Combining Compiler and Runtime IPC Predictions to Reduce Energy in Next Generation Architectures[*]

Saurabh Chheda
BlueRISC, Inc.
Massachusetts, USA
saurabh@bluerisc.com

Osman Unsal
Intel Research Center
Barcelona, Spain
osmanx.unsal@intel.com

Israel Koren
koren@ecs.umass.edu

C. Mani Krishna
krishna@ecs.umass.edu

Csaba Andras Moritz
moritz@ecs.umass.edu

The Department of Electrical & Computer Engineering
University of Massachusetts, Amherst
Massachusetts, USA

ABSTRACT

Next generation architectures will require innovative solutions to reduce energy consumption. One of the trends we expect is more extensive utilization of compiler information directly targeting energy optimizations. As we show in this paper, static information provides some unique benefits, not available with runtime hardware-based techniques alone. To achieve energy reduction, we use IPC information at various granularities, to adaptively adjust voltage and speed, and to throttle the fetch rate in response to changes in ILP. We evaluate schemes that are based on static IPC, runtime IPC and also combined, hybrid approaches.

We show that IPC-based adaptive voltage scaling schemes can reduce energy consumption significantly, but the approach that also uses static IPC information in combination with runtime IPC, better captures program ILP burstiness and helps meet applications' target performance: an important criterion in the real-time domain. We have found that static IPC-based fetch-throttling works very well, in most cases performing similarly or better than hardware-only runtime IPC-based schemes. Overall, static IPC based resource throttling alone can save up to 14% energy in the processor with less than 5% IPC degradation. The hybrid scheme saves somewhat more energy but at the expense of higher performance degradation than the static-only approach. In fact, we obtain the lowest IPC degradation with the static IPC-based scheme.

[*]This work was supported in part by National Science Foundation under the projects NSF/CCR:015516 and NSF/ITR:0205212. This work was done when Saurabh Chheda and Osman Unsal were at The University of Massachusetts, Amherst. Saurabh Chheda is now with BlueRISC, Inc., Hadley, Massachusetts - 01035, USA. Osman Unsal is now with Intel Barcelona Research Center, Barcelona, Spain.

Categories and Subject Descriptors

C.4.3 [**Computer Systems Organization**]: Computer System ImplementationMicrocomputers[Microprocessors]

General Terms

Instruction Level Parallelism, Fetch Throttling, Adaptive Voltage Scaling

Keywords

Low power design, compiler architecture interaction, instruction level parallelism, fetch throttling, adaptive voltage scaling

1. INTRODUCTION

In the past decade microprocessor performance has witnessed phenomenal advances. The rate of increase in power and energy dissipation has been, however, much higher than that of performance. This increases demands on battery capacity in mobile applications and makes the problem of heat dissipation more challenging. Processors are, in fact, becoming so power-hungry that reducing their power consumption without significantly degrading their performance has become a major focus of microprocessor vendors.

This paper describes a different approach to address this problem by evaluating tightly integrated compiler architecture techniques. Specifically, we address chip-wide power and energy reduction in processors based on instructions per cycle (IPC) information. We use statically estimated IPC information and runtime predicted IPC, to adaptively adjust voltage and speed, and to throttle the fetch rate in response to changes in ILP. Our goal is to evaluate a variety of schemes based on static IPC, runtime IPC, and hybrid solutions. We show that by adding static IPC information we can often provide significant benefits - not available with runtime IPC based techniques alone.

1.1 Instructions per Cycle

IPC is a measure of instruction level parallelism (ILP) in the program. IPC can be predicted at runtime or estimated at compile time. There are fundamental differences between the two approaches.

Predictions of IPC at runtime are generally made by observing the IPC over a certain window and then assuming that this will continue to hold over some future interval. If we have programs whose IPC is stable for long periods of time, this approach works well. On the other hand, many programs exhibit irregular or bursty IPC behaviour at the chosen window size: for these, runtime predictions are often wrong. One can always increase the sensitivity of runtime methods to rapid IPC changes, by shortening the window over which observations and predictions are made. This, however, often comes at a higher overall performance/power cost, especially when IPC is used to guide resource management such as in voltage scaling.

Static IPC is a compile-time estimate of the actual IPC based on program analysis. Static IPC could therefore provide an indication of a sharp change in ILP (or ILP burstiness). A source of inaccuracy in static IPC is due to the compiler's inability to capture dynamic effects, such as branch prediction errors and cache misses. Another difference between static and dynamic IPC is that static IPC is typically estimated for a program phase (e.g., basic block, loop) rather than a fixed number of instructions. This results in a variable and application-dependent window size.

1.2 Contributions of this Paper

As noted, there are apparent advantages and disadvantages to both IPC estimates. While static IPC is a good indicator of ILP burstiness, runtime IPC can be more accurate when there is no change in ILP. When used for resource management, an accurate IPC is essential to avoid exceeding application target performance (by for example slowing down the processor too much). This is especially important in real-time applications.

Our objective is to provide a comprehensive study of using static and dynamic IPC information in the context of energy saving techniques. We explore a variety of schemes including hybrid ones. This paper expands on our initial work on static IPC prediction based fetch throttling [33], that for the first time, evaluated the usefulness of static IPC predictions for energy optimizations.

The IPC based framework in this paper is used in two energy optimization contexts: (1) we adjust voltage, speed, and the granularity of voltage scaling, in response to changes in ILP; and (2) we control the fetch rate in response to changes in ILP. The first optimization provides energy savings by running the processor at a lower speed and voltage when there is slack predicted compared to the target performance or application deadline. Speed and voltage is continuously adjusted as a function of predicted IPC. The second optimization provides energy savings by avoiding unnecessary instruction fetches and unnecessary switching activity, in program phases with low predicted ILP.

Some of the key contributions and insights in this paper include:

- We develop and evaluate a suite of adaptive techniques, for voltage, speed, and resource adjustments, leveraging static and/or dynamic IPC information, as a coherent strategy for chip-wide energy reduction in the processor.
- We develop new compiler techniques, including compiler driven static IPC estimation schemes at various program granularities, from basic blocks to loop and procedure levels. Our static IPC estimation approach is based on monotonic dataflow analysis and simple heuristics, performed at various program granularities in compiler backends.
- We develop a static IPC prediction scheme and a hybrid IPC-based scheme, and compare them with known runtime schemes. We use the static IPC prediction scheme to drive our fine-grained fetch-throttling energy-saving heuristic. We have experimented with a variety of architectural configurations using multimedia and Spec 2000 benchmarks. We obtain up to 14% total energy savings in the processor with generally little performance degradation. Moreover, this scheme is the one that has the smallest IPC degradation of all the studied schemes. In general, we have found that static IPC-basedfetch-throttling works very well if preserving performance is important. We investigate whether dynamic factors such as cache misses, branch prediction and pipeline depth would dilute the efficiency of our static IPC-prediction-based heuristic. We find the efficiency variation to be small.
- As expected, IPC-based voltage scaling is an efficient way to adjust energy to match target performance in case there is a slack, i.e., when the application does not need all of the performance provided by the superscalar processor. We have found that the various static and/or dynamic schemes are comparable in energy savings, but that the target performance is often not met with hardware-only schemes for bursty applications. By contrast, the voltage scaling scheme that uses static information about program burstiness has been successful in preserving applications' target performance.

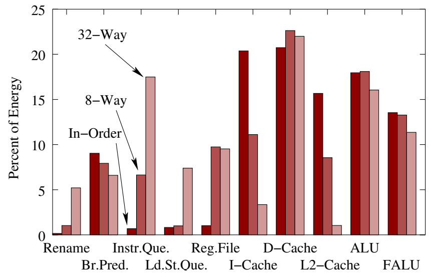
1.3 Organization of Paper

The rest of this paper is organized as follows. Section 3 discusses compiler and architectural implementation issues related to our Static-IPC prediction scheme and the experimental setup. The energy saving heuristic for the IPC-based Adaptive Voltage Scaling and IPC-based Fetch Throttling is explained in Section 4. Our experimental results are presented in Section 5. In Section 6, we discuss related work and comment on its relevance to this paper.

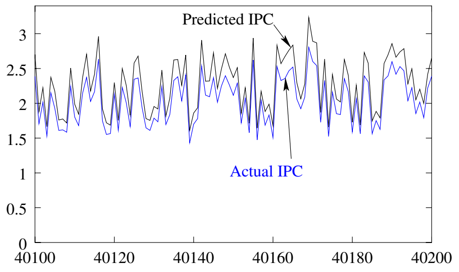
2. MOTIVATION

To determine which processor blocks are going to be major power drains and thereby choose which sections of the processor to apply our energy saving methods to, we conducted a preliminary study. We analyzed the percentage of energy contribution of different blocks for three architectural configurations. See Figure 1a. Following [6], we assume that clock power consumes a constant ratio of the power across the components of the chip. The results show the average for 8 multimedia applications from the Mediabench suite.

