

Compiler-Based Adaptive Fetch Throttling for Energy-Efficiency*

Huaping Wang, Yao Guo, Israel Koren, C. Mani Krishna
Department of Electrical and Computer Engineering
University of Massachusetts, Amherst, MA 01003
E-mail: {hwang,yaoguo,koren,krishna}@ecs.umass.edu

Abstract

Front-end instruction delivery accounts for a significant fraction of energy consumption in dynamically scheduled superscalar processors. Different front-end throttling techniques have been introduced to reduce the chip-wide energy consumption caused by redundant fetching. Hardware-based techniques, such as flow-based throttling, could reduce the energy consumption considerably, but with a high performance loss. On the other hand, compiler-based IPC-estimation-driven software fetch throttling (CFT) techniques result in relatively low performance degradation, which is desirable for high-performance processors. However, their energy savings are limited by the fact that they typically use a predefined fixed low IPC-threshold to control throttling.

In this paper, we propose a Compiler-based Adaptive Fetch Throttling (CAFT) technique that allows changing the throttling threshold dynamically at runtime. Instead of using a fixed threshold, our technique uses the Decode/Issue Difference (DID) to assist the fetch throttling decision based on the statically estimated IPC. Changing the threshold dynamically makes it possible to throttle at a higher estimated IPC, thus increasing the throttling opportunities and resulting in larger energy savings. We demonstrate that CAFT could increase the energy savings significantly compared to CFT, while preserving its benefit of low performance loss. Our simulation results show that the proposed technique doubles the energy-delay product (EDP) savings compared to the fixed threshold throttling and achieves a 6.7% average EDP saving.

1. Introduction

Power consumption has emerged as a significant factor in computer architecture. Out-of-order processing of instructions, speculative execution, and register renaming techniques improve performance significantly compared to in-order execution, but they also introduce significant energy overhead necessary to keep track of all the instructions and their dependencies.

A large fraction of power in modern high-performance processors is dissipated by the front-end of the pipeline, including the fetch and decode units. Conventional superscalar processors attempt to maximize the number of “in-

flight” instructions at all times in order to achieve high performance. Following a branch misprediction, they begin fetching at full speed and continue doing so until the next branch misprediction flushes the pipeline or until the issue queue (or re-order buffer) is full. No matter how low the instruction-level parallelism (ILP) may be, instructions are still fetched, decoded and then put into the issue queue. This not only increases the energy consumption of the issue queue logic with additional wake-ups and selections, but also adds more pressure to the register file.

A number of front-end throttling techniques have been proposed for improving the energy efficiency during the fetching process in superscalar pipelines. These techniques can be categorized into hardware-based runtime [3, 7, 9, 10, 11] and software-based static [8, 13, 18] techniques. Hardware-based techniques assume that the program state is stable and use the recent history information to predict future behavior. These can catch dynamic behavior such as cache misses but cannot catch irregular situations such as abrupt phase changes. For example, a flow-based fetch throttling technique [3] uses the instruction Decode/Commit rate (DCR) in the previous cycle to decide whether to stall instruction fetching in the next cycle. Throttling is triggered when high DCR values occur as a result of branch misprediction. Because this approach cannot catch the bursty behavior of programs, it will cause substantial performance degradation (more than 8% for some benchmarks).

Software-based throttling techniques can estimate the ILP based on compile-time program analysis and provide indications of sharp changes in ILP (or ILP bursts). Static techniques may, however, produce inaccurate predictions due to their inability to capture dynamic effects such as branch mispredictions and cache misses. Previous research [18] employed compiler techniques to estimate the IPC and used the estimated IPC to drive its fine-grained fetch-throttling energy-saving heuristic. A fetch will be stalled in the following cycle if the estimated IPC is lower than a predefined threshold, which in [18] has been set to 2 for an 8-way issue processor. Throttling using such a low threshold will have only a small negative effect on performance, but will also yield relatively small energy savings.

There are two potential problems using a fixed low value of the IPC-threshold to drive fetch throttling. The first one is that it limits the throttling opportunities at high IPC values. If there are many instructions left unexecuted in the previous cycle, we could throttle at a higher IPC-threshold with probably no performance loss. The second problem is that the fixed IPC-threshold technique may throttle at an

*This work has been supported by NSF under grant EIA-0102696.

inappropriate time, resulting in a performance loss. Assume for example that the estimated IPC in the following cycle is 2, but there are no instructions left in the issue queue from the previous cycle; a throttling at this time is inappropriate and will result in a performance loss. Therefore, by using adaptive rather than fixed IPC-thresholds for fetch throttling, we could overcome both problem and obtain better results.

In this paper, we present a new approach called Compiler-based Adaptive Fetch Throttling (CAFT), which allows changing the throttling IPC-threshold adaptively at run-time. Our technique is based on compile-time static IPC estimation, but we use the Decode/Issue Difference (DID) to assist the fetch throttling decision based on the statically estimated IPC. DID is the difference between the numbers of decoded and issued instructions in the previous cycle, which can be considered as the recent history information. The IPC-threshold is changed dynamically according to the DID value, making it possible to throttle at a higher estimated IPC if appropriate. This increases the throttling opportunities and thereby results in larger energy savings.

