Revolver: Processor Architecture for Power Efficient Loop Execution

Abstract

With the rise of mobile and cloud-based computing, modern processor design has become the task of achieving maximum power efficiency at specific performance targets. This trend, coupled with dwindling improvements in single-threaded performance, has led architects to predominately focus on energy efficiency. In this paper we note that for the majority of benchmarks, a substantial portion of execution time is spent executing simple loops. Capitalizing on the frequency of loops, we design an out-of-order processor architecture that achieves an aggressive level of performance while minimizing the energy consumed during the execution of loops. The Revolver architecture achieves energy efficiency during loop execution by enabling "in-place execution" of loops within the processor's out-of-order backend. Essentially, a few static instances of each loop instruction are dispatched to the out-of-order execution core by the processor frontend. The static instruction instances may each be executed multiple times in order to complete all necessary loop iterations. During loop execution the processor frontend, including instruction fetch, branch prediction, decode, allocation, and dispatch logic, can be completely clock gated. Additionally we propose a mechanism to pre-execute future loop iteration load instructions, thereby realizing parallelism beyond the loop iterations currently executing within the processor core. Employing Revolver across three benchmark suites, we eliminate 20, 55, and 84% of all frontend instruction dispatches. Overall, we find Revolver maintains performance, while resulting in 5.3%-18.3% energy-delay benefit over loop buffers or micro-op cache techniques alone.

1. Introduction and Motivation

Although transistor densities continue to scale, the associated per-transistor energy benefit normally obtained from successive process generations is rapidly disappearing. This phenomena, known as the end of Dennard scaling, forces architects to limit transistor switching by means of structural optimization, functional specialization, or clock regulation [9, 10]. Furthermore, the need for improved computational efficiency has been highlighted by increased demand in the power conscious mobile and server markets.

To cope with these increasing energy constraints, future out-of-order processors must further streamline common execution patterns, thereby eliminating unnecessary pipeline activity. For common applications on modern processors, the energy required by instruction execution is relatively small. Instead these applications expend the majority of energy on control overheads, such as instruction fetch and scheduling [26]. This energy distribution is particularly visible in recent mobile processors, such as ARM's Cortex-A15, where frontend power accounts for 25-45% of all core energy [2, 20].

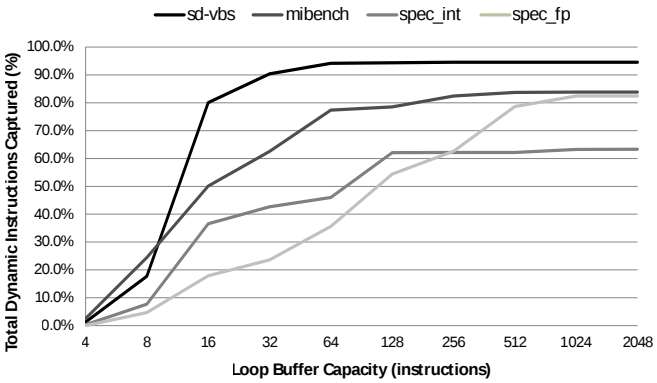

Figure 1: Ideal Loop Buffer Performance.

To reduce these frontend energy overheads, architects have proposed many pipeline-centric instruction caching mechanisms that capitalize on temporal instruction locality [6, 21, 3, 19, 15]. The majority of these proposals target capturing loop instructions, in a decoded or encoded form, into a small buffer for inexpensive retrieval on future iterations. Figure 1 shows the percentage of instruction accesses that can be serviced from ideal[1] loop buffers of varying sizes across multiple benchmark suites. As seen, loop buffers can be quite effective, with a common loop buffer size of 32 instructions able to capture 24-90% of all instruction accesses.

The observed effectiveness of loop buffers has resulted in their rapid industrial adoption [11, 20, 25, 1, 18]. Shown in Figure 2, industry has progressively implemented more deeply embedded loop buffers in processor pipelines as a means to reduce frontend activity. Notably, this trend has created the ability to bypass not only fetch, but decode overheads as well. Although early designs, like AMD's 29k, only saved instruction cache accesses and supported very small loops, more modern designs from Intel and ARM bypass more pipeline stages and store significantly larger loops.

