A Power-efficient Co-designed Out-of-Order Processor

Abhishek Deb*, Josep Maria Codina† and Antonio González*†
*Universitat Politecnica de Catalunya, Barcelona Spain
†Intel Research Labs Barcelona, Barcelona, Spain
Email : abhishek@ac.upc.edu, {josep.m.codina, antonio}@intel.com

***Abstract*—A co-designed processor helps in cutting down both the complexity and power consumption by co-designing certain key performance enablers. In this paper, we propose a FIFO based co-designed out-of-order processor. Multiple FIFOs are added in order to dynamically schedule, in a complexity-effective manner, the μ-ops.**

We propose a commit logic that is able to commit the program state as a superblock commits atomically. This enables us to get rid of the Reorder Buffer (ROB) entirely. Instead to maintain the correct program state, we propose a four/eight entry Superblock Ordering Buffer (SOB). We also propose the per superblock Register Rename Table (SRRT) that holds the register state pertaining to the superblock. Our proposed processor dissipates 6% less power and obtains 12% speedup for SPECFP; as a result, it consumes less energy.

Furthermore, we propose an enhanced steering heuristic and an early release mechanism to increase the performance of a FIFO based out-of-order processor. We obtain performance improvement of nearly 25% and 70% for a four FIFO and for a two FIFO configurations, respectively. We also show that our proposed steering heuristic based processor consumes 10% less energy than the previously proposed steering heuristic.

I. Introduction

A Co-designed Processor translates instructions from the Source ISA to the host ISA, using virtual machine technology. It further optimizes the RISC like μ-ops using its run-time software. Microarchitectural support is added to both eliminate virtual machine related overhead, and to support various optimizations. This kind of virtual machine is referred to in literature as a Co-designed Virtual Machine [1].

The goals of a co-designed processor include performance, power-efficiency, and design simplicity by co-designing the processor in hardware and software. These goals can be achieved by introducing new microarchitectural features, or changing the underlying microarchitecture entirely or co-designing key performance enablers.

A. Co-designing the Commit Logic

Our proposed co-designed processor uses the Virtual Machine Monitor (VMM) to form superblocks [2]. We use superblocks that have a single-entry and a single-exit point, by converting the branches (except the last branch) into asserts, similar to rePlay [3]. Such a superblock, however, needs to be committed atomically.

Since the μ-ops execute out-of-order, a re-ordering logic is required at the back end in order to maintain the correct program order. One could use a conventional ROB in order to achieve this. However, the atomic commit constraint of the superblock implies that if all the μ-ops of the superblock have not executed then the μ-ops need to wait in the ROB to commit. This puts pressure on the ROB and related structures and leads to stalls.

In order to mitigate these problems we propose a ROB-less re-ordering logic. Two structures, namely a Superblock Ordering Buffer (SOB) and Superblock Register Rename Tables (SRRTs) are proposed that together maintain the correct program state. Each superblock is allocated a SRRT and an entry in a Superblock Ordering Buffer (SOB). The entry at the head of SOB is considered for commit at every cycle. A SOB entry contains various fields; that not only indicate whether a superblock is ready to commit, but also locates the program state associated with the superblock.

B. Simplifying the Issue Logic

Generally, a CAM based issue logic enables out-of-order execution of μ-ops, using a wake-up and select logic. This mechanism helps in exploiting the ILP, but it comes at the cost of higher complexity [4] and power dissipation [5].

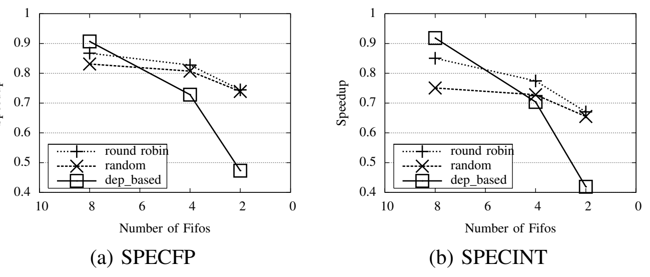


Fig. 1: Performance of Dependence-based Steering Heuristic

A FIFO based issue logic, on the other hand, helps in reducing both the complexity and the power. Palacharla et al. [4] proposed a dependence-based steering heuristic that steers each μ-op to a chosen FIFO at the dispatch stage. As the number of FIFOs are decreased the performance of the dependence-based heuristic declines sharply, for both integer and floating-point benchmarks, as shown in Figure 1. A similar observation was also made by Canal et. al. [6]. In fact a policy as simple as round-robin or random performs better with fewer number of FIFOs compared to dependence-based heuristic.

We propose a co-designed out-of-order processor that uses multiple, but fewer FIFOs, in order to achieve high performance. Since the VMM forms the superblocks it performs various optimizations similar to that in [7], [8], [9]. Our steering heuristic is based upon the dependence-based heuristic, and we make simpler modifications by analyzing the stalls at the dispatch stage. This modified steering heuristic reduces the stalls at dispatch, drastically, leading to a significant performance benefit.

C. The Early Release

Modern out-of-order processor hold issue queue entries for a small number of cycles, in order to recover from load-hit miss-speculation. By releasing some of the issue queue entries earlier, we gain in achieving a higher dispatch throughput and in reducing the pressure on issue queue. For FIFO based issue queues, this mechanism is even more important as it releases the performance critical entries, from the head of a FIFO.