The contributions of this paper are:

- We present a flexible compiler-based fetch throttling technique that can change the throttling threshold adaptively with the help of dynamic information. Because CAFT can throttle more cycles than CFT, it saves more energy.
- We provide a detailed analysis of the distribution of IPC-thresholds for the CAFT technique. It shows that over half of the throttle cycles are performed at IPC-thresholds above 2, allowing significant improvements in the throttling opportunities. As a result, our technique can save more energy than a fixed low IPC-threshold scheme and still preserve performance.
- We make comparisons to DEP - the dependence-based fetch throttling scheme [3] - a flow-based hardware fetch throttling technique, and to JIT - the Just-In-Time instruction delivery scheme [10] - a hardware-based fetch throttling technique which uses information about *in-flight* instructions to control the front-end instruction fetching. Both methods are based on the same underlying principles that we use in CAFT. We demonstrate that although CAFT has only a small relative reduction in energy consumption, its performance loss is considerably lower compared to DEP and JIT.

The rest of the paper is organized as follows. In the next section we briefly describe the compiler-based static IPC estimation approach. In Section 3 we present our compiler-based adaptive fetch throttling technique. For comparison purposes, we also describe the dependence-based hardware-only throttling technique, the just-in-time instruction delivery technique and the DID-based technique. Our evaluation methodology is presented in Section 4 followed by our numerical results in Section 5. Section 6 describes some related work. Finally, we present a summary of our work in Section 7.

2. Compiler-based IPC Estimation

In our work, we use compile-time static IPC-estimation to drive throttling, which is similar to what has been proposed by Unsal *et al.* [18]. A brief introduction to the compiler-level IPC-estimation scheme is provided in this section.

Our implementation considers only true data dependencies (Read-After-Write or RAW) to check if instructions depend on each other or can be executed in parallel. As mentioned in [12], a major limitation of increasing ILP is the presence of true data dependencies. Tune *et al.*[17] 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 [4]. In our experiments, we consider a standard, non-value speculating out-of-order architecture. However, note that the compiler-driven framework is equally applicable to an architecture with value speculation, only the compiler-level passes need to be modified.

We statically determine true data dependencies using data dependency analysis at the assembly-code level. Our post-register allocation scheme uses monotone data flow analysis, similar to [2]. We identify data dependencies at both registers and memory accesses. Register analysis is straightforward: the read and written registers in an instruction can be identified easily, since registers do not have aliases. However, for memory accesses, there are three implementation choices: no alias analysis, complete alias analysis, or alias analysis by instruction inspection [14]. We perform an approximate and speculative alias analysis by instruction inspection that 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 a true dependency.

We use SUIF [19]/MachSUIF [16] as our compiler framework. SUIF does high-level passes while MachSUIF performs machine-specific optimizations. The final MachSUIF pass produces Alpha assembly code. We have added new passes to both SUIF and MachSUIF to annotate and propagate the static IPC-estimation. Our IPC-estimation is at the basic block or loop level: loop beginnings and endings serve as natural boundaries for the estimation. 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 runs just prior to assembler code generation.

3. CAFT: Compiler-Based Adaptive Fetch Throttling

As mentioned before, the previous compile-time static IPC-estimation based fetch throttling framework fixes the

throttling IPC-threshold at 2 and assumes that throttling at such a low IPC will have little effect on performance. This approach limits the energy savings because it ignores many throttling opportunities which exist at a higher IPC. However, if we fix the throttling IPC-threshold at a high value, the performance will rapidly decrease due to too frequent throttling.

An adaptive throttling IPC-threshold, which would allow us to throttle at a higher estimated IPC, could be beneficial if it can still keep the performance loss low. If we can change the IPC-threshold adaptively and throttle at a higher IPC only when appropriate, we can reduce the number of cache accesses considerably. As a result, instructions are fetched just in time to exploit the available parallelism. Also, since fetching will proceed at a slower pace, we will reduce the number of incorrectly speculated instructions that enter the pipeline.

After instructions are decoded, they are put into the issue queue and executed on individual functional units if all the operands are ready and enough functional units are available. In a perfect machine with no constraints such as true data dependencies, the number of instructions through the decode and issue stages should be identical during a period of time. When the number of decoded instructions surpasses the number of issued instructions, this means that sufficient parallelism does not exist and instruction decode has to be suspended. If we continue to fetch and decode instructions at this time we only add to the overall energy consumption, especially in the I-cache, while not improving performance. A large Decode/Issue Difference (DID) value means that many instructions were left unexecuted, and the performance will not be affected if we throttle for one cycle.

