

Cool-Fetch: A Compiler-Enabled IPC Estimation Based Framework for Energy Reduction

Osman S. Unsal
Intel Barcelona Research Center,
Intel Labs UPC Barcelona, Spain
Email: osmanx.unsal@intel.com

Israel Koren, C. Mani Krishna, Csaba Andras Moritz
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
Email: {koren,krishna,moritz}@ecs.umass.edu

Abstract— With power consumption becoming an increasingly important factor, it is necessary to reevaluate traditional, power-intensive, architectural techniques and their relative performance benefits. We believe that combined architecture-compiler efforts open up new and efficient ways to retain the performance benefits of modern architectures while addressing their power impact.

In this paper, we present Cool-Fetch, an architecture-compiler based approach to reduce energy consumption in the processor. While we mainly target the fetch unit, an important side-effect of our approach is that we obtain energy savings in many other parts of the processor. The explanation is that the fetch unit often runs substantially ahead of execution, bringing in instructions to different stages in the processor that may never be executed. We have found that although the degree of Instruction Level Parallelism (ILP) of a program tends to vary over time, it can be statically estimated by the compiler. Our Instructions Per Clock (IPC) estimation scheme uses monotonic dataflow analysis and simple heuristics, to guide a fetch-throttling mechanism. We develop the necessary architecture support and include its power overhead. Using Mediabench and SPEC2000 applications, we obtain up to 15% total energy savings in the processor with generally little performance degradation. We also provide a comparison of Cool-Fetch with previously proposed hardware-only dynamic fetch-throttling schemes.

I. INTRODUCTION

For modern processors, the rate of increase in power and energy dissipation is greater than that of performance. Processors are also increasingly used in energy-constrained battery-powered applications. This has forced designers to reformulate their optimization criteria: power and energy are complementing performance as additional design goals. In this paper we present a new framework to address chip-wide power reduction in processors by leveraging static information speculatively. This framework is based on tight cooperation and integration between compiler and architecture.

Specifically, we examine compiler-driven static approaches for increased energy efficiency with only minor performance degradation. Our approach is based on the static estimation of the rate of instructions per clock (IPC) which is a measure of instruction level parallelism (ILP). Most current dynamic architectural energy savings methods depend on analyzing past ILP behavior to estimate future ILP. In contrast, we use static

information about the future ILP that is inherently embedded in the program. To the best of our knowledge, this is the first work that uses compiler-driven static-only IPC estimation for Out-of-Order (OOO), superscalar processor energy savings. Our contributions in this paper are:

- We develop a compiler-driven static IPC estimation scheme that is based on dependence testing and simple heuristics in compiler backends. This information provides an estimate for the upper-bound of available ILP.
- We use this estimation 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 15% chipwide energy savings in the processor with generally little performance degradation.
- We compare the energy and performance aspects of the architectural-compiler level Cool-Fetch with previously-proposed microarchitectural-only fetch-throttling mechanisms.
- We investigate whether dynamic factors such as cache misses, branch prediction, instruction window size and pipeline depth could dilute the efficiency of our static IPC-estimation-based heuristic. For the test cases, we find such efficiency variation to be small.

The compiler-driven IPC estimation approach coupled with fetch-throttling forms the *Cool-Fetch* framework. Various microarchitectural-level front-end mechanisms exist for processor power savings[4], [6], [17], [21]. In comparison, Cool-Fetch exploits the close coupling between the compiler and microarchitectural levels.

The rest of this paper is organized as follows. In Section II, we perform an energy audit to see which blocks in the processor consume the most energy. We use this information to target those blocks for energy savings. Section III discusses compiler and architectural implementation issues related to our IPC estimation and energy savings scheme followed by the experimental setup. The results are presented in Section IV. In Section V, we discuss related work and comment on its relevance to this paper. We conclude in Section VI with a brief discussion.

Supported in part by NSF grant EIA-0102696.

This work was performed when Osman S. Unsal was with the University of Massachusetts, Amherst.

II. MOTIVATION

Power and energy are crucial design parameters not only at the device-level but at the architectural and compiler levels as well. Let us explore the architectural level first. 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 1. Following [5], we assume that the clock 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 IV-A. We scale every resource accordingly; the first configuration is a simple single-issue in-order machine, the second is an 8-way OOO 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 [29].

We now consider the compiler layer. In Figure 2, we present a snapshot from the execution profile of the Spec 2000 application *quake*. The plot shows the actual IPC against our compiler-driven static IPC estimation as averaged over windows of 100 cycles each. Based on this figure, estimated IPC provides a reasonably accurate estimate of actual IPC and we are therefore, motivated to use the static estimation for energy savings by throttling resources when they are not needed. We devote the rest of the paper to exploring and explaining this scheme.

III. IPC-ESTIMATION IMPLEMENTATION

We use a static approach to IPC estimation. It is sufficiently accurate and it is easy to implement, extend and retune. 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 [19], a major limitation of increasing ILP is the presence of true data dependencies. Tune et al. [25] 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 [8]. However, note that the compiler-driven Cool-Fetch framework is equally applicable to an architecture with value speculation, only the compiler-level passes need to be replaced. 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, instructions 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 framework. Here, we consider a standard, non value speculating OOO architecture in our experiments.