The key contributions of this paper are as follows :

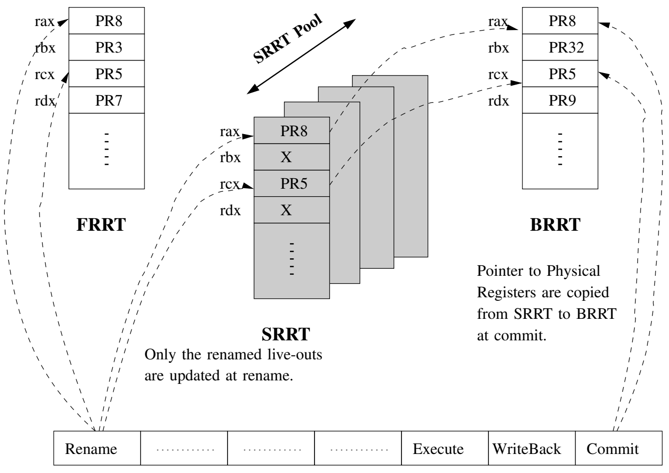
- Superblock Ordering Buffer (SOB) is proposed, that commits the program state in the original program order. As a result of the SOB and related structures, the need of conventional Reorder Buffer (ROB) is eliminated.
- Per Superblock Register Rename Table (SRRT) is proposed, that holds the register of the corresponding superblock and is committed atomically.
- Enhanced dependence-based steering logic is proposed, that reduces various stalls at dispatch stage due to the unavailability of empty FIFOs. This provides significant performance benefit, by increasing the decoder throughput.
- Early release logic is proposed, that releases few issue queue entries at issue time. This reduces the pressure on FIFO based issue queues and provides major improvement in performance in a FIFO constrained scenario.

The rest of the paper is structured as follows. First, the co-designed commit logic is proposed in Section II. In Section III the steering logic and the early release logic are discussed in greater detail. The performance and the power results are discussed in Section IV. The related work is discussed in Section V and the conclusions are drawn in Section VI.

II. Co-designing the Commit Logic

As a consequence of out-of-order execution, reorder logic is required at the back-end in order to maintain a correct program state. We propose a ROB-less reorder logic in order to maintain the correct program state. A special commit operation updates the program state, both the register and the memory state, at once.

Front-end state is maintained by the Front-end Register Rename Table (FRRT), while the Back-end (committed) state is maintained by the Back-end Register Rename Table (BRRT), similar to the Netburst microarchitecture [10]. However, since the superblocks are atomic, the BRRT is updated only when all the μ-ops of the superblock have successfully written to the Physical Register File. Similarly, the memory state is committed to the D-cache when all the μ-ops of the superblock have completely executed.



rax
rbx
rcx
rdx
PR8
PR3
PR5
PR7
FRRT
SRRT Pool
PR8
X
PR5
X
SRRT
Only the renamed live-outs are updated at rename.
PR8
PR32
PR5
PR9
BRRT
Pointer to Physical Registers are copied from SRRT to BRRT at commit.
Rename
Execute
WriteBack
Commit

Fig. 2: Register State Rename and Commit. rax and rcx are the live-outs of the current superblocks. Hence, rbx and rdx entries in the SRRT are invalid. As a result at commit only rax and rcx are updated in the BRRT.

A. Register State

In order to commit the register state atomically, we propose the per superblock register rename table (SRRT). The SRRT, like FRRT or BRRT, holds mappings from architected register to physical registers. However, unlike FRRT or BRRT, it holds mappings of only those architected registers that are live-outs of the superblock[1], as illustrated by the example in Figure 2.

A SRRT is assigned to a superblock when the head μ-op is being renamed. The source operands mappings are read from the FRRT and the destination operand is updated in the FRRT, in a conventional manner. However, if the μ-op being renamed is a live-out, then the SRRT, corresponding to the superblock, is updated. When all the μ-ops, of the superblock, have executed the register state is committed, by copying only the valid mappings from the SRRT to the BRRT; and the SRRT is made available.

SRRT like FRRT can be designed either using a RAM or a CAM based structure. We, however, use a RAM based structure as it is more scalable [4]. In a typical Register Rename Table a RAM cell consists of a shift register cell in order to shadow the mapping. However, the SRRT unlike FRRT does not need any shift register cell.

Moreover, unlike conventional rename tables, the SRRT is not read at the rename stage. Its only written at the rename stage by the μ-ops that are live-outs of the superblock. The read ports are accessed only at the commit stage when the valid mappings from the SRRT are copied to the BRRT.

Total number of ports in a conventional FRRT for a four-way superscalar processor is twelve[2]. Whereas, six read and two write ports are sufficient for a SRRT. Hence if a superblock has more live-outs than six, the commit is split into multiple cycles. This has no consequence on the performance

[1]Live-outs of the superblocks are marked by the VMM.
[2]Eight read ports and four write ports assuming two operand μ-ops.

as commit is not in the critical path. Similarly, no more than two live-outs could be renamed in a given cycle. Our experiments have shown that the average number of live-outs per superblocks are nearly 8 and 4 for SPECFP and SPECINT, respectively.

The delay of a Rename Table is given by $T_{decode} + T_{wordline} + T_{bitline} + T_{senseamp}$ [4]. The T_{decode}, and $T_{bitline}$ depend upon the total number of ports and the number of entries. As shown above we reduce the number of ports to eight, while the number of entries are still the same[3]. This is in turn reduces these delays and the corresponding power.

Furthermore, the $T_{wordline}$ depends on the number of shift-register cells, the number of ports and the width of each entry. SRRTs do not need any shift register cells, has fewer ports, and the width is same[4]; and hence has a smaller delay. As a result, the overall delay of SRRT is lower than that of FRRT, and it does not fall into the critical path. However, as there are four/eight SRRTs, additional power is dissipated, which is quantified in Section IV-E.

