Cool-Fetch: Compiler-Enabled Power-Aware Fetch Throttling

Osman S. Unsal, Israel Koren, C. Mani Krishna, Csaba Andras Moritz
Dept. of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003
E-mail: {ounsal,koren,krishna,moritz}@ecs.umass.edu

Abstract— **In this paper, we present 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 in 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 predicted by the compiler with considerable accuracy. Our Instructions Per Clock (IPC) prediction scheme is using a dependence-testing-based analysis and simple heuristics, to guide a front-end fetch-throttling mechanism. We develop the necessary architecture support and include its power overhead. We perform experiments over a wide number of architectural configurations, using SPEC2000 applications.**

Our results are very encouraging: we obtain up to 15% total energy savings in the processor with generally little performance degradation. In fact, in some cases our intelligent throttling scheme even *increases* performance.

Keywords— **Low power design, compiler architecture interaction, instruction level parallelism, fetch-throttling**

I. Introduction

Power consumption is becoming an important design criterion. Our previous work explored power reduction in the caching subsystem by leveraging static information speculatively[7], [8], [9]. In this paper we apply the same approach to address chip-wide power reduction in processors. This framework is based on a 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 prediction of the rate of instructions per clock (IPC) which is a measure of instruction level parallelism (ILP). Most current dynamic microarchitectural energy savings methods depend on analyzing past ILP behavior to predict future ILP. In contrast, we use static information about the future ILP that is inherently embedded in the program. The compiler-driven IPC prediction approach coupled with fetch-throttling forms the *Cool-Fetch* framework. To the best of our knowledge, this is the first work that uses compiler-driven static-only IPC prediction for out-of-order, superscalar processor energy savings. In this paper,

- We develop a compiler-driven static IPC prediction scheme that is based on dependence testing and simple heuristics in compiler backends. This information is a measure of the upper-bound of available ILP.
- We use this prediction scheme to drive our fine-grained fetch-throttling energy-saving heuristic. We have experimented with a variety of architectural configurations using Spec 2000 benchmarks. We obtain up to 15% total energy savings in the processor with generally little performance degradation.
- For some specific applications our compiler-driven fetch-throttling scheme actually results in *increased* performance. We discuss the reasons behind this somewhat surprising result.
- We compare the energy and performance aspects of Cool-Fetch with previously-proposed microarchitectural-only fetch-throttling mechanisms.

Supported in part by NSF grant EIA-0102696. Manuscript submitted: 22 Mar. 2002. Manuscript accepted: 2 Apr. 2002. Final manuscript received: 9 Apr. 2002.

II. Motivation

In Figure 1, 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 100 cycles each. 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.

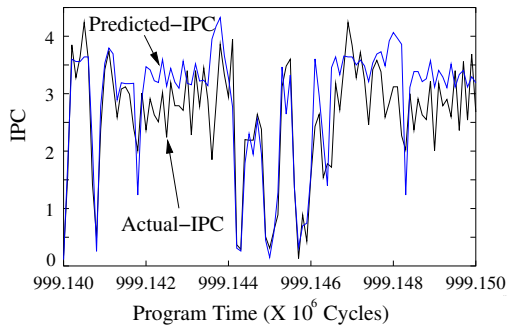

Fig. 1. Predicted versus actual IPC for the *equake* Spec 2000 application. Each point on the x-axis is an average of 100 cycles.

III. IPC-Prediction Implementation

We use a static approach to IPC prediction. It is sufficiently accurate, is easy to implement, to 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 or can be executed in parallel. As expressed by Tune et al. [6], the bottleneck for many workloads on current processors is true dependencies in the code. While antidependencies (Write-After-

Read or WAR) or output dependencies (Write-After-Write or WAW) could be eliminated by register renaming, 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 was severely constrained in its register renaming resources. However, contemporary processors usually have enough resources to eliminate most false dependencies. Of course, there are other, dynamic, factors that influence IPC, such as branch prediction or cache misses. Surprisingly, in our experiments, we found that the impact of cache misses on the efficiency of our static-only approach is actually smaller than envisioned. We next discuss the compiler and architectural level issues related to static IPC-prediction.

A. Compiler-Level 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 code. We added new passes to both SUIF and Machsuif to annotate and propagate the static IPC-prediction. Our IPC-prediction is at the loop level: loop beginnings and ends serve as natural boundaries for the prediction. The high-level loop annotation pass works with expression trees and traverses the structured control flow graph (CFG) of each routine.

