

# Power Modeling and Reduction of VLIW Processors \*

Weiping Liao and Lei He

Electrical and Computer Engineering Department  
University of Wisconsin, Madison, WI 53706

## Abstract

In this paper, we first present a cycle-accurate power simulator based on the IMPACT toolset. This simulator allows the designer to evaluate both VLIW compiler and microarchitecture innovations for power reduction. Using this simulator, we then develop and compare the following techniques with a bounded performance loss of 1% compared to the case without any dynamic throttling: (i) clock ramping with hardware-based prescan (*CRHP*), and (ii) clock ramping with compiler-based prediction (*CRCP*). Experiments using SPEC2000 floating point benchmarks show that the power consumed by floating point units can be reduced by up to 31% and 37%, in *CRHP* and *CRCP* respectively.

## 1 Introduction

Power is rapidly becoming one of the primary design constraints for modern processor design due to increased complexity and speed of the system. Cycle-accurate microarchitecture level power simulations such as Wattch [1], SimplePower [2], and  $TEM^2P^2EST$  [3] have been developed, and used extensively to validate power-efficient microarchitecture innovations including clock gating [4], dynamically reconfiguring resources [5], etc. However, all aforementioned work focuses on the superscalar architecture.

In this paper, we study the power modeling and reduction for the VLIW architecture. Our contributions include:

- We integrate the Cai-Lim power model [6, 7] into the IMPACT toolset [8], and develop a cycle-accurate power simulator named PowerImpact. This simulator allows the designer to evaluate both VLIW compiler and microarchitecture innovations for power reduction.
- We develop and compare the following techniques with a bounded performance loss of 1% compared to the case without any dynamic throttling: (i) clock ramping with hardware-based prescan (*CRHP*), and (ii) clock ramping with compiler-based prediction (*CRCP*).

- Experiments using PowerImpact and SPEC2000 floating point benchmarks show that the power consumed by floating point units can be reduced by up to 31% for *CRHP* and 37% for *CRCP*, respectively.

The rest of the paper is organized as follows. Section 2 describes the power simulator for VLIW architecture. Sections 3 presents implementations of *CRHP* and *CRCP*. Section 4 shows the experiment results. Section 5 concludes the paper and discusses the ongoing work.

## 2 Cycle-Accurate VLIW Power Simulation

Existing work [1, 2, 3] considers superscalar architecture. There is hardly any architectural level power simulator for Very Long Instruction Word(VLIW) architecture. In this section, we first introduce the IMPACT infrastructure for VLIW architecture[8] and the Cai-Lim power model. We then present the power simulation enhancement to IMPACT.

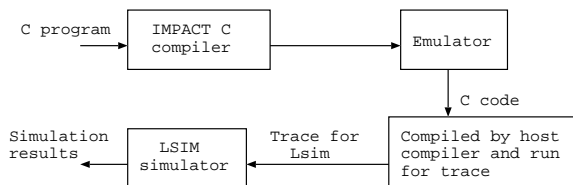


Figure 1: Flow diagram for IMPACT

### 2.1 IMPACT Architecture Framework

The IMPACT toolset (<http://www.crhc.uiuc.edu/IMPACT/>) contains the IMPACT EPIC architecture, compiler, and emulation and simulation tools (see Figure 1). The IMPACT compiler compiles C benchmark with both front-end and back-end optimizations. The emulator translates the intermediate representation to C code, which can be compiled by host compiler and be executed to generate a trace for Lsim, a cycle-accurate microarchitectural-level performance simulator. This toolset has been successfully utilized to conduct

\*This research was partially supported by the NSF CAREER Award 0093273 and SRC grant 2000-HJ-782. We used computers donated by HP and SUN Microsystems. Address comments to lhe@ece.wisc.edu.