B. Memory State

We use gated store buffers [11] in order to hold the data, corresponding to a store μ-op. A buffer entry is allocated to the store at the rename stage. When all the μ-ops of the superblock have successfully executed a special commit operation commits the store buffer data to the cache hierarchy.

C. Superblock Ordering Buffer

We propose Superblock Ordering Buffer (SOB) in order to commit the superblock register and memory state in program order. Since SOB is a circular buffer, a superblock is committed only when the entry corresponding to it in the SOB is at the head of SOB. For a four-way out-of-order processor a SOB could have four or eight entries, many-fold smaller than a conventional ROB for a four-way superscalar processor.

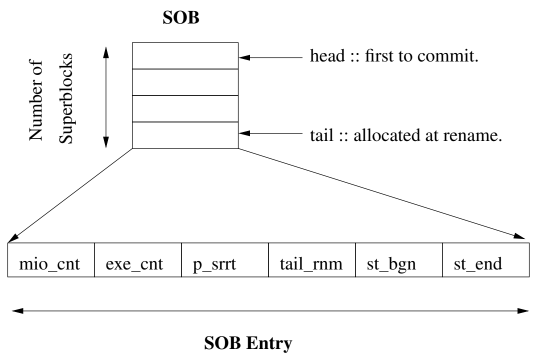


Fig. 3: Superblock Ordering Buffer (SOB)

Each SOB entry consists of six fields as shown in Figure 3. The first field (mio_cnt) indicates the number of μ-ops in a superblock. The second field (exe_cnt) indicates the number of μ-ops that have successfully executed. The third field (p_srrt) is a pointer to the per superblock register rename table (SRRT). The fourth field (tail_rnm) is a bit indicating whether the tail μ-op of the superblock has been renamed. The fifth (st_bgn) and the sixth (st_end) fields indicate the beginning and the end index of the gated store buffer, respectively.

As the head μ-op is renamed, a SOB entry and a SRRT is allocated to the superblock. The p_srrt field of the SOB entry is made to point to the SRRT. As other μ-ops are renamed mio_cnt is incremented, and the tail_rnm is set when the tail is renamed. The st_bgn is updated when the first store is at rename, while st_end is updated when the last store is updated.

As a μ-op reaches the commit stage, the exe_cnt field of the SOB entry is incremented. The head of SOB is considered for commit when the exe_cnt equals the mio_cnt, and the tail_rnm (bit) field is set[5]. The register state held by SRRT is updated to BRRT; and the gated store buffer commits the memory state.

We have modeled SOB using a RAM array, and observed that the delay and power dissipated is lower than the ROB. Since the SOB consists of only eight or four entries, the total access delay and power dissipated is lower.

D. Physical Register Recycling

Physical registers are held in the conventional physical register file. The VMM marks the *non live-out* architected registers and finds the number of consumers for each one of them. This information is used to update the counter associated with the corresponding Physical Register. As the consumer μ-ops execute the counter, associated with the source physical registers, is decremented. The Physical Registers that have their counters equal to zero are freed by the conventional Register Recycling mechanism [12].

On the other hand, the *live-out* architected registers of a superblock are handled similar to the conventional out-of-order processor. Even after the superblock is committed the counters associated with these physical register are not equal to zero. This is because they hold the program register state and at least the BRRT holds reference to them. Only when another superblock with same *live-out* architected register is committed the physical register is freed.

III. Out-of-Order Logic

A FIFO based issue logic is used in order to implement out-of-order execution, in a complexity-effective manner. Such an issue logic was proposed earlier by Palacharla et al. [4]. Multiple FIFOs are used to issue μ-ops in parallel and out-of-order, while μ-ops from a FIFO are always issued in the program order.

A. Dependence-based Steering Heuristic

Since our proposed steering logic depends partially on the one proposed by Palacharla et al [4], we first describe Palacharla's approach and later point out its drawbacks.

Let I be the instruction that is ready to be dispatched. Depending upon the availability of I's operands, the steering decisions made are:

[3]Number of Architected Registers.

[4]Size depends upon number of Physical Registers.

[5]This implies the μ-ops have renamed and hence executed.

- If all the operands of I are ready, then steer I to an empty FIFO.
- If only one source operand of I is available, then steer I to a FIFO whose tail produces the required operand. If no such FIFO found steer I to an empty FIFO.
- If both the source operands of I is unavailable, then steer I to a FIFO whose tail is the operand producer of either of the operands, giving priority to left source operand.

If the desired FIFO is full or an empty FIFO is not available then dispatch is stalled. The steering logic required in the dispatch stage consists of a SRC_FIFO table. This table is indexed by physical register, and contains the identity of the FIFO buffer that contains the μ-op that produces the architected register value.

Figure 1 shows the performance of the above mentioned dependence-based dispatch policy when implemented in our co-designed processor.

B. Enhanced Steering Heuristic

As observed above, a major drawback of the dependence-based scheme is a high frequency of stalls incurred at the dispatch stage, as the number of FIFOs are halved. The three conditions that causes the dispatch to stall for the dependence-based scheme are:

- Rdy_no: Empty FIFO is unavailable for a μ-op whose source operands are ready at dispatch.
- Tail_no: Empty FIFO is unavailable for a μ-op, neither of whose source operands are ready nor are any of the producers a tail of a FIFO.
- Tail_FIFO: FIFO is available, but is full, for a μ-op whose producer is the tail of the FIFO.