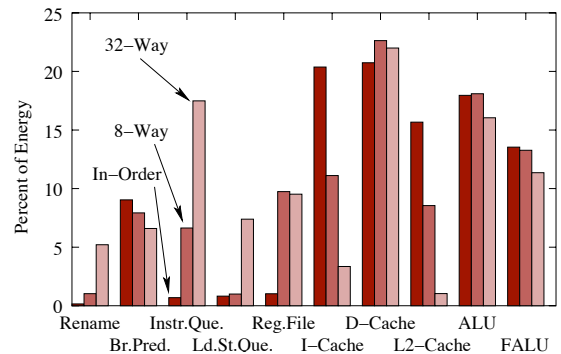


Fig. 1. Percentwise energy consumption of major processor blocks.

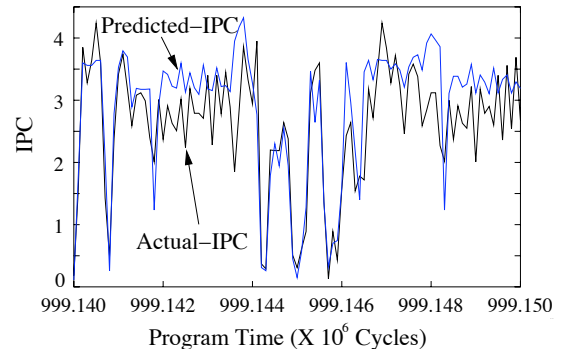


Fig. 2. Estimated versus actual IPC for the *quake* Spec 2000 application. Each point on the x-axis is an average of 100 cycles.

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.

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. In this work, we have enough FU resources in our baseline so this does not become one of the ILP limiting bottlenecks. Of course, there are other, dynamic, factors that influence IPC, such as branch prediction and cache misses. For our test cases, we found that the impact of those dynamic components on the efficiency of our static-only approach is actually smaller than we expected.

A. 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 analysis*, 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 makes high-level passes while Machsuif makes machine-specific optimizations. The final Machsuif pass produces Alpha assembly. We added new passes to both SUIF and Machsuif to annotate and propagate the static IPC-estimation: see Figure 3. We use the available compiler passes and optimizations in SUIF for aggressive extraction of ILP. Our IPC-estimation is at the loop level: loop beginnings and ends serve as natural boundaries for the estimation. Therefore, we need to annotate the beginning and end of every loop. The loop annotation pass accomplishes this: the high-level pass is invoked immediately after we convert source code into SUIF intermediate format and link and merge the various sources. Therefore, this pass works with expression trees and traverses the structured control flow graph (CFG) of each routine.

The other added pass, the IPC-estimation pass, is an assembler-level MachSuif pass that is run just prior to assembler code generation. This way, we guarantee that no compiler level optimizations such as instruction scheduling, which might result in instructions being moved and/or modified, are performed after our pass. As mentioned above, we identify true data dependencies at memory and register accesses in the IPC-estimation pass. The pass is over the linear instruction list of each routine: see Algorithm 1. The algorithm examines each routine and passes the routine as an argument to the function

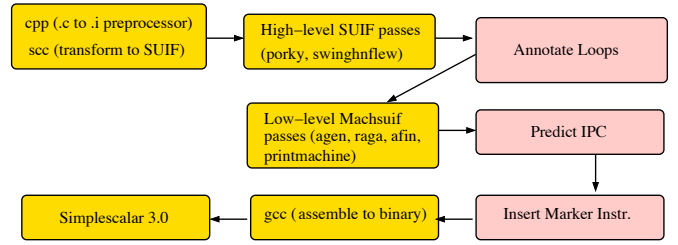


Fig. 3. Compiler flow diagram, our added compiler passes are on the right.

Algorithm 1 The IPC Estimation Algorithm

```

for each routine do
  Call TraverseInst(routine);
end for
/* For each routine, traverse linear instruction list */
TraverseInst(input)
treeList=expTree(input); /* recast input as a linear instruction list */
initialize dependencyList;
while !empty(treeList.instr) do
  ipcCount++;
  if treeList.instr is write then
    add treeList.instr.writtenItem to dependencyList
  end if
  if treeList.instr is read then
    while !empty(dependencyList) do
      if treeList.instr.readItem in dependencyList then
        insert Annotate(estimatedIpc,ipcCount);
        ipcCount=0;
        initialize dependencyList;
      end if
    end while
  end if
  if treeList.instr is loopBegin or loopEnd then
    insert Annotate(estimatedIpc,ipcCount);
    ipcCount=0;
    initialize dependencyList;
  end if
  treeList.instr=treeList.next;
end while
  
```

TraverseInst. The function *expTree* in *TraverseInst* recasts the routine as a linear instruction list. The instruction list is then traversed in instruction order and true dependencies are annotated. Thus the routine is divided into annotation blocks. Each block carries a unique annotation in the beginning of the block, which is simply a count of the instructions in the block. Whenever we come across a true data dependency, we end the block. All the instructions in the block except the last one can potentially be issued in the same cycle. Note that we also end our estimation block at the beginning and end of each loop. This implicitly constitutes a simple static branch prediction mechanism. By terminating the IPC-estimation block at the loop boundaries, we assume that the loop branch is likely to

be taken.