FUB's name	corresponding hardware	FUB's name	corresponding hardware
npclog	next pc generation logic	decodepla	Instruction decoder
btblog	BTB logic	decodemisp	Misprediction handling logic
btbcac	BTB cache	fuint	Integer execution unit
rsbcac	Return Stack Buffer	fufp	Floating point execution unit
itlbcac	Instruction TLB	ul2log	Unified L2 cache logic
dtlbcac	Data TLB	ul2tag	Unified L2 cache tag
illlog	L1 instruction cache logic	ul2cac	Unified L2 cache array
illtag	L1 instruction cache tag	reglog	Register File logic
illcac	L1 instruction cache array	reg	Registers
dlllog	L1 data cache logic	vcachelog	Victim cache logic
dlltag	L1 data cache tag	vcachetag	Victim cache tag
dllcac	L1 data cache array	vcachecac	Victim cache array
biu	Bus/IO buffer		

Table 1: FUBs in our power model

the system level architectural experiment and new code optimization [8, 9].

In this paper, we implement our power modeling within Lsim, but implement our power reduction techniques within the back-end compiler, emulator and Lsim simulator.

## 2.2 Cai-Lim Power Model

The Cai-Lim power model [6] was originally used for superscalar architecture. It partitions the processor into Functional Unit Blocks (FUBs). Each FUB is characterized by areas ( $A(n)$ ) of four circuit types - dynamic, static, PLA (programmable logic array), and clock and memory to implement this FUB, as well as active power densities  $P_a(n)$  and inactive power densities  $P_i(n)$  for each type of circuit. Then, the energy dissipation of a FUB is given by

$$E = \sum_n A_n \cdot (P_a(n) \cdot active\_cycles + P_i(n) \cdot inactive\_cycles) \quad (1)$$

where  $n$  iterates over the four circuit types, and numbers of active and inactive cycles are collected by cycle-accurate performance simulator. The energy dissipation computed for each FUB separately is added up to get the total power dissipation.

## 2.3 PowerImpact

We integrate the Cai-Lim power model into IMPACT toolset, and name the resulting new toolset as PowerImpact. Figure 2 illustrates the overall structure of PowerImpact. We develop an interface between Lsim and the power models of FUBs. According to the structure of Lsim simulator, we partition the VLIW architecture supported by Lsim into total twenty five Functional Unit Blocks(FUBs) shown in Table 1. PowerImpact reads the user specified power information and the

system configuration, then the activities and corresponding power information for FUBs are collected in every clock cycle. The PowerImpact toolset is able to simulate the performance, average power, and step power (i.e., the power difference between two consecutive cycles) for every functional block and the whole system for given benchmark programs.

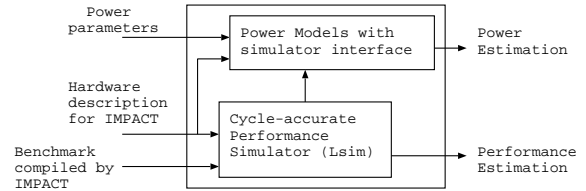


Figure 2: Overall structure of the PowerImpact

## 3 Clock Ramping

Clock gating is effective to reduce the dynamic power consumption of functional units. Most existing work [10, 11, 12] assume that the dynamic throttling can be achieved *instantly*. However, turning on/off a functional unit in a short time (e.g. within one clock cycle) will lead to a large surge current. A large surge current requires higher design and manufacturing costs for on-chip voltage supply, reduces the circuit reliability, and limits the voltage scaling for further power reduction.

To reduce the surge current by these clock gating technologies, Tiwari et al [13, 14] first proposed to extend the switch on/off time by inserting "waking up" and "going to sleep" time between the on and off state. In this case, the clock gating takes a few cycles and can be called *clock ramping* for differentiation from the conventional clock gating approach in [10]. To avoid the performance penalty introduced by the extra switching cycles, the clock ramping with hard-

ware prescan (*CRHP*) is proposed in [15]. An extra set of fetch and decode logic<sup>1</sup> are used to prescan the incoming instructions so that the clock gated functional units can be ramped up *in time* for the upcoming instructions. The super-scalar architecture is assumed in [15].