The difference between decoded and issued instructions can be considered as the recent history information that is used to change the throttling IPC-threshold dynamically. With both recent history information (runtime DID value) and future estimation (compiler-time IPC estimation), we can capture the properties of the pipeline behavior more accurately than by using the hardware-based dynamic throttling or the compiler-based static throttling individually. Coordinating compiler-provided information and runtime DID values is crucial to achieving higher energy savings and at the same time avoiding high performance penalties.

3.1. The Algorithm

Instead of using a fixed IPC-threshold such as 2, CAFT dynamically changes the IPC-threshold between 2 to 5. Even when the estimated IPC is 2, we still selectively throttle based on the runtime DID value, instead of throttling whenever the IPC is below 2 as in [18]. The reason is that although the estimated IPC may be low in some cycles, we should not throttle if execution units are idle and waiting for incoming decoded instructions. If we do, throttling will impact the performance. On the other hand, we limit our highest IPC-threshold to 5. Our experiments show that throttling above an estimated IPC of 5 is very rare.

As mentioned above, we use the instruction Decode/Issue Difference (DID) to assist the IPC-estimation throttling

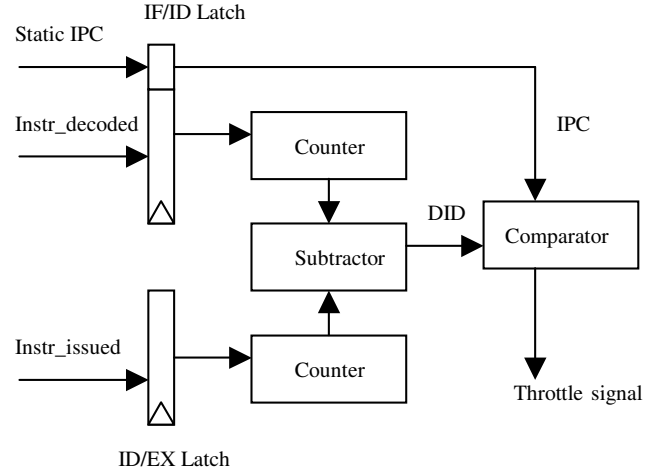


Figure 1. Hardware implementation for CAFT

technique to throttle at changeable thresholds. If the instruction decoding rate matches the instruction issuing rate (i.e., the DID value is zero), no fetch throttling is needed. If the DID value in the last cycle is greater than zero, which means that redundant instructions were decoded, there exist opportunities to throttle the fetch at the next cycle. Additional fetching will introduce the possibility of miss-fetching and increase the number of Icache accesses, resulting in a waste of energy. For example, if the DID in the previous cycle is 3 and the IPC estimate in the next cycle is less than 3, we can safely throttle for one cycle during instruction fetching. If the instructions left unused in the previous cycle can provide the needs of the next cycle, stopping fetching for one cycle will not hurt the performance. The algorithm can be summarized as follows:

```
IF Estimated_IPC ≤ DID
THEN throttle for one cycle
```

For different DID values, we throttle for all the estimated IPCs up to the DID value. The DID value captures dynamic effects such as cache misses and branch mispredictions, which are not captured by the fixed IPC-threshold compiler-based fetch throttling methods.

3.2. Architecture-level Implementation

The structure of our architecture-level design is similar to [18], which uses a fixed IPC-threshold. Estimated IPC values are inserted into the binary code and forwarded to the pipeline during decoding. It requires 2-3 bits to encode the estimated IPC values. If enough flexibility exists in the ISA of the target processor, this information can be encoded directly into the instructions, eliminating the need for a special instruction.

We show the architectural implementation of CAFT in Figure 1. The compiler-supplied estimated IPC value is identified and latched at the decode stage. We also add two counters to monitor the number of instructions decoded and issued in the previous cycle. The values of these counters are subtracted to calculate the DID, which is then compared

with the estimated-IPC latched at the decode stage. If the estimated-IPC is smaller than the DID, a fetch throttling signal is generated and transmitted to a clock-gater to stall fetching for one cycle [18].

3.3. Hardware-based Fetch Throttling

Dependence-Based (DEP) For comparison sake, we also implemented the hardware Dependence-based (DEP) scheme [3]. DEP inspects the instructions currently being decoded and counts the dependencies among them. Whenever the number of dependencies exceeds a pre-specified threshold, a throttling signal is triggered. The justification for this scheme is that a large number of dependencies is an indication of a long and probably critical computation path. Consequently, it is unlikely that prefetching additional instructions will significantly improve performance.

Instead of throttling for both the fetch and the decode stages as in [3], we only throttled the fetch stage, which makes it similar to our CAFT. Also, we compared the number of dependencies among the decoded instructions to the number of decoded instructions instead of to the decode-width as done in [3]. Comparing to the decode-width does not consider the fact that the number of decoded instructions in one cycle affects the number of dependencies in that cycle. Our experiments showed that these modifications will result in higher energy savings than the scheme in [3]. We obtained the best results when, once the number of dependent instructions among the decoded instructions was greater than half the number of decoded instructions, a throttle was triggered at the following cycle. We show the comparison of this approach with our CAFT approach in Section 5.

