Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA

Baruch Solomon, Avi Mendelson, Doron Orenstien, Yoav Almog, and Ronny Ronen
Intel Corporation
Intel Israel (74) Ltd., P.O. Box 1659, Haifa 31015, Israel.
{baruch.solomon, avi.mendelson, doron.orenstien, yoav.almog, ronny.ronen}@intel.com

ABSTRACT

We introduce the Micro-Operation Cache (Uop Cache – UC) designed to reduce processor's frontend power and energy consumption without performance degradation. The UC caches basic blocks of instructions – pre-decoded into micro-operations (uops). The UC fetches a single basic-block worth of uops per cycle. Fetching complete pre-decoded basic-blocks eliminates the need to repeatedly decode variable length instructions and simplifies the process of predicting, fetching, rotating and aligning fetched instructions. The UC design enables even a small structure to be quite effective.

Results: a moderate-sized UC eliminates about 75% instruction decodes across a broad range of benchmarks and over 90% in multimedia applications and high-power tests. For existing Intel P6 family processors, the eliminated work may save about 10% of the full-chip power consumption with no performance degradation.

General Terms: Performance, Design

Keywords: instruction fetch, instruction cache, micro-operation cache, power reduction.

1. INTRODUCTION

A processor frontend fetches stream of instructions from the memory hierarchy and supplies valid decoded instructions to the execution core. The frontend predicts which instructions should be fetched next, decode them, and move them with *low latency* and *high bandwidth* to the backend (renaming, execution and retire).

High bandwidth fetching and decoding is a challenge for every microprocessor. It is extremely challenging for the IA-32 processor family, which features a variable length Instruction Set Architecture (ISA). An IA-32 instruction length may vary between 1 to 15 bytes; it may reside in any byte address, and it may be translated into one or more *micro-operations* or *uops* (e.g., an add-register-to-memory instruction consists of a separate load, add and store micro-operations). A complex logic is needed to maintain a decoding rate of several variable length instructions per cycle. The IA32 frontend lasts several pipe-stages and consumes about 28% of the overall processor power [Mann98].

Traditional processors use *instruction caches* (IC) to store instructions. In the race for higher performance, several attempts have been made to improve instruction fetch and decode. The main strategy for increasing the fetch bandwidth is to use auxiliary structures to store instructions in their program execution order rather than in their memory address order. Most known structures are the Trace Cache (TC) [Pele94, Rote96, Frie97, Upto00], the Basic Block Cache (BBC) [Blak99], and the eXtended Block Cache (XBC) [Jour00][1]. These novel structures attempt to dynamically create long instruction sequences. Instructions are first fetched from memory in the traditional way. Later, instructions are grouped into traces according to their execution order; heuristic is used to decide trace start and end points. Finally, the collected traces are assembled and stored in the auxiliary structure. Later on, when stored instructions are needed, they are fetched from the auxiliary structure.

These structures are not designed for power efficiency. Indeed, to reduce latency they cache already decoded uops so they avoid repeated decoding and save power. But, they also involve complex logic that consumes a lot of power. This logic implements mechanisms such as multiple-branch predictors, big caches, trace ending heuristics and more.

Our goal is to provide an alternative frontend for the Intel P6 processor family that delivers competitive fetch bandwidth as existing P6 family processors at lower power consumption. The *Micro Operation Cache* (Uop Cache, UC) is the base of this frontend. The UC, as the XBC and the TC, avoids repeated decoding by caching already decoded micro-operations. The UC works on basic blocks, not traces, so its logic is simpler and consumes less power than XBC and TC.

The remainder of the paper is organized as follows. Section 2 describes the UC structure and algorithms and lists the design space alternatives. Section 3 and 4 explain the experimental methodology and bring the experimental results: comparing the UC to the current IC and exploring design space alternatives. Section 5 analyzes the power consumption and estimates the power saving. Finally, we conclude in section 6.

[1] [jour00] includes a good overview of instruction fetch structures.

2. UOP CACHE STRUCTURES

2.1 The P6 Microarchitecture – The Frontend

The frontend pipeline of the P6 family consists of several pipe stages that accomplish the following tasks:

- Determining the next Instruction Pointer (IP) to fetch from.
- *IC tag lookup*.
- *IC data fetch*, including aligning and rotating as needed.
- *Instruction length decoding*.
- *Instruction decoding* – translating instructions into micro-operations (uops).