In this paper, we develop a new compiler optimization technology, which automatically inserts ramp-up instructions (*RUI*) based on hyperblock scheduling to instruct the *in-time* ramping up of functional units. Therefore, no extra fetch and decode logic used in the hardware prescan is needed. We call the new clock ramping technology as clock ramping using compiler-based prediction (*CRCP*). For comparison, we also implement an improved CRHP technique for VLIW architecture, with a finer clock ramping granularity to achieve more power reduction compared to [15].

In the following subsections, we present first the improved CRHP and then the new CRCP. Because FPUs consume almost 10-20% power of the processor, we use FPUs to illustrate our ideas based on SPEC benchmark simulation.

### 3.1 Clock Ramping with Hardware Prescan (*CRHP*)

The conventional floating point unit (FPU) design only has two states: *inactive state* and *active state* (see Figure 3(a)). When there are floating point instructions executed, the FPU is in the active state and consumes active power ( $P_a$ ). On the other hand, FPUs have no activity in the inactive state and dissipate leakage power ( $P_i$ ), about 10% of the active power ( $P_a$ ) in present process technology. When any floating point instruction gets into the FPU, the FPU will jump up from the inactive state to the active state in one clock cycle. This approach may lead to a large surge current.

To reduce the surge current at the architecture level, we assume that the power level does not change within a clock cycle, and define the *step power* as the power difference between the previous and present cycles. Further, we assume that the bigger the step power, the larger the surge current. Therefore, the step power can be used as a figure of merit of the surge current. Then, we can insert a few cycles between the fetch and execution stages and introduce intermediate power consumption levels between the inactive and active states to reduce the step power. Figure 3(b) illustrates the clock ramping technique first proposed in [14, 13]. This approach may result in a big performance loss however.

In comparison, Figure 3(c) shows that our *clock ramping with instruction prescan* method. It prescans the coming instructions before these instructions are fed into the instruction fetch (IF) stage, and ramps up the corresponding FPUs based on the result of prescanning.

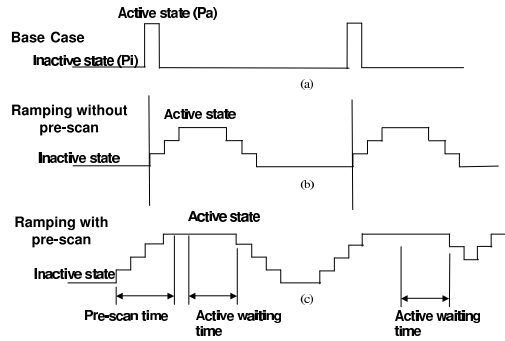


Figure 3: The relationship of states.

For the microarchitecture in the Lsim, there are two clock cycles from IF to EXE stages. If we prescan a floating point instruction  $N$  clock cycles earlier before it gets into the IF stage, we can have  $(N+2)$  clock cycles to gradually power up the target FP unit to the active state if there is no functional unit stall. We call  $N$  the *prescan time* ( $T_p$ ). Further, we define the time to ramp up a functional unit as *ramping time*  $T_r$ .  $T_r$  of a functional unit is decided by the design constraints on the surge current, and is assumed to be independent of the pipeline stall. When there is no pipeline stall,  $T_p + 2 = T_r$  is required to ensure no performance loss and is assumed in [15]. This assumption will be removed in this paper for better performance and more power reduction.

As in [15], we define the *active waiting time* ( $T_a$ ) as the time that an idle FPU remains in the active state before its ramping down. It helps to exploit the spatial and temporal locality of FP instructions.