Just-In-Time Instruction Delivery (JIT) We next compared our CAFT method to the Just-In-Time (JIT) instruction delivery scheme [10] which is another hardware-based fetch throttling technique similar to CAFT. It uses information about *in-flight* instructions to control the front-end instruction fetching. When the number of *in-flight* instructions exceeds the MAXcount, instruction fetching is inhibited. The MAXcount value can be dynamically adjusted to the least value such that performance is not reduced by some threshold amount, e.g., 2%.

Because our processor configuration is different from that in [10], the tuning parameters in adjusting the MAXcount value were also different. After extensive experiments, we concluded that when the initial value of MAXcount is 32 and the MAXcount increment is 16 in every 100K instructions interval, the energy savings are maximal. With these parameters, our measured performance reduction was similar to that in [10] (i.e., 3%). Another difference is that when MAXcount was an “optimal” value, we only restarted tuning if the performance (IPC) changed by more than some “noise” margin. We did not consider branch changes as a reason for retuning, because in some benchmarks, considering branches as “noise” causes many unnecessary tunings, resulting in a substantial performance reduction.

Decode/Issue Difference (DID) In order to verify that the DID-directed CAFT is more efficient than any of

Processor Speed	1.5GHz
Process Parameters	0.18 μm , 2V
Issue	Out-Of-Order
IF, ID, IS, IC Width	8-way
Fetch Queue Size	32
Instruction Queue Size	128
Branch Prediction	2K entry bimodal
Int.Functional Units	4 ALUs, 1 Mult./Div.
FP Functional Units	4 ALUs, 1 Mult./Div.
L1 D-cache	16KB, 4-way, writeback
L1 I-cache	16KB, 4-way, writeback
Combined L2 cache	128KB, 4-way associative
L2 Cache hit time	20 cycles
Main memory hit time	100 cycles

Table 1. Baseline parameters

the individual schemes alone, we also tested the hardware-only Decode/Issue Difference (DID) technique. This technique assumes that insufficient parallelism exists when the number of instructions decoded exceeds the number of instructions issued, and continuing fetching will make the instructions stay longer in the issue queue wasting the wake-up and selection energy of the issue logic. In such a case, we can throttle the fetch.

In addition to the differences, we also tested different *ratios* between the numbers of decoded and issued instructions. The best results were obtained when the number of decoded instructions was twice the number of issued instructions. When we set the ratio to a lower value, the performance decreased rapidly. This is different from CAFT, which can throttle at a very low Decode/Issued Difference value if the IPC estimation in the next cycle is low. We show the numerical results in Section 5.

4. Evaluation Methodology

LARGE

The baseline architecture is described in Table 1. Our baseline processor configuration has 128 entries in its instruction queue; therefore we use a 128 element Register Update Unit (RUU). 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.

We used the SimpleScalar [6]/Wattch [5] framework to run the binaries and collect the energy results. We ran our baseline application without any annotations and without any throttling, and all other throttling versions were compared against this baseline. SimpleScalar has been modified to recognize the compiler-generated IPC flags and IPC values. In Wattch, we used the activity-sensitive power model with aggressive conditional clocking. The rationale for this choice was to compare our fetch-throttling framework to an unthrottled baseline that is already power-efficient. Wattch can be retuned for the 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 2-bit field

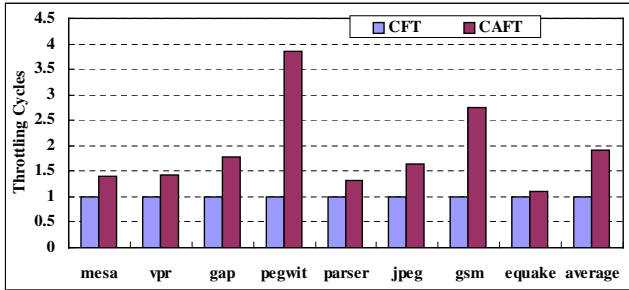


Figure 2. Normalized throttling cycles

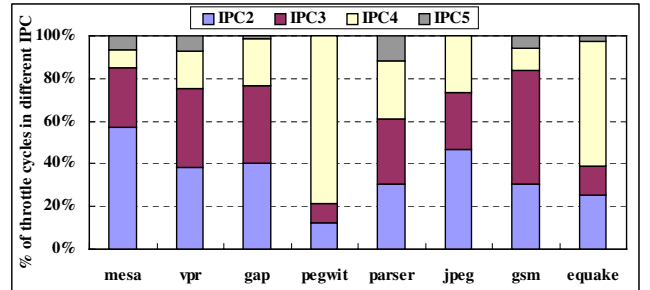


Figure 3. IPC distribution when throttling using CAFT

decoding in the dispatch stage and the comparison hardware logic.