Instructions are split into uops. Simple instructions consist of up to 4 uops; *complex instructions* consist of 5 uops or more. Complex instructions have their uops come from the micro-sequencer that extracts them out of a special micro-code ROM at the rate of 3 uops per cycle.

Current implementation of the P6 family can sustain execution rate of 3 uops per cycle. The enhanced frontend should maintain at least this fetch rate.

2.2 Uop Cache Overview

We define *basic block* as a *single-entry* / *single-exit* sequence of instructions. The Uop Cache (UC) stores basic blocks of instructions pre-decoded into uops. The block is mapped into the UC according to the address of its first instruction. To simplify the UC structure, a basic block may be broken into fixed length UC lines - each contains a fixed number of uops slots. Some slots contain active uops, while other may remain empty. A basic block can span over one or more UC lines.

2.3 Uop Cache Data Structure

The UC is a regular *n*-associative cache with tags and sets. Each set contains number of lines (as the number of ways) and each line contains a number of uops. Each line is addressed by the IP of its first instruction. This forces all uops originated from the same IA-32 instruction to reside in the same UC line. The additional data associated with each UC line includes:

- Number of valid uops stored in the UC line.
- The total length of the original IA-32 instructions that constitutes the basic block stored in this UC line.

This data is stored along with the UC tag array and is accessed during a cache lookup.

2.4 Basic Blocks

Basic blocks are entered only via the first instruction in the block. A new block is started following:

1. A control flow instruction: e.g., Branch (conditional or unconditional), Call, Return.
2. A complex instruction. In this case, a dummy uop, used by the micro-sequencer, is stored in the UC line.
3. An exception occurred in the course of instruction fetch (e.g. page fault).

2.5 Execution modes

The frontend operates in one the following two modes:

- *Build-Mode*: Fetching instructions from the IC, decoding them into uops, and storing them into the UC.
- *Stream-Mode:* Fetching uops from the UC.

Mode Switch occurs when moving from build-mode to stream-mode and vice-versa.

2.5.1 Build-Mode

Instructions fetched from the IC are decoded into uops. In parallel to their issue, the uops are also stored in the UC fill buffer. Uops from consecutive instructions are packed together to fill a UC line. After a UC line is built, it is placed in the UC. A new UC line is started for each new basic block, or when the current UC line does not have enough empty slots to store all the uops generated by the coming instruction.

UC lines are not fully utilized. The build procedure may leave empty uop slots in the UC line. In addition, the UC may suffer some level of redundancy - this happens when control flow leads to an instruction that is already in the UC, but not as the first in its UC line. Fortunately, these inefficiencies do not hurt the UC performance.

2.5.2 Stream-Mode

When in stream-mode, after reading a UC line, a new IP is computed by adding the previous IP with the length of the instructions stored in that line (see 2.3 above). A new UC lookup is done using the new computed IP.

2.6 Uop Cache Pipeline

The integration of the UC within the P6 microarchitecture pipeline is illustrated in Figure 1. The heavy shaded blocks are the new UC related added blocks, the light shaded and white blocks are a slightly modified blocks of the existing P6-like microarchitecture.

The main motivation of this pipeline is to ensure that there is no bubble on a switch from either stream-mode to build-mode (i.e. on a UC miss) or build-mode to stream-mode. First, we split the lookup and fetch into two separated stages for both the IC and the UC, making both caches serial. The IC and the UC lookups are done in the first stage, and based on the hit/miss indicator we decide which way in which cache should be accessed for actual instructions/uops fetch.

To save power, the actual fetch of uops from the UC is done only one stage before storing them in the uop buffer, and the new added latches hold only the location (set and way numbers) of the relevant UC line.

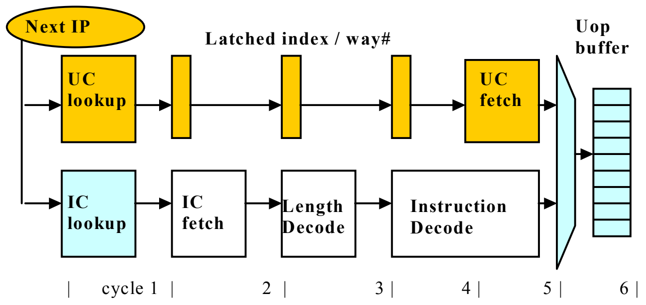

Figure 1: Frontend Pipeline with Uop Cache

A closer look at the frontend pipeline in Figure 1 shows:

- There is no bubble when switching from a stream-mode to a build-mode, and vice versa.
- All blocks in the white area can be shutdown when working in stream-mode.