However, despite potential energy savings, no commercial out-of-order processor has attempted to bypass all frontend pipeline stages, including allocation and dispatch, during loop execution. This boundary has not been pushed because of assumed obstacles related to program ordering, operand dependence linking, and resource allocation.

In this paper we propose the Revolver architecture, an aggressive out-of-order processor which obviates the complexities related to moving loop buffering into the processor back-

[1] No branch mispredictions or limitations on in-loop control.

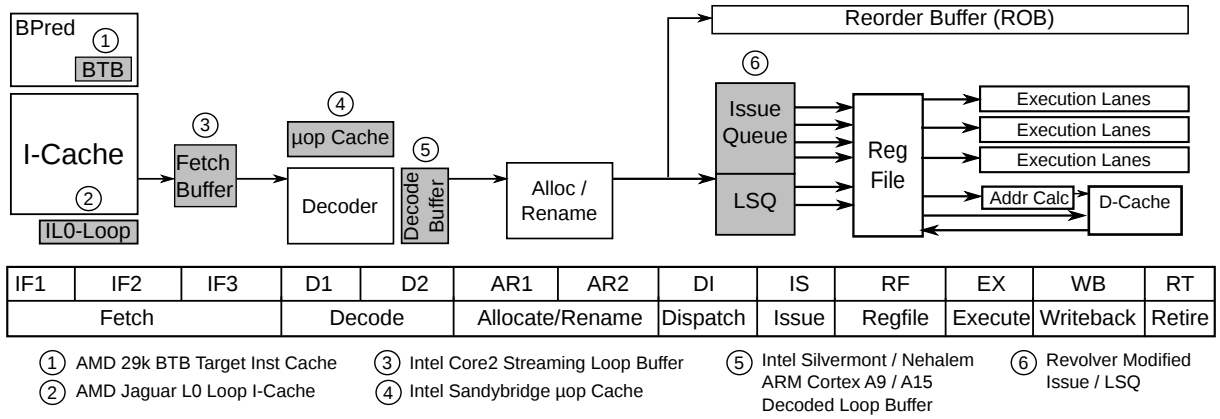

Figure 2: Instruction Reuse Caching Methods. Structures modified/added by each method highlighted.

end. During loop execution only a few static instances of each loop instruction are dispatched to the out-of-order backend. After which, each instruction instance may be executed in-place multiple times in order to complete all necessary loop iterations. During subsequent executions, no additional resources are allocated and no frontend structures are accessed. Revolver enables this through a series of insights and straightforward modifications to a traditional processor pipeline. Additionally we propose a mechanism which enables the pre-execution of future loop iteration loads, enabling single-cycle loads in many instances on future loop iterations.

Revolver's loop execution is particularly novel due to its:

- Moving operand dependence linking into the out-of-order backend
- Eliminating the need to re-allocate resources between instruction re-executions
- Reducing the branch misprediction penalty for variable iteration loops
- Enabling the pre-execution of future loop iteration loads

2. Overview

The Revolver architecture is an aggressive out-of-order processor which supports two primary modes of operation: non-loop and loop. The overall design is similar to a normal out-of-order processor with few structural differences.

During non-loop execution mode, instructions flow through the entire processor pipeline as in a conventional out-of-order core. The key structural difference between Revolver and a traditional out-of-order core is the lack of a register allocation table (RAT) within the processor frontend. Instead, dependence linking between instructions is performed in the processor backend by a simple structure called the Tag Propagation Unit (TPU) that is accessed in parallel with issue select. Other than this structural modification, which is detailed further in Section 4.2, the Revolver backend operates like a normal out-of-order processor during non-loop mode.

In loop mode, the Revolver architecture works by detecting and dispatching loop bodies to an out-of-order backend which is capable of self-iterating, thus eliminating all frontend activity during loop execution.

To enable loop mode, additional loop detection logic is placed at decode. Once a loop's starting address and number of required resources[2] have been calculated, a subsequent decoding of the first loop instruction initiates loop mode. During loop mode, the loop body is unrolled as many times as allowed by the resources present within the out-of-order backend. Fundamental to Revolver's operation is its ability to eliminate the need for any additional resource allocation once a loop has been dispatched. With respect to the frontend, allocation of most resources proceeds normally. However, as later detailed, allocation of destination registers requires special handling. After dispatch, the loop body resides within the issue queue and will execute multiple times until all loop iterations are complete.

