

# Power-efficient Instruction Delivery through Trace Reuse\*

Chengmo Yang and Alex Orailoglu  
Computer Science and Engineering Department  
University of California, San Diego  
9500 Gilman Drive, La Jolla, CA 92093  
{c5yang, alex}@cs.ucsd.edu

## ABSTRACT

*As power dissipation inexorably becomes the major bottleneck in system integration and reliability, the front-end instruction delivery path in a traditional out-of-order superscalar processor needs to deliver high application performance in an energy-effective manner. This challenge can be addressed by efficiently reusing the work of fetch and decode performed during preceding loop iterations and resident mostly within the processor itself. As a large percentage of the instructions currently under fetch have previously dispatched copies resident in the **Reorder Buffer (ROB)**, in this paper we develop a mechanism to utilize the ROB as a storage location for previously decoded instructions. Thus instructions can be fed directly from the ROB into the rename and issue stages, enabling the gating off of the fetch and decode logic for large periods of time so as to deliver significant power savings. Power and performance criticality of the ROB requires an efficient reuse identification mechanism; we outline such a cost-efficient **Reuse Identification Unit (RIU)** which enables effective identification of the matches between the ROB entries and the instructions currently under fetch. Simulation results on both multimedia and SPEC 2000 benchmarks confirm that incorporating the proposed technique on traditional out-of-order superscalar processors results in not only a slight improvement in performance, but also significant savings in the overall system power dissipation, achieved within a limited hardware budget.*

**Categories and Subject Descriptors:** C.1.3 [Processor Architectures]: Other Architecture Styles—*Adaptable architectures*

**General Terms:** Performance, Design

**Keywords:** Low-power design, adaptive processor, instruction delivery

## 1. INTRODUCTION

Power efficiency has already been well established as an important product quality characteristic for embedded processors, as they typically need to satisfy stringent constraints of

battery life and heat dissipation. General purpose processor design, on the other hand, has placed more emphasis on exploiting instruction-level parallelism (ILP), resulting in power efficiency being addressed mainly at the technology level, through lower supply voltages, smaller transistors, silicon-on-insulator (SOI) technology, etc. Nevertheless, the continuing advances in semiconductor technology have caused power dissipation to become the major issue in current system integration and reliability. In order to control packaging/cooling costs and to prevent chip malfunctions caused by thermal issues [1], microarchitecture designers must also take power constraints into consideration.

The increasing sensitivity to power consumption forces a reevaluation of traditional performance focused design decisions for general-purpose processors, especially for superscalar processors with dynamic scheduling capability. More specifically, one of the most complex and power consuming components in an out-of-order superscalar processor is the *front-end* instruction delivery path, consisting of the fetch, decode, register access and rename, as well as the issue stages. Because the rate and accuracy at which instructions enter the pipeline set an upper limit to sustained performance and determine the efficiency of energy consumption [2], the front-end is usually forced to perform instruction fetch and decode in the peak bandwidth, as early as possible, by making use of sophisticated branch prediction algorithms. According to data reported in [2], such an aggressive instruction delivery strategy consumes about 35% of the total processor power. However, such power consumption is unnecessarily elevated, as this aggressive strategy not only brings in a lot of useless instructions on mis-speculated paths, but also causes instructions to spend many needless cycles in the issue queue waiting for dependencies to be resolved. Accordingly, previous power-aware instruction delivery techniques have focused on either reducing mis-speculated fetches, such as pipeline gating [3] based on confidence estimation, or preventing mismatches between the speeds of decode and commit, such as issue queue adaptation [4, 5] and instruction flow-based fetch [7, 8] or decode [6] gating.

Focusing solely on branch prediction and issue queue design falls short of achieving significant power savings, since each instruction still needs to go through all the stages (fetch, decode, rename and issue) on the instruction delivery path. In other words, to achieve more aggressive energy savings, for a large fraction of the total issued instructions, the work performed by the front-end should be reused through bypassing one or more stages on the instruction delivery path. Previously proposed techniques usually employ small caches to achieve work reuse. These caches can be placed either be-

\*This work is supported in part by NSF Grant 0082325.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PACT'06, September 16–20, 2006, Seattle, Washington, USA.  
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

tween the processor and the L1 cache [9, 10], or after the decode stage to store decoded instructions [12], or even more aggressively after the issue stage to store issued instructions [2]. While particular shortcomings for each technique, detailed in Section 2, exist, these reuse techniques also share the common shortcoming of introducing non-negligible hardware and performance overhead, as these dedicated caches need specific address mapping mechanisms and impose sizable latency in the case of cache misses.

In this paper, instead of adding and searching a dedicated cache, work reuse is achieved more efficiently through the use of information associated with in-flight instructions. By noticing that all the in-flight instructions have their information stored within the processor, more precisely, in the *Reorder Buffer* (ROB), we propose to utilize the ROB as a storage location for previously decoded instructions. Because the functions performed by the fetch and decode stages are identical each time a specific trace from the program is executed, we propose to gate off the fetch and gating unit when a trace is reexecuted, and deliver a corresponding previously decoded copy from the ROB directly into the rename stage. Since the ROB is a critical component that affects the overall performance of the out-of-order processor, we also propose an efficient approach to thread the *basic blocks* stored in the ROB into a cost-efficient *Reuse Identification Unit* (RIU). As a result, matches between the fetching trace and a previously decoded copy can be identified efficiently by searching the power-efficient RIU only upon encountering a change in control flow, instead of searching the whole ROB for each given fetch address. The small RIU furthermore enables the next branch address to be calculated as soon as a match has been identified, thus completely eliminating the power hungry BTB lookups for non-branch instructions normally occurring at every execution cycle.

The remainder of this paper is organized as follows. Section 2 reviews previously proposed techniques for power-aware instruction delivery. Section 3 examines the opportunities and challenges of reusing the information stored in the ROB to achieve power reduction. Sections 4 and 5 respectively present the proposed reuse identification and instruction delivery mechanisms. Our framework is evaluated both theoretically and experimentally in Sections 6 and 7, respectively. Section 8 summarizes this paper.

## 2. PREVIOUS WORK

As sensitivity to power consumption becomes one of the defining challenges of out-of-order superscalar processor design, increasing research attention is getting paid to the issue of power/performance tradeoff exploration of instruction delivery mechanisms.

One class of techniques achieves power reduction through gating off one or more pipeline stages on the instruction delivery path. Their goals consist of avoiding energy inefficiencies caused by incorrect control flow speculation, as well as making the instruction delivery more demand-driven. More specifically, instruction delivery can be postponed if a branch prediction confidence estimator detects a large probability of wrong path execution [3], or if instructions sent to the decode unit cannot fully fill the decode width [6], or if the number of in-flight instructions exceeds an adaptive upper bound [7]. The limitation here is that incorrect estimations made by the heuristics may cause starvation of useful instructions in the pipeline back-end, thus degrading performance significantly.