Shutting down the IC lookup during stream-mode can save more power, but at the cost of a bubble while switching modes.

For simplicity, the UC and IC share the same branch prediction unit (BPU). To save power, the UC could have used a separate, yet simpler, BPU that predicts only one branch per cycle.

2.7 Design space

The prime target of the UC is to reduce the power of the frontend while providing enough uop bandwidth. We target uop fetch bandwidth of at least 3 uops per cycle. There are several design alternatives that can be considered for the UC, each affecting performance, bandwidth, area and power:

- Cache size, associativity and line size: these parameters influence hit rate, cache space utilization, switch rate and output bandwidth. Larger cache size and high associativity increase hit rate and reduce switch rate. Larger line size increases uop bandwidth. Each of these parameter influences the UC power. Higher associativity increases the UC lookup power; longer lines consume more power on UC fetch.
- Usage of "access counters". An optional counter is attached to each IC line. Instructions are stored in the UC only when the counter of the corresponding UC line reaches a certain predefined value. Access counters significantly reduce the number of line-builds and line-replacements in the UC.
- The targeted average output bandwidth of the frontend can make a big difference in the UC design. Our target is to sustain a fetch rate of at least 3 uops per cycle.

The following parameters affect performance, bandwidth, and power as well, but in order to narrow the design space, we choose a certain value for each and used it for the studies presented here.

- Serial or parallel caches. Whether cache lookup and cache fetch are done in the same cycle, or split into 2 separate cycles. Serial caches consume less power but require an extra cycle in the frontend pipeline – slightly increasing the misprediction penalty. We assume that both the UC and the IC are serial.
- Zero/Non-zero "Stream to Build" switch penalty. Whether IC and UC lookups are done in parallel, or, while in stream-mode, the IC lookup is done only after a UC miss. The latter allows complete shutdown of the IC while the UC operates in stream-mode, but incurs a one-cycle bubble on each stream-mode to build-mode switch. Note that this feature is orthogonal to the serial/parallel cache feature. We assume zero-switch penalty.
- The mapping between the IP and the UC. This determines the memory block size mapped to a single set in the UC. Bigger offset causes more uops to map to the same UC set. It takes higher UC associativity or longer UC lines to support that. We assume 16 bytes memory blocks.

3. EXPERIMENTAL METHODOLOGY

3.1 Machine Model

The results were obtained using a trace-driven, stand-alone, frontend simulator that models the Uop Cache, Instruction Cache, a decoder, and a micro-sequencer. We assume one line fetched from the UC and the IC per cycle. IC fetches are 16 bytes long. We also assume a perfect IC and a perfect branch predictor[2].

Complex instructions are fetched from a micro-sequencer at a rate of 3 uops/cycle. A dummy uop points from the UC to the micro-sequencer and takes one uop slot. The model assumes a frontend pipeline similar to the one depicted in Figure 1 above. The model assumes serial IC and serial UC.

3.2 Benchmarks

The model process instruction traces. Each trace consists of 30 million consecutive x86 instructions translated into uops. Traces record both user and kernel activities. Results are reported for 49 traces grouped into 8 suites:

- *SpecInt:* 9 traces from the SPECint2000 benchmark.
- *SpecFP*: 9 traces from the SPECfp2000 benchmark.
- *Win2K:* 6 traces from Windows2000 benchmark.
- *WinSt99:* 7 traces of from Winston99 benchmark.
- *Smark98NT:* 9 traces from SYSmark32 benchmark.
- *WB99_3D:* 3 traces of popular 3D games.
- *MM99:* 3 traces of video processing with MMX.
- *HighPower:* 3 traces of a power virus application.

4. DESIGN SPACE EXPLORATION

This section explores the design space of the UC. UC size (s), line size (l), associativity (w), and access-counter values (ac) are explored.