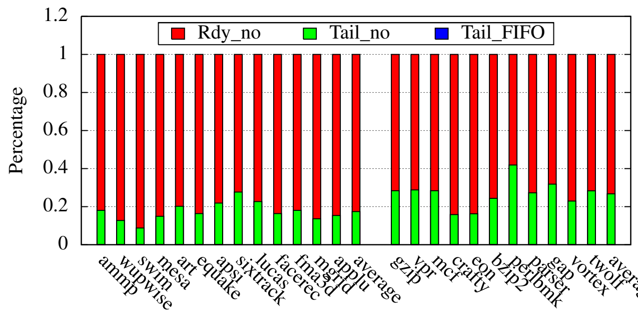


Fig. 4: Dispatch Stall Condition Distribution

Figure 4 provides distribution of the these three stalling conditions. Nearly 80% of stalls are due to Rdy_no, while Tail_FIFO hardly causes any stall. We exploit this observation by modifying the steering logic.

Our modified steering logic builds upon the dependence-based steering logic. It, however, reduces the stalls due to Rdy_no by steering a μ-op, whose operands are ready, to a FIFO whose tail is ready. The FIFO whose tail is ready implies that all μ-ops ahead of it in the FIFO must be ready, as the μ-ops are steered based on their dependencies.

Figure 5 shows the enhanced steering logic that requires an additional tail-check logic. Shaded blocks are those that have

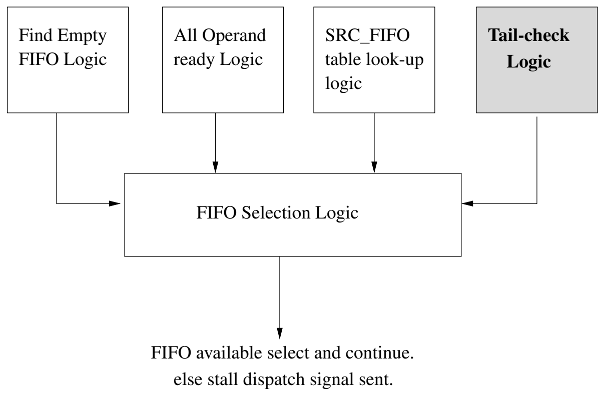

Find Empty FIFO Logic
All Operand ready Logic
SRC_FIFO table look-up logic
Tail-check Logic
FIFO Selection Logic
FIFO available select and continue.
else stall dispatch signal sent.

Fig. 5: Steering Logic

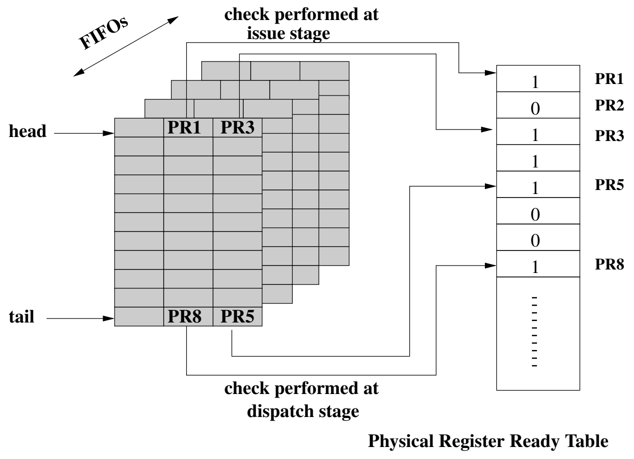

FIFOs
check performed at issue stage
head
tail
PR1
PR3
PR8
PR5
1
0
1
1
1
0
0
1
PR1
PR2
PR3
PR5
PR8
check performed at dispatch stage
Physical Register Ready Table

Fig. 6: Illustration of head and tail check

been added. The Tail-check logic basically uses the Register ready table, which is also checked by the head of FIFO at the issue stage, as illustrated in Figure 6.

The Physical Register Ready table has same rows as the SRC_FIFO table. However, the size of a column is smaller than SRC_FIFO as it contains just a single bit. This implies the access delay to this table is smaller than that for the SRC_FIFO table. Furthermore, since the check is made in parallel to other checks, as shown in Figure 5, the tail-check logic does not add to the critical path. However, it does dissipate extra power and is evaluated in Section IV.

As a consequence of the proposed enhancements to the steering logic, the likelihood of multiple ready μ-ops in a FIFO increases. This can be further exploited by increasing the issue width per FIFO. The μ-op immediately after the head will only be issue if the head is ready to issue.

We further try to decrease the stalls at dispatch by reducing stalls due to Tail_no. We reuse the Tail-check logic by steering a μ-op, that encounters Tail_no, to a FIFO whose tail is ready. We use the same intuition, as stated above, that a FIFO whose tail is empty is as good as an empty FIFO.

C. Early Release

Modern out-of-order processors, for example Alpha 21264 [13], speculate on the loads to hit in the L1-Cache. In order to guarantee back-to-back execution, dependent μ-ops of a load are issued speculatively. Issue Queue entries of all the issued μ-ops are held for a couple of cycles. In case of a miss-speculation, the μ-ops from the execution pipeline are squashed and issue queue entry is re-validated.