We selected a mix of computation-bound and multimedia applications from the SPEC2000 [1] and Mediabench benchmark suites. We randomly chose four applications from each suite: *pegwit*, *gsm*, *jpeg*, *mesa* from Mediabench; *gap*, *parser*, *vpr*, *equake* from SPEC2000. We ran all Mediabench applications to completion. For the SPEC CPU2000 benchmarks we skipped past the initialization stage and simulated the next 500 million instructions using the reference input set. To skip the initialization phase, we fast-forwarded by the number of instructions as prescribed by Sair *et al.* [15] in their SPEC CPU2000 initialization segment analysis.

5. Results

In this section, we first show the throttling cycles and the IPC distribution when throttling using CAFT in the different benchmarks. Then, we present the execution time and energy results for different throttling schemes in different benchmarks. In explaining how fetch throttling can save energy, we also show the reduction in miss-speculated instructions and the distribution of the fetch width for each fetch operation. Finally, we show the EDP and ED²P for the different throttling techniques.

5.1. IPC Distribution

In order to show how many throttle cycles are added after changing the threshold adaptively, we counted the number of throttle cycles in different benchmarks for both CFT and CAFT, normalizing the number of throttle cycles of CAFT to that of CFT. The results appear in Figure 2. From this figure we can see that CAFT has more throttling cycles than CFT, especially for *pegwit*, where the value is up by 3.8x. More throttling cycles result in higher energy savings. On average, the total number of throttling cycles for CAFT is almost doubled compared to CFT.

The total number of throttling cycles increases substantially because CAFT can throttle fetching at higher estimated IPC values. In order to identify how many times CAFT throttles with an estimated IPC above 2, we analyzed the estimated IPC distribution when throttling, shown in Figure 3. From this figure, we can see that more than half of all the throttling cycles have an IPC which is above the threshold of 2, in all the benchmarks except *mesa*. In many

of the throttles the IPC is 3 or 4 but very few have an IPC of 5, since throttling which such a high IPC value may result in a significant performance penalty.

5.2. Execution Time and Energy

Figure 4 shows the execution time and energy consumption of five different fetch throttling schemes, normalized to the baseline without fetch throttling (No-Throttle). The schemes are: hardware dependence-based (DEP), just-in-time instruction delivery (JIT), decode/issue difference (DID), compiler-based fixed threshold (CFT) and our compiler-based adaptive scheme (CAFT).

First, we observe that in most cases, the hardware-based fetch throttling schemes (DEP, JIT and DID) have a longer execution time than the static IPC-estimation based fetch throttling technique. On average, DEP increases execution time by 4%, JIT increases execution time by 3.3%, while CAFT increases it by only 1.5%. For the *equake* benchmark the execution time is increased by more than 8% when using DEP, which is undesirable for high performance processors.

Hardware-based schemes cause a large performance loss because such techniques can only capture the history information and use the past behavior to drive fetch throttling. They assume that the program behavior in the past and the near future is stable, yet many programs exhibit irregular or bursty behavior which cannot be detected solely based on the past behavior. Although JIT can dynamically adjust the future MAXcount as a function of past program behavior, the tuning process itself may cause significant performance degradation. Also, such changes can be detected only after a large interval (e.g., 10K instructions).

Static IPC, on the other hand, is a compile-time estimation of the actual IPC based on program analysis, and thus it can provide an indication of a sharp change in ILP. CFT uses a fixed low IPC-estimation as the throttling threshold and has a small performance loss. For CAFT, although it can throttle at higher estimated IPCs and for more cycles, the performance loss is still low. The combination of DID and future IPC estimation can capture the program behavior more accurately than software- or hardware-only fetch throttling techniques. With the help of recent history information, dynamic effects like cache misses and branch misprediction will have a smaller effect on static IPC-estimation-based fetch throttling. Thus, when DEP cannot catch such bursty program phase changes, it causes a significant performance