Because a large fraction of the renamed instructions are stalled in the issue queue waiting for dependencies to be resolved, a significant amount of research attention has also been paid on efficient issue queue utilization. The issue queue size can be dynamically adapted to fit the ILP content of the workload, according to issue queue utilization [4] or parallelism-based metrics [5]. More recently, fetch gating and issue queue adaptation have been combined together in [8] to achieve further reduction in energy consumption. In this technique, fetch gating is driven by issue queue utilization in order to track the level of program parallelism, while the remaining underutilized issue queue entries are shut down using the issue queue adaptation approach. However, the achievable overall energy-delay savings are still limited to around 6% according to the data reported in [8]. This is because only focusing on issue queue design cannot provide significant power savings, since each instruction still needs to go through all the stages on the instruction delivery path.

To achieve more aggressive energy savings, the work performed by the front-end should be reused. The first research step in work reuse is a smaller and thus more energy-efficient cache, proposed to reduce the energy dissipation of instruction fetch. The *Filter Cache* technique [9] adds a smaller cache between the processor and L1 cache to store recently accessed cache blocks, in the process incurring the penalty of longer average memory access time. A similar mechanism is presented in [10] to store the instructions within loops in an extra buffer. In [11] the loop buffer is extended to capture more complicated loops, such as loops with internal branches or with a number of separate tail-sections that all return to the same loop head. The power reduction achievable by these techniques is quite limited, as only the work performed by the fetch unit is reused.

In order to achieve increased reuse, caches can be placed after the decode stage to store *decoded* instructions, such as *Trace Caches* [12]. A hit in the trace cache allows decoded instructions to be delivered directly into the pipeline from the trace cache, thus enabling the shutdown of the whole decode stage for significant periods of time during program execution. However, trace caches themselves are extremely power-consuming. Firstly, they require much more complicated indexing and mapping mechanisms to search for frequently executed traces. Secondly, since different directions of a conditional branch correspond to distinct traces, one block of instructions can be stored multiple times at different locations in the trace cache, implying highly inefficient storage and energy utilization.

Building on the use of trace caches, dynamic vectorization techniques [13] have been proposed to detect repetitive control flow at run-time and capture the corresponding dynamic loop body in a vector form. This enables multiple iterations to be issued from the single copy of the loop body in the instruction window, thus eliminating the need to re-fetch the loop body for each iteration. However, in addition to the power inefficiency associated with the trace cache, the technique furthermore imposes the constraint that a trace line must start at a loop head (a target of a backward branch) and terminate at the corresponding loop tail. Because multiple iterations of a loop are speculatively delivered into the processor to boost performance, this technique also wastes a significant amount of power if the program includes complicated loops with internal branches and multiple tail-sections.

More recently, an *Execution Cache* which places the cache after the issue stage to store *renamed* instructions in issue

order, has been proposed in [2]. This technique allows more reuse to be achieved as a hit in the execution cache enables the bypass of the whole instruction fetch, decode, rename and issue stages. Nonetheless, the applicability of this technique is strongly limited by its indispensable shortcomings; because the cache stores rename instructions, reuse of traces needs a dedicated register renaming technique to handle *pseudo data dependences*. More importantly, as instructions are stored in issue order, they lose their original logical order, implying that traces can only be retrieved sequentially to maintain semantic correctness. Furthermore, a complicated trace look up step needs to be performed at each trace end. In conjunction with the requirement of an in-order initiation of each trace, this set of constraints leads to non-negligible performance penalty associated with each trace change. According to the data reported in [2], the 30% overall energy savings achieved by this technique are attained at a cost of 9% performance degradation.

### 3. TECHNICAL MOTIVATION

To aggressively exploit instruction level parallelism (ILP), traditional superscalar processors usually incorporate dynamic scheduling and speculative execution capabilities. In addition, the front-end is forced to aggressively fetch and decode instructions in the peak bandwidth and as early as possible. This performance focused strategy therefore causes two significant sources of power inefficiency. One of them may be caused by the mismatch between the static instruction sequences stored in the I-cache and the dynamic instruction sequences to be delivered into the pipeline. Because the conventional fetch engine can deliver only contiguous cache locations at any time instance, an increasingly high number of useless instructions following a taken branch are brought into the front-end as the fetch bandwidth grows. Although predicting multiple branches per cycle allows noncontiguous instructions to be fetched at one time, this requires a complicated branch prediction scheme. Actually, in order to hide the latency of branch execution, dynamic superscalar processors have to look up the branch target buffer (BTB) for each fetch address during the first cycle to perform speculative execution, even if no branch instruction exists. The non-contiguous instructions obtained from the cache furthermore must be assembled into the dynamic sequence, requiring extra work to shift and align them in order.

More importantly, another source of power inefficiency may stem from repeating work performed for the same instruction trace across loop iterations. Each time a specific trace from the program is executed, the front-end repeats its job of fetch, decode, register rename, and dispatch. While the destination register of one instruction can be renamed to distinct physical registers at various loop iterations, the functions performed by the fetch and decode stages are identical for the same instruction. On the other hand, as dynamic out-of-order processors keep increasing the number of in-flight instructions to exploit ILP while the sizes of traces remain constant, multiple copies of the same trace can exist simultaneously within the processor. This implies that the power spent in fetching and decoding a new copy of the trace is actually unnecessary, as the decoded information can be extracted from an old copy that still exists within the processor.

To reduce the power spent on instruction delivery, in our framework the work performed at previous iterations is reused as much as possible. By noticing that all the in-flight instruc-

tions have their information stored within the processor, more precisely, in the *Reorder Buffer* (ROB), we propose to gate off the fetch and decode units when a trace is reexecuted, and deliver a corresponding, previously decoded copy from the ROB directly into the rename stage. This ROB-based instruction delivery mechanism completely eliminates the repetitive fetch and decode cycle for the same trace. This technique furthermore eliminates the power that used to be spent in fetching useless instructions following a taken branch. As the ROB records dynamic instruction sequences, delivering instructions across taken branches is performed automatically with no requirement for either complicated branch prediction schemes or dedicated alignment hardware. Because most of the instructions stored in the ROB constitute the most frequently executed instructions, the technique we propose actually enables an implicit reuse of the most frequently executed traces.

While utilizing the ROB for instruction delivery engenders significant power savings for the most frequently executed traces, the extra power spent in achieving the potential work reuse should be minimized. This introduces additional challenges in identifying reuse as well as in delivering decoded instructions. Fundamentally, given the instruction currently under fetch, reuse can only be achieved if a ROB entry contains a previously dispatched copy of the exact instruction. In other words, the index of the ROB entry corresponding to a previous copy, if any, should be identified. As multiple instructions are inserted into and retired from the ROB in every cycle, the ROB strongly affects the overall performance of the superscalar, implying that a straightforward search of the ROB for reuse identification introduces non-negligible performance and power overhead. Therefore, an efficient approach for quick identification of a match between the current fetch address and the *PCs* (*program counters*) recorded in the ROB is necessary in the proposed framework. Furthermore, in order to be delivered into the rename stage directly, an instruction obtained from the ROB should contain all the information in a decoded form, that is, the operation code, the destination and source registers, as well as the immediate field, if any. Unlike the instructions stored in the cache, decoded instructions introduce variability in instruction lengths, imposing additional challenges in efficiently storing all the decoded information in the ROB.