4.1 Measured Parameters

Several parameters that affect power and performance are measured:

- *Instruction hit-rate*: The number of instructions fetched from the UC relative to the total number of fetched instructions. A complex instruction is counted as one instruction. Instruction hit-rate correlates to the power saved by the reduction in decoding IA-32 instructions.
- *Line hit-rate*: measured as the reduction in the number of IC line fetches in a UC/IC system relative to the number of IC line fetches in a system without a UC. Line hit-rate correlates to the power saving gained by the reduction in IC lookups, fetches, line aligning and rotating.
- *Overall Fetch Ratio*: The number of line fetches (from both the UC and the IC) relative to the number of IC line fetches in an IC only system. Overall fetch rate correlates to the extra power required to perform UC lookups and fetches.

The last two items deserve an explanation: an IC line may contain one or more basic blocks. A single IC fetch in an IC only configuration is counted once. Fetching the same bytes in a combined IC/UC configuration may involve several UC fetches

[2] Perfect BPU can be assumed since we fetch one block per cycle. Studies involving fetching multiple blocks (e.g. TC) cannot assume that.

and possibly an IC fetch. The latter occurs in case only part, not all, of the instructions within an IC line reside in the UC.

- *Uop bandwidth*: The number of uop fetched from the UC per cycle. We
- count only uops coming directly from the UC or uops of complex instructions whose pointer reside in the UC. We do not count uops coming from the IC. We divide this number by the number of stream-mode cycles. This method makes sure we evaluate a real, pure, UC potential.
- *Number of Builds*: This number correlates to the extra power required to perform UC line builds and replacements.
- *Number of Switches*: The number of switches from stream-mode to build-mode (and vice-versa). The number of switches correlates to the lost performance in case of non-zero switch mode penalty.

4.2 UC size, UC line size and UC associativity

A basic block can span over several UC lines, sharing the same UC set. Several basic blocks can be mapped to the same UC set. This should be taken into account when dealing with line size and associativity.

Increasing the UC line size is expected to increase the UC fetch bandwidth. However, bigger line size may increase the number of empty uop slots, thus reducing the UC hit rate for a given UC size. Since a non-complex instruction can have up to 4 uops, the minimum possible UC line size is 4. However, the minimal practical UC line size is 5. Having 5 uops per line ensures that a block that spans over more than one UC line delivers at least 3 uops per cycle, while a 4 uops UC line size may deliver as low as 2.5 uops per cycle (e.g. 3 uops in the first UC line and 2 in the second).

Higher associativity is expected to increase hit rate by reducing the conflict misses. Larger UC decreases the number of capacity misses, thus improving the overall hit rate.

Figure 2 and Figure 3 show the *Line hit-rate* and the *Instruction hit-rate* of different UC configurations. 64 sets are used for all configurations. The number of ways and the UC line length are specified for each column (*w, l*: associativity, line size).

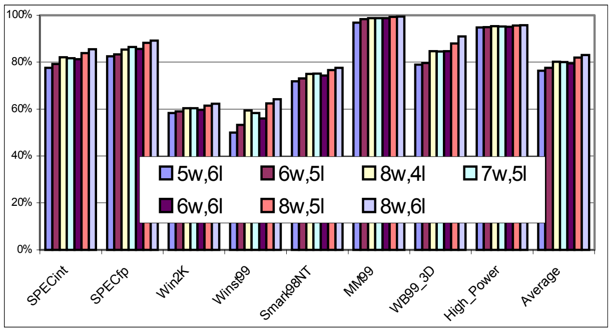

Figure 2: UC Line Hit-Rate

As expected, the instruction hit-rate is somewhat higher than the line hit-rate. By and large, bigger UC size (*64×w×l*) increases hit rate, but bigger line size reduces it (e.g., 6*w*,5*l* is better than 5*w*,6*l* and 7*w*,5*l* is better than 6*w*,6*l*). The average instruction hit rate is in the range of 80-85%.

Hit rate trends in our configuration follow the hit rate trends of traditional IC (not shown). For example, Win2K and Winst99, which are notorious for having a rather big working set, exhibit lower hit rates.