The details of the benchmarks are explained in Section 5.2. We scale every resource accordingly; the first configuration is a simple single-issue in-order machine, the second is an 8-way out-of-order configuration and the third is a 32-way machine. The last configuration, while impractical, gives an idea of the power distribution if one were to have essentially unlimited resources. We include this as an asymptotic case. Note that the fetch- and issue-related logic, the L1 data cache and the ALUs become dominant as the complexity of the architecture is increased. These results agree with the findings in Zyuban and Kogge's study [35]. In Figure 1b., we present a snapshot from the execution profile of the Spec 2000 application *equake*. The graph shows the actual IPC against our compiler-driven static IPC prediction as averaged over windows of 10000 cycles. Since predicted IPC provides a reasonably accurate estimate of actual IPC, we are motivated to use the static prediction for energy savings by throttling resources when they are not needed.



(a) Percentwise energy consumption of major processor blocks



(b) Predicted versus actual IPC for the *equake* Spec 2000 application

Figure 1: Where and how to save energy

3. STATIC IPC-ESTIMATION

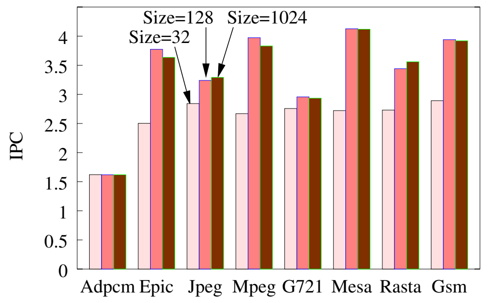
In our implementation, we only consider true data dependencies (Read-After-Write or RAW) to check if instructions depend on each other. As mentioned in [27], a major limitation of increasing ILP is the presence of true data dependencies. Tune et al. [32] also remark that the bottleneck for many workloads on current processors is true dependencies in the code. Although the impact of true dependencies can be mitigated through the use of value speculation, the energy overhead of value speculation hardware has been found to be prohibitively high. Therefore, we consider a standard, non-value-speculating, out-of-order, architecture in our experiments. For this architectural configuration, antidependencies (Write-After-Read or WAR) or output dependencies (Write-After-Write or WAW) could be eliminated by register renaming, but even infinite resources cannot eliminate true dependencies. However, note that the compiler-driven fetch throttling framework is equally applicable to an architecture with value speculation: only the compiler-level passes need to be replaced.

It is also possible to handle false dependencies in the compiler passes: this would be a viable option if the processor were severely constrained in its register renaming resources. However, contemporary processors usually have enough resources to eliminate most false dependencies. Another possible use for compiler-driven ILP estimation could be the static analysis and determination of the Functional Unit (FU) needs of the application. A back-end energy-saving heuristic would then dynamically turn off unnecessary FUs (such as ALUs) during statically predicted periods of low-FU usage. Of course, there are other, dynamic, factors that influence IPC, such as branch prediction and cache misses. Surprisingly, in our experiments, we found that the impact of those dynamic components on the efficiency of our static-only approach is actually smaller than we expected.

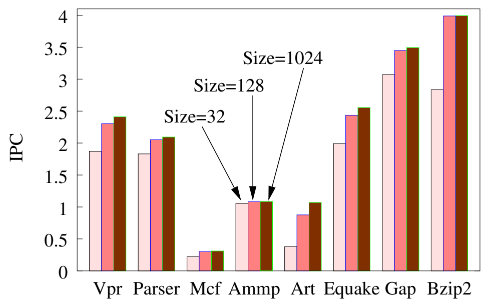
Another issue that needs to be discussed is the impact of the Out-of-Order architecture on loop-level parallelism. Intuitively, if the instruction window is large enough, intructions across loop iterations could be scheduled out-of-order creating an effect that is similar to software pipelining, which is not yet captured in our compilation framwork. However this is not the case: see Figure 2. Here, for Mediabench and Spec200 applications (see Section 5.2), a very large instruction window of size 1K does not influence the IPC. We next discuss the compiler and architectural level issues related to static IPC-estimation.

3.1 Compiler-Level Implementation

We statically determine true data dependencies using an assembly-code level data dependency analysis. The advantage of doing the analysis at this level instead of at the source code level is that the instruction level parallelism is fully exposed at the assembly code layer. Our post-register allocation scheme uses monotone data flow analysis, similar to [3]. However, our scheme has two important distinctions: first, we use monotone data flow analysis to identify the data dependencies, not for instruction scheduling. Second, our method is speculative, whereas [3] requires complete correctness. We identify data dependencies at both registers and memory accesses. Register analysis is straightforward: the read and written registers in an instruction can be established easily, since registers do not have aliases. The determination of reaching uses is achieved using the well-known algorithm in [2]. However, for memory accesses, this is not the case and there are three implementation choices: no alias analysis, complete alias analysis, or alias analysis by instruction inspection. *No alias analysis* is too speculative for IPC estimation: it assumes that a memory load instruction is always dependent on a preceding store instruction. This model would apply if there were no load/store queues or multiple memory ports in the processor but modern out-of-order architectures are typically equipped with those resources. Another alternative is doing *full alias anal-*



(a) Mediabench Applications



(b) Spec2000 Applications

Figure 2: The impact of instruction window size on IPC. The processor resources have been chosen large enough to highlight the impact of changing instruction window size. We assume an 8-way issue with 10 FU's, 128K L1 D and I-Caches,64K bimodal + 64K Gshare with 64K selector hybrid branch predictor.

ysis, although it requires considerable overhead to implement, this option would ensure full correctness. Still, we have found that our approximate and speculative *alias analysis by instruction inspection* provides ease of implementation and sufficient accuracy. In this scheme, we distinguish between different classes of memory accesses such as static or global memory, stack and heap. We also consider indexed accesses by analyzing the base register and offset values to determine if different memory accesses are referenced. If this is the case, we do not consider this pair of read-after-write memory accesses as true dependencies. We follow with a more detailed description of the implementation.

We use SUIF/Machsuif as our compiler framework. SUIF does high-level passes while Machsuif does machine-specific optimizations. Our IPC-estimation is at the basic block or loop level.

3.2 Architectural-Level Implementation

We exploit the statically-estimated IPC for fetch-throttling and voltage frequency scaling. We present a suite of IPC-driven schemes for fetch throttling and voltage scaling that use statically-estimated (i.e. compile-time) IPC and/or dynamic (i.e. runtime) IPC. Central to all schemes is the use of compile-time estimated IPC, either alone or in conjunction with other runtime metrics, to drive our energy saving heuristics.

When code is generated for static or hybrid fetch throttling, an assembler level pass examines the estimated IPC to assess the need for fetch throttling. If the estimated IPC is below a threshold, established at compile-time, a *throttling flag* is inserted.

In a similar fashion, when code is generated for static or hybrid voltage scaling, an assembler level pass examines the variation in estimated IPC over basic blocks to assess the need for voltage change and also estimates the degree of voltage scaling. For example, if the variation in estimated IPC from one basic block to the next exceeds a threshold (also fixed at compile-time), a *voltage scaling flag* is inserted. The degree of change in estimated IPC over basic blocks is used to decide the degree of voltage scaling (i.e. the amount by which voltage is raised/lowered). The granularity at which IPC is estimated can be changed, from basic block level to loop level, or even higher.

In the static or hybrid voltage scaling scheme, the number of bits in the voltage scaling flag depend on the number of voltage levels available to the processor core. If the core voltage of a processor can switch between four levels, then the voltage scaling flag requires two bits to encode the desired voltage level. For a hybrid voltage scaling scheme, a single bit is enough to indicate if the core voltage needs to be raised or lowered. Note that the fetch throttling flag requires only a single bit. If enough flexibility exists in the ISA of the target processor, then the flag can be inserted directly into the instructions eliminating the need for a special instruction. In our experiments, we take this approach and also consider the additional power dissipation stemming from this extension: see Section 5.1. If there is not enough flexibility in the ISA, then special flag instructions should be added. This may raise the question of increased code size due to the additional instructions. The voltage scaling related instructions are typically at a coarser granularity so they don't affect performance significantly.

We have analyzed the worst-case code size increase due to our approach: assuming that every IPC-estimation marker results in a throttling hint. This is unrealistic but gives an upper bound. Figure 3 shows the percentage increase for the Mediabench and Spec 2000 applications. The average code size increase is modest at 5.1%.

4. ENERGY-SAVING FRAMEWORK

4.1 IPC-based Fetch Throttling

The fetch-throttling scheme latches the compiler-supplied predicted IPC at the decode stage. If the predicted IPC is below a certain *threshold*, then the fetch stage is throttled, i.e., new instruction fetch is stopped for a specified *duration* of cycles. The rationale is that frequent true data dependencies which are at the core of our IPC-prediction scheme, will cause the issue to stall. Therefore, the fetch could be throttled to relieve the I-cache and fetch/issue queues and thereby save power without paying a high performance penalty. We have done extensive experiments to determine the threshold value and the duration. The results suggested that a threshold of 2 and duration of 1 is the best choice. That is, we stop instruction fetch for 1 cycle when we encounter an IPC prediction that is at most 2. This heuristic is similar to the front-end throttling scheme by Baniasadi et al. [7]. Besides

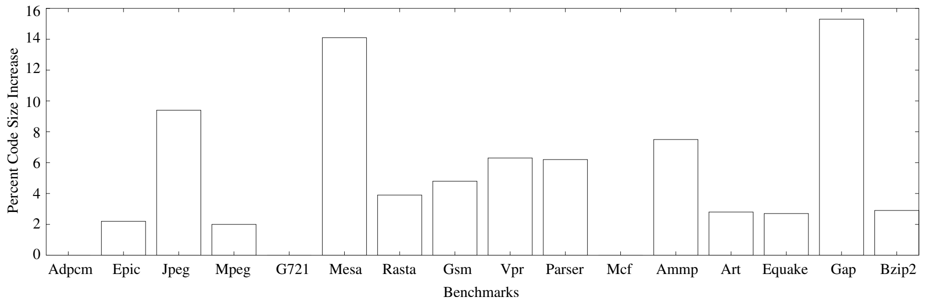


Figure 3: Percent code size increase due to ISA augmentation with IPC-marker instructions. See Section 5.2 for details of the particular benchmarks.