## 4. POWER-EFFICIENT REUSE IDENTIFICATION

Identifying potential reuse needs a mechanism to search for matches between the instruction currently under fetch and instructions stored in the ROB, that is, to compare the next fetch address against the PCs recorded in the ROB entries. However, identifying reuse at the instruction level is a straightforward yet inefficient solution, as in the worst case each fetch address needs to be compared against all the PC values stored in the ROB to identify a potential match. Consequently, in this section we propose a block level approach, in which a search for a match needs to be initiated solely upon a change in control flow.

### 4.1 Identifying Reuse at Block Level

Our block level reuse identification approach is motivated by the observation that both the instructions under fetch and the instructions stored in the ROB are ordered as sequences

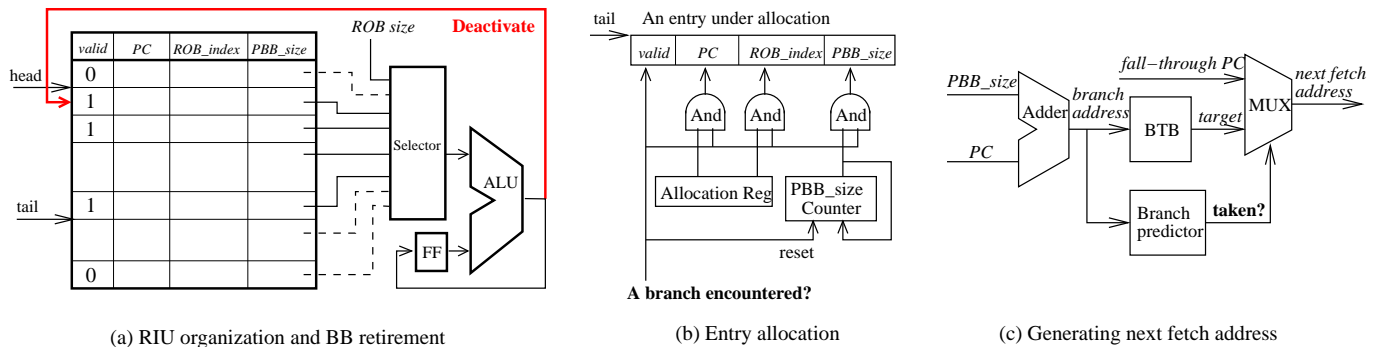


Figure 1: Implementing Reuse Identification

of *basic blocks* (BBs)<sup>1</sup>. Actually, because the ROB is implemented as a FIFO structure, the BBs presented in the ROB, except for the **newest** (the BB currently under dispatch) and the **oldest** (the BB currently being replaced by the newest BB), have all of their instructions stored in the ROB in program order. This implies that if a match has already been identified for the starting instruction of a BB, the rest of the ROB entries belonging to that BB are qualified for reuse with no additional search needed. In fact, because the program control flow can only be altered by branch instructions, the search for a new match only needs to be performed for the instruction following a branch. This observation enables us to furthermore reduce the number of necessary searches by grouping one or multiple BBs into a *pseudo basic block* (PBB), a linear sequence of instructions that always ends up with a branch instruction. By doing this, all the existing matches can be captured through comparing the predicted fetch address after a branch against the starting PCs of the PBBs stored in the ROB.

Because the search for a new match needs to be initiated upon encountering a branch instruction, a mechanism to effectively identify branch addresses in advance is critical for the efficiency of the proposed reuse identification technique. Instead of constantly looking up the branch target buffer (BTB) during the first fetch cycle for each fetch address, the framework we propose enables early branch identification through recording at the beginning of a PBB the distance to the subsequent branch, that is, the size of the PBB. By recording the size of each PBB in the ROB, the next branch address can be calculated as soon as a match has been identified, and the generated branch address can be sent to the BTB and the branch predictor immediately to predict a new fetch address.

With the starting PC and the PBB size information being extracted dynamically during execution for each PBB under dispatch, tracking the content of the ROB and searching for new matches can be performed efficiently. During dynamic execution, once a match has been identified, the index of the corresponding ROB entry can be obtained, and the following instructions start to be delivered from the ROB directly into the rename stage. Meanwhile, the obtained PBB size is added to the starting PC to calculate the next branch address. The generated branch address is delivered to the BTB and the branch predictor to obtain a new predicted fetch address, for which the search for a new match is initiated in turn.

Using this technique, the number of comparisons needed for reuse identification can be effectively reduced. A comparison of the worst case analysis indicates that the proposed technique only needs to compare the predicted fetch address after a branch against all the recorded starting PCs, while the baseline idea of searching the whole ROB needs to compare each fetch address against all the instructions stored in the ROB. Obviously, if the average PBB size of a program is  $M$ , the complexity of the proposed approach is reduced to  $1/M^2$  of the baseline idea.

## 4.2 Reuse Identification Unit

In the proposed framework, a specialized hardware structure, namely the *Reuse Identification Unit* (RIU), is employed to efficiently track the content of the ROB at block level. According to the last section, an RIU entry is allocated for each PBB stored in the ROB to record the starting PC, the corresponding ROB entry index, as well as the PBB size. Figure 1a presents the detailed organization of the RIU. As can be seen, each RIU entry includes a *valid* bit, a *PC* field, an *ROB\_index* field, as well as a *PBB\_size* field. Because all the PBBs in the ROB are stored sequentially, the RIU is also organized in a FIFO structure, with the allocation and retirement of RIU entries corresponding to the movement of the tail and head pointers, respectively.

During dynamic execution, the RIU performs three functions: threading a new PBB under dispatch, retiring an old PBB to track the content of the ROB, as well as searching for new matches.

**PBB threading:** In our framework, PBB threading is performed by sequentially allocating a new RIU entry for each PBB under dispatch. This is performed after the decode stage in parallel with inserting instructions into the ROB, through the use of an *allocation register* and a *PBB\_size counter*, as presented in Figure 1b. More precisely, when the first instruction following a branch is being inserted into the ROB, both the instruction address and the index of the associated ROB entry are recorded in the *allocation register*, while the *PBB\_size counter* is initiated to 0. As the subsequent instructions get inserted into the ROB, the value of the *PBB\_size counter* gets incremented until a branch instruction is encountered. At that time, the RIU entry indexed by the RIU tail pointer is reallocated to record the value of the *allocation register* as well as the *PBB\_size counter*. This whole process is repeated when the next instruction, that is, the first instruction following the branch just encountered, is inserted into the ROB. In most cases the RIU entry indexed

<sup>1</sup>A *basic block* (BB) is a linear sequence of instructions with single entry and exit points.

by the tail pointer constitutes a dead entry<sup>2</sup> which is free for reallocation. However, in the extreme case where the whole content of the ROB is composed of tiny PBBs that cannot be recorded in the RIU simultaneously, the RIU only records the most recently decoded PBBs for later reuse, implying that a live RIU entry has to be reallocated.