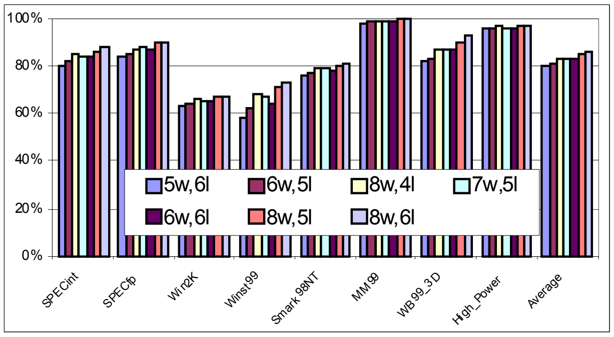

Figure 3: UC Instruction Hit-Rate

Figure 4 shows the overall fetch ratio. Both the IC fetches and UC fetches are shown, relative to an IC only configuration. On average the usage of UC decreases the number of IC fetches by about 80%, but increases the overall number of fetches (UC and IC) by 50% to 70%.

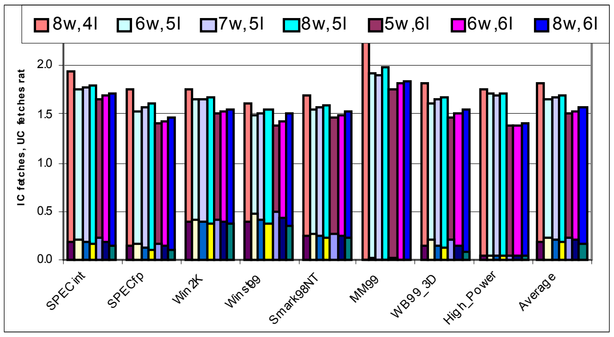

Figure 4: UC Fetch Ratio

Figure 5 shows the fetch bandwidth for various cache configurations. As expected, the bandwidth depends mainly on the UC line size. The results show that in order to sustain a fetch bandwidth of over 3 uops per cycle, a 6-uop UC line size is recommended. Overall, the (*6w,6l*) configuration seems to be the sweet spot that provides a reasonable hit-rate and an acceptable bandwidth for its size.

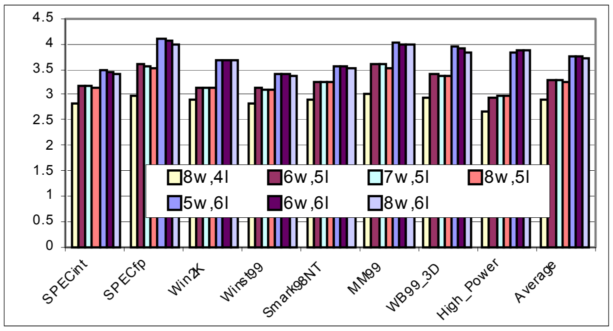

Figure 5: UC Bandwidth (uops/cycle)

Figure 6 shows the switch rate, measured as number of uops per switch. The number of uops per switch correlates well with the UC instruction hit rate. The switch rate helps us approximate the loss of using non-zero switch penalty (explaining why we measure and present it in uops per switch). For example, assuming overall fetch bandwidth of about 3 uops per cycle, an average switch every 100 uops means a stalled cycle every 33 cycles, or potential performance loss of ~ 3%.

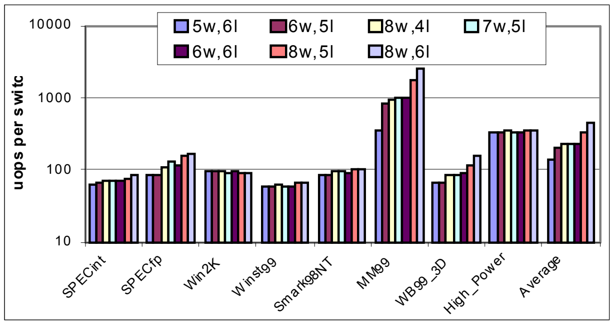

Figure 6: UC stream-to-build Switch Rate

4.3 Access counters

Many UC lines are accessed only when they are built and are replaced without being re-executed even once, wasting power to build them and throwing potentially usable UC lines. By applying access counter filter when storing lines into the UC, we can decrease the number of such "wasted" builds.

Figure 7 shows the reduction in the number of line builds for various access counter values. Results are recorded as the number of builds per 1K instructions.