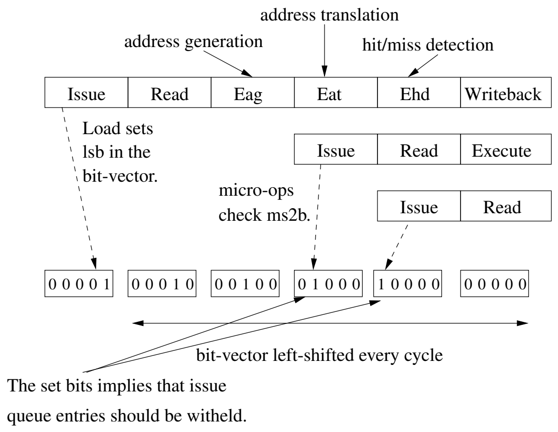


Fig. 7: Illustration of early release

This holding of issue-queue entries for a few number of cycles adds pressure on the issue queue. For FIFO based issue queues this has an even more dramatic impact on the performance, especially for the dependence-based steering heuristic.

We propose an early release mechanism, that releases issue queue entries, at the issue stage, for all the μ-ops issued in that cycle; given its safe to do so. The proposed early release mechanism is not just applicable to FIFO based issue logic, but can be applied to CAM based Issue logic, as it is.

Our proposal introduces two[6] bit-vectors that keep track of whether a load was issued in a given cycle for a small history. The length of these bitvectors is determined by the latency of the load μ-op, which is five in our case. At any given cycle, if a load issues the least significant bit is set. Both the bit-vectors are left-shifted, at every cycle, by one.

Figure 7 provides an illustration of the early release scheme. To determine whether a load was issued X cycles back, the bit at position $X + 1$[7] is checked. Since the bit-vectors are five bits wide, the set bit at most significant or the second most significant position implies a load was issued four or three cycles back, respectively. This information is sufficient to determine whether the issue queue entries, of the μ-ops issuing in the current cycle, could be released.

IV. Evaluation

A. Experiment Methodology

Our proposed HW/SW co-designed processor is modeled using PTLSim [14]. We have implemented a FIFO based out-of-order processor core along with a Cd-VM. Table I provides detailed information of the microarchitecture of the simulated processor.

[6]Since there are two Load Units.

[7]With the least significant bit position being zero.

4-way out-of-order Processor Parameters	
I-Cache	16 KB, 4-way, 64 B line, 2 cycle access
Branch Predictor	Combined Predictor 64 K 16-bit history 2-level, 64 K bi-modal, 1K RAS
Fetch Width	4 micro-ops / x86 instructions up-to 16 bytes long
Issue Width	4 (2 LD, 2 FP, 2 INT, 2 ST)
L1 Data Cache	32 KB, 4-way, 64 B line, 2 cycles
L2 Cache	256 KB, 16-way, 64 B line, 6 cycles
L3 Cache	4 MB, 32-way, 64 B line, 14 cycles
Main Memory	154 cycles
Rename	8 source, 4 destination operands
FIFO-based Issue Queue	8 entry, (2/4/8) FIFOs
CAM-based Issue Queue	16 entry, 1 LD, 2 ALU, 1 FP
Functional Units	2 LDU, 2 ALU, 2 AGU, 2 FPU
Register File	128-entry INT RF, 128-entry FP RF, 4 write ports each
Gated Store Buffer	32 entry
Load Fill Request Queue	8 entry
Miss Buffer	8 entry

TABLE I: Baseline processor configuration

We have evaluated our proposed scheme using the SPEC2000 benchmark suite. These benchmarks have been compiled with gcc version 4.1.3 using -O3. Using the developed infrastructure, we have simulated the benchmarks for 100 million x86 instructions after the initialization and a cache warm-up period of 2 million x86 instructions.

B. Enhanced Steering Heuristic performance

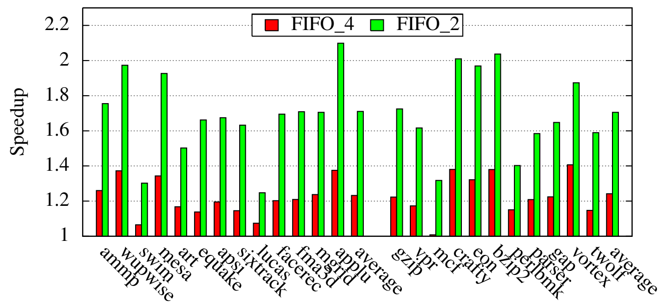


Fig. 8: Performance of Enhanced Steering Heuristic

Figure 8 shows the performance of our enhanced steering heuristic. The two bars show the normalized speedup of our heuristic with respect to the dependence-based steering heuristic. As evident from the figure we obtain speedup of nearly 70% and nearly 25% for a two FIFO and a four FIFO configuration, respectively.

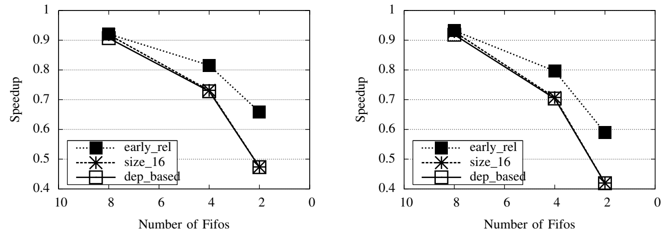
1) Stalls at Dispatch: As mentioned in Section III that our heuristic increases the dispatch throughput by reducing the unnecessary stalls. We observed that for a four FIFO configuration our heuristic reduces stalls at dispatch by 55% and 62% for SPECFP and SPECINT, respectively. For a two FIFO configuration we obtain even more reduction in stalls, as the speedup obtained is higher for a two FIFO configuration.