With respect to Revolver's backend, the primary innovation is the ability to allow loop instructions to maintain their provided resources across multiple executions. Instructions retain issue queue entries after issue select and reuse them immediately for the next loop iteration upon commit. The load/store queue is also modified to enable reuse of entries while properly maintaining program order. Finally, as detailed in Section 4.4, each result producing loop instruction simply alternates writing one of two pre-allocated physical registers. Revolver's TPU is designed to allow dependent instructions to properly access source registers even with alternating register dependencies.

On loop exit, all instructions are removed from the out-of-order backend and the loop fall through path is immediately dispatched. Immediate dispatch is possible since, before clock-gating, the processor frontend redirects to the fall through path after successful loop dispatch.

The following sections provide a more detailed and clarified operation of Revolver's loop execution mode.

[2]Resources being physical registers as well as issue queue, load queue, and store queue entries.

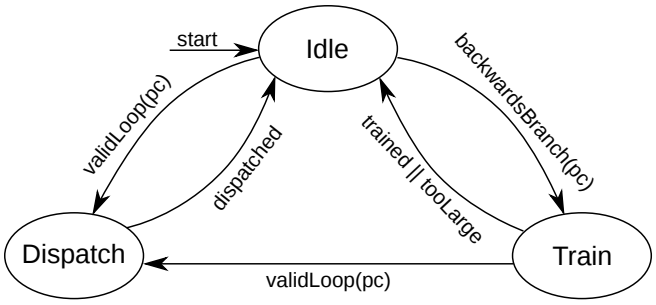

Figure 3: Loop Detection Finite State Machine.

v	start addr	fallthrough addr	num_insts	max_unroll	profitability

Figure 4: Loop Address Table (LAT) Structure.

3. Loop Detection and Training

Loop-mode dispatch in Revolver is regulated by two key mechanisms: detection logic in the processor frontend and feedback provided by the processor backend. The processor frontend is responsible for detecting and guiding loop body dispatch. The processor backend provides feedback relating to the profitability of loop execution. If a loop is deemed unprofitable, backend feedback will result in the frontend disabling future loop-mode dispatches.

3.1. Detection Operation

The loop detection logic in Revolver is similar to that used in previous loop buffer proposals [21, 16]. Loop detection is controlled by the simple finite state machine (FSM) shown in Figure 3. This state machine operates in one of three possible states: *Idle*, *Train*, or *Dispatch*.

In the *Idle* state, instructions propagate normally through decode until the start of a profitable loop is identified or a taken PC-relative backwards[3] branch/jump is encountered. Profitable loops are identified by consulting a small (4 entry) structure containing known loops called the Loop Address Table (LAT). The LAT, detailed in Figure 4, is a small direct-mapped structure that records information relating to the loops composition and profitability. In the event a profitable loop is encountered, the detection FSM transitions to the *Dispatch* state and begins loop-mode dispatch. If no profitable loop is identified and a backwards branch or jump is encountered, the detection FSM instead transitions to the *Train* state.

The *Train* state exists to record a previously unknown loop's start address, end address, and allowable unroll factor[4] in the LAT. Once entering the *Train* state, until the loop ending branch, resources required by the loop body are recorded. After the ending branch is encountered, the loop information is entered into the LAT and the FSM transitions to the *Idle* state again. If a loop requires too many resources to be contained by the backend, the LAT is not updated. The fall through address for a loop is set to the next sequential memory address. It should be noted that, if a loop start instruction is encountered at any time, training will be aborted and the FSM will immediately transition to the *Dispatch* state. Finally, if another backwards control instruction is encountered, the resource usage information is reset and the *Train* state is re-entered.

In the *Dispatch* state, the decode logic guides the dispatch of loop instructions into the out-of-order backend by specially tagging them as loop instructions. The loop body is unrolled as many times as possible, subject to available backend resources. After unrolling all loop instances, the frontend is redirected to the fall through path. Once the fall through path fills the frontend, the frontend stalls and clock gates.