Note that we apply clock ramping to each individual FPU. In the implementation presented in [15], all FPUs are treated as a whole floating-point block, and are ramped up and down simultaneously. Clearly, not all FPUs are used at the same time. Figure 4 shows the run-time utilization rates of FPUs for SPEC2000 FP benchmarks *equake* and *art*, with the hardware configuration of 6-issue width and totally 4 FPUs. Clearly only a small fraction of total FPUs are required most time. It is easy to predict that our ramping of each individual FPU can reduce more power compared to the ramping of the whole FP block in [15].

### 3.2 Clock Ramping with Compiler-based Prediction (*CRCP*)

Alternative to hardware based prediction, compiler can be used to predict incoming FP instructions. In our compiler-based clock ramping method, the compiler decides when and how many FPUs are needed by the incoming floating-point

<sup>1</sup>We can also use a larger instruction buffer to avoid the extra set of fetch and decode logic. But the performance in our experiment became much worse due to branches.

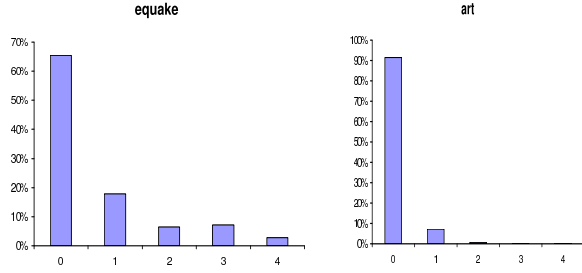


Figure 4: Utilization rate for FPUs

instruction. Such decisions can be coded into a special type of instructions called ramp-up instructions (*RUI*), and be inserted into the instruction sequence. When RUIs are fetched, the hardware will ramp up FPUs as many as needed. We call this method clock ramping based on compiler prediction (*CRCP*). Note that the ramping down is still decided by the hardware in *CRCP*.

In VLIW architecture, instructions are grouped into bundles. An interesting observation is that bundles are not full most time. Figure 5 shows the utilization rates of bundles for SPEC2000 floating-point benchmark programs *equake* and *art*. Clearly only a small fraction of bundles are full. Therefore, RUIs can be inserted into empty slots of bundles. The basic *CRCP* algorithms and a variety of improvement will be discussed below.

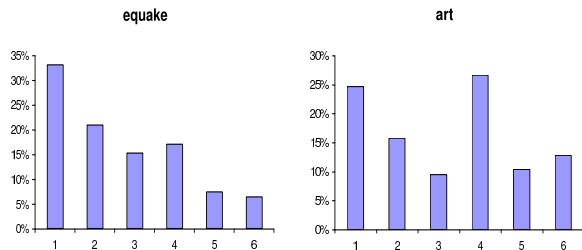


Figure 5: Distribution of instruction numbers in bundles, with bundle width = 6

### 3.2.1 Basic *CRCP* Algorithm

We choose hyperblock [16] as the basic structure in our *CRCP* algorithm. A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations. The motivation behind hyperblock is to group a number of basic blocks from different control flow paths into a single manageable block for compiler optimization and scheduling [16].

We first define two concepts for the ease of description: (1) the latency of a bundle, as the maximum latency of the instructions in the bundle; (2) the distance between two bun-

dles A and B, as the sum of the latencies of all bundles between A and B, including the latency of bundle A.

We apply our *CRCP* algorithm as an extra back-end compiler optimization after the compiler finishes performance-related optimization and scheduling. Our algorithm searches each hyperblock for floating-point instructions (*FPI*). During our search, once we find a bundle with FP instructions called *FP bundles*, we go upstream with distance  $D$  and reach the bundle called the *target* bundle. If we succeed in inserting a RUI into the target bundle, the distance  $D$  is called prediction time  $T_p$ . It is the counterpart of the prescan time  $T_p$  in *CRHP* so we use the same symbol to represent them. When there is no pipeline stall,  $T_p + 2 = T_r$  is required to prevent performance loss. Figure 6(a) illustrates how we choose the target bundle. In this figure, bundle B is the FP bundle and bundle A is the target bundle. The distance between A and B is  $T_p$ . In this case, The RUI contains only the number of FPUs needed by the correspondent FP bundle.