C. Detailed analysis

In order to better understand the performance benefits, we quantify them in this section. For this purpose we have chosen three FIFO configurations: two, four and eight.

It would also be interesting to compare FIFO based out-of-order processor to conventional issue logic based out-of-order processor. For this purpose we have replaced the FIFO logic with a CAM issue logic[8], in our out-of-order processor. All the numbers shown in this section are normalized to this CAM issue logic.

1) Effect of Early Release: Figure 9 shows the impact in performance of the early release mechanism if applied to a dependence-based heuristic, as shown by early_rel. Clearly, early_rel is 10% and 14% better than dep_based for a four FIFO configuration for SPECFP and SPECINT, respectively. Moreover, for a two FIFO configuration early release mechanism results in a gain of nearly 40% with respect to dep_based heuristic.

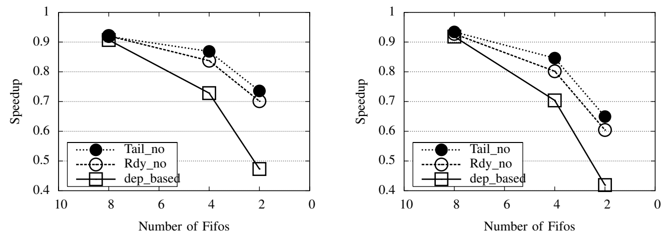


(a) SPECFP (b) SPECINT

Fig. 9: The Effect of early release

We also compare the early release to a FIFO with double the number of entries to sixteen, as shown by size_16 in Figure 9. Increasing the size of a FIFO merely adds entries to the tail of the FIFO. Whereas, the early release mechanism releases entries from the head, which are critical for steering heuristic.

2) Enhanced Steering Heuristics: Figure 10 shows the performance of our enhanced steering heuristics. Rdy_no is the one that reduces the stalls due to Rdy_no (see Section III-B), whereas Tail_no is the one that reduces the stalls due to both Rdy_no and Tail_no.

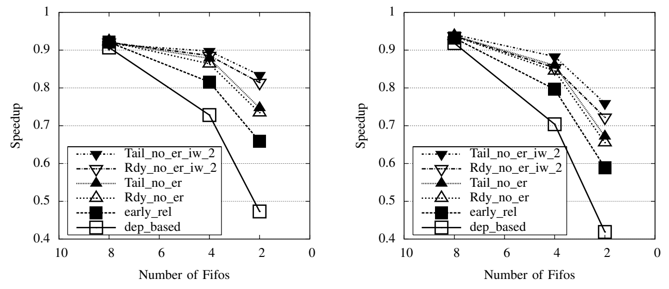


(a) SPECFP (b) SPECINT

Fig. 10: Enhanced Steering Heuristic

3) Effect of per FIFO Issue Width: The performance of enhanced heuristics with FIFOs that could issue two micro-ops is shown by Rdy_no_er_iw_2 and Tail_no_er_iw_2 in

[8]Details in Table I.



(a) SPECFP (b) SPECINT

Fig. 11: The Effect of Per FIFO Issue Width

Figure 11. Clearly, the benefit of increasing per FIFO issue width is evident for a two FIFO configuration. For a two FIFO configuration in particular with Tail_no_er heuristic we obtain a speedup of 13% and 10% for SPECFP and SPECINT, respectively with respect to the configuration where per FIFO issue width is one.

D. Commit Logic Study

As explained earlier the atomic commit constraint of superblocks causes the conventional ROB based processor to stall. Figure 12 shows the reduction in stalls obtained by our SOB/SRRT commit logic with respect to conventional ROB based processor. Nearly 25% and 22% of resource related stalls are reduced in SPECFP and SPECINT, respectively.

The larger superblocks in SPECFP leads to more stalls, as it results in more μ-ops waiting in the ROB. Hence the performance improvement obtained in SPECFP is notably larger than that in SPECINT. Figure 12 shows that in SPECFP we obtain a speedup of 12%, and 1.5% in SPECINT, with respect to conventional ROB based processor.

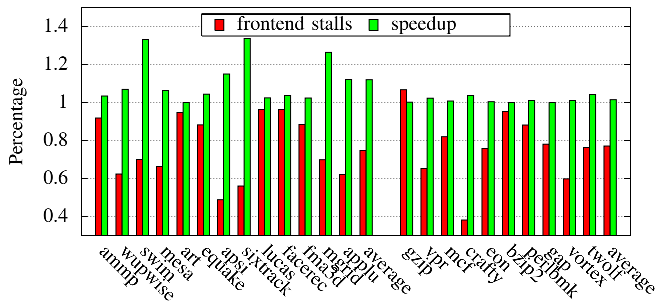


Fig. 12: SOB vs ROB

In Figure 13 we show the impact in performance by limiting the number of simultaneous superblocks in the pipeline. In order to limit the number of simultaneous superblocks, we limit the number of SRRTs. This is exactly equivalent of limiting number of SOB entries. We show speedups of three different configurations with eight, four and two SRRTs. The speedups are normalized to the one with unbounded number of SRRTs.