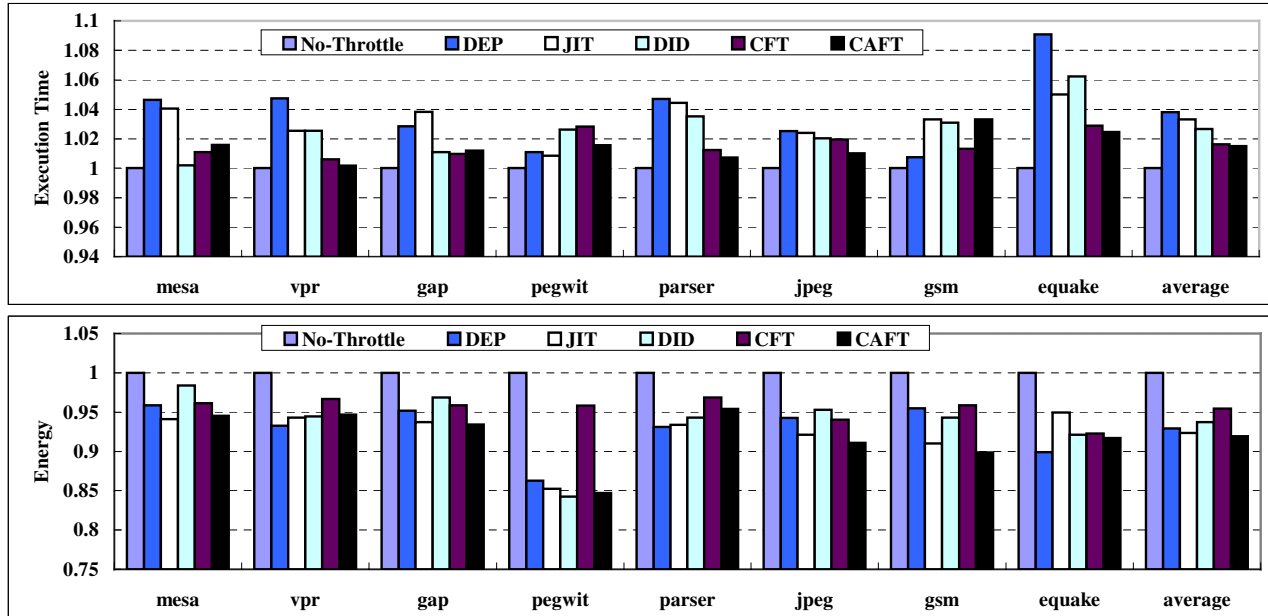


Figure 4. Normalized execution time and energy

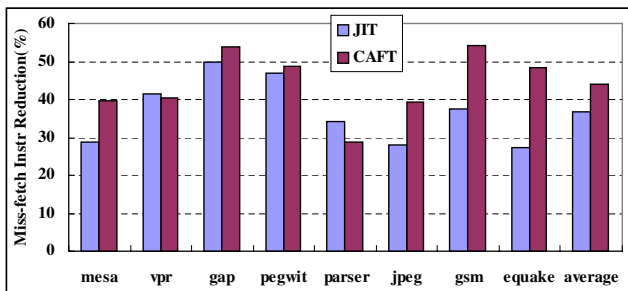


Figure 5. Reduction of miss-fetched instructions relative to non-throttling case

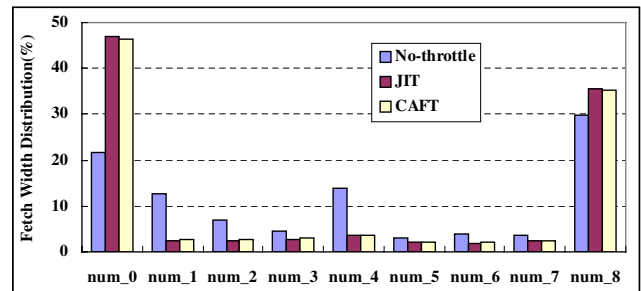


Figure 6. Fetch width distribution (averaged across all benchmarks)

decrease. This is especially evident in Figure 7 which shows that the ED²P for DEP is worse than for the No-Throttle baseline.

When comparing the normalized energy in Figure 4, CFT is the worst scheme, with only 4.6% total energy savings compared to the baseline. The reason is that CFT loses many throttling opportunities when the estimated IPC is high, due to its low IPC-threshold of 2. CAFT, which changes the throttling threshold adaptively based on the application characteristics, can throttle at a high estimated IPC and thus throttles more cycles than CFT. The use of DID information ensures that CAFT does not increase the execution time compared to CFT. CAFT achieves greater energy reduction than CFT and the total average energy reduction is almost 8%. CAFT also has higher energy savings when compared to the hardware-based fetch throttling schemes DEP, JIT and DID.

From Figure 4, we also observe that DID, when applied alone, cannot match the energy/performance benefits of CAFT. Like DEP and JIT, DID can not catch bursty pro-

gram phase changes. As a result, it has a larger performance decrease than the software-based schemes CFT and CAFT.

The increase in the number of throttling cycles is the main cause for the reduction in energy consumption in CAFT. A higher number of throttling cycles means a greater reduction in the number of fetched and executed miss-speculated instructions and a reduction in Icache accesses. With fewer instructions fetched, it not only saves Icache accesses energy, but also reduces the actions of forwarding instructions through pipeline stages, resulting in whole chip energy savings. As shown in Figure 5, the average reduction in miss-fetched instructions in CAFT is near 45% relative to the non-throttle scheme, while JIT achieves only a 36% reduction in miss-fetched instructions. Fetch throttling can greatly reduce the unnecessary miss-fetched flushes and save energy in different pipeline stages. Also, with a higher number of throttling cycles compared to CFT, CAFT can greatly reduce the number of Icache accesses and cause the number of instructions in each Icache access to be either 0 or 8 most of the time. The reason is that the number of available entries of the fetch buffer determines the number of instruc-