**PBB retirement:** To track the content of the ROB, the PBB indexed by the head pointer of the RIU is retired from the RIU if the complete set of instructions of that PBB is not fully resident in the ROB. More specifically, as entries in the ROB are set up in program order at the time of instruction dispatch, an entire PBB will remain in the ROB until the tail pointer moves across its beginning instruction. Consequently, tracking the PBBs presented in the ROB can be achieved by ensuring that the sum of the size of all the live PBBs is less than the total ROB size. As shown in Figure 1a, in our framework this function is implemented simply through the use of an ALU to calculate the sum of all the live RIU entries’ *PBB\_size* value minus the ROB size, and ensuring the result is less than 0. Upon the allocation of a new RIU entry, if adding its *PBB\_size* to the previous result of the ALU causes the sum to become negative, the PBB in the oldest RIU entry is retired by resetting the *valid* field and moving across the head pointer. As can be seen in Figure 1a, this function is implemented through writing the most significant bit of the new ALU result into the *valid* field of the RIU entry indexed by the head pointer, implying that a positive ALU result (with the most significant bit equal to 0) always invalidates the oldest entry. The retired RIU entry’s *PBB\_size* value is in turn subtracted from the current positive ALU result, and this retirement process is repeated until the ALU result becomes negative.

**Match searching:** In the proposed framework, a search for a match is performed at the fetch stage, as soon as the next fetch address has been generated by the branch prediction subsystem. If instructions are currently being delivered through the ROB-based path, the given predicted fetch address is first compared against the *PC* field of the RIU entry following the current matching RIU entry in order to reduce search complexity. As the PBBs stored in the ROB automatically compose the most frequently executed trace, most existing matches can be captured in this way, implying that search complexity can be significantly reduced. However, if no match is captured, or if instructions are currently being delivered through the conventional path, a search against all the PCs recorded in the RIU should be initiated. In this situation, the RIU can be viewed as a fully associative cache, with the *PC* fields being the tags. To reduce the power spent in tag comparison, the whole *PC* field is implemented as a *Content-Addressable Memory* (CAM) array [15]. If a thorough search indicates the existence of multiple matches for the given fetch address, the latest recorded RIU entry is selected, implying a reuse of the most recently executed trace. Starting from the next clock cycle, the *ROB\_index* value of the selected RIU entry is obtained to access the ROB for instruction delivery. Meanwhile, the value in the *PBB\_size* field is immediately added to the PC to obtain the next branch address, which is delivered to the BTB and the branch predictor to obtain a new predicted fetch address, as shown in Figure 1c.

<sup>2</sup>An RIU entry becomes dead if the corresponding PBB is no longer present in the ROB.

A further implementation aspect that needs consideration is the RIU size, that is, the number of RIU entries. As one RIU entry is allocated for each PBB stored in the ROB, this architectural parameter is highly related to the ROB size and the average PBB size of a program. While increasing the RIU size may be seen as desirable and affording insurance against application variance, the associated area and power overhead should also be taken into consideration. As the average PBB size of most programs can be observed to be constrainable to no less than 4 instructions, in our framework we set the RIU size to be equal to 1/4 of the number of ROB entries. In this way, most of the time the RIU can be used to track all the PBBs present in the ROB, except for the aforementioned extreme cases wherein the whole content of the ROB is composed of tiny PBBs.

In most cases a single RIU access suffices for the delivery of a whole PBB stored in the ROB as well as for obtaining the subsequent branch address. The only exception occurs in the situation where the corresponding PBB size exceeds the representational limits of the *PBB\_size* field<sup>3</sup>, which results in consecutive RIU entries to be allocated and thus multiple lookups to be performed to obtain a large PBB size. While performing multiple lookups implies more power consumption, no search for a new match is needed in this case, as a PBB with a large PBB size is always recorded in consecutive RIU entries. In addition, the latency of accessing multiple RIU entries can also be completely hidden, since the corresponding large PBB size indicates the next change in control flow to be at least more than  $2^n$  instructions away, assuming an  $n$ -bit width of the *PBB\_size* field.

## 5. ROB-BASED INSTRUCTION DELIVERY

### 5.1 Alternative Instruction Delivery Path

A match identified by the RIU enables significant energy reductions achieved through gating off the fetch and decode stages during trace reexecution using pipeline gating techniques [3]. As presented in Figure 2, our reuse technique adds another instruction delivery path to the superscalar architecture. Instructions sent to the rename stage and inserted into the ROB are either delivered through the fetch and decode stages, or obtained directly from the ROB entries identified by the RIU. However, after the rename stage, these two instruction delivery paths become identical for the rest of the superscalar pipeline, including the rename, issue, and forwarding logic, as well as the functional units. Consequently, our ROB-based instruction delivery path does not introduce significant complexity to the existing superscalar architecture. In addition, because instructions are delivered through the ROB if a trace is captured by the RIU, the ROB in our framework can be viewed as a “trace cache”, implying that the selection of delivery paths for a trace cache can also be adopted in our framework to reduce design complexity.

When instructions are delivered through the ROB-based path, with the PBB size information being extracted in advance, search for a new match only needs to be performed when the subsequent branch instruction is encountered. Here, either a conservative or a speculative delivery policy can be adopted. The main difference between these two cases is whether subsequent instructions stored in the ROB be-

<sup>3</sup>The proposed framework employs a 5-bit *PBB\_size*; statistical data that we have collected indicates that on average 98% of all the dynamic executed branches can be thus captured.

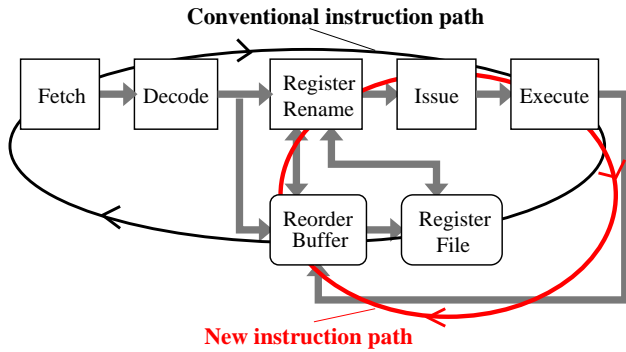


Figure 2: Using the ROB for instruction delivery

yond the incoming branch can be speculatively delivered into the pipeline or not before the next match has been established by a new search. In fact, speculative delivery of subsequent PBBs beyond the incoming branch implies reuse of the branch outcome resolved<sup>4</sup> during an earlier loop iteration as a prediction for the exact branch in the current iteration. If a speculative delivery strategy is adopted and the result of the next match search indicates a speculation to be incorrect, the instructions on the mis-speculated path should be flushed. As a result, the conservative strategy always displays a reduced energy consumption, while a speculative strategy can boost performance if branch results are highly biased to enable a highly accurate prediction. However, compared to the conventional instruction delivery path, the ROB-based instruction delivery path has fewer pipeline stages, implying the associated branch mis-prediction penalty is always cheaper for both the speculative and the conservative instruction delivery strategies.