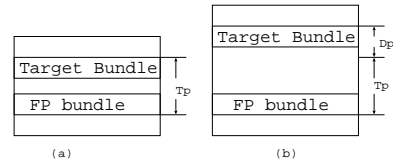


Figure 6: Insert ramp-up instructions

It is possible that the target bundle is full, and there is no slot to insert RUI into this bundle. In this case, we choose to continue going upstream until we find a bundle with at least one empty slot to insert the RUI. However, in this case  $T_p + 2 > T_r$ , which means the hardware will ramp up FPUs too early and cause unnecessary power consumption. To avoid this, we record the distance  $D_p$  (as shown in Figure 6) between the ideal location for RUI and the first feasible location for RUI. The hardware will not ramp up FPUs right after it fetches an RUI, but ramps up FPUs  $D_p$  cycles later.

Further, if we reach the head of the entrance point of a hyperblock, we should consider each branch, except those off-trace branches, to this block and continue searching upstream on each branch point. Figure 7 shows this case. Clearly it may introduce extra RUIs and increase power consumption. But such RUIs are necessary to improve performance.

When a RUI is fetched, the hardware obtains the  $D_p$  and the number of FPUs that is needed by the incoming FP bundle. After  $D_p$  cycles, the hardware checks the states of all FPUs, then ramps up as many FPUs as needed. For example, if the incoming FP bundle has four FP instructions as indicated by RUI and there are already two FPUs in the active state, then only two extra FPUs will be ramped up. It is easy to see that in our *CRCP* approach, the hardware is much simpler than that in *CRHP*. There is no extra set of fetch or

Fetch/Decode/Issue/Execute width	6			
BTB size	1024 entries 2-way associative			
Memory	page size 4096 bytes, latency 30 cycles			
Memory bus bandwidth	8 bytes/cycle			
Functional Unit	number	Latency		
Integer Unit	4	1		
FPU	4	2 for FP add and FP multiply, 15 for FP divide		
Cache	number of sets	block size	associativity	Replace Policy
L2 Cache	4096	256	1	LRU
L1 Instruction Cache	1024	64	2	LRU
L1 Data Cache	512	64	4	LRU
dcache 1st-level TLB	32	4096	fully	LRU
dcache 2nd-level TLB	256	4096	fully	LRU

Table 2: System configuration for experiments

decode logic needed.

After an FPU is used, it is kept in active state for the length of active waiting time. This is the same as that in CRHP. The rest of this subsection describes improvements over the basic algorithm.

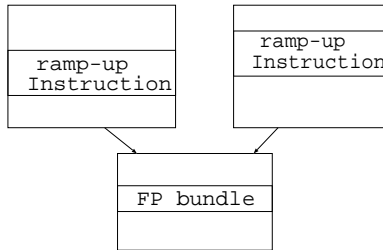


Figure 7: Insertion of ramp-up instructions beyond the current Hyperblock

### 3.2.2 Reduce Redundant Ramp-up Instructions

Inside each block, if the distance between two FP bundles is smaller than the active waiting time, and the latter FP bundle has FPIs no more than previous one, then we can simply skip the latter FP bundle and do not need to insert RUI for it. Because the two bundles are in the same block, it is very possible (but not definitely because we choose hyperblock, not basic block) that the previous FP bundle is executed before the latter one. So within the active waiting time, if the latter one has fewer FP instructions, its requirement will be met for sure. For this reason, we avoid inserting RUI for the latter FP bundle so that we can get rid of redundant RUIs and save power.

### 3.2.3 Control Flow

If we confront a procedure call instruction when searching upstream, we find the return instructions of the procedure and continue searching upstream from the return instruction.

Also, when we move out of a block while searching upstream, we should check if this is the head of a procedure. If so, we need to search the whole program, find every procedure call to the current procedure, and continue searching upstream from every procedure call instruction.