B. Architectural-Level Implementation

After the compiler-passes, we use an assembler level pass to find every IPC-estimation annotation, and insert a marker instruction with the associated IPC number for each. The Alpha assembly marker instruction *bis* is an xor operation with the first source and the destination being the zero-register (\$31), and the second source being the IPC-estimation. We exhaustively checked all our benchmarks by doing a disassembly (to check system inserted code as well): we were not able to find any naturally occurring instruction that xors the zero register with an immediate value and saves the result in the zero register again.

An important distinction between Cool-Fetch and dynamic architectural-level throttling schemes is that the throttling decision is made statically by the compiler in Cool-Fetch. A final pass examines each marker instruction and if the IPC-estimation is below a threshold, it inserts a *throttling flag* at that point. It is this throttling flag, not the marker instruction, that is passed to the hardware layer.

Note that the 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 III-C. If there is not enough flexibility in the ISA, then special throttling flag instructions should be added. This may raise the question of increased code size due to the additional instructions. Although we do not implement this model, we include an analysis of worst-case code size increase due to this approach: we assume that every IPC-estimation marker results in a throttling hint. This is unrealistic but gives an upper bound. Across Mediabench and Spec2000 applications, this bound is modest at 5.1% average code size increase.

For the purposes of this work we have chosen a fine-granularity front-end fetch-throttling scheme. However, the compiler-directed approach is amenable to back-end energy optimization schemes as well. The fetch-throttling scheme latches the compiler-supplied throttling flag at the decode stage. If the flag is set, i.e., the estimated IPC is below a certain threshold, then the fetch stage is throttled and 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-estimation 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. In Cool-Fetch the decision to throttle the fetch is taken by the compiler and therefore the throttling threshold could be returned to a different value depending on the context. By comparison, the throttling threshold value is committed to hardware in architectural-only methods and is therefore fixed. 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 estimation that is at most 2. We include the architectural implementation of our energy saving heuristic in Figure 4. Here, when the throttling flag is set, GATEH 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 [13].

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 y-axis denotes the actual IPC as averaged over 10,000 (coarser granularity) and 500 (comparatively finer granularity) cycles. 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 estimation framework would work equally well with a coarse granularity scheme as well.

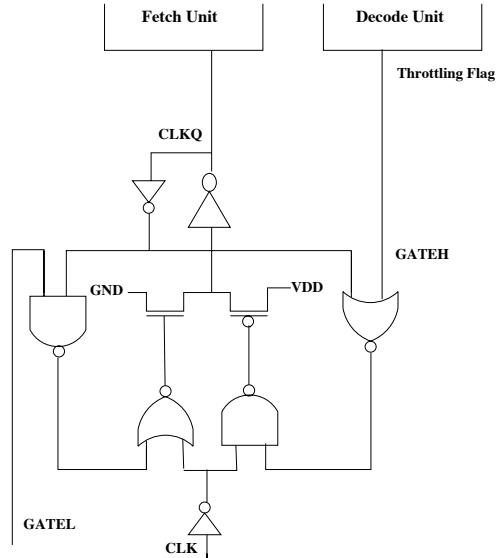


Fig. 4. Architectural implementation of front-end throttling. GATEH is asserted when there is a throttling flag.

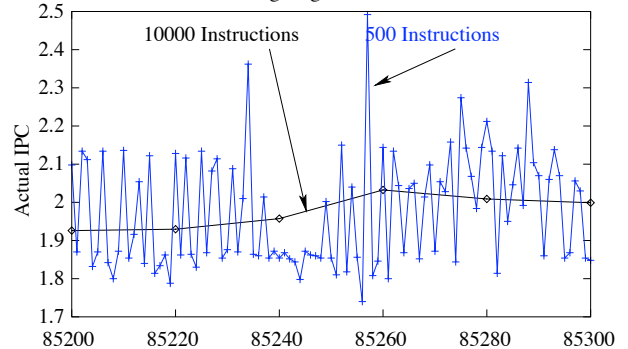


Fig. 5. Coarser versus finer granularity.

C. Architectural Simulator Setup

The baseline architecture reflects the state-of-the-art in current processor designs. Table I contains a description of the baseline parameters. The trend is towards wider issue: Henry et al. [11] 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. [16]. Based on the preceding analysis, we selected an 8-wide issue, 128 entry instruction queue as our baseline. Note that the Cool-fetch baseline includes the throttling logic block which we explained in Section III-B.