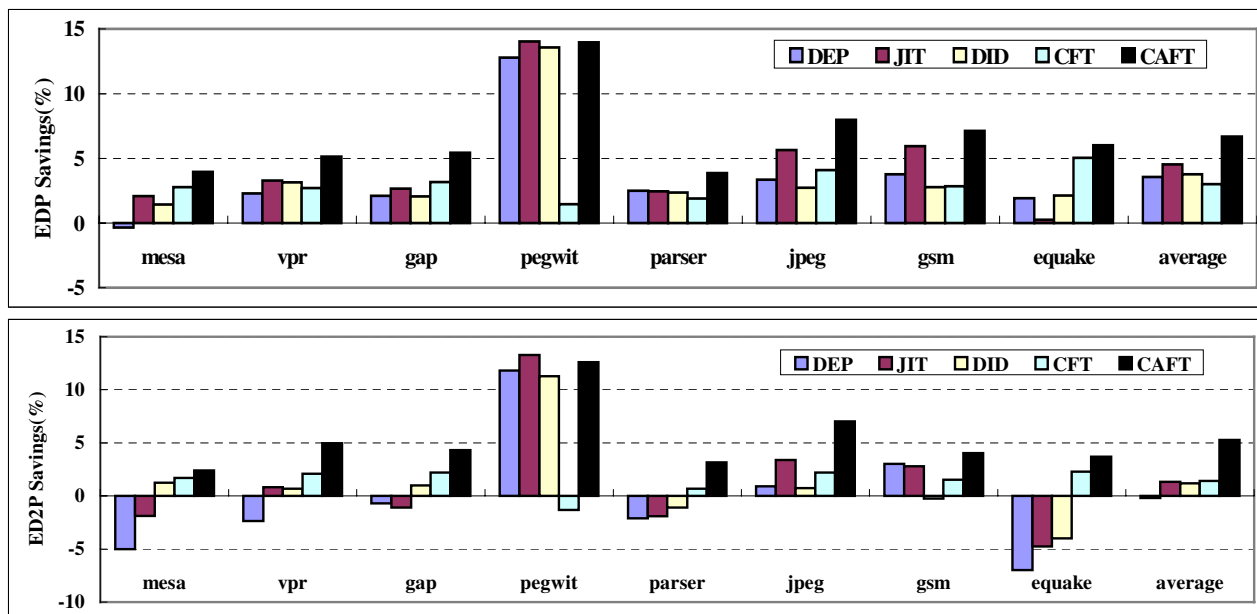


Figure 7. Energy-delay product and energy-delay² product savings

tions that can be fetched. Fetch throttling can cause the fetch buffer to be drained, allowing it to accommodate a full fetch (i.e., 8 instructions) fairly often. With the same number of instructions to run, the increase in the number of full fetches will decrease the number of total Icache accesses. From Figure 6, we can see that for CAFT and JIT the number of fetches is most of the time either 0 or 8 while in the non-throttle case the number of instructions fetched is distributed between 1 to 7 half of the time.

5.3. EDP/ED²P

In order to highlight the impact on performance, we show the EDP and ED²P savings in Figure 7. Although CAFT has no significant energy savings relative to DEP and JIT, the EDP reduction is significant due to the smaller performance decrease. Similarly, although CAFT has the same low performance loss as CFT, the EDP reduction for CAFT is larger because it saves more energy. CAFT is thus more beneficial than software- or hardware-only fetch throttling techniques when EDP is used as the metric.

As shown in Figure 7, DEP achieves 3.5% EDP reduction and CFT achieves 3% reduction on average. The EDP savings of CAFT are more than doubled for several benchmarks compared to either DEP (e.g., *mesa* and *jpeg*) or CFT (e.g., *pegwit* and *gsm*), averaging 6.7% reduction on all benchmarks. When considering ED²P, DEP has no benefit at all because of the high performance penalty. The other three schemes (JIT, DID and CFT) have only an average reduction of less than 2%, while the average ED²P improvement for CAFT is 5.3%.

6. Related Work

Prior related work can be divided into two groups: software-based techniques [8, 13, 18] and hardware-based

techniques [3, 7, 9, 10, 11].

Unsal *et al.* [18] propose a compiler-driven static IPC-estimation-based fetch throttling scheme that is based on dependence testing in the compiler back-end. This scheme throttles fetching at a low estimated IPC value. They fix the throttling IPC-threshold to 2 to lower the effect on performance, but with a low threshold the energy savings are not significant.

Mehta *et al.* [13] present the Fetch Halting technique that suspends instruction fetching when the processor is stalled by a critical long latency instruction. In order to characterize critical instructions, they use software-profiling techniques to annotate the critical load instruction. Then, if the hardware predicts L2 cache or main memory misses when executing a critical load instruction, fetch will be halted.

An early hardware-based front-end technique is the pipeline gating work of Manne *et al.* [11]. The authors inhibit speculative execution when such execution is very likely to fail. They analyze the likelihood of a branch 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.

Aragon *et al.* [9] also focused on reducing the power dissipated by mis-speculated instructions. They propose Selective Throttling as an effective way of triggering different power-aware techniques (fetch throttling, decode throttling or disabling the selection logic). For branches with a low confidence prediction, the most aggressive throttling mechanism is used whereas high confidence branch predictions trigger the least aggressive techniques.