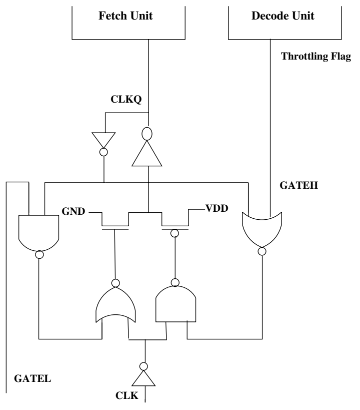


Figure 4: Architectural implementation of front-end throttling. GATEH is asserted when there is a throttling flag.

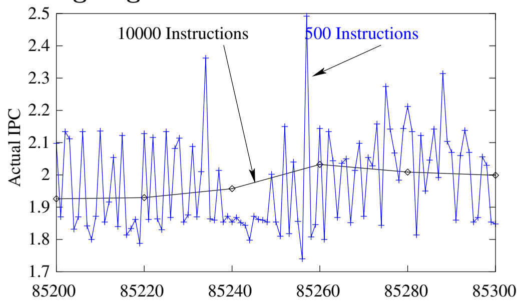


Figure 5: Coarser versus finer granularity.

using a compiler-directed approach; our scheme uses static 'future-information' to predict future behavior, whereas [7] uses the past-behavior to predict future behavior. We include the architectural implementation of our energy saving heuristic in Figure 4. Here, when the predicted IPC is 1 or 2, GATEL is asserted and the fetch stage is throttled by using a clock gater. To prevent glitches, a low-setup clock gater is used which allows the qualifier to be asserted up to 400ps after the rising clock edge without producing a pulse [20].

We preferred a fine-grained heuristic over a coarse-grained one. Coarse-grained heuristics usually average available ILP-information over a large number of cycles, which can lead to loss of accuracy. Consider Figure 5, where a slice of the Epic multimedia benchmark is shown. The curves show the actual IPC as averaged over 10000 (coarser granularity) and 500 (comparatively finer granularity) cycles, respectively. It is evident that a coarser granularity scheme would be less accurate than one using a comparatively finer granularity scheme. However, we should note that our compiler-layer IPC prediction framework would work equally well with a coarse granularity scheme as well.

In addition to the static IPC-driven scheme for fetch throttling, we have developed one hybrid scheme for fetch throttling that combines the static-only scheme with runtime, front-end fetch-throttling schemes by Baniasadi et al. Our energy saving heuristic in the hybrid schemes uses the statically inserted fetch-throttling flags in conjuction with the runtime mechanisms proposed by Baniasadi et al. to throttle the fetch unit.

4.2 IPC-based Adaptive Voltage Scaling

Modern electronic systems run workloads whose performance demands typically vary and cannot be predicted in advance. For such systems, voltage scaling is ideal since we can change voltage (and hence frequency) dynamically to meet performance goals. As energy is decreased quadratically by reduced voltage, this is an efficient way to improve energy efficiency.

Changing voltage with variation in IPC is an attractive solution to reduce the energy consumption without sacrificing the performance goals. An IPC-driven voltage adaptation requires, in order to maintain goal performance, to continuously adjust supply voltage and speed in response to the IPC variation presented by the application. When the amount of IPC seen for a code fragment is predicted to be lower than the average IPC required to meet the performance target, we could raise the supply voltage to make up for the limited IPC. In code fragments with higher IPC, the supply voltage can be similarly reduced such that the processor can run slower, save energy, and still achieve its target performance.

Given a performance goal, such as a specified MIPS rate (or an average application IPC rate at a given voltage and frequency), we propose a suite of voltage scaling heuristics that use static and runtime IPC to make voltage scaling decisions. At the end of a selected window, voltage and speed adjustments are made depending on the IPC and target performance.

A hardware-only, runtime IPC based dynamic voltage scaling system calculates the observed IPC over the previous window as a MIPS rate, and calculates a new frequency (and hence, voltage level) for the next window to meet the target MIPS rate.

$$f_{new} = f_{old} \times \frac{MIPS_{goal}}{MIPS_{observed}} \quad (1)$$

If f_{new} is high or low enough, the processor selects a new appropriate voltage level (e.g., from the voltage points available in the processor) that achieves a larger or equal frequency as the desired frequency f_{new}.

The hardware-only, dynamic voltage scaling scheme is based on constant-size intervals. At every interval, control is transferred to a software routine that computes the runtime IPC of the last interval and then makes a decision to scale voltage for the current interval. This incurs a performance penalty on every interval due to transfer of control to the software routine, irrespective of whether voltage is changed or not. A fixed interval-based scheme mispredicts the voltage on a sudden variation in ILP since its decision is based on past IPC information. In this paper, we also use static IPC to complement the runtime predicted IPC as a measure of the actual program IPC.

When the voltage scaling heuristic employs static IPC, in addition to runtime IPC, this ILP misprediction can be avoided. *Static IPC, although possibly an optimistic estimate, gives a good indication of the relative variation of IPC in a current interval compared to a previous one.* Thus, a voltage scaling scheme that also employs static IPC is expected to be better suited to handle bursts in IPC (and thus bursty applications).

In our solution, we augment the hardware-only IPC scheme with *statically-obtained voltage scaling hints.* At every compiler-determined program interval (such as loop boundaries), special hardware compares the current and previous compiler-introduced voltage scaling flags during runtime. Any difference between these flags is an indication of change in IPC (predicted by the compiler) and hence a hint that a voltage scaling decision needs to be made. Clearly, if the voltage scaling decision were made based on the runtime IPC of the previous window (such as in a hardware-only solution), it would often result in an incorrect change in voltage. In case there is an IPC burst predicted by the compiler, the static IPC estimation is used to make the decision (the first time) rather than the runtime IPC of the previous window. When there is no burstiness expected, our compiler-enabled scheme uses the actual runtime IPC of the previous window. Note that the window size is program variable and not fixed (such as in the scheme based on runtime IPC alone).

5. EXPERIMENTS

5.1 Architectural Simulator Setup

The system model is a typical out-of-order superscalar processor. See Figure 6 for the modeled processor blocks. Note that the baseline includes the shaded throttling logic block; which we explained in Section 3.2.

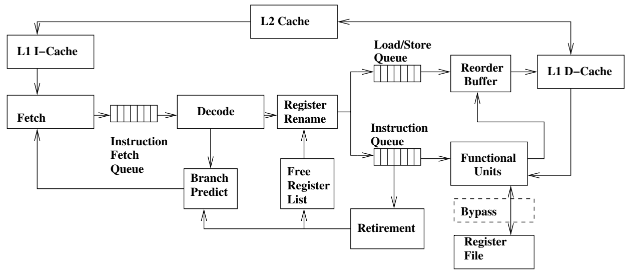


Figure 6: A superscalar out-of-order core.

The baseline architecture reflects the state-of-the-art in current processor designs. Table 1 contains a description of the baseline parameters. The trend is towards wider issue as evidenced by the proposed 8-way Alpha 21464, and the recently introduced dual 4-way issue processors such as the POWER4 from IBM [4] and MAJC 5200 from SUN [21]. Henry et al. [16] propose novel circuits that scale to 8-way issue; they also present results for a 128-entry issue/reorder buffer. An actual implementation of a large instruction queue is the 1.8GHz 64-entry instruction window buffer by Leenstra et al. [23]. Based on the preceding analysis, we selected an 8-wide issue, 128 entry instruction queue as our baseline.

We use Wattch [8] to run the binaries and collect the energy results. Wattch is based on the Simplescalar [9] framework. Our baseline processor configuration has 128 entries in its instruction queue, therefore we use a 128 element RUU (Register Update Unit). RUU is a simple and elegant scheme [31], adapted by Simplescalar, to facilitate out-of-order superscalar execution. The RUU includes the instruction queue as well as the physical register files and the reorder buffer. We use a size of 64 for the Load-Store Queue (LSQ). In Wattch, we use the activity-sensitive power model with aggressive conditional clocking. We use a 0.18 μm, 1.0Ghz, 1.75V process. We extended the power dissipation model in Wattch so that it accounts for the extra power overhead due to the 1-bit throttling flag field decoding in the dispatch stage. The static IPC-based voltage scaling is done with inserted special instructions at a coarser granularity (e.g., loop nest level).