Processor Speed	1.5GHz
Process Parameters	0.18 μm , 2V
Issue	Out-of-order
Fetch, Issue, Decode, Commit	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, writeback
L1 I-cache	128Kb, 4-way, writeback
Combined L2 cache	1Mb, 4-way associative
L2 cache hit time	20 cycles
Main memory hit time	100 cycles

TABLE I
BASELINE PARAMETERS.

We use Wattch [7] 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). 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). We run our baseline application without any annotations and compare this against the IPC estimated version. SimpleScalar has been modified to recognize the compiler-generated throttling flag. In Wattch, we use the activity-sensitive power model with aggressive conditional clocking. The rationale for this choice is to compare our fetch-throttling framework to an unthrottled baseline that is already power-efficient. Wattch can be retuned for state-of-the-art technology scaling parameters, we use a 0.18 μm , 1.5GHz, 2V 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.

To extract the maximum available ILP and therefore achieve higher IPC, some contemporary wide-issue processor designs such as the AMD AthlonXP [1] 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 [12] and the 12-stage AMD

Hammer [28]. 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 addition of extra pipeline stages.

IV. EXPERIMENTS

A. Benchmarks

We use the Mediabench[15] and Spec CPU2000[24] benchmarks in our experiments. We select eight applications from each suite: adpcm, epic, g721, gsm, jpeg, mesa, mpeg, rasta from Mediabench and bzip2, gap, mcf, parser, vpr, ammp, art, equake from Spec2000. Of the Spec2000 suite, five (bzip2,gap,mcf,parser,vpr) are Integer, while three (ammp,art,equake) are Floating Point benchmarks. We run all Mediabench applications to completion. For the Spec CPU2000 benchmarks we skip past the initialization stage and simulate the next 1 billion instructions using the reference input set. To skip, we fast-forward by the number of instructions as prescribed by Sair et al. [22] in their Spec CPU2000 initialization segment analysis. If the prescribed number is less than 1 billion, we fast-forward by 1 billion instructions.

B. Baseline Results

We first present our results for the baseline case. See Figure 6a 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 6b, 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 our findings on the percentage decrease of the fetch and instruction queue occupancy. For Mediabench and Spec2000, on average, 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. Notice that the front-end throttling scheme decreases the average queue occupancy of the back-end issue queue as well.

We now examine the percentage of energy savings per processor block: see Figure 7. As expected, the block with the highest overall savings is the fetch stage. However, note

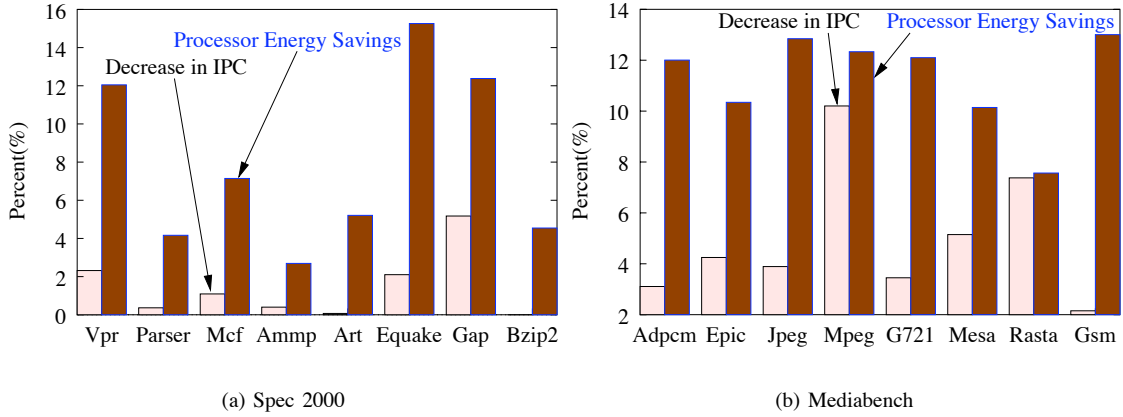


Fig. 6. Impact of compiler IPC-estimation driven fetch throttling

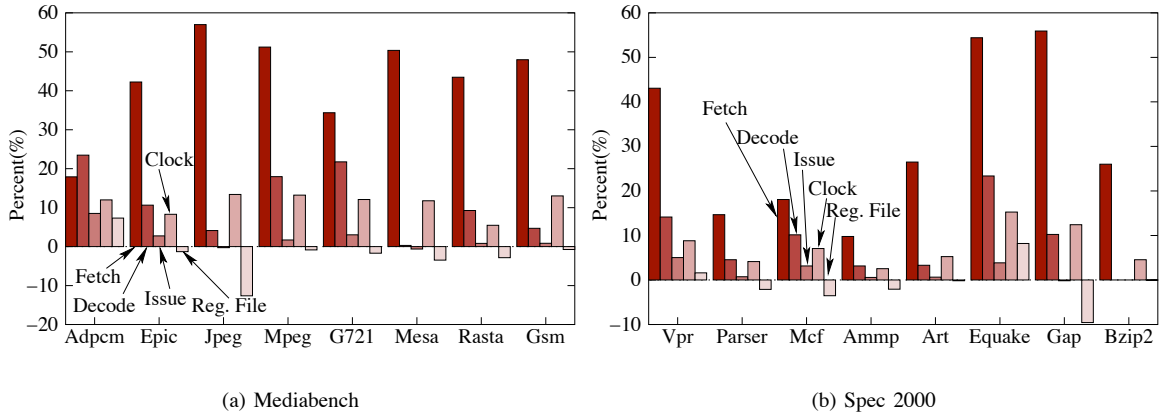


Fig. 7. Percentage Energy Reduction in Processor Blocks

that even the issue stage benefits from fetch-throttling.