The other added pass; the IPC-prediction pass, is a lower-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. In the IPC-prediction pass, we identify true data dependencies at memory and register accesses. Since the approach is speculative and is not required to be correct all the time, we employ a straightforward, easy-to-implement memory dependence analysis at the instruction operand level. A non-speculative, exhaustive, memory dependence approach would require sophisticated alias analysis. However, such analysis is unnecessary: our results suggest that the speculative scheme performs sufficiently well. The pass is over the linear instruction list of each routine. We divide the program 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 prediction block at the beginning and end of each loop.

B. Architectural-Level Implementation

An important distinction between Cool-Fetch and dynamic microarchitectural-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 conducted an analysis of worst-case code size increase due to this approach: we assume that every IPC-estimation marker results in a throttling flag. This is unrealistic but gives an upper bound. The average code size increase is modest at 5.2%.

The fetch-throttling scheme latches the compiler-supplied throttling flag 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 for our configuration. That is, we stop instruction fetch for 1 cycle when we encounter an IPC prediction that is at most 2. When the throttling flag is set, 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 [4].

The system model is a typical out-of-order superscalar processor. Table I contains a description of the baseline architectural parameters. The baseline case reflects the trend towards wider issue as evidenced by the 8-way Alpha 21464.

General Parameters	0.18 μm, 2V, 1.5GHz
Issue	8-way Out-of-order
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- and I-cache	Each: 128Kb, 4-way
Combined L2 cache	1Mb, 4-way
L2 cache hit time	20 cycles
Main memory hit time	100 cycles

TABLE I
BASELINE PARAMETERS

C. Architectural Simulator Setup

We use Wattch [2] to run the binaries and collect the energy results. Wattch is based on the Simplescalar framework. Simplescalar has been modified to recognize the compiler-generated IPC predictions. 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 in the dispatch stage.

IV. Experiments

A. Benchmarks

We use the Spec CPU2000 benchmarks in our experiments. We randomly select eight applications. The first five (bzip2,gap,mcf,parser,vpr) are from the Integer and the last three (ammp,art,equake) are from the Floating Point 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. [5] 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 2. The compiler IPC-prediction driven front-end throttling yields excellent results for this architectural configuration: on the average, we get 8% processor energy savings with a performance degradation of 1.5%.

To illustrate how fetch-throttling saves resources, we

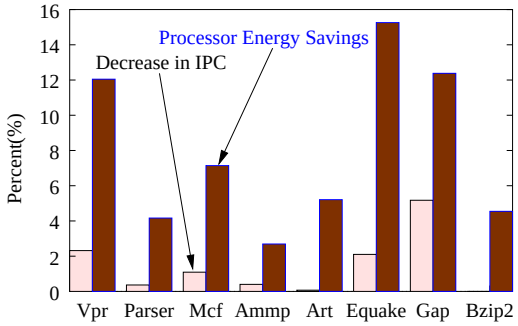

Fig. 2. Impact of compiler IPC-prediction driven fetch throttling.

studied the percentage decrease of the fetch and instruction queue occupancy for the benchmarks. The percent of time that the queues are full is decreased by 14.7% for fetch; and 7.7% for issue. The average queue size is decreased by 10.4% for fetch; and 2.0% for issue.

We now examine the percentage of energy savings per processor block: see Figure 3. As expected, the block with the highest overall savings is the fetch stage. However, note that even the issue stage benefits from fetch-throttling.

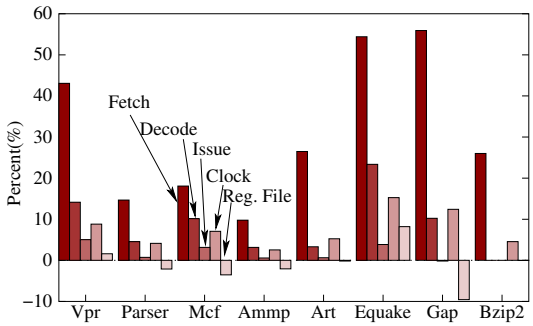

Fig. 3. Percentage Energy Reduction in Processor Blocks.

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. [1]. 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 run-time information while Cool-Fetch utilizes only compile-time information. We implemented DCR and DEP following the guidelines in [1]. The performance results are given in Figure 4. By contrast with DCR and DEP, Cool-Fetch substantially preserves the original performance of the applications. The energy results in Figure 5 indicate that on the average, Cool-Fetch is as energy-efficient as DCR. Note that for some applications such as the Parser, DCR saves more energy, however, the energy savings come at the expense of performance, i.e., DCR trades off performance for energy. On the other hand, DEP saves more energy than Cool-Fetch: 12.1% on average. However, DEP incurs a higher performance penalty, an average degradation of 5.6%.