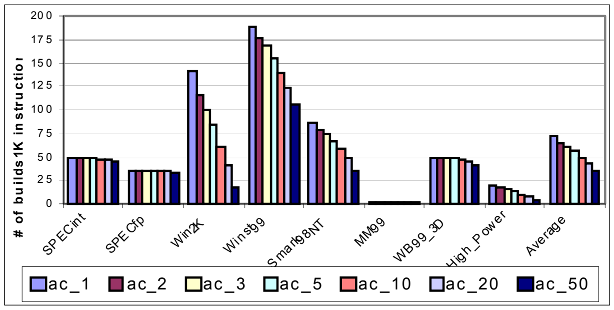

Figure 7: UC line Builds vs. Access counter (for 6w,6l)

Figure 8 shows the impact of various access counters on the hit rate. As expected, bigger access counter values decrease the number of line builds, but also reduce the hit rate (when taking high enough access counter values). Different access counter values have no impact on the overall fetch bandwidth (not shown). Access counters have minimal impact on applications with small working sets (e.g., SpecInt, SpecFP). They significantly decrease the number of builds for application with large working sets (Win2K, Winst99) at a small decrease in the hit-rate. For example, with access counter of 10, Win2K enjoys over 2X reduction in UC line builds for small decrease in the line hit-rate (58.5% vs. 60%).

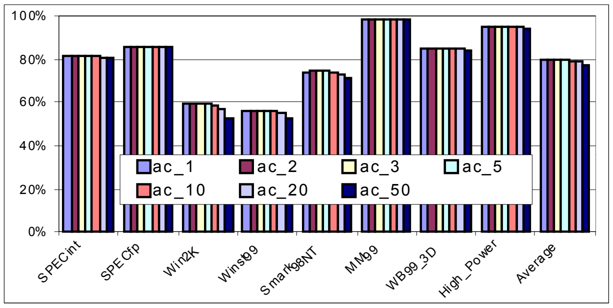

Figure 8: UC line Hit-rate vs. Access counter (for 6w,6l)

The above numbers may deceive. The benefit of the access counters should be assessed based on the actual power saved by eliminating line builds against the extra power required by the additional IC accesses. This benefit depends on the power needed to build a line and the power needed to fetch an IC line and decode its instructions.

5. FRONTEND POWER REDUCTION

A full-blown power estimate requires detailed power breakdown that is not fully available and is out of the scope of this paper. Nevertheless, we will show that with reasonable assumptions, a UC has a significant power saving potential.

According to [Mann98], the Pentium® Pro processor consumes about 14% of its power in instruction fetch and length decode, and another 14% in instruction decode.

The UC contributes to the frontend power in various ways:

1. In stream-mode, the power hungry Instruction Length Decoder and the Instruction Decoder are inactive.
2. IC fetch is replaced by a serial UC fetch.
3. Converting the IC into a serial cache saves power even when in build-mode.
4. Usage of access counters reduces the average power needed to build UC lines.

On the other hand, using the UC may add power since:

1. IC lookups are done in parallel to UC lookups.
2. We service about 50% more line fetches because UC lines are effectively shorter.
3. A bigger mux in front of the uop buffer is needed.

5.1 Frontend Power Model

We examine the power requirements in build-mode and stream-mode.

1. Stream mode. The following units are active:
   (a) UC lookup
   (b) Serial IC lookup
   (c) UC line fetch
   (d) Uop store into the uop buffer.
   The rest of the units are inactive.
2. Build-mode. The following units are active:
   (a) Serial IC lookup
   (b) Serial IC line fetch
   (c) Instruction Length Decode
   (d) Instruction Decode and uop store into the uop buffer
   (e) UC line fill and replace - as needed
   (f) UC lookup (following a control flow instruction).

The power consumed by the micro-sequencer when fetching instructions out of the micro-code ROM is independent of the UC usage and is not counted. Indeed, the presence of many complex instructions may limit the UC power saving potential.

5.2 Frontend Power Numbers Elaboration

An IC lookup and IC fetch are done for each accessed IC line. Instruction length decoding and instruction decoding are more frequent as they are done for each instruction. Out of the power spent in the frontend, only about 25% is spent on IC lookup and fetch. The rest is spent on instruction length decoding and instruction decoding.