C. Comparison With a Dynamic-Only Architectural Scheme

We now compare Cool-Fetch to two previously proposed microarchitectural-level front-end throttling schemes: Decode/Commit Rate (DCR) and Dependence-Based (DEP) heuristics by Baniasadi et al. [6]. 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 cool-fetch, DEP is dependency-based, however DEP makes use of runtime information while Cool-Fetch utilizes only compile-time information. We implemented DCR and DEP following the guidelines in [6]. The performance results are given in Figure 8. By contrast with DCR and DEP, Cool-Fetch substantially preserves the original performance of the applications. The

energy results in Figure 9 indicate that on the average, Cool-Fetch 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 Cool-Fetch, DEP saves more energy however trades off performance.

D. Sensitivity Analysis

In this section, we examine the impact of resource and control dependencies on our Cool-fetch. We start with resource dependencies and analyze the effects of cache misses. Then, we proceed to another resource dependency and 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.

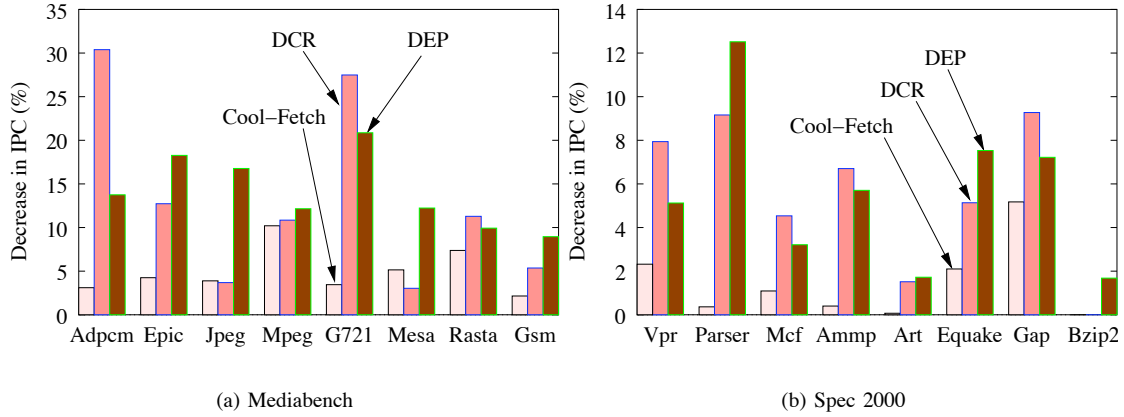


Fig. 8. Performance of Cool-Fetch versus DCR and DEP.

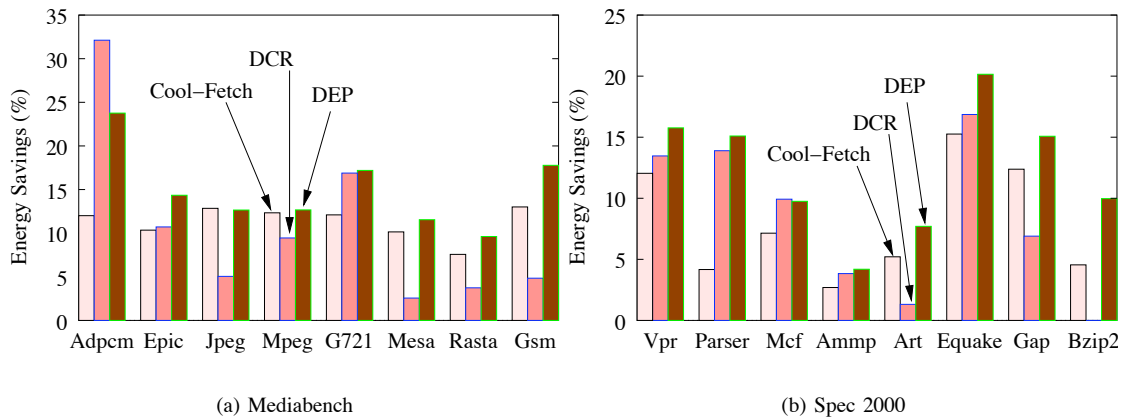


Fig. 9. Energy Efficiency of Cool-Fetch versus DCR and DEP.