3.2. Detection Discussion

In this section we highlight multiple aspects of the previously described loop detection mechanism.

First, the Revolver architecture allows almost unlimited control flow, including function calls/returns, within a loop body. The only limitation is that predicted execution paths are statically determined at the time of loop dispatch. Thus loops with unstable control flow make poor candidates for loop-mode and backend feedback is responsible for eventually disabling loop-mode dispatch of such loops.

Secondly, loop-mode is disabled for a given loop if it contains *serializing* instructions. Examples of *serializing* instructions include system calls, memory barriers, and load-linked/store-conditional pairs.

3.3. Training Feedback

The backend feedback serves one primary purpose: relaying information about the profitability of a loop body.

Shown in Figure 4, the LAT contains a *profitability* field that acts as a 4-bit saturating counter. Upon insertion into the LAT, loops receive a default profitability of 8. Loop-mode dispatch is enabled if profitability is greater than or equal to 8. Feedback from the backend adjusts a loops profitability to impact its likelihood of loop-mode dispatch.

The following factors impact a trained loops profitability. If the dispatched unrolled loop body iterates more than twice the profitability is incremented by 2, otherwise it is decremented by 2. If a branch within the loop body mispredicts to an address that other than the fall through, the loops profitability is set to zero. For disabled loops, the frontend increments the profitability by 1 for every two sequential successful dispatches observed.

Adjusting by these factors ensures that only highly profitable loops are enabled for loop-mode dispatch, thus mitigating any potential negative performance impact while capturing the majority of potential benefit.

[3] Branch target instruction address less than current instruction addresss.
[4] As constrained by physical resources.

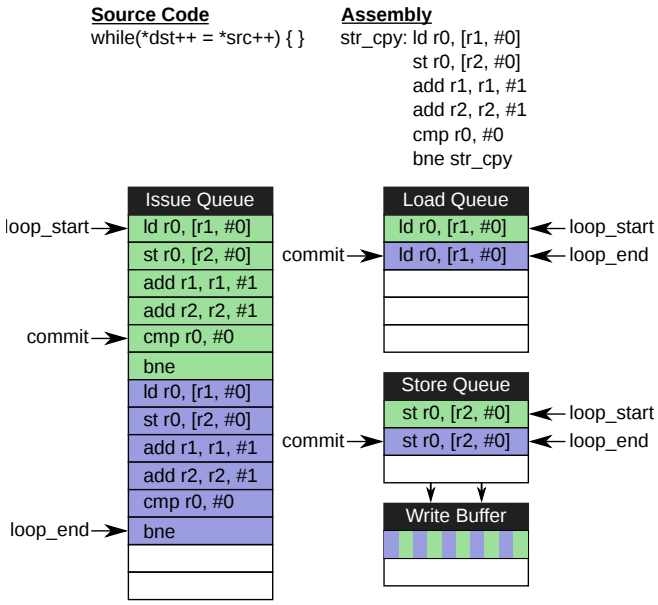

Figure 5: Revolver Out-of-Order Backend Example.

4. Loop Execution

Revolver's out-of-order backend supports loop-mode execution through a series of simple modifications to the issue queue, load/store queue, and commit logic. These modifications allow loop instructions within the backend to be executed multiple times in order to complete all necessary loop iterations. During subsequent executions, all instructions retain their initially allocated resources. In this section we provide an overview of backend operation as well as the required structural modifications.

4.1. Overview

To summarize backend functionality, Figure 5 provides an example of loop-mode execution performing a string copy operation[5]. In this example, the six instruction string copy loop is unrolled twice into the issue queue. The first (green) loop body performs all odd-numbered iterations while the second (blue) loop body completes all even-numbered loop iterations. This partial unrolling allows parallelism across iterations during loop-mode execution.

For maintenance and ordering, loop start and end pointers are tracked by each backend queue. The queues also maintain a commit pointer that identifies their oldest, uncommitted entry. Shown in Figure 5, the commit pointer walks from the loop start until the loop end entry. After committing the loop end instruction, the commit pointer wraps to the loop start to begin committing the next loop iteration. Upon commit, issue queue entries are reset and can be immediately reused for the next loop iteration. Load queue entries are simply invalidated on commit, while store queue entries drain into a small write combining buffer. Draining stores into a write buffer allows

[5]Copy bytes from source array to destination array until encountering null.