Because the proposed ROB-based instruction delivery path is shorter than the conventional path, a special mechanism to switch between these two paths is necessary for the proposed framework. Because these two instruction paths only differ before entering the rename stage, as long as instructions are sent to the rename stage and inserted into the ROB tail in program order, semantic correctness will be ensured. More specifically, a switch to the conventional path happens if instructions are delivered from the ROB and **no** match exists for the predicted next fetch address. As the ROB-based path is shorter than the conventional path, instructions delivered from the decode stage will always enter the rename stage later than the instructions delivered from the ROB. Therefore, the fetch and the decode stage can be gated off either in parallel or in serial. On the other hand, a switch to the ROB-based path occurs if instructions are delivered from the fetch and decode stages and a match for the predicted next fetch address is identified. However, as the conventional instruction delivery path is longer, the fetch and the decode units cannot be gated off simultaneously. Instead, to ensure that instructions still enter the rename stage in semantically correct program order, a switch needs to be performed in two steps. First, the fetch unit is gated off, while the decode stage still decodes previously fetched instructions. At the second step, after the instructions in the decode stage have been sent to the rename stage as well as the ROB, the decode stage is

<sup>4</sup>In the extreme case, the branch in question may not even have been resolved yet. Therefore, the predicted branch outcome is used in this case.

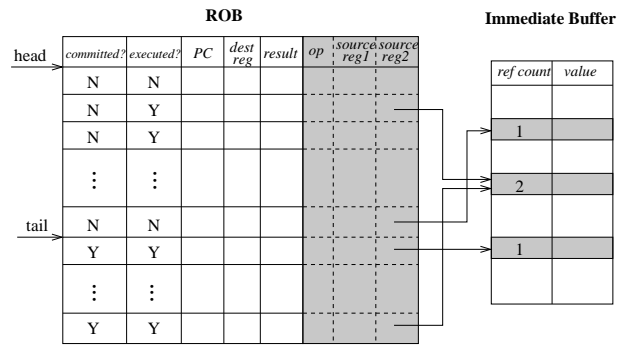


Figure 3: Proposed ROB organization and the Immediate Buffer

gated off, and instructions start to be delivered from the ROB into the pipeline.

## 5.2 Extended ROB and Immediate Buffer

In the traditional out-of-order superscalar architecture, each ROB entry only holds the destination register operand tag, the results produced by functional units, the PC to maintain precise exception, and two flags that respectively indicate whether the instruction has been executed and committed. On the other hand, to be delivered into the rename stage directly, an instruction obtained from the ROB should contain all the pieces of the decoded information, that is, the operation code, the destination and source registers, as well as the immediate, if any. As can be seen, among these pieces of decoded information, only the destination register tag is stored in a conventional ROB for instruction retirement. Consequently, the proposed framework presumes the extension of the ROB to additionally include the rest of the decoded information. While the sizes of opcode and the source register tags are small enough for them to be contained within the ROB directly, extending the ROB to include decoded immediates introduces non-negligible hardware overhead. To solve this problem, a specialized hardware structure, namely an *Immediate Buffer*, is added to the proposed framework to hold decoded immediates. In this way, the corresponding ROB entries only need to record the buffer indices as shown in Figure 3, instead of the second source register tags.

Unlike register operands that can be renamed differently, immediate values are always identical each time the exact instruction is executed. Therefore, to efficiently utilize the Immediate Buffer, we allow multiple copies of the same instruction to share the same entry in the Immediate Buffer. As shown in Figure 3, this is implemented by adding a *reference counter* to each entry of the Immediate Buffer, and incrementing the counter when a new copy of the instruction enters the ROB. As a result, an entry in the Immediate Buffer is only allocated if the ROB contains no previous copy of that instruction, that is, if the instruction is delivered through the conventional path. On the other hand, when an instruction containing an immediate is delivered through the proposed ROB-based path, no Immediate Buffer entry is allocated for the new instruction copy. Instead, the *source reg2* value obtained from the previous copy of the exact instruction is directly written to the ROB entry under allocation, and the *reference counter* of the corresponding Immediate Buffer entry is incremented. Decrementing the reference counter occurs whenever a copy of the corresponding instruction contained

in the ROB is replaced by a new dispatched instruction. An Immediate Buffer entry becomes free for reallocation if no reference from the ROB exists.

In general, when an instruction is delivered through the proposed ROB-based path, the values of the *op*, *dest reg*, *source reg1* and *source reg2* fields belonging to the matching ROB entry are sent to the rename stage. Meanwhile, the whole content of the matching ROB entry is copied to the ROB tail, that is, the ROB entry currently under allocation. In this process, if the instruction previously held in the reallocated ROB entry has an immediate, the reference counter of the corresponding Immediate Buffer entry, indexed by the *source reg2* field, is decremented. Similarly, if the instruction under delivery has an immediate, the reference counter of the corresponding Immediate Buffer entry is incremented, and the obtained immediate value is sent to the pipeline thereafter. Although in this case the ROB and the *Immediate Buffer* are accessed serially, the latency to access the Immediate Buffer can be completely hidden, as the value of the immediate is not needed at the rename stage.

## 6. EFFECTIVENESS ANALYSIS

### 6.1 Expected Reuse Analysis

The ROB-based reuse technique is motivated by the observation of the strong locality associated with programs, in line with previously proposed reuse techniques. Fundamentally, the fraction of all the executed instructions that can be delivered from the ROB depends on two factors: the average code size of the repetitive executed traces and the number of entries in the ROB. A well known principle in computer architecture is that 90% of the cycles are spent in executing only 10% of the code. The locality associated with emerging applications, especially multimedia applications that spend most of their execution time within a set of heavily utilized DSP kernels, is even higher. While the sizes of the program hot spots remain constant, the size of the ROB grows rapidly. This is because out-of-order superscalar processors tend to increase the fetch bandwidth and the pipeline depth to exploit more ILP, thus retaining a large number of instructions in flight. As the number of ROB entries constitutes the upper bound of the number of in-flight instructions, a larger ROB is necessitated, implying a growing probability for the instructions currently under fetch or decode to have corresponding entries in the ROB that were set up during previous loop iterations.

Furthermore, the instructions contained in the ROB for possible reuse are not limited to be only in-flight instructions. Actually, during dynamic execution, the ROB contains not only in-flight instructions, but also recently committed instructions. Since the ROB is essentially a FIFO queue, the functions of dispatch and commit are respectively implemented through the movement of the tail and head pointers. Entries in the ROB are set up in program order at the time of instruction dispatch. This implies that an entry of ROB will maintain its previous value until the tail pointer moves across it, that is, a committed instruction still stays in the ROB until it is replaced by a new dispatched instruction. Consequently, during program execution the ROB consistently holds the most recently dispatched instructions, up to the number of ROB entries, independent of whether they have been committed or remain uncommitted. Obviously, all these recently dispatched instructions are qualified for reuse.

### 6.2 Power Analysis

Incorporating the proposed technique on traditional out-of-order superscalar architecture enables power savings to be achieved in three ways. Primarily, the fetch and the decode unit can be gated off when instructions are delivered through the ROB-based path. Additionally, the power spent in fetching useless instructions following a taken branch and in aligning noncontiguous instructions obtained from the cache is completely eliminated, as delivering instructions across taken branches is performed automatically in the ROB-based path. The technique further eliminates power hungry BTB lookups for non-branch instructions normally occurring at every cycle, since a hit in the RIU allows the next branch address to be generated in advance, based on the extracted PC and PBB size information.