### 3.2.4 Load Instructions

Load instructions have pre-defined latencies in IMPACT. However, the actual run-time latencies for load instructions can be much larger than the pre-defined value when cache misses happen. Because the ramping of FPUs doesn't stall when the pipeline is stalled, if the load latency becomes larger than the sum of the FPU ramp-up time and active waiting time, the FPU will ramp down before the instruction arrives at the execute stage, which may cause a large performance loss.

To reduce the performance loss, we apply the following simple amendment. If we detect during the decode stage a data hazard due to a load instruction, we simply pick one active FPU and keep it in the active state until the load instruction finishes. Because an FP bundle is most likely to contain one FP instruction (see Figure 4), keeping one FPU in the active state can prevent a large performance loss with small power consumption overhead as shown by experiment results in section 4.

## 4 Experiment Results

In this section, SPEC2000 FP benchmark programs *equake* and *art* are used to study the performance and power impacts of various power reduction techniques. We mea-

sure performance in IPC, and compare our performance and power to those without any dynamic throttling. The system configuration used in our experiment is summarized in Table 2. Similar to the Intel Itanium architecture (<http://developer.intel.com/design/ia-64/>), our configuration has a fetch width of six and four FPUs.

Figures 8-11 show the performance loss and power reduction achieved by the CRHP and CRCP approaches, for the benchmark programs *equake* and *art*, respectively. The two parameters in the figures are the active waiting time  $T_a$  and prediction/prescan time  $T_p$ . We assume that the ramping time is  $T_r = 10$  in all experiments in this paper.

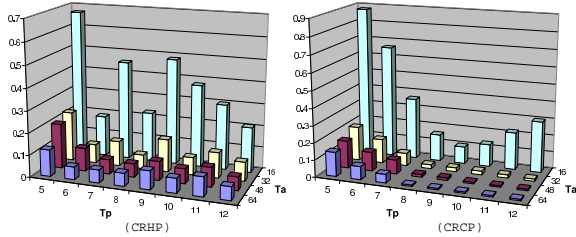


Figure 8: Performance loss (in percentage as the Z-axis variable) of CRHP and CRCP approaches for *equake*.

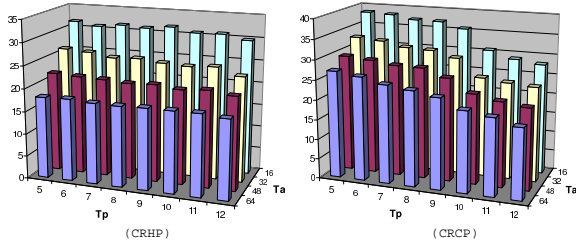


Figure 9: Power reduction (in percentage as the Z-axis variable) of CRHP and CRCP approaches for *equake*.

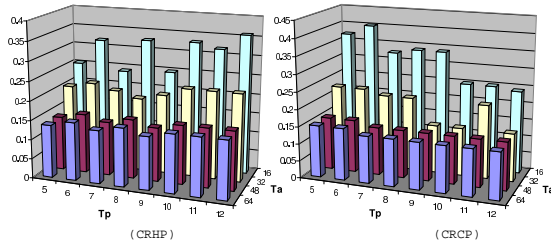


Figure 10: Performance loss (in percentage as the Z-axis variable) of CRHP and CRCP approaches for *art*.

According to these figures, the longer the active waiting time, the better the performance. Further, one can easily see

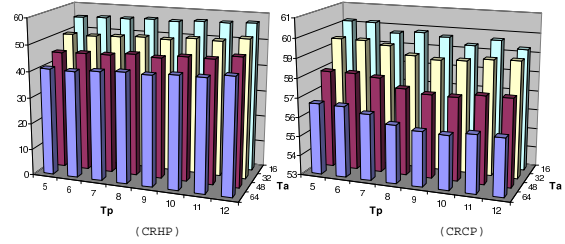


Figure 11: Power reduction (in percentage as the Z-axis variable) of CRHP and CRCP approaches for *art*.

that  $T_a = 16$  can satisfy the bounded performance loss of 1%. Therefore, we will assume  $T_a = 16$  in the rest of this paper.