This is not the case for the test cases considered. In Table II, 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 [10]. Consider the very high miss rates for the MCF, AMMP and ART. This suggests that extraction of available ILP is affected by dynamic memory performance in those benchmarks. Yet, as seen from Figure 6b, the performance degradation due to our scheme for those applications is not worse compared to other, lower miss-rate, applications.

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 II

MISS RATES FOR THE BASELINE L1 DATA-CACHE (128K, 4 WAY)

We now present the results for more constrained resources. In Figure 10, 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), even though the 2K bimodal predictor had good prediction rates for the selected Spec2000 benchmarks (with the exception of Equake which had a 77% rate). 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 sav-

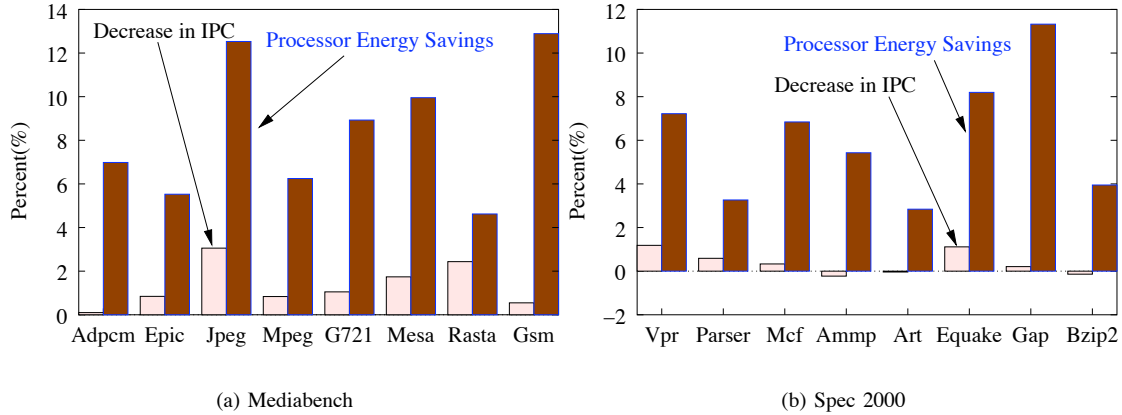


Fig. 10. Compiler IPC-estimation driven fetch throttling for smaller fetch and instruction queues

ings, 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 11. The deeper pipeline allows an exploration for different *threshold* and *duration* parameters. Figure 11a and 11b show the case with a throttling threshold of 2 and duration of 1 cycles. Figure 11c and 11d are for a threshold of 2 and an expanded throttling duration of 2 cycles. Figure 11e and 11f 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.

V. PREVIOUS WORK

Previous analyses of limits of available ILP reported either pessimistic or optimistic results. The pessimistic camp includes the work of Wall [27], who assumes a processor with perfect memory disambiguation, perfect register renaming, unlimited fetch bandwidth and a large number of functional units. For a wide range of benchmarks, this processor achieves a maximum speedup of only about 7 times that of a realistic baseline processor. On the other hand, Nicolau and Fisher [20] are in the optimistic camp: they report that three digit IPC values are achievable for some loop-dominated applications.

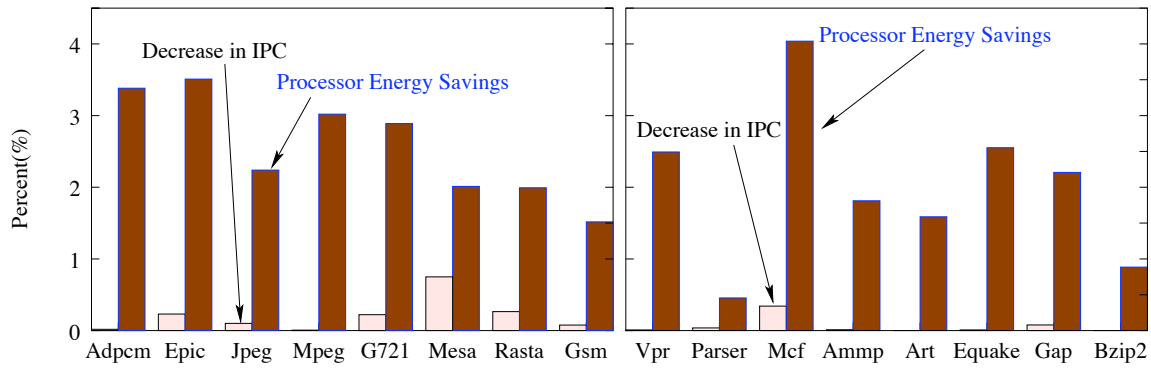
IPC estimation for superscalar processors is in some ways orthogonal to compiler-level dependence analysis for instruction scheduling in VLIW processors. For VLIW processors, the compiler statically and non-speculatively determines

dependence-free instructions that are then bundled into a long instruction. The literature for VLIW compilers is vast, for the sake of brevity we refer the reader to Schlansker et al. [23] for compiler-architecture interaction techniques which achieve high levels of ILP in VLIW processors. IPC estimation is similar to superword-level-parallelism [14] in the sense that it can be profitable when inherent ILP is scarce.