The proposed framework also consumes extra power in implementing the ROB-based instruction delivery path, more precisely, in searching and updating the RIU as well as in obtaining decoded instructions from the ROB and the Immediate buffer. Extra leakage power is consumed in the RIU, the immediate buffer, as well as the extended fields of the ROB. As the ROB is extended to store all the decoded information, traditional ROB functions such as inserting and retiring instructions also consume more power.

In most situations, the power consumed in implementing work reuse is not comparable to the three aforementioned sources of power reduction. On one hand, the fetch and decode units, together with the BTB, consume a large fraction of the total processor power. On the other hand, both the RIU and the Immediate buffer are small hardware units, thus only consuming a small amount of dynamic access power as well as leakage power. Furthermore, because the function of retirement needs to access a subset of the ROB fields, a circuit level technique such as subbanking [14] can also be employed in our framework to reduce the power spent in ROB accesses. As all three sources of power reduction are proportional to the probability of reuse while the extra power dissipation remains constant, a significant amount of power reduction can be achieved if a large fraction of the total executed instructions can be delivered through the ROB-based path, that is, if the program displays a strong temporal locality.

### 6.3 Performance Analysis

Compared to traditional superscalar architectures, the proposed ROB-based path enables noncontiguous instruction fetch to be performed speculatively across taken branches. Accordingly, performance improvement can be achieved if these speculatively delivered instructions are useful, that is, if the corresponding taken branches are predicted correctly. In general, branch predictions are generated in the same way on both instruction delivery paths. However, automatic delivery of instructions across PBB boundaries implies reuse of an earlier branch outcome for the exact branch in the current iteration, which may slightly reduce prediction accuracy as no inter-branch correlation is considered in this case.

As a large fraction of instructions are delivered into the pipeline from the ROB, the proposed technique also reduces the number of L1 I-cache accesses. At first sight it seems that the number of cache misses can hardly be reduced, because the reduced I-cache accesses correspond to recently executed traces that typically hit in the cache. However, while not significant, delivering instructions from the ROB still can reduce the probability of cache conflicts since the L1 I-cache is

Fetch queue size:	8 (16) entries
Ld/St queue size:	32 (64) entries
BTB :	1024 sets, 4-way
Branch predictor:	combination of bimodal and gshare selector: 1024 entries bimodal: 2048 entries gshare: 4096 entries, 12-bit history
Branch mis-prediction penalty:	4 (5) cycles
L1 I-cache:	16KB 1-way, 32B block, 1 cycle
L1 D-cache:	32KB 2-way, 32B block, 1 cycle
L2 unified cache:	512KB 4-way, 64B block, 8 cycles
TLB	128 entries, 8-way, 16K page size

**Table 1: Simulation parameters**

accessed much less frequently. As a result, in the case where the L1 I-cache falls short of storing the whole code of a recently executed trace due to cache conflicts, the ROB-based instruction delivery path enables a sizable reduction in the cache miss occurrence rate.

As discussed before, the proposed ROB-based path displays a shorter pipeline than the conventional path, thus further resulting in two distinct aspects in affecting performance. On one hand, while switches from the conventional path to the ROB-based path cause no performance degradation, each switch to the conventional path needs an additional penalty cycle spent in searching the RIU to ensure that no match exists for the given predicted fetch address. On the other hand, a shorter pipeline implies that fewer instructions need to be flushed in case of a branch mis-prediction, thus significantly reducing the branch mis-prediction penalty.

In sum, the achievable performance improvement outperforms the incurred performance penalty if the program in execution displays a large amount of taken branches that can be predicted correctly, that is, if a large fraction of useful instructions can be delivered through the ROB-based path within a relatively small amount of switches.

## 7. SIMULATION RESULTS

To evaluate the proposed instruction delivery techniques for different types of applications, a set of experimental studies have been performed on a subset of the *Mediabench* [16] and the *SPECint 2000* benchmarks. The *SimpleScalar* toolset [17] is employed to simulate an aggressive superscalar out-of-order processor. The simulator has been modified to include an RIU model and to bypass cache accesses in the case of RIU hits. As the effectiveness of the proposed ROB-based instruction delivery technique is a function of issue width and ROB size, three configurations are evaluated during our simulation process: a 4-way superscalar with a 128-entry ROB, a 4-way superscalar with a 256-entry ROB, and an 8-way superscalar with a 256-entry ROB. In all three cases, the RIU size is set to be 1/4 of the ROB size. The rest of the simulation parameters are summarized<sup>5</sup> in Table 1. To save simulation time, for each *SPECint 2000* benchmark 100 million of instructions are simulated, with the simulation point selected using the Simpoint [18] technique.

Table 2 presents the basic benchmark characteristics obtained on a baseline 4-way processor with a 256-entry ROB. The first column lists the value of IPB (instructions per branch), a numerical characteristic of the branch instruction density.

<sup>5</sup>Some parameters have different values for the 4-way and the 8-way superscalar processors. In these cases, the parameters listed within parentheses are the ones used for the 8-way case.

	IPB	IPC	pred hit	L1-I miss
mcf	6.60	0.214	95.50%	0.000%
gcc	7.45	1.308	92.72%	2.648%
gzip	10.20	1.865	93.30%	1.468%
vpr_route	9.70	1.242	94.89%	0.000%
parser	6.24	1.603	94.20%	0.020%
twolf	8.22	1.096	87.52%	1.473%
adpcm	3.49	1.292	81.32%	0.006%
epic	6.80	2.755	96.67%	0.009%
gsm	20.58	2.678	94.39%	0.449%
mpeg2	5.88	1.585	80.22%	0.008%
average	8.52	1.564	91.08%	0.608%

**Table 2: Benchmark Characteristics**

It can be observed that the IPB value for most benchmarks is consistently in the range of 5 to 10, a typical situation for real applications. The significantly large IPB of 20.6 for the *gsm* benchmark is due to the existence of a basic block containing more than 300 instructions. The second column lists the value of IPC (instructions per cycle). The next two columns present the prediction accuracy and the L1-I cache miss rate, respectively. As can be observed, the selected benchmarks display a wide spectrum of diverse characteristics; two multimedia benchmarks, *adpcm* and *mpeg2*, display quite low branch prediction accuracy, while three SPECint 2000 benchmarks, *gcc*, *gzip* and *twolf*, display quite high cache miss rates.

### 7.1 Reuse Effectiveness

Figure 4 presents the reuse opportunity results including the ratio of all the fetched instructions that can be captured in the ROB, as well as the fraction of reusable instructions remaining in the ROB that can be captured in the RIU. The first set of results reflects the average size of repetitive traces of the benchmark, while the second set is influenced by the average PBB size of the captured traces. It can be observed that the proposed reuse technique is quite effective, as on average 50% of all instructions under fetch can be captured in a 128-entry ROB and 95% of all these reusable instructions can also be captured in a 32-entry RIU. For a configuration of a 256-entry ROB and a 64-entry RIU, the corresponding results further increase to 58% and 97%, respectively.