As is evident from the figure that an eight SRRT configuration is as good as an unbounded SRRT configuration. Moreover, a four SRRT configuration provides performance

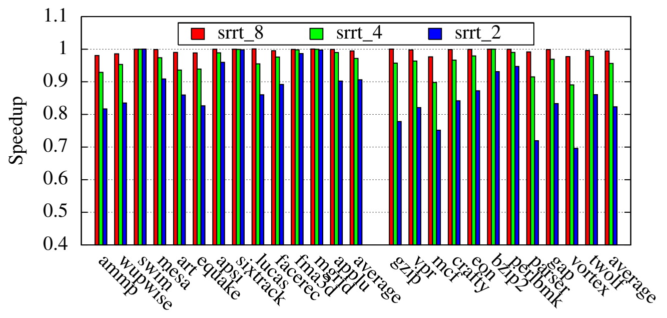


Fig. 13: The Effect of SRRT

that is within 3% of the unbounded case. Hence, a pool of four SRRTs and a SOB with four entries seems to be a good trade-off between performance and complexity.

E. Power and Energy Results

We use Wattch 1.02 [15] to quantify the power dissipated and energy consumed. The power results shown here are using the conditional clock gating[15]. All the results in this section are normalized to the CAM issue logic configuration.

1) Power Results: Figure 14 shows the normalized power dissipated by the co-designed processor implementing different steering heuristics[9]. With respect to the processor with dependence based steering heuristic our enhanced steering heuristic dissipates 12% and 14% more power for SPECFP and SPECINT, respectively.

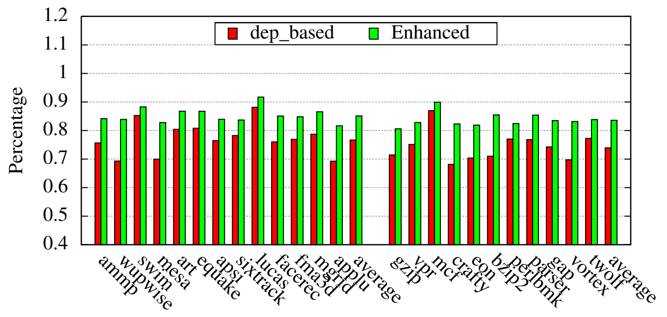
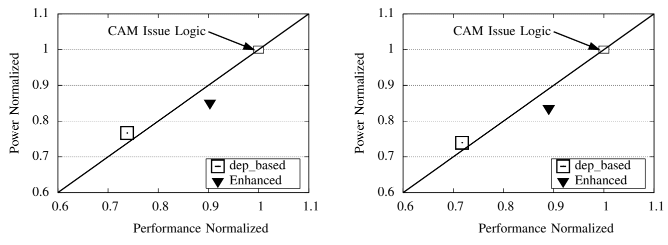


Fig. 14: Normalized Power Results

We also measured the power dissipated by the individual units. Together the issue and the dispatch logic of the proposed enhanced steering heuristic dissipates nearly 56% and 52% more power for SPECFP and SPECINT, respectively, compared to dependence-based heuristic. The power dissipated by these units is still nearly 2% of the power dissipated by the processor.

Even though our heuristic dissipates more power, it is still more power-efficient than both the dependence-based heuristic and the CAM issue logic. By plotting power against performance normalized to CAM issue logic we quantify this claim in Figure 15. Any point that falls below the line $y = x$ is more power-efficient than CAM issue logic.

[9]Four FIFO configurations.



(a) SPECFP (b) SPECINT

Fig. 15: Normalized Power and Performance Results. The point 1,1 corresponds to CAM issue logic.

2) Power Results SOB and SRRT: We modeled both the SOB and SRRT using RAM array, to quantify the power dissipated. We assumed an eight-entry SOB and eight SRRT configuration[10]. The SOB has eight ports and each entry is five bytes wide.

We observe that our proposed processor dissipates 6% less power compared to a ROB-based processor. This main gain is due to the fact the SOB is multiple-folds smaller than a ROB. Even though there are multiple rename tables, at any given cycle at most two SRRTs will be accessed. A write access to a SRRT is made when a μ-op being renamed is a live-out of the superblock, while a read access to a SRRT is made when state is committed to BRRT.

3) Energy Results: For devices that run on battery, energy is an important design constraint. If a processor finishes executing the same task by consuming lesser energy, it would be preferable[11]. Figure 16 shows that the enhanced steering heuristic consumes 7% less energy than CAM issue logic. The dependence-based heuristic, on the other hand, consumes 3-4% more energy.

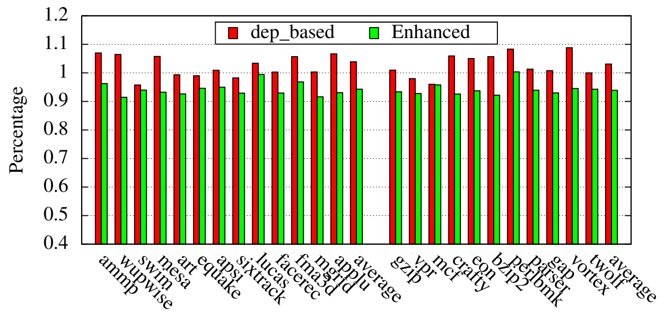


Fig. 16: Normalized Energy Results

V. Related Work

IBM Daisy [7] and BOA [8] projects have proposed a VMM to binary translate Power PC instructions to VLIW instructions. The generated VLIW instructions are scheduled by the VMM at the scope of tree-regions, superblocks or atomic blocks. This provides a much better scope in scheduling

[10]Four-entry SOB and four SRRTs are sufficient, but a bigger configuration provides an upper bound on power dissipation.

[11]Given the power and execution time are reasonable.

than that provided by basic blocks. Using a VLIW microarchitecture drastically cuts down the complexity in the logic and power.