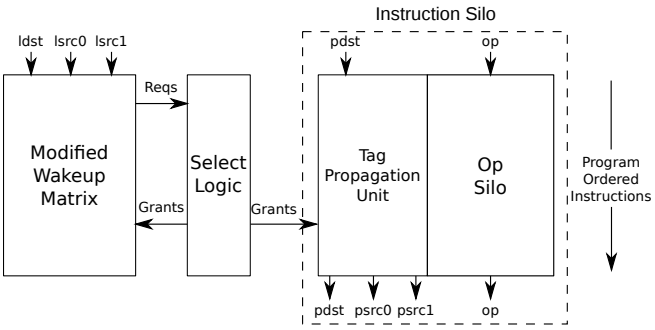

Figure 6: Revolver Out-of-Order Issue Queue Design.

the store queue entry to be immediately reused in the next loop iteration. In the rare instance when a store cannot drain into the write buffer, commit stalls. Finally, loop-mode reuse of LSQ entries requires no modification to the age-based ordering logic of the LSQ. LSQ ordering logic must already support wrap-around based upon the relative position of a commit pointer in a conventional out-of-order. To demonstrate this, given the example's relative position of the commit pointer in Figure 5, the second (blue) loop body store is properly ordered as older than the first (green) loop body store.

Loop-mode execution completes when any branch, loop terminating or otherwise, resolves to the loop's fall through path. Allowing any branch which resolves to the fall through path to terminate loop-mode execution means that loops end gracefully even on iteration counts that are not evenly divisible by the unrolling factor. Additionally, this resolution handling allows *break* statements within loop bodies to quickly resolve without being treated as mispredicts. After termination, the loop's out-of-order resources may be freed and the fall through path immediately proceeds through dispatch.

In the remainder of this section we describe the precise structural modifications necessary to support this operation.

4.2. Scheduler Modifications

Key to supporting loop-mode execution in Revolver is the structure and operation of the instruction scheduler. The overall design of the Revolver's scheduler, shown in Figure 6, is similar to the matrix scheduler presented in [24] by Sassone et al. The main components of this scheduler are a wakeup array for identifying ready instructions, select logic for arbitration between ready instructions, and the instruction silo for producing the opcode and physical register identifiers of selected instructions.

Three primary modifications make the Revolver scheduler distinctive from [24]. First, Revolver's scheduler is strictly managed as a queue and maintains program order among entries. Secondly, the wakeup array utilizes logical register identifiers and position dependence, rather than physical register identifiers for wakeup. Finally, Revolver uses a Tag Propagation Unit (TPU) to provide physical register mappings, instead of a frontend RAT combined with backend tag storage.