An alternative front-end approach is the fetch/decode throttling proposed by Baniasadi *et al.* [3]. 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 mentioned by the authors,

traffic per pipeline stage is used as an indirect, and approximate, metric of power dissipation.

Karkhanis *et al.* [10] suggested throttling based on Just In Time Instruction Delivery. This scheme monitors and dynamically adjusts the maximum number of in-flight instructions in the processor. A counter for the number of in-flight instructions is incremented when an instruction is fetched, and decremented when an instruction is committed. Another register, MAXcount, sets the limit on the allowable in-flight instructions. Whenever the in-flight instruction count exceeds MAXcount, instruction fetching is stopped. The algorithm searches for the “optimal” number of in-flight instructions and changes the value of MAXcount at intervals of 100K committed instructions.

Buyuktosunoglu *et al.* [7] introduced an issue-centric fetch-gating scheme based on issue queue utilization and application parallelism characteristics. The issue queue utilization is obtained by tracking the occupancy of the issue queue and the application parallelism characteristics are obtained by monitoring from how deep in the Reorder Buffer (ROB) are instructions being issued. Fetching is stopped if over half of the instructions that were issued are located in the lower half of the ROB and the issue queue is at least half full.

7. Conclusions

Throttling at a fixed low IPC limits the capability to reduce energy for static IPC-estimation-based fetch throttling techniques. In this paper, we propose a Compiler-based Adaptive Fetch Throttling (CAFT) technique, which attempts to change the throttling IPC-threshold adaptively and still maintain a good performance. Compared to the previous fixed threshold approach (CFT) [18], we show that CAFT achieves a 3.7% additional EDP saving and 6.7% overall EDP reduction. In comparison with previous hardware dependence-based fetch throttling schemes (DEP and JIT), CAFT has a lower performance degradation and a higher EDP reduction.

References

- [1] The standard performance evaluation corporation. <http://www.spec.org>, December 2000.
- [2] W. Amme, P. Braun, F. Thomasset, and E. Zehendner. “Data Dependence Analysis of Assembly Code”. In *International Journal of Parallel Programming*, volume 28, number 5, pages 431–467, 2000.
- [3] A. Baniasis 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*, pages 16–21, 2001.
- [4] R. Bhargava and L. K. John. Latency and energy aware value prediction for high-frequency processors. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 45–56, New York, NY, USA, 2002. ACM Press.
- [5] D. Brooks, V. Tiwari, and M. Martonosi. “Wattch: A Framework for Architectural-level Power Analysis and Optimizations”. In *Proceedings of the 27th Annual International Symposium on Computer Architecture, ISCA-27*, pages 83–94, 2000.
- [6] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.
- [7] A. Buyuktosunoglu, T. Karhanis, D. H. Albonesi, and P. Bose. “Energy Efficient Co-adaptive Instruction Fetch and Issue”. In *Proceedings of the 30th Annual International Symposium on Computer Architecture, ISCA-30*, pages 147–156, San Diego, CA, May 2003.
- [8] T. M. Jones, M. F. P. O’Boyle, J. Abella, and A. González. “Software Directed Issue Queue Power Reduction”. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture, HPCA-11*, pages 144–153, 2005.
- [9] J. G. Juan Aragon and A. Gonzalez. “Power-aware Control Speculation through Selective Throttling”. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture, HPCA-9*, pages 103–112, 2003.
- [10] T. Karkhanis, J. E. Smith, and P. Bose. “Saving Energy with Just in Time Instruction Delivery”. In *Proceedings of the 2002 international symposium on Low power electronics and design, ISLPED '02*, pages 178–183, 2002.
- [11] S. Manne, A. Klauser, and D. Grunwald. “Pipeline Gating: Speculation Control for Energy Reduction”. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA-25*, pages 132–141, Barcelona, Spain, 1998.
- [12] R. Maro, Y. Bai, and R. I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *PACS '00: Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 97–111, London, UK, 2001. Springer-Verlag.
- [13] N. Mehta, B. Singer, R. I. Bahar, M. Leuchtenburg, and R. S. Weiss. “Fetch Halting on Critical Load Misses”. In *22nd IEEE International Conference on Computer Design, ICCD-04*, pages 244–249, 2004.
- [14] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [15] S. Sair and M. Charney. “Memory Behavior of the SPEC2000 Benchmark Suite”. Technical report, IBM T.J. Watson Research Center, 2000.
- [16] M. Smith. Extending suif for machine-dependent optimizations. In *Proc. First SUIF Compiler Workshop*, Jan. 1996.
- [17] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. “Dynamic Prediction of Critical Path Instructions”. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture, HPCA-7*, pages 185–196, 2001.
- [18] O. S. Unsal, I. Koren, C. M. Krishna, and C.A.Moritz. “Cool-fetch: Compiler-enabled Power-aware Fetch Throttling”. In *IEEE Computer Architecture Letters*, volume 1, pages 6–10, 2002.
- [19] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. Lam, and J. L. Hennessy. SUIF: A parallelizing and optimizing research compiler. Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.