Energy reduction through ILP monitoring is a fertile research area. Most approaches use hardware-based heuristics to predict ILP behavior based on past profiling information. This dynamic-only estimation 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. Here, we focus on the front-end.

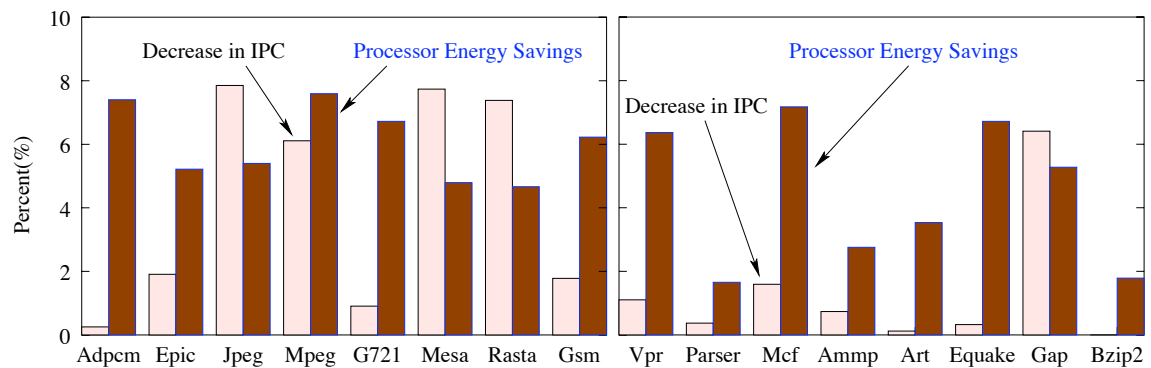
An early example for front-end techniques is the pipeline gating work of Manne et al. [17]. The authors inhibit speculative execution when such execution is highly likely to fail. They analyze when a branch is likely to mispredict and exclude wrong-path instructions from being fetched into the pipeline. Their results show a 38% reduction in wrong-path executions with a 1% performance loss. A more recent work by Parikh et al. [21] 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 Baniyadi et al. [6]. 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.

To the best of our knowledge, there have been no efforts on incorporating compiler-driven static-only techniques for



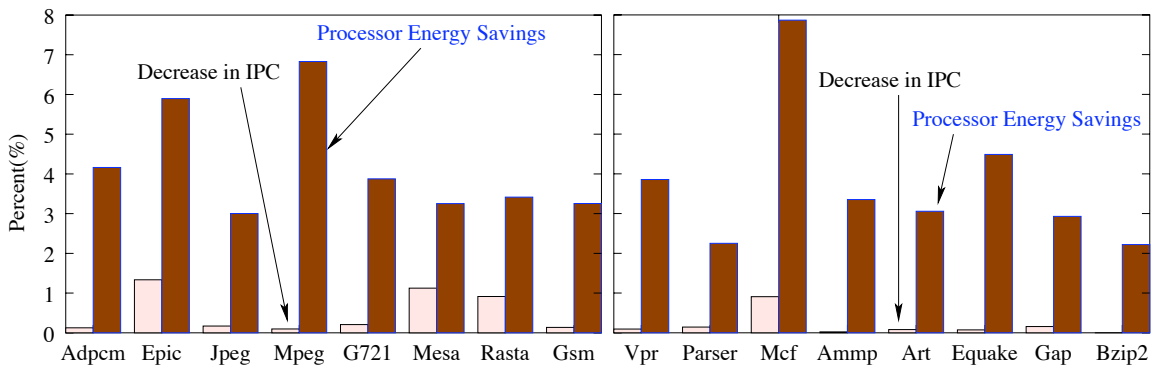
(a) Mediabench with throttle threshold of 2 and duration of 1 cycle

(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

Fig. 11. Results for 11-stage pipeline.

determining ILP for power and energy savings. Marculescu [18] proposes a dynamic compiler-assisted technique that adaptively selects the number of instructions to be fetched and executed in parallel. She employs a profile-driven methodology to find the optimal number of instructions to be executed in parallel for each basic block. However, this requires pre-execution of the program for n times to gather profiling data for

each basic block, where n is the maximum available fetch or execution rate. The granularity of the approach is at the basic block level, consequently there is a single fetch and execute rate per block.

VI. SUMMARY

We have shown in this paper that compiler-driven static IPC estimation is a powerful approach for achieving chipwise energy savings in superscalar out-of-issue processors. We report up to 15% processor energy savings with Cool-Fetch. The impact on performance is minimal and depending on the application, the method can even lead to performance improvements. We include the power dissipation overhead of our technique in experiments across a wide spectrum of architectural configurations. We find that the efficiency of our static technique is quite stable in the presence of dynamic program behavior such as cache misses and branch mispredictions.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions.