The worst capture capability of the RIU occurs when a 32-entry RIU is employed to capture the reuses obtained by a 128-entry ROB for the *adpcm* benchmark. The fundamental reason is that *adpcm* displays a quite low IPB of 3.49 as listed in Table 2, resulting in the inability of the RIU to simultaneously track all the PBBs resident in the ROB. However, the results of *adpcm* also shows that doubling both the ROB and the RIU sizes can significantly enlarge the capture capability of the RIU. This is because increasing the ROB size causes a larger probability for the ROB to contain multiple copies of a single PBB, thus increasing the probability for that PBB to be captured in the RIU.

Obviously, for most benchmarks a sizable increase in the capture capability of the ROB can be observed when the ROB size is enlarged from 128 to 256. The only exception is *gsm*, which contains a basic block of more than 300 instructions that can be captured by neither a 128-entry nor a 256-entry ROB. On the other hand, for most benchmarks increasing the fetch bandwidth results in a limited or even a negative increase in the capture capability of both the ROB and the RIU. This is because as the fetch bandwidth grows, more useless instructions on mis-prediction paths are delivered into the pipeline. These useless instructions can even pollute the

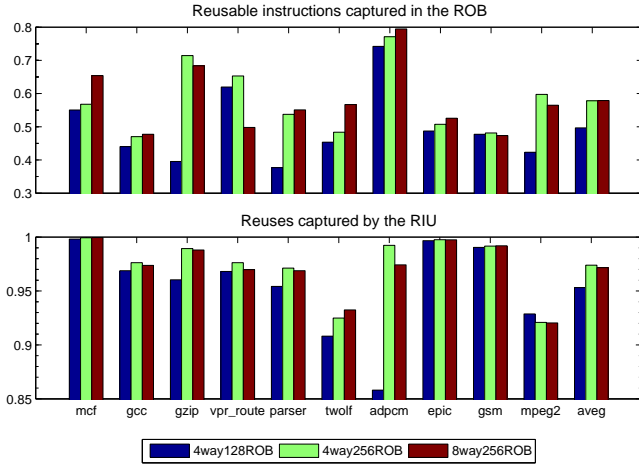


Figure 4: Reuse effectiveness

ROB, thus further reducing the amount of useful reuse that can be captured.

## 7.2 Performance Results

Figure 5 presents the performance related results when incorporating the proposed technique to the baseline superscalar processor for all the three selected configurations. The values presented here include the reduction in the number of L1 I-cache misses, the increase in branch mis-prediction rate, and the fraction of IPC improvement. As can be seen, except for *gcc*, *vpr\_route*, and *mpeg2*, the ROB-based technique enables more than 5% reductions in the number of cache misses, especially in the case where a 256-entry ROB is employed. On the other hand, except for *epic*, the increase in branch mis-prediction rate is less than 5%. Consequently, the ROB-based technique results in performance improvement for most benchmarks, except for *mcf* which has a quite low IPC (as shown in Table 2), and for *epic* which displays a 29% increase in branch mis-prediction rate<sup>6</sup> when the benchmark is executed on a 4-way processor.

Because the proposed ROB-based path displays a shorter pipeline than the conventional path, fewer instructions need to be flushed in case of a branch mis-prediction. Consequently, although the ROB-based path delivers instructions more aggressively and causes more mis-predictions, notable performance improvement still can be obtained for most benchmarks. In most cases the achievable performance improvement is less than 4%. However, a significant improvement of 20% occurs when *gzip* is executed on an 8-way processor with a 256-entry ROB. This performance improvement is caused by the significant reduction of 76% in the number of cache misses. More specifically, the direct-map L1 I-cache in the baseline architecture causes a sizable miss rate of 1.47% due to cache conflicts, most of which are effectively eliminated by the ROB-based instruction delivery mechanism as the frequency of cache accesses can be significantly reduced.

For most benchmarks, performance improvement can be observed when the fetch bandwidth is increased. The only exceptions are *vpr\_route* and *parser*, for which increasing the fetch bandwidth even causes performance degradation. This is because increasing the issue bandwidth causes more useless

<sup>6</sup>Because the original mis-prediction rate is quite low, this translates to only a 1% reduction in prediction accuracy.

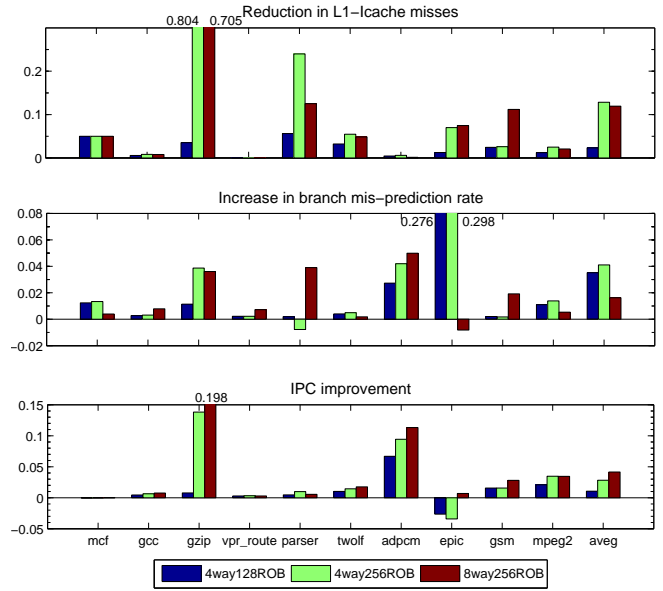


Figure 5: Performance related results

instructions on mis-prediction paths to be delivered into the pipeline, thus further polluting the ROB and reducing the amount of useful reuse. Accordingly, for *vpr\_route* a sizable reduction in the amount of reuse captured by the ROB can be observed from Figure 4, while for *parser* a sizable increase in both the branch mis-prediction rate and cache miss rate can be observed from Figure 5.

## 7.3 Power Results

In our simulation framework, *WATTCH* [19] is utilized to analyze the energy consumption characteristics of all benchmarks for the baseline out-of-order architectures. In addition, *CACTI* [20] is employed to analyze the access power characteristics of the ROB, the BIU, and the Immediate Buffer, since all of them are implemented as standard SRAM structures. Because no tag comparison is needed for accessing the ROB and the Immediate Buffer, their power characteristics are extracted by modelling a direct-mapped cache structure and subtracting the power consumption of the tag array and the comparator cells. As outlined in Section 4.2, in most cases an RIU access only need to compare the given predicted fetch address against the PC field of the RIU entry following the current matching RIU entry. Accordingly, this type of RIU access is modelled as accessing a direct-mapped cache structure, while thorough searches of the RIU are modelled as accessing a fully associative cache. The line widths for both the RIU and the Immediate Buffer are fixed at 8 bytes, the minimal word size allowed by *CACTI*, which is about the right amount of storage for both the RIU and the Immediate Buffer. These extracted access power characteristics, together with the number of accesses reported by the modified SimpleScalar out-of-order simulator, are used to evaluate the total energy consumption of the hardware units we propose. As energy values vary heavily across various design styles and fabrication processes, in this section relative percentages of energy savings rather than technology dependent energy values are reported.