Moreover, there exists a optimal  $T_p$  for the given active waiting time. In general, a  $T_p$  that is too small or too large is not beneficial for performance. Because a too small or too large  $T_p$  does not ramp FPUs in time and contribute to the performance loss. However, a large  $T_p$  degrades performance less than a small  $T_p$  does. This is due to the fact that the FPU is kept active for the active waiting time and therefore the performance loss by a too early ramping can be compensated.

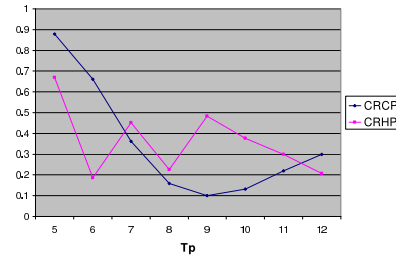


Figure 12: Performance loss (in percentage) for  $T_r = 10$  and  $T_a = 16$

Figure 12 shows the performance for CRCP and CRHP when  $T_a = 16$  for benchmark *equake* (*art* has a similar trend). Clearly, the performance of CRCP is a convex curve with the single local optimal  $T_p = 9$ . However, the performance of CRHP is not a convex curve, and has a few local optimal  $T_p$  values. Therefore in the theoretic sense, an exhaustive enumeration of  $T_p$  is needed to find the best  $T_p$  for CRHP while the best  $T_p$  for CRCP can be easily found as a local optimal value without exhaustive enumeration.

Figure 13 shows the power reduction for CRCP and CRHP when  $T_a = 16$  for benchmark *equake* (again, *art* has a similar trend). With respect to the best  $T_p = 6$  for CRHP and the best  $T_p = 9$  for CRCP, the energy consumed by FPUs can be reduced by 31% and 37% for CRHP and CRCP respectively while the performance loss is negligible 0.2% and

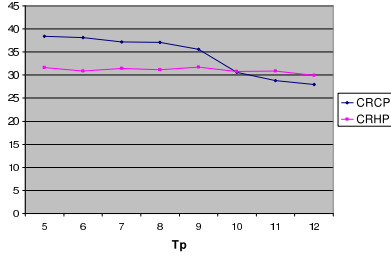


Figure 13: Power reduction (in percentage) for  $T_r = 10$  and  $T_a = 16$

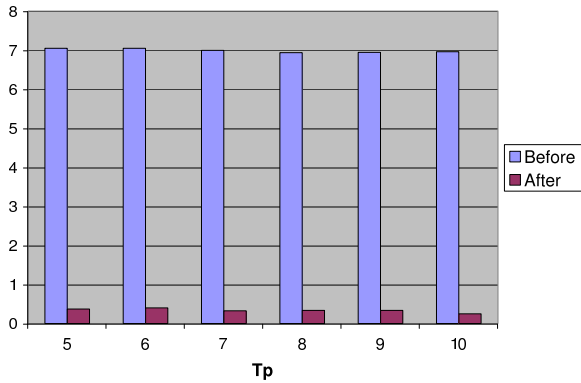


Figure 14: Performance Loss (in percentage) before and after the amendment for load instruction, for  $T_r = 10$ ,  $T_a = 16$  and  $T_p = 9$ .

0.1% for CRHP and CRCP respectively. It is worthwhile to point out that we don't consider the power dissipation and cache miss for hardware prescan. So the actual power and performance by CRHP will be worse than those in Figure 8 and 12. Given that CRCP has a higher performance and uses less energy, the compiler-based CRCP is recommended for VLIW processors.