REFERENCES

- [1] Advanced Micro Devices Inc., "QuantiSpeed Architecture," *AMD White Paper*, September 2001.
- [2] Aho A. V., Rethi R., Ullman J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading MA, 1988
- [3] Amme W., Braun P., Thomasset F., Zehendner E., "Data Dependence Analysis of Assembly Code," *International Journal on Parallel Programming* 28, 5 (2000), 2000.
- [4] Aragon J. L., Gonzalez J., Gonzalez A., "Power-Aware Control Speculation Through Selective Throttling," *Proceedings of the 9th International Symposium on High Performance Computer Architecture, HPCA9*, February 2003.
- [5] Bahar R. I., Manne S., "Power and Energy Reduction Via Pipeline Balancing," *Proceedings of the 28th International Symposium on Computer Architecture, ISCA28*, June 2001.
- [6] Baniasadi A., Moshovos A., "Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors," *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED01*, August 2001.
- [7] Brooks D., Tiwari V., Martonosi M., "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proceedings of the 27th International Symposium on Computer Architecture, ISCA27*, June 2000.
- [8] Bhargava R., John L. K., "Latency and Energy Aware Value Prediction for High-Frequency Processors," *Proceedings of the 16th Annual ACM International Conference on Supercomputing, ICS02*, June 2002.
- [9] Burger D., Austin T. D., "The Simplescalar Tool Set, Version 2.0," *University of Wisconsin-Madison Computer-Sciences Department Technical Report #1342*, June 1997.
- [10] Cantin J. F., Hill M. D., "Cache Performance for Selected SPEC CPU2000 Benchmarks," *Computer Architecture News, Vol. 29, No. 4*, September 2001.
- [11] Henry D. S., Kuszmaul B. C., Loh G. H., Sami R., "Circuits for Wide-Window Superscalar Processors," *Proceedings of the 27th International Symposium on Computer Architecture, ISCA27*, June 2000.
- [12] Hinton G., et al., "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal Q1*, 2001.
- [13] Kever W., et al., "A 200MHz RISC Microprocessor with 128kB On-Chip Caches," *1997 IEEE International Solid-State Circuits Conference, ISSCC 1997*, February 1997.
- [14] Larsen S., Amarasinghe S., "Exploiting Superword Level Parallelism With Multimedia Instruction Sets," *In Proceedings of the SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.
- [15] Lee C., Potkonjak M., Mangione-Smith W. H., "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *Proceedings of the 30th Annual International Symposium on Microarchitecture, MICRO30*, December 1997.
- [16] Leenstra J., et al., "A 1.8 GHz Instruction Buffer", *2001 IEEE International Solid-State Circuits Conference, ISSCC 2001*, February 2001.
- [17] Manne S., Klausner A., Grunwald D., "Pipeline Gating: Speculation Control for Energy Reduction," *Proceedings of the 25th International Symposium on Computer Architecture, ISCA25*, June 1998.
- [18] Marculescu D., "Profile-Driven Code Execution for Low Power Dissipation," *Proceedings of the International Symposium on Low Power Electronics and Design, ISLPED00*, July 2000.
- [19] Maro R., Bai Y., Bahar R. I., "Dynamically Reconfiguring Processor Resources to Reduce Power Consumption in High-Performance Processors," *Workshop on Power-Aware Computer Systems PACS'00, In conjunction with ASPLOS IX*, 2000.
- [20] Nicolau A., Fisher J. A., "Measuring the Parallelism Available for Very Long Instruction Word Architectures," *IEEE Transactions on Computers, C-33(11)*, November 1984.
- [21] Parikh D., Skadron K., Zhang Y., Barcella M., Stan M. R., "Power Issues Related to Branch Prediction," *8th International Symposium on High-Performance Computer Architecture, HPCA8*, February 2002.
- [22] Sair S., Charney M., "Memory Behavior of the SPEC2000 Benchmark Suite," *IBM T. J. Watson Research Center Technical Report*, 2000.
- [23] Schlansker M., Rau B. R., Mahlke S., Kathail V., Johnson R., Anik S., Abraham S. G., "Achieving High Levels of Instruction-Level Parallelism With Reduced Hardware Complexity," *Technical report, HPL-96-120, Hewlett Packard Laboratories*, February 1996.
- [24] The Standart Performance Evaluation Corporation, <http://www.spec.org>, December 2000.
- [25] Tune E., Liang D., Tullsen D. M., Calder B., "Dynamic Predictions of Critical Path Instructions," *7th International Symposium on High Performance Computer Architecture, HPCA7*, January 2001.
- [26] Unsal O.S., Koren I., Krishna C.M., Moritz C.A., "Cool-Fetch: Compiler-Enabled Power-Aware Fetch Throttling," *IEEE Computer Architecture Letters*, Vol. 1, 2002.
- [27] Wall D. W., "Limits of Instruction-Level Parallelism," *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV*, 1991.
- [28] Weber F., "Hammer: The Architecture of AMD's Next-Generation Processors," *Microprocessor Forum*, October 2001.
- [29] Zyuban V. V., Kogge P. M., "Inherently Lower-Power High-Performance Superscalar Architectures," *IEEE Transactions on Computers Vol. 50 No. 3*, March 2001.