Transmeta Crusoe [16] is a commercial co-designed processor that uses a Code Morphing Software (CMS) [17] to perform multi-staged emulation. Shadow copy of Register File is used to check-point the register state before a superblock starts executing. Working copy, as the name suggests, holds the working register set of the superblock. Whereas, the memory state is held in gated store buffers [11]. A special commit operation updates the register and the memory state, at once. Similar to DAISY [7] and BOA [8], Crusoe uses VLIW processors to cut down complexity and power. Aliasing HW is added to [18] detect any memory ordering violation.

The only similarity between our work and that of IBM DAISY, BOA or Crusoe is the VMM part. Our microarchitecture is different since we use a FIFO based out-of-order logic. Secondly, we use SOB and SRRT based atomic commit and reordering mechanism.

Akkary et al. [19] have proposed a large instruction window processor using checkpoint processing and recovery. They replace the conventional ROB with a checkpoint buffer, which is similar to the SOB that we have proposed. However, they do not execute superblock, instead checkpoints are taken at branches that have low-confidence of being predicted correctly. We instead checkpoint when the first instruction of the superblock is renamed. Their proposal still requires a buffer to hold instructions in order to bulk commit. We instead propose per superblock map table that holds the register state of the superblock.

Moreover, since we use VMM to form superblock and get rid of confidence-based predictor. VMM also helps in providing early register recycling as number of consumer of non live-out register are determined. More importantly, our focus in this paper is not to provide a large instruction window processor, but to provide a co-designed out-of-order processor which is more power-efficient.

Conventional Issue Queue logic is based on CAM and RAM structures [4], leading to high complexity and power dissipation [5]. Palacharla et al. [4] propose a multiple FIFO based issue logic, where instructions are issued from the head of FIFOs. They propose a dependence-based heuristic as described in the paper to steer instructions to the FIFOs. As shown in the Section I, the performance with this heuristic declines sharply as the number of FIFOs are halved.

Canal and González [6] proposed several schemes to issue instructions. In one of their schemes instructions are placed in a buffer that is indexed by the physical register identifier. It is based on the observations that nearly one-quarter of the dynamic instructions have one of their operands available at dispatch. Their another scheme is based on computing the issue cycle of each instruction at dispatch.

VI. Conclusion

This paper presents a complexity-effective co-designed out-of-order processor. Our proposed steering heuristic, compared to the dependence-based heuristic, obtains speedups of 25% and 24% for SPECINT and SPECFP, respectively. We have also shown that our proposed steering heuristic based processor consumes 10% less energy than the previously proposed steering heuristic.

We have also proposed an early issue queue entry releasing mechanism. Issue queue entries are released at the issue stage; given that no loads were issued, a fixed number of cycles, earlier. This helps in reducing the pressure on issue queues.

In order to efficiently execute superblocks, we codesign the commit logic. We have proposed two structures - Superblock Order Buffer (SOB) and Superblock Register Rename Tables (SRRT) - in order to acheive this. Such a processor dissipates 6% less power than a conventional ROB based out-of-order processor and performs 12% better over a conventional ROB based processor.

References

[1] J. Smith *et al.*, *Virtual Machines: A Versatile Platform for Systems and Processes*. Elsevier Inc., 2005.

[2] W. W. Hwu *et al.*, “The superblock: An effective technique for vliw and superscalar compilation,” *THE JOURNAL OF SUPERCOMPUTING*, 1993.

[3] S. Patel *et al.*, “rePLay: A Hardware Framework for Dynamic Optimization,” *IEEE Transactions on Computers*, 2001.

[4] S. Palacharla *et al.*, “Complexity-effective superscalar processors,” in *IEEE Intl. Symp. on Computer Architecture*, 1997.

[5] M. Gowan, “Power considerations in the design of the alpha 21264 microprocessor,” in *Design Automation Conference*, 1998.

[6] R. Canal *et al.*, “A low-complexity issue logic,” in *Intl. conference on Supercomputing*, 2000.

[7] K. Ebcioglu *et al.*, “DAISY: Dynamic compilation for 100% architectural compatibility,” in *IEEE Intl. Symp. on Computer Architecture*, 1997.

[8] E. Altman *et al.*, “BOA: The Architecture of a Binary Translation Processor,” IBM, Tech. Rep., 1999.

[9] A. Deb *et al.*, “SoftHV : A HW/SW Co-designed Processor with Horizontal and Vertical Fusion,” in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011.

[10] G. Hinton *et al.*, “The Microarchitecture of the Pentium 4 Processor,” *Intel Technology Journal*, 2001.

[11] M. J. Wing *et al.*, “Gated store buffer for an advanced microprocessor,” U.S. Patent 6 011 908, 2000.

[12] J. Smith and G. Sohi, “The microarchitecture of superscalar processors,” *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609 –1624, 1995.

[13] R. Kessler *et al.*, “The Alpha 21264 Microprocessor Architecture,” in *IEEE Intl. Conf. on Computer Design*, 1998.

[14] M. Yourst, “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator,” in *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2007.

[15] D. Brooks *et al.*, “Wattch: a framework for architectural-level power analysis and optimizations,” in *Proceedings of the 27th annual international symposium on Computer architecture*, 2000.

[16] A. Klaiber, “The technology behind Crusoe Processors,” *Transmeta Technical Brief*, 2000.

[17] J. Dehnert *et al.*, “The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges.” in *IEEE Intl. Symp. on Code Generation and Optimization*, 2003.

[18] E. J. Kelly, “Memory controller for a microprocessor for detecting a failure of speculation on the physical nature of a component being addressed,” U.S. Patent 5 832 205, 1998.

[19] H. Akkary *et al.*, “Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.