Tools for Constrained Embedded Syst.* Washington DC, USA.

Yang, H., Gao, G. R., and Leung, C. 2002. On achieving balanced power consumption in software pipelined loops. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*. Grenoble, France, 210–217.

Yun, H. and Kim, J. 2001. Power-aware modulo scheduling for high-performance VLIW processors. In *Proc. Int. Symp. on Low Power Electronics and Design*. Huntington Beach, California, USA., 40–45.

Zaccaria, V., Sami, M., Sciuto, D., and Silvano, C. 2003. *Power estimation and optimization methodologies for VLIW-based embedded systems*. Kluwer, Boston.

Received ; revised ; accepted