Figure 6 presents the reduction in the number of L1 I-cache accesses as well as the energy savings achieved in instruction

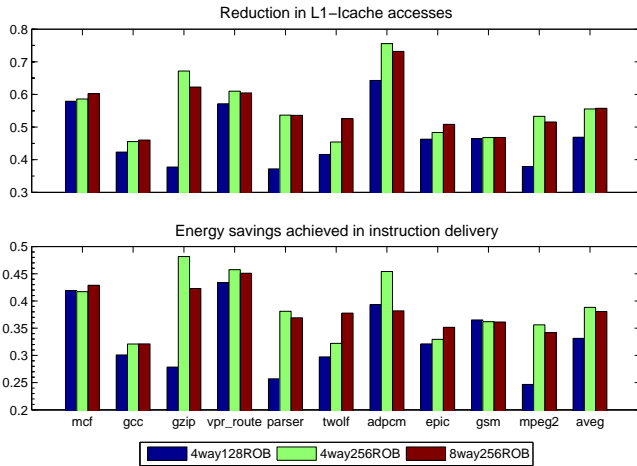


Figure 6: Power related results

delivery by incorporating the proposed technique into the traditional out-of-order processor. Because the ROB-based and the conventional paths only differ in the stages preceding the rename stage, the data we report here is focused on the energy consumed in the L1 I-cache, the BTB and the branch predictor, the decode unit, the ROB, the RIU, as well as the Immediate Buffer. As can be observed from Figure 6, on average more than 50% of the cache accesses can be redirected to the ROB, enabling significant energy savings achieved through gating off instruction fetch and decode units. The most significant energy savings occur when *gzip* is executed on a 4-way superscalar processor with a 256-entry ROB.

For most benchmarks, increasing the issue bandwidth slightly reduces the achieved energy savings, as more useless instructions on mis-prediction paths are delivered into the pipeline, thus further reducing the amount of useful reuse that can be captured. On the other hand, a significant increase in energy savings can be observed when the ROB size is enlarged from 128 to 256. This is because the ratio of the achievable energy savings is proportional to the reuse opportunity, as outlined in Section 6.2. In general, the SPEC 2000 benchmarks can obtain a larger increase in energy savings than the multimedia benchmarks when the ROB size is enlarged. This is because the SPEC 2000 benchmarks are usually composed of longer and more complex traces, thus requiring a relatively larger ROB to capture the potential reuse. For benchmarks with small IPB values such as *adpcm*, enlarging the ROB size increases the probability for a PBB to have multiple reusable copies, thus also enlarging the reuse opportunity since the PBB is more likely to be captured in the RIU. In sum, the proposed technique is quite effective as significant energy savings can be achieved, ranging from 25% to 48%, with the average equal to 38%.

## 8. CONCLUSIONS

We have presented a methodology to design a power-efficient instruction delivery mechanism in this paper. Through extending the reorder buffer (ROB) as a storage location for recently executed traces, decoded instructions can be fed directly from the ROB into the rename stage, and the fetch and decode stages can be gated off for large periods of time with significant power savings. A *Reuse Identification Unit* (RIU) has also been proposed to enable effective identifica-

tion of the matches between the ROB entries and instructions currently under fetch, by searching the cost-efficient RIU only once per basic block. Once instructions are delivered through the ROB-based path, this RIU furthermore enables the elimination of the power expensive BTB lookups that used to be performed for non-branch instructions. The proposed ROB-based instruction delivery mechanism also enables the simultaneous delivery of instructions across taken branches, without necessitating complicated branch prediction schemes. Simulation results on both multimedia and SPEC 2000 benchmarks show that for an 8-way out-of-order processor with a 256-entry ROB, the technique we propose not only enables a significant average energy savings of 38% in instruction delivery, but also slightly improves performance by 4% on average.

## 9. REFERENCES

- [1] D. M. Brooks, P. Bose, *et al.*, "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," *IEEE Micro*, 20(6):26–44, Nov. 2000.
- [2] E. Talpes and D. Marculescu, "Execution cache-based microarchitecture for power-efficient superscalar processors," *IEEE Transactions on VLSI Systems*, 13(1):14–26, Jan. 2005.
- [3] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: Speculation control for energy reduction," *In Proc. 25th ISCA*, pp. 132–141, June 1998.
- [4] A. Buyuktosunoglu, S. E. Schuster, *et al.*, "A circuit level implementation of an adaptive issue queue for power-aware microprocessors," *In 11th Great Lakes Symposium on VLSI*, pp. 73–78, Mar. 2001.
- [5] D. Folegnani and A. Gonzalez, "Energy-effective issue logic," *In Proc. 28th ISCA*, pp. 230–239, June 2001.
- [6] A. Baniasadi and A. Moshovos, "Instruction flow-based front-end throttling for power-aware high-performance processors," *In Proc. ISLPED'01*, pp. 16–21, Aug. 2001.
- [7] T. Karkhanis, P. Bose, and J. E. Smith, "Saving energy with just in time instruction delivery," *In Proc. ISLPED'02*, pp. 178–183, Aug. 2002.
- [8] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose, "Energy efficient co-adaptive instruction fetch and issue," *In Proc. 30th ISCA*, pp. 147–156, June 2003.
- [9] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache: An energy efficient memory structure," *In Proc. Micro-30*, pp. 184–193, Dec. 1997.
- [10] L. Lee, B. Moyer, and J. Arends, "Instruction fetch energy reduction using loop caches for embedded applications with small tight loops," *In Proc. ISLPED'99*, pp. 267–269, Aug. 1999.
- [11] J. A. Rivers, S. Asaad, J.-D. Wellman, and J. H. Moreno, "Reducing instruction fetch energy with backwards branch control information and buffering," *In Proc. ISLPED'03*, pp. 322–325, Aug. 2003.
- [12] B. Solomon, A. Mendelson, D. Orenstein, and R. Ronen, "Micro-operation cache: A power aware frontend for variable instruction length ISA," *In Proc. ISLPED'01*, pp. 4–9, Aug. 2001.
- [13] S. Vajapeyam, P. J. Joseph, and T. Mitra, "Dynamic vectorization: a mechanism for exploiting far-flung ILP in ordinary programs," *In Proc. 26th ISCA*, pp. 16–27, May 1999.
- [14] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," *In Proc. ISLPED'99*, pp. 70–75, Aug. 1999.
- [15] J. Montanaro, R. T. Witek, *et al.*, "A 160-MHZ, 32-B, 0.5-W COMS RISC microprocessor," *IEEE Journal of Solid-State Circuits*, 31(11):1703–1714, Nov. 1996.
- [16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," *In Proc. Micro-30*, pp. 330–335, Dec. 1997.
- [17] D. C. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *Computer Architecture News*, 25(3):13–25, June 1997.
- [18] <http://www.cse.ucsd.edu/users/calder/simpoint/>.
- [19] D. Brooks, V. Tiwari, and M. Martonosi, "WATTCH: A framework for architectural-level power analysis and optimizations," *In Proc. 27th ISCA*, pp. 83–94, May 2000.
- [20] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power and area model," Technical report, Western Research Lab, Aug. 2001.