Processor Speed	1.5GHz
Process Parameters	0.18 μm, 1.75V
Issue	Out-of-order
Fetch, Issue, Decode	
Commit Width	8-way
Fetch Queue Size	32
Instruction Queue Size	128
Branch Prediction	2K entry bimodal
Int. Functional Units	4 ALUs, 1Mult./Div.
FP Functional Units	4 ALUs, 1Mult./Div.
L1 D-cache	128Kb, 4-way
L1 I-cache	128Kb, 4-way
Combined L2 cache	1Mb, 4-way
L2 cache hit time	20 cycles
Main memory hit time	100 cycles

Table 1: Baseline Parameters.

To extract the maximum available ILP and therefore get higher IPC, some contemporary wide-issue processor designs such as the AMD AthlonXP [19] use short pipelines; we take a similar approach and use the default 5-stage pipeline structure in our architectural simulator (fetch, dispatch or decode, issue, writeback, commit) as the baseline. However, other recent competing processors use deeper pipelines to achieve higher clock rates at the expense of IPC. Examples of these are the 20-stage Intel Pentium 4 [17] and the 12-stage AMD Hammer [34]. Therefore, we also model and analyze the impact of a deeper 11-stage pipeline (2 fetch, 4 decode, 2 issue, 2 writeback, 1 commit stages) in our sensitivity analysis. The Simplescalar pipeline stages are extended from 5 to 11 and a branch penalty of 10 cycles is assumed for this analysis. We also extended the Wattch power models to account for the additional pipeline stages.

For our voltage and frequency scaling experiments, we retuned Wattch to support mulitple levels of core voltages and corresponding frequencies. Some current processors support dynamic voltage scaling, including Transmeta's TM 5400, Intel's XScale, and AMD's K6-IIIE+. We have adapted Wattch to support five levels of voltage (and corresponding frequency), as in Intel's XScale 2.

Frequency (MHz)	Core Voltage (V)
200	0.7
466	1.0
600	1.2
800	1.4
1,000	1.75

Table 2: Core Voltage and Frequency of Operation for Intel XScale

Though it is possible to dynamically change voltage and clock speed, current systems suffer from a high latency penalty for changing voltage. For every change in voltage, the processor must drain the instruction pipeline, and request a change in voltage. During the period when voltage is changing and stabilizing, the processor is in the idle state. This latency can be as high as 75-150μs (for the AMD K6-IIIE+), and is dependent on the type of voltage regulation used. Using an external DC-DC regulator to change voltage is common and comprises of bulk of the latency penalty. Newer DC-DC regulators/converters can switch between voltages in less than 5μs, thereby bringing down the latency penalty for switching voltage. Another viable alternative is to provide multiple supply voltages on-chip to choose from. Multiple supply voltages already exist on many CPU's, that reduces the latency penalty significantly.

5.2 Benchmarks

We use the Mediabench[22] and Spec CPU2000[1] benchmarks in our experiments. We randomly select eight applications from each suite: see Table 3.

5.3 IPC Based Fetch-Throttling Results

We first present our results for the baseline parameters. See Figure 7a for the Spec 2000 applications. For the Spec 2000 benchmarks in this architectural configuration, compiler IPC-estimation driven front-end throttling yields excellent results: on the average, we get 8% processor energy savings with a performance degradation of 1.4%. As shown in Figure 7b, for the Mediabench applications we get 11.3% average energy savings, however this comes at the price of an average 4.9% performance degradation. This is due to the fact that multimedia programs have typically a higher ILP than general purpose applications such as Spec 2000: although the low IPC estimated instructions are stalled at the issue queue, later and higher IPC instructions could have all their operands available and issued out-of-order if there is sufficient ILP available. This implies that for this configuration running media benchmarks, a coarser-granularity scheme or a hybrid static/dynamic heuristic could yield better results.

To present how fetch-throttling saves resources, we include results on the percentage decrease of the fetch and instruction queue occupancy: See Figure 8. Notice that the front-end throttling scheme decreases the average queue occupancy of the back-end issue queue as well. For Mediabench and Spec, the time that the queues are full is decreased by 28.6% and 14.7% for fetch; and 17.2% and 7.7% for issue, respectively. The average queue size is decreased by 19.2% and 10.4% for fetch; and 4.1% and 2.0% for issue, respectively.

We now examine the percentage of energy savings per processor block: see Figure 9. As expected, the block with the highest overall savings is the fetch stage. However, note that even the issue stage benefits from fetch-throttling.

5.3.1 Comparison With Dynamic-Only Architectural Schemes

We now compare the static fetch-throttling scheme to two previously proposed microarchitectural-level front-end throttling schemes: Decode/Commit Rate (DCR) and Dependence Based (DEP) heuristics by Baniasadi et al. [7]. Both DCR and DEP are also fine-grained schemes; however they solely rely on dynamic information. DCR throttles fetch when the number of instructions passing through decode exceeds significantly the number of instructions that commit. As such DCR exposes a purely dynamic property by inhibiting fetch during branch mispredictions. DEP analyzes the decoded instructions every cycle and throttles fetch if the number of dependencies exceeds a threshold of half the decode width. Similar to the static fetch-throttling scheme, DEP is dependency-based; however DEP makes use of run-time information while the static fetch-throttling scheme utilizes only compile-time information. We implemented DCR and DEP following the guidelines in [7]. The performance results are given in Figure 10. By contrast with DCR and DEP, our scheme substantially preserves the original performance of the applications. The energy results in Figure 11 indicate that on the average, the static fetch-throttling scheme is as energy-efficient as DCR. However, for some applications such as the ADPCM, DCR saves more energy. Note that this energy savings comes at the expense of performance, i.e., DCR trades off performance for energy. Compared to the static fetch-throttling scheme, DEP saves more energy however trades off performance. This becomes apparent when we compare the energy-delay signatures of the static fetch-throttling scheme with DCR and DEP in Figure 12; The static fetch-throttling scheme is substantially more energy-delay efficient, especially so for media applications.

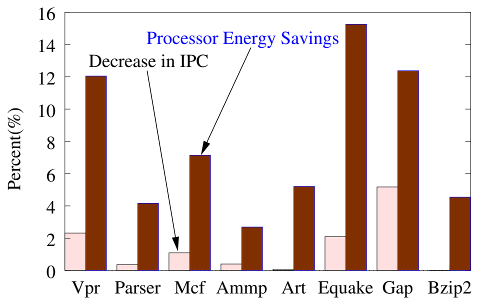
5.3.2 Sensitivity Analysis for Fetch-Throttling

In this section, we examine the impact of resource and control dependencies on Cool-fetch. We start with resource dependencies and analyze the effects of cache misses. Then, we experiment with a smaller instruction queue size. Finally, we test the impact of using a larger branch predictor and also present an extended pipeline experiment which is essentially a test of control dependencies since it amplifies branch misprediction penalties. We now describe the results of each experiment in turn.

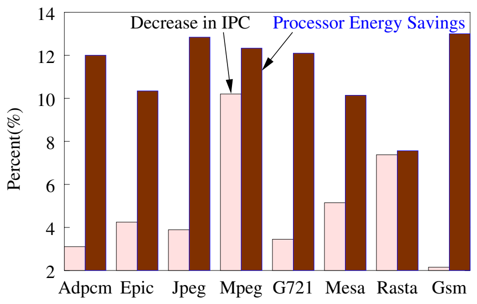
One would expect that since our energy-saving heuristic depends on a static approach, dynamic program behavior such as cache misses would dilute the efficiency of our method. Somewhat surprisingly, this is not the case. In Table 4, we present the data cache miss rates for the Spec 2000 benchmarks. The results are in agreement with data gathered from a recent Spec 2000 cache performance analysis [11]. Consider the very high miss rates for the MCF,

Benchmark	Description
ADPCM	Adaptive differential pulse code modification audio coding
EPIC	Image compression coder based on wavelet decomposition
G721	Voice compression coder based on G.711, G.721 and G.723 standards
GSM	Rate speech transcoding coder based on the European GSM standard
JPEG	A lossy image compression decoder
MESA	OpenGL graphics clone: using Mipmap quadrilateral texture mapping
MPEG	Lossy motion video compression decoder
RASTA	Speech recognition front-end processing
BZIP2	Compression
GAP	Group theory, interpreter
MCF	Combinatorial optimization, single-depot vehicle scheduling
PARSER	Word processing, synthetic parser of English
VPR	CAD FPGA circuit placement and routing
AMMP	Computational chemistry
ART	Neural network for object recognition in a thermal image
EQUAKE	Simulation of seismic wave propagation in large valleys

Table 3: Mediabench and Spec CPU2000 benchmarks used.



(a) Spec 2000



(b) Mediabench

Figure 7: Impact of compiler IPC-estimation driven fetch throttling

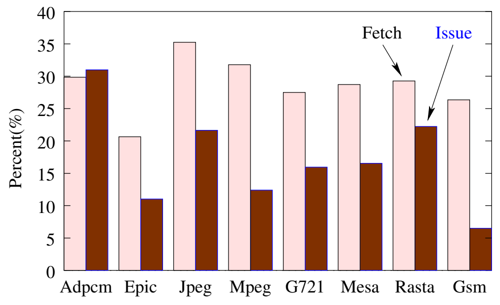
AMMP and ART. This suggests that extraction of available ILP is affected by dynamic memory performance in those benchmarks. Yet, as seen from Figure 7b, the performance degradation due to our scheme for those applications is not worse compared to other, lower miss-rate, applications.