We have considered our load amendment in Figures 8-11. To appreciate the contribution of this amendment, we show in Figure 14 the performance before and after our amendment for CRCP approach. Benchmark *art* is used as it has a relatively low cache hit rate for load instructions. Surprisingly, this simple amendment can reduce the performance loss from over 6% to less than 1%.

## 5 Conclusions and Discussions

In this paper we first present the PowerImpact, the cycle-accurate power simulation based on IMPACT infrastructure for VLIW processors. We then use PowerImpact to study the following power reduction techniques with a bounded performance loss of 1% compared to the cases without any

dynamic throttling: (i) clock ramping with hardware-based prescan (CRHP), and (ii) clock ramping with compiler-based prediction (CRCP). Experiments using SPEC2000 floating point benchmarks show that the power consumed by floating point units can be reduced by up to 31% and 37%, for CRCP approach and CRHP approach respectively.

An limitation of our work is that IMPACT is designed originally as a C compiler. There is only a few SPEC FP benchmarks written in C, while most SPEC FP benchmarks are written in Fortran. As we know the Fortran front-end for IMPACT is under development and will be available soon. More floating point benchmarks will be tested then.

## References

- [1] D.Brooks, V.Tiwari, and M.Martonosi, "Wattch: A framework for architectural-level power analysis optimization," in *ISCA*, 2000.
- [2] W.Ye, N.Vijaykrishnan, M.Kandemir, and M.J.Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *DAC*, 2000.
- [3] A. Dhodapkar, C. Lim, G. Cai, and W. Daasch, "Tem<sup>2</sup>p<sup>2</sup>est: A thermal enabled multi-model power/performance estimator," in *Workshop on Power-Aware Computer Systems, in conjunction with the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [4] V. Tiwari, D. Singh, S. Rajgopal, and G. Mehta, "Reducing power in high-performance microprocessors," in *DAC*, 1998.
- [5] R. Maro, Y. Bai, and R. Bahar, "Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors," in *Workshop on Power-Aware Computer Systems, in conjunction with the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [6] G. Cai and C. Lim, "Architectural level power/performance optimization and dynamic power estimation," in *Cool Chips Tutorial colocated with MICRO32*, November 1999.
- [7] S. Ghiasi and D. Grunwald, "A comparison of two architectural power models," in *Workshop on Power-Aware Computer Systems, in conjunction with the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [8] P. Chang, S. Mahlke, W. Chen, N. Warter, and W. Hwu, "Impact: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th ISCA*, May 1991.
- [9] D. August, D. Connors, and e. a. S.A. Mahlke, "Integrated predicated and speculative execution in the impact epic architecture," in *Proceedings of the 25th ISCA*, July 1998.

- [10] S.Manne, A.Klauser, and D.Grunwald, "Pipeline gating: Speculation control for energy reduction," in *ISCA*, 1998.
- [11] N.Vijaykrishnan, M.Kandemir, M.J.Irwin, and H.S.Kim, "Energy-driven integrated hardware-software optimization using simplepower," in *ISCA*, 2000.
- [12] E.Musoll, "Predicting the usefulness of a block result: a micro-architectural technique for high-performance low-power processors," in *32nd Annual International Symposium on Microarchitecture*, November 1999.
- [13] M. Pant, P. Pant, D. Wills, and V.Tiwari, "An architectural solution for the inductive noise problem due to clock-gating," in *Proc. Int. Symp. on Low Power Electronics and Design*, pp. 255–257, 1999.
- [14] M. Pant, P. Pant, D. Wills, and V. Tiwari, "Inductive noise reduction at the architectural level," in *International Conference on VLSI Design*, pp. 162–167, 2000.
- [15] Z. Tang, N. Chang, S. Lin, W. Xie, S. Nakagawa, and L. He, "Ramp up/down floating point unit to reduce inductive noise," in *Workshop on Power-Aware Computer Systems, in conjunction with the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.
- [16] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proc. of Micro 25*, pp. 45–54, 1992.