The above activities (except IC lookups) are not needed when UC lines are fetched. There are 50% more fetches in the IC/UC configuration relative to IC only configuration (see Figure 4), but the work on each UC line and on each uop coming from it is much smaller.

Our back of the envelope power estimate is based on the information we gathered in section 4 and the following assumptions:

1. Fetching uops from the UC and storing them in the uop buffer consume together about the same power as an IC fetch (IC and UC areas are assumed to be similar).
2. UC lookup and serial IC lookup consume much less power than UC fetch, hence their power can be neglected.

Assuming about 80% *line hit-rate*, and taking into account (2) above we conclude that build-mode power consumption decreases to about 20% of the original frontend power.

Based on (1) above, and given about 50% overall added fetches, we conclude that the UC consumes 50% more power than the original IC fetch power. That is 25%*1.5=37.5% of the original frontend power.

Overall, the IC/UC based frontend consumes less than 60% (37.5%+20%) of the original frontend power. Using serial IC and UC caches provides additional saving, which offsets the fact that we have ignored the cost of UC line builds and other small activities. The real picture is even better – the power demanding applications exhibit high UC hit rates (close to 90%) that results in bigger power saving when most important.

Since the frontend consumes 28% of the overall processor power consumption,, saving 40% of it constitutes about 10% of the full chip power (40%*28% = 11%).

6. CONCLUSIONS AND FUTURE WORK

In this paper we presented the Uop Cache (UC), an alternative instruction supply mechanism aimed at power reduction. We showed that the UC provides a competitive fetch bandwidth as an existing microarchitecture based on a conventional instruction cache (IC), while consuming significantly less power. As opposed to more aggressive novel instruction supply mechanisms, such as trace cache, even a small UC with moderate instruction hit-rate is still attractive in terms of performance and power. We have shown that the UC may save over 40% of the frontend power, which equals to over 10% of the full chip power.

The concept of the UC can be extended in several directions:

- Refine the Power Estimation of the Frontend including the IC and UC possible implementations.
- Tuning the sizes of IC and the UC for optimal power and performance. It may be beneficial to use part of the transistor budget to increase the IC size. This reduces the number of L1 misses and decreases the total energy consumption.
- Allow multiple-exit basic blocks. Multiple-exit blocks may contain the fall-through instructions following a conditional branch. They reduce the number of very short blocks, which decrease the overall fetch bandwidth.
- Enhance the UC efficiency. For example, UC continuation lines (lines that do not start basic blocks) may be stored in a separate structure rather than consuming UC lines. This way, long basic blocks do not consume several ways in the same set, thus reducing the rate of UC conflict misses.
- Annotate the uops in the UC lines to simplify other repeated operations (e.g., provide intra-block renaming information along with the uops).
- Increase overall fetch bandwidth. For example, fetching together all UC lines that belong to the same basic block.

Actual directions will be mainly influenced by the final goal: more bandwidth for the same power or same bandwidth for less power?

7. ACKNOWLEDGMENTS

We would like to thank Bob Valentine for his helpful ideas for integrating the UC into the frontend. We want to thank Daniel Baumgarten for his help in implementing the UC simulator.

8. REFERENCES

[Blac99] B. Black, B. Rychlik, and J. Shen, "The Block-based Trace Cache," in Proceedings of the *26th International Symposium on Computer Architecture*, May 1999.

[Frie97] D. Friendly, S. Patel, and Y. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism," in *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[Jour00] S. Jourdan, L. Rappoport, Y. Almog, M. Erez, A. Yoaz, R. Ronen, "eXtended Block Cache", in Proceedings of the *International symposium on High Performance Computer Architecture*, January 2000.

[Mann98] S. Manne, D. Grunwald, A. Klauser, "Pipeline gating: Speculation control for energy reduction", in Proceedings of the *25th International Symposium on Computer Architecture*, June 1998.

[Pele94] A. Peleg and U. Weiser, "Dynamic Flow Instruction Cache Memory Organized around Trace Segments Independent of Virtual Address Line*," U.S. Patent Number 5,381,533, Intel Corporation*, 1994.

[Rote96] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching**,**" in Proceedings of the *29th International Symposium on Microarchitecture*, November 1996.

[Upto00] Mike Upton, "The Intel Pentium® 4 Processor", http://www.intel.com/pentium4, October 2000.