We now present the results for more constrained resources. In Figure 13, the fetch and instruction queues are 8 and 32 instructions, respectively. For the Spec 2000 benchmarks, we again get excellent results: 6.13% energy savings with 0.37% performance penalty. For the Ammp and Bzip2 applications, we even have a slight performance gain with our compiler-directed throttling heuristic. By fetch-throttling at times of low-ILP, the branch prediction can be more effective. Indeed, for those applications the ratio of committed to fetched instructions is higher for the throttled configuration. This in turn leads to slightly increased performance. For the multimedia applications, we achieve good results for this configuration: 8.5% average energy savings with a 1.3% performance penalty. To check the narrow-issue case, we also replicated our experiments for a 4-way issue configuration, the results are similar and not included here for the sake of brevity.

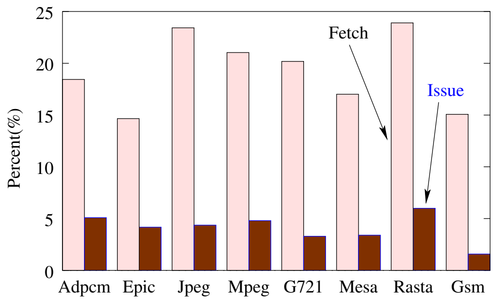
For branch mispredictions, we experimented with a larger and better hybrid branch predictor (64K bimodal + 64K Gshare with 64K selector). We use the Spec applications for this experiment since the branch prediction rates of Spec applications are typically lower compared with the loop-dominated media applications and thus can benefit more from a larger branch predictor. Compared with the unthrottled case with the same branch predictor configuration, the 2K bimodal predictor results in 1.4% average performance degradation and 8% energy savings, while the hybrid predictor has 1% performance degradation and 7.5% energy savings.

As discussed before, we analyzed the impact of increasing the pipeline depth to 11. The results are shown in Figure 14. The deeper pipeline allows an exploration for different *threshold* and *duration* parameters. Figure 14a and 14b show the case with a throttling threshold of 2 and duration of 1 cycles. Figure 14c and 14d are for a threshold of 2 and an expanded throttling duration of 2 cycles. Figure 14e and 14f show the impact of using a throttling duration of 1 cycle, but a threshold value of 3. There are interesting tradeoffs here. The 1 cycle throttling duration case gives the least performance degradation but with modest energy savings. The 2 cycle duration case has the highest energy savings, however the performance penalty is larger, especially for the media applications. The threshold of 3 and delay of 1 cycle gives good energy savings results with a small drop in performance, clearly this case is the optimum among the three policies studied. The throttling duration of 2 cycles is long for wide-issue architectures, and requires substantial changes to the throttling logic. However, using the higher threshold of 3 with a duration of 1 cycle requires minimal change to throttling logic and is a better match for a deeper pipeline.

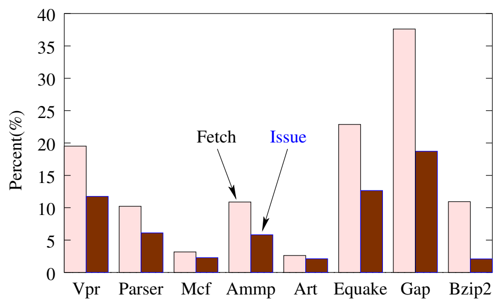
Media applications save more energy through DCR and DEP, while the static-only Fetch Throttling scheme suffers



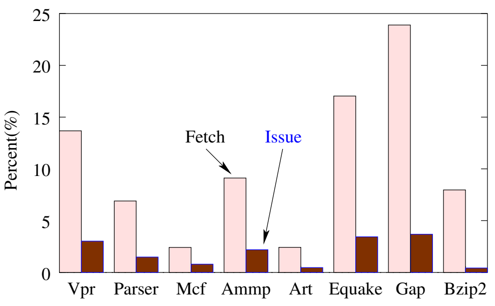
(a) Mediabench: Decrease in queues being full



(b) Mediabench: Decrease in average queue size

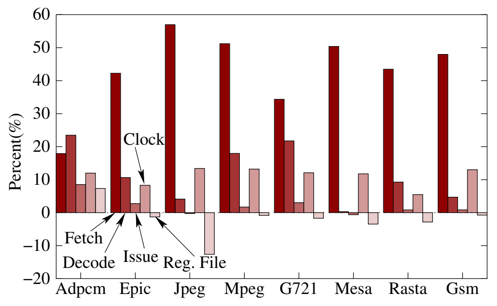


(c) Spec 2000: Decrease in queues being full

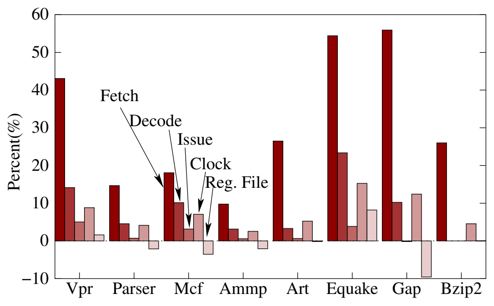


(d) Spec 2000: Decrease in average queue size

Figure 8: Percentage Decrease in Fetch and Issue queue sizes.



(a) Mediabench



(b) Spec 2000

Figure 9: Percentage Energy Reduction in Processor Blocks

less performance penalty. For those applications, therefore, a combined microarchitectural-compiler heuristic offers to be a promising approach. We developed such a heuristic that throttles fetch when the statically-estimated IPC is low, however, additional fetch throttling is applied if the statically-estimated IPC is high but DCR detects high decode-to-commit rate. As seen in Figure 15, this scheme achieves highherer energy savings than static-only fetch throttling scheme with less performance degradation than DCR.

5.4 Voltage Scaling Results

In all the voltage scaling schemes evaluated in this paper, we save energy by constantly adjusting the processor core voltage in response to changes in IPC, while still maintaining target performance. We can save energy when the voltage is reduced since any decrease in voltage gives quadratic savings in energy. As mentioned earlier in Section 4.2, to meet target performance, at each (fixed or compiler determined) window boundary we adjust the frequency and voltage of the processor such that the estimated MIPS of the subsequent window is made equal or slightly larger than the target MIPS.

We first evaluate the contraints that a performance goal places on the energy savings possible with any adaptive voltage scaling scheme. Assuming that we have perfect knowledge of the runtime IPC at the commit stage (i.e., the true runtime IPC at every cycle is known beforehand), we evaluate if we can obtain any energy savings with adaptive voltage scaling when the performance goal is close to the maximum

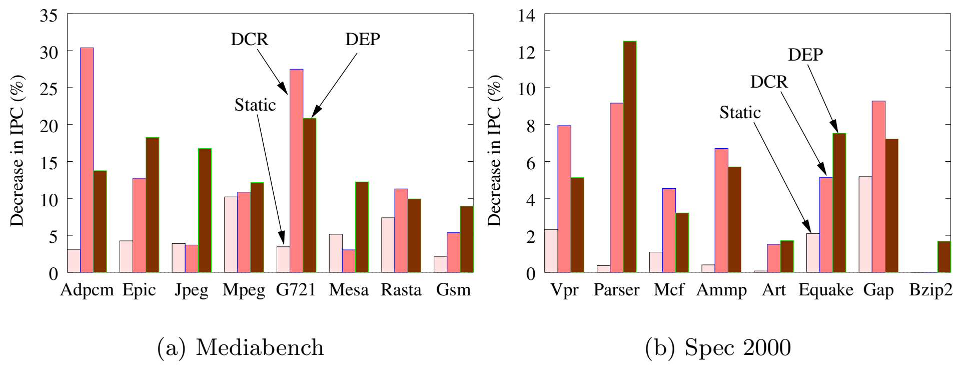


Figure 10: Performance of Static Fetch-Throttling versus DCR and DEP.

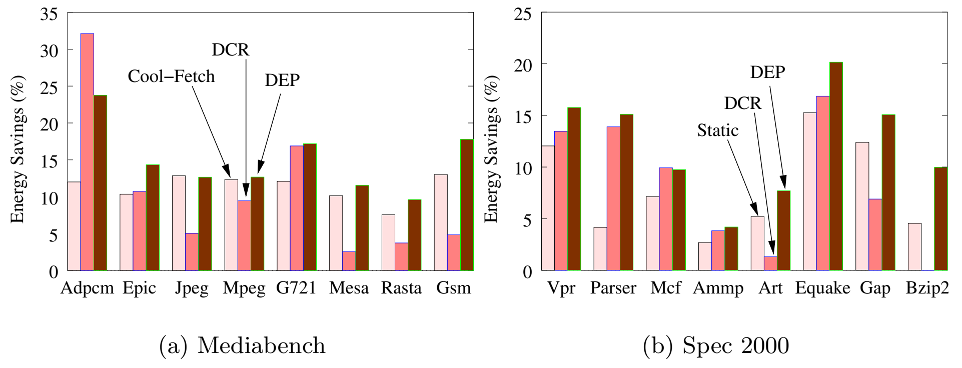


Figure 11: Energy Efficiency of Static Fetch-Throttling (Cool-Fetch) versus DCR and DEP.