D. Sensitivity Analysis

One would expect that since our energy-saving heuristic depends on a static approach, dynamic program behavior such as cache misses and branch mispredictions would dilute the efficiency of our method. Somewhat surprisingly, this isn't the case. In Table II, we present the data cache miss rates. The results are in agreement with data gathered from a recent Spec 2000 cache performance analysis [3]. Consider the very high miss rates for the MCF, AMMP and ART. This suggests that extraction of available ILP is effected by dynamic memory performance in

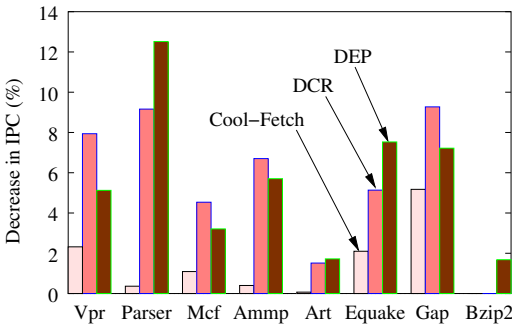

Fig. 4. Performance of Cool-Fetch versus DCR and DEP.

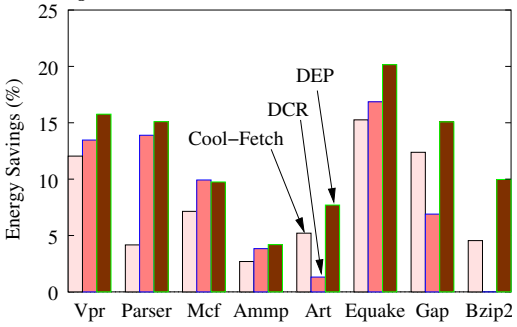

Fig. 5. Energy Efficiency of Cool-Fetch versus DCR and DEP.

those benchmarks. Yet, as seen from Figure 2, the performance degradation due to our scheme for those applications is not worse compared to other, lower miss-rate applications. For branch mispredictions, we experimented with a larger and better hybrid branch predictor (64K bimodal + 64K Gshare with 64K selector). Compared with the unthrottled case with the same branch predictor configuration, the 2K bimodal predictor results in 1.5% average performance degradation and 8% energy savings, while the hybrid predictor has 1% performance degradation and 7.5% energy savings.

We now present the results for more constrained resources. In Figure 6, the fetch and instruction queues are 8 and 32 instructions, respectively. 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. 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.

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)

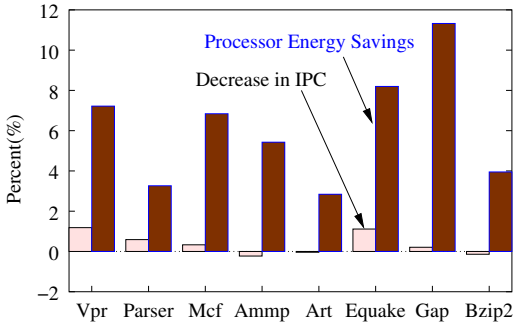

Fig. 6. Compiler IPC-prediction driven fetch throttling for smaller fetch and instruction queues.

V. Discussion

In this paper, we have shown that compiler-driven static IPC prediction is a powerful engine for chipwise energy saving heuristics 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 are extending our experiments by using deeper pipelining and more aggresive branch predictors in our baseline. We will also include the MediaBench benchmarks and use different *thresholds* and *durations* for throttling. We are currently investigating applying similar compile-time approaches to back-end energy optimizations which require slightly different, coarser granularity, schemes.

References

[1] 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

[2] 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, ISCA'00,* June 2000

[3] Cantin J. F., Hill M. D., "Cache Performance for Selected SPEC CPU2000 Benchmarks," *Computer Architecture News, Vol. 29, No. 4,* September 2001

[4] Kever W., et al., "A 200MHz RISC Microprocessor with 128kB On-Chip Caches," *Proceedings of the International Solid-State Circuits Conference, ISSCC-97,* February 1997

[5] Sair S., Charney M., "Memory Behavior of the SPEC2000 Benchmark Suite," *IBM T. J. Watson Research Center Technical Report,* 2000

[6] 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

[7] Unsal O.S., Wang Z., Koren I., Krishna C.M., Moritz C.A., "On Memory Behavior of Scalars in Embedded Multimedia Systems," *Workshop on Memory Performance Issues, ISCA28,* June 2001.

[8] Unsal O.S., Ashok R., Koren I., Krishna C.M., Moritz C.A., "Cool-Cache for Hot Multimedia," *34th Annual International Symposium on Microarchitecture, MICRO34,* December 2001

[9] Unsal O.S., Koren I., Krishna C.M., Moritz C.A., "The Minimax Cache: An Energy-Efficient Framework for Media Processors," *8th International Symposium on High-Performance Computer Architecture, HPCA8,* February 2002