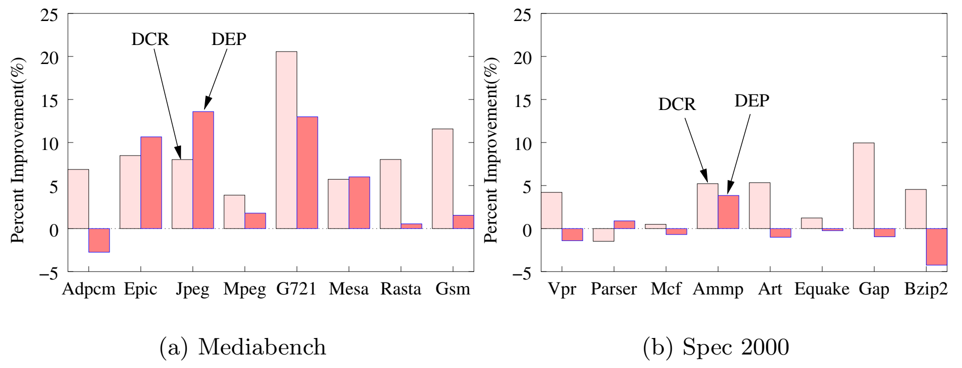


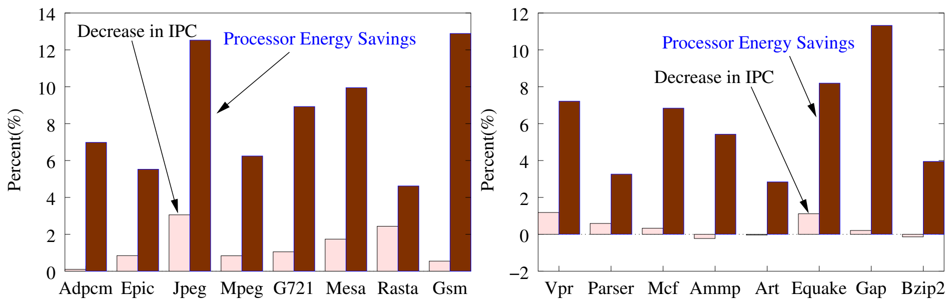
Figure 12: Energy-Delay Efficiency of Cool-Fetch versus DCR and DEP.

performance available (i.e., performance extracted by the processor when operating at fixed highest voltage). This is shown in Figure 16, where we compare the energy and delay of a fixed-voltage scheme at 1.2V, 600MHz with a voltage scaling scheme based on true runtime IPC, and a hybrid, compiler-enabled scheme that uses static IPC obtained at the basic-block granularity. The goal performance is close to the performance at the fixed voltage (i.e. 1.2V, 600MHz). For this experiment we assume ideal clock gating with no voltage scaling cost.

We can conclude from Figure 16, that if the performance goal is close to the performance when the maximum performance available, even a voltage scaling scheme with perfect knowledge of the runtime IPC at the commit stage does not save energy. This can be attributed to two factors: first, if the goal performance is close to maximum available performance, the processor is running at the highest possible voltage for the bulk of the execution time and hence getting energy savings is more difficult, and secondly, the different pipeline stages have resource utilization that is different from the runtime IPC measured at the commit stage which can result in inaccurate predictions that offset the benefits that are obtained by lowering the voltage at high IPC levels. This shows that we cannot expect to get significant energy savings if there is no (or little) slack. Slack is defined as the difference between the desired execution time and the execution time at baseline, fixed, highest voltage and clock rate.

Benchmark	Rate	Benchmark	Rate	Benchmark	Rate	Benchmark	Rate
VPR	1.1	PARSER	1.5	MCF	29.2	AMMP	14.3
ART	16.8	EQUAKE	1.1	GAP	0.3	BZIP2	2.0

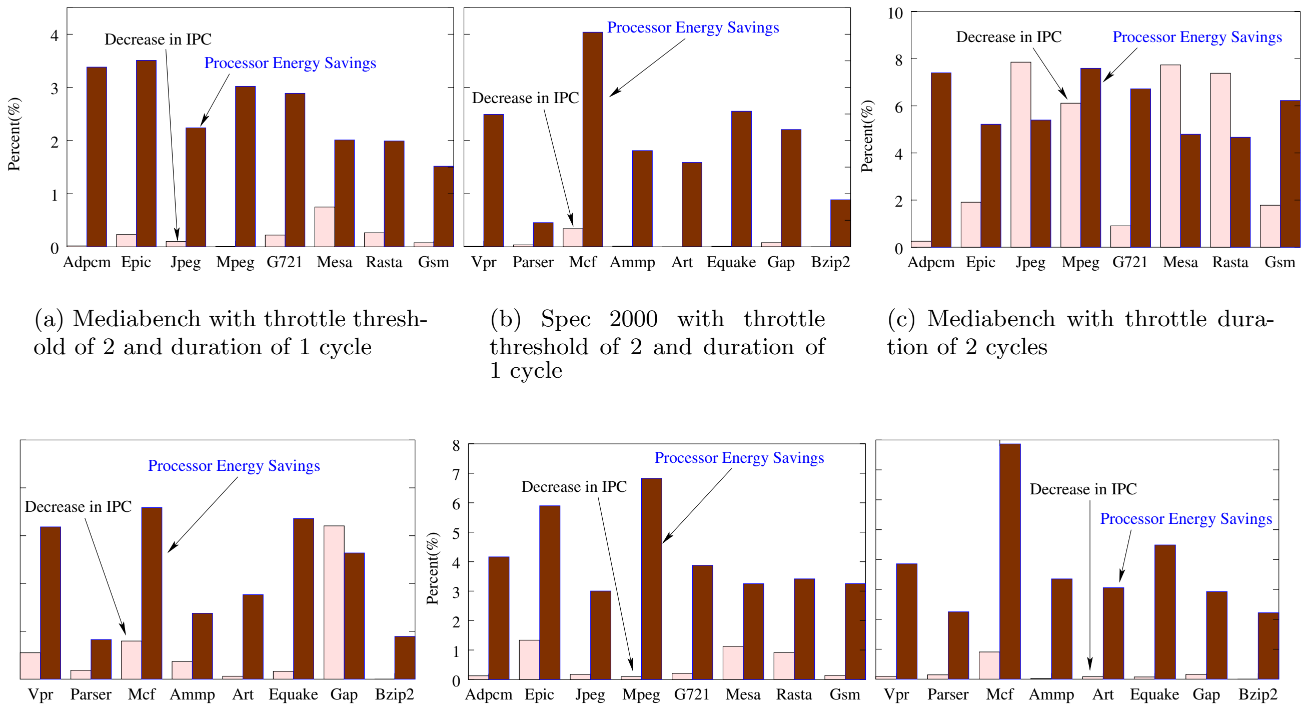
Table 4: Miss rates for the baseline L1 data-cache (128K, 4 way)



(a) Mediabench

(b) Spec 2000

Figure 13: Compiler IPC-estimation driven fetch throttling for smaller fetch and instruction queues



(a) Mediabench with throttle threshold of 2 and duration of 1 cycle

(b) Spec 2000 with throttle threshold of 2 and duration of 1 cycle

(c) Mediabench with throttle duration of 2 cycles

(d) Spec 2000 with throttle duration of 2 cycles

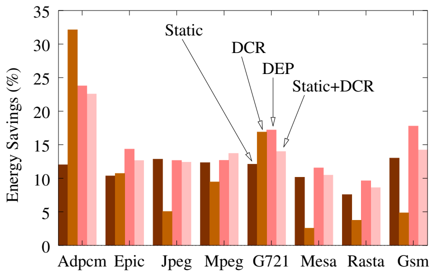
(e) Mediabench with throttle threshold of 3 and duration of 1 cycle

(f) Spec 2000 with throttle threshold of 3 and duration of 1 cycle

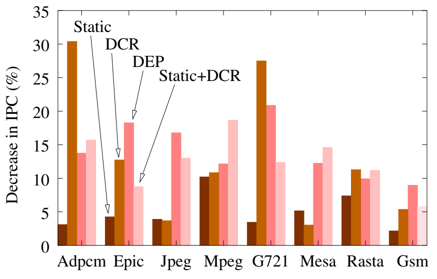
Figure 14: Results for 11-stage pipeline.

Next we compare the proposed compiler-enabled, called hybrid, scheme with the hardware-only runtime IPC based voltage scaling scheme, in the presence of slack (i.e., when the performance goal is not close to the maximum performance). We are particularly interested to see if the static IPC is useful in predicting ILP burstiness. We show experiments for a slack of 20%; we have found the results for larger slacks (not included) to be consistent with our findings at 20% slack. We have found that slacks smaller than 10% are hard to capitalize on due to the reasons explained above.

As mentioned earlier in Section 4.2, the hybrid scheme uses *compile-time generated voltage scaling flags* (based on statically-predicted IPC) to decide the voltage scaling points. In addition, it uses the runtime IPC of the past window, to decide the processor core voltage. Voltage is changed based on static IPC when two consecutive compiler-inserted flags

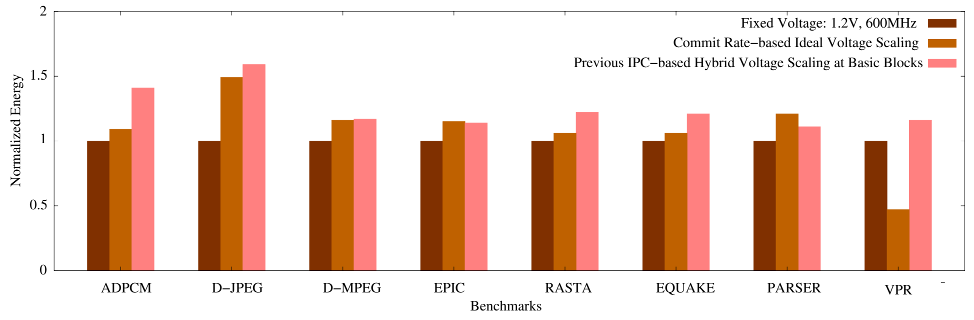


(a) Energy

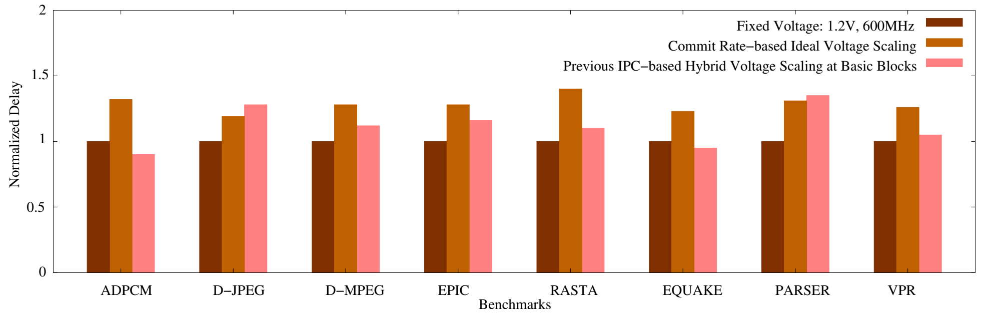


(b) Performance

Figure 15: Comparison between static-DCR and other schemes



(a) Energy Savings (normalized with energy consumption while running at 1.2V, 600MHz)



(b) Performance Impact (normalized with performance while running at 1.2V, 600MHz)

Figure 16: Comparison of Fixed Voltage Scheme with Ideal (at Commit Stage) Voltage Scaling and the Hybrid Scheme

indicate an ILP burst. Otherwise, the application is run based on the runtime IPC predicted for the previous window.

Figure 17 compares the proposed hybrid scheme with a hardware-only scheme with sampling window of 100 cycles, and 10,000 cycles. Figure 17 (a) shows the energy reduction. Figure 17 (b) shows the percentage increase (or decrease) in performance relative to the target performance (corresponds to 0 in the figure).

As seen, energy consumption is reduced significantly: by as much as 15% to 50%. When the application's performance fails to meet the target but degrades performance compared to the target performance, it is possible that more energy reduction is achieved, e.g., in *d-jpeg*. In many soft real-time applications, the quality of perceived performance rolls off slowly as the actual performance degrades.

We can note that the runtime IPC based schemes degrade performance significantly in two applications: *d-jpeg* and *rasta*. Rasta's performance is off only at a 100 cycle granularity. D-jpeg is off by more than 50% at both 100 and 10,000 cycle granularities. The explanation is that this application has ILP burstiness at both window sizes, that a runtime IPC based scheme cannot predict accurately. The actual IPC variation, shown at two granularities in Figure 18, confirms this. We can, for example, note from subfigure (a) that at a window size of 10,000 we would have a good chance to have very different ILP in consecutive windows (that would make our runtime IPC based scheme frequently misspredict the actual ILP).

We can also observe that even the hybrid compiler-enabled scheme degrades performance for one application, *parser*, by 20%. By contrast to the runtime scheme, this is not a fundamental limitation however. We have found that the reason the hybrid compiler-enabled scheme does not meet performance in *parser* is that there are too few voltage scaling flags introduced by the compiler (voltage scaling is introduced primarily at loop boundaries in this example). The hybrid approach could be easily adjusted to fix this problem: for example, by simply implementing loop-splitting or loop-peeling in the compiler to add more voltage scaling decision points. Another solution would be to adjust voltage not only after compiler-inserted points but also at a runtime, fixed window size similarly to the runtime IPC based schemes. We are currently pursuing research in this direction.

Overall, our results confirm our belief that an adaptive voltage scaling scheme that relies solely on the previous window's runtime IPC to predict IPC cannot match the target performance in bursty applications. In addition, we have found that static IPC, in combination with runtime IPC, is useful to make correct voltage decisions even in bursty applications. Interestingly, we have found that static IPC based schemes degrade performance much less than runtime schemes in both voltage scaling and fetch throttling optimizations. An explanation for this finding is that the inaccuracy introduced by static IPC estimation due to dynamic effects is often much less significant than the IPC mispredictions in runtime schemes due to ILP burstiness.

6. PREVIOUS WORK

Most existing ILP based approaches use hardware-based heuristics to predict ILP behavior based on past profiling information. This dynamic-only prediction is then used to drive a throttling, gating or resource-resizing mechanism to save energy. The work can be divided into two broad categories: front-end and back-end methods. Front-end techniques focus on the fetch and decode block, i.e., the earlier stages of the pipeline. The back-end methods, on the other hand, utilize the later stages, i.e., the issue stage. Other researchers have also looked at using the dynamic-only prediction to drive a voltage and frequency scaling mechanism to save energy.

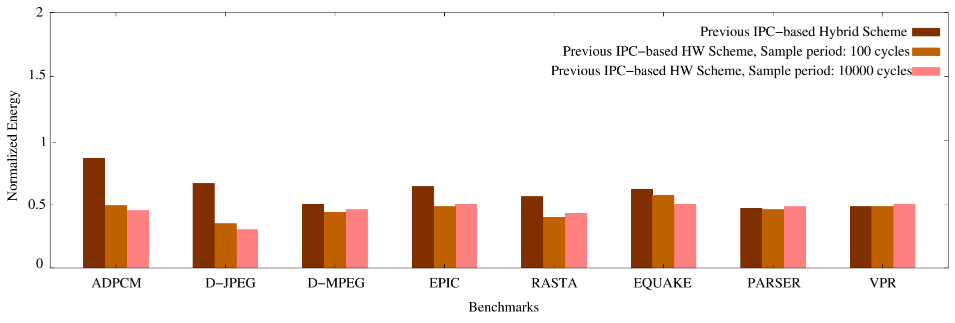
An early example for front-end techniques is the pipeline gating work of Manne et al. [25]. The authors inhibit speculative execution when such execution is highly likely to fail. They analyze when a branch is likely to mispredict and exclude wrong-path instructions from being fetched into the pipeline. Their results show a 38% reduction in wrong-path executions with a 1% performance loss. A more recent work by Parikh et al. [28] also examines power issues related to branch prediction. A key observation of the paper is that chip-wide energy consumption could be reduced by improving branch prediction accuracy even if this leads to spending more power in the branch unit. An alternative front-end approach is fetch/decode throttling by Baniasadi et al. [7]. This fine-grained approach utilizes the information passing through each pipeline stage to estimate the ILP. Based on this information, the fetch/decode stage is stalled when insufficient parallelism exists. However, as also expressed by the authors, traffic per pipeline stage is used as an indirect, approximate, metric of power dissipation.

Back-end approaches concentrate on the issue logic. One popular technique is dynamically resizing the instruction queue size; most researchers take a coarse-grained approach. Folegnani and Gonzales [14] demonstrate that a critical power component in modern microprocessors is the part devoted to extracting parallelism from applications at run-time. Every 1000 cycles, they check if the instruction queue can be resized based on past ILP information in the form of IPC. Combined with other issue queue management methods, they report 14.9% energy savings with 1.7% performance degradation. However, they do not consider the additional power dissipation due to the added logic. Ponomarev et al. [29] also dynamically resize the instruction queue. They dynamically and independently adjust the size of the issue queue, the reorder buffer and the load/store queue. Up to 70% of the power associated with those structures are eliminated. Buyuktosunoglu et al. [10] applied a similar technique, but included the power impact of added logic as well. They report an identical result of up to 70% reduction. An alternative technique, as applied by Bahar and Manne [6] disables clusters of execution units as well as the issue queue when the IPC is predicted to be low. The prediction is done every 256 cycles, so the scheme is coarse-grained. Ghiasi et al. [15] use a coarse granularity scheme to change the issue mode of the processor according to its IPC needs. The overall target IPC is specified by the OS, and is determined by a separate profiling run of the application.

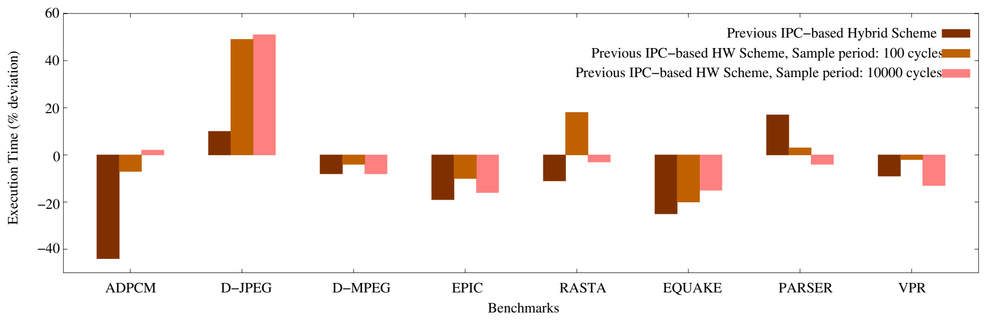
Aggressive supply voltage scaling and process optimization to reduce energy for active logic circuits are being examined in [24, 12]. A recent work by [13] monitors a program's ILP and adjusts processor voltage in response to the amount of observed ILP. They use a profile-driven approach to determine the ILP by setting up an interrupt handler that calculates the observed ILP every 2μs. Another scheme proposed by Marculescu [26] proposes the use of multiple supply voltages at a microarchitectural level by exploiting the difference in latencies of different pipeline stages. The work in [18] proposes a trace-based compiler strategy that is used to scale the voltage in application phases with low CPU utilization. A profile-driven dynamic voltage scheme, shown in [5], uses compiler techniques to establish program checkpoints for voltage scaling at an intra-task level. [30] proposed a static voltage scheduling algorithm for hard real-time systems. This scheme exploited slack times within individual task boundaries to power-down the system during minimal resource utilization periods.

7. CONCLUSION

Energy and power consumption reduction are critical design objectives in modern processors. This paper describes a suite of compiler-architecture techniques to obtain chip-wide energy reduction in the processor, centered around the idea of leveraging IPC information for power-aware resource management. We use IPC information to adaptively adjust voltage and speed, and to throttle processor resources, in re-

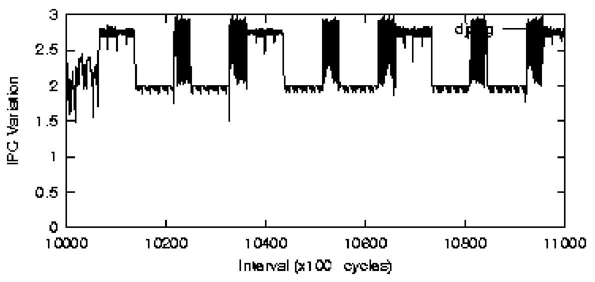


(a) Energy Savings (normalized with energy consumption while running at 1.75V, 1GHz)

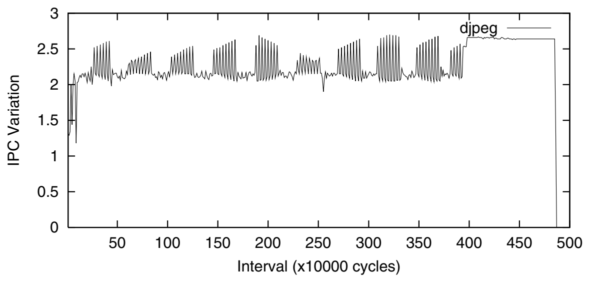


(b) Increase/decrease in execution time (normalized with performance while running at 1.2V, 600MHz with a slack of 20%)

Figure 17: Comparison of previous IPC-based Hybrid Scheme with previous IPC-based Hardware-only Schemes with different voltage scaling interval



(a) IPC burstiness seen at 100 cycle window



(b) IPC burstiness seen at 10,000 cycle window

Figure 18: IPC burstiness observed in d-jpeg for various window sizes

sponse to changes in ILP. Overall, static IPC based resource throttling alone can save up to 14% energy in the processor with less than 5% IPC degradation. We have found that static IPC, in combination with runtime IPC, is necessary to make correct voltage decisions in bursty applications. Interestingly, we have found that the static IPC based schemes degrade performance much less than the runtime-only schemes, in both voltage scaling and fetch throttling optimizations. This is because the inaccuracy introduced by static IPC estimation in compiler-enabled schemes due to dynamic effects (such as cache misses) is often much less significant than IPC mispredictions caused by the program ILP burstiness in runtime schemes.

8. REFERENCES

[1] The standard performance evaluation corporation. *http://www.spec.org*, Decmember 2000.

[2] A. Aho, R. Rethi, and J. Ullman. Compilers: Principles, techniques, and tools. Addison-Wesley.

[3] W. Amme, P. Braun, F. Thomasset, and E. Zehendner. Data dependence analysis of assembly code. In *International Journal on Parallel Programming 28, 5*, 2000.

[4] C. J. Anderson and et al. Physical design of a fourth-generation power ghz microprocessor. In *Proceedings of the 2001 IEEE International Solid-State Circuits Conference (ISSCC '01)*, February 2001.

[5] A. Azevedo and et al. Profile-based dynamic voltage scheduling using program checkpoints. In *Work in*

submission at the European Design Automation and Test Conference (DATE '02). ACM Press, June 2000.

[6] I. R. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, June 2001.

[7] A. Baniasadi and A. Moshovos. Instruction flow-based front-end throttling for power-aware high-performance processors. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '01)*, August 2001.

[8] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00).* ACM Press, June 2000.

[9] D. Burger and T. D. Austin. The simplescalar tool set, version 2.0. In *University of Wisconsin-Madison Computer-Sciences Department Technical Report #1342*, June 1997.

[10] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Proceedings of the Workshop on Power-Aware Computer Systems, ASPLOS IX*, November 2000.

[11] J. Cantin and D. Hill. Cache performance for selected spec cpu2000 benchmarks. *Computer Architecture News, Vol. 29, No. 4*, September 2001.

[12] A. Chandrakasan and R. W. Brodersen. Low-power digital cmos design. Kluwer Academic Publishers.

[13] B. Childers, H. Tang, and R. Melhem. Adapting processor supply voltage to instruction-level parallelism. In *Proceedings of the KoolChips Workshp, in conjunction with MICRO '00*, December 2000.

[14] D. Folegnani and A. Gonzalez. Energy-effective issue queue. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA '01)*, June 2001.

[15] S. Ghiasi, J. Casmira, and D. Grunwald. Using ipc variation in workloads with externally specified rates to reduce power consumption. In *Proceedings of the Workshop on Complexity-Effective Design, ISCA27*, June 2000.

[16] D. Henry, B. Kuszmaul, G. Loh, and R. Sami. Circuits for wide-window superscalar processors. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*, June 2000.

[17] G. Hinton and et al. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[18] C.-H. Hsu and U. Kremer. Compiler-directed dynamic voltage scaling based on program regions. *Technical Report, Department of Computer Science, Rutgers University (DCS-TR-461)*, November 2001.

[19] A. M. D. Inc. Quantispeed achitecture. *AMD White Paper*, September 2001.

[20] W. Kever and et al. A 200mhz risc microprocessor with 128kb on-chip caches. In *Proceedings of the 1997 IEEE International Solid-State Circuits Conference (ISSCC '97)*, February 1997.

[21] A. Kowalczyk and et al. First-generation majc dual processor. In *Proceedings of the 2001 IEEE International Solid-State Circuits Conference (ISSCC '01)*, February 2001.

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97).* ACM Press, December 1997.

[23] J. Leenstra and et al. A 1.8 ghz instruction buffer. In *Proceedings of the 2001 IEEE International Solid-State Circuits Conference (ISSCC '01)*, February 2001.

[24] D. Liu and C. Svensson. Trading speed for low power by choice of supply and threshold voltages. *IEEE Journal of Solid-State Circuits (IEEE JSSC)*, 1993.

[25] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25nd International Symposium on Microarchitecture (MICRO '98)*, June 1998.

[26] D. Marculescu. Power efficient processors using multiple supply voltages. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, December 2000.

[27] R. Maro, Y. Bai, and I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Workshop on Power-Aware Computer Systems PACS'00, In conjunction with ASPLOS IX.*

[28] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA8)*, February 2002.

[29] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, December 2001.

[30] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the Design Automation Conference (DAC '01).* ACM Press, June 2000.

[31] G. Sohi and S. Vajapayem. Instruction issue logic for high-performance interruptable pipelined processors. In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA14)*, June 1987.

[32] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic predictions of critical path instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA7)*, January 2001.

[33] O. S. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. Cool-fetch: Compiler-enabled power-aware fetch throttling. *IEEE Computer Architecture Letters*, 2002.

[34] F. Weber. Hammer: The architecture of amd's next-generation processors. *Microprocessor Forum*, October 2001.

[35] V. Zyuban and P. Kogge. Inherently lower-power high-performance superscalar architectures. In *IEEE Transactions on Computers Vol. 50 No. 3*, March 2001.