With the high level structure of the scheduler defined, the

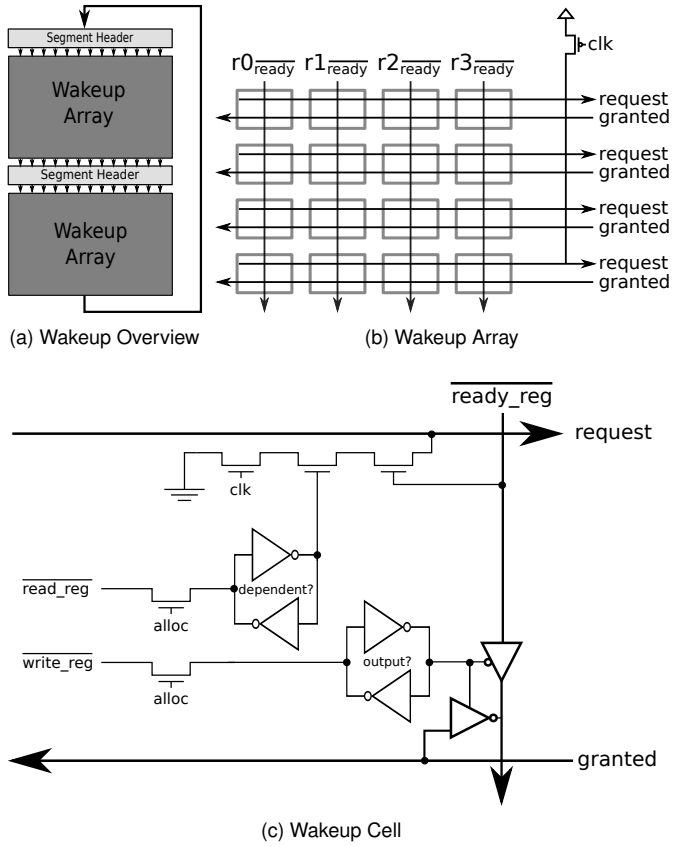

Figure 7: Revolver Wakeup Logic

following Sections 4.3 and 4.4 explain the function and operation of the Revolver's instruction wakeup and Tag Propagation Unit.

4.3. Wakeup Logic

The purpose of instruction wakeup in an out-of-order processor is to observe results generated by scheduled instructions in order to identify new instructions capable of being executed. To perform this task, Revolver's instruction wakeup utilizes program-based ordering of instructions and logical register identifier broadcasts. This differs from many conventional schedulers, which use physical register-based broadcasts and do not require ordering. This primary benefit from this organization is that Revolver is able to perform instruction wakeup without requiring frontend renaming.

4.3.1. Wakeup Operation Figure 7 shows the Revolver wakeup logic at multiple levels of granularity. At the highest level, in Figure 7a, instruction wakeup is organized as a segmented, program-ordered circular queue. The segmented wakeup arrays within the scheduler are interconnected via a unidirectional ring that transmits logical register broadcasts. In our designs, segments are sized equal to the machine's dispatch width and broadcasts along the ring interconnect travel at the rate of eight instruction entries per cycle. At any given time, one segment in the machine will be designated the *architected* segment, with incoming operands implicitly ready.

Inside the wakeup array, shown in Figure 7b, instructions are distributed along rows, while columns correspond to logical registers. Upon allocation into the wakeup array, instructions mark their respective logical source and destination registers. Unscheduled instructions within the wakeup array cause their downstream logical destination register column to be deasserted. This deassertion prevents younger, downstream dependent instructions from waking up. Once all necessary source register broadcasts are received, an instruction requests scheduling. After being granted by select, the instruction asserts its destination register to wakeup younger dependent instructions.

Close examination of the wakeup array's logic cell, shown in Figure 7c, demonstrates how the wakeup operation is possible. Our modified wakeup cell design draws from earlier work in [24]. The wakeup cell holds two state bits that designate the instruction as sourcing or producing a logical register. The request signal to the select logic is implemented in dynamic logic. If an instruction is dependent and the incoming ready signal is not asserted, the request signal is pulled down. This pulldown operation works as a logical NOR. If no unbroadcast dependents remain, the request signal to select will remain asserted. With respect to outputs, if an instruction has not been scheduled and produces the logical register, the outgoing ready signal will be deasserted. The ready signal and grant signal are implemented with static logic. Once the result producing instruction is granted, the grant signal will result in the downstream ready being asserted. For loop-mode operation, after commit, the grant signal is deasserted and the cell is free to reevaluate based upon a new incoming ready signal.

4.3.2. Wakeup Example In this section we work through a simple example of wakeup logic operation. Figure 8 provides an example with three instructions. All three instructions must be serially executed due to dependencies on logical register *r0*. The diagram is color coded with ready register columns colored in blue, active request signals represented in green, and active grants represented in red.

On cycle 0, the ready signals for logical registers propagate downwards unless inhibited by an instruction. As *Instruction 1* produces register *r0*, it gates the downstream ready broadcast until it has been scheduled. This prevents improper wakeup of *Instruction 2* and *Instruction 3*. Logical registers *r1* and *r2* are produced by no instruction, thus their ready broadcasts are uninhibited and continue propagation. During cycle 0, *Instruction 1* satisfies all dependencies and asserts its request vector.

In cycle 1, the grant signal for *Instruction 1* returns acknowledging issue. Signalling issue, *Instruction 1* ceases inhibition of the downstream *r0* ready signal. With *r0* now asserted, *Instruction 2* asserts its request vector. As *Instruction 2* has not been granted, it maintains downstream inhibition of *r0*, preventing *Instruction 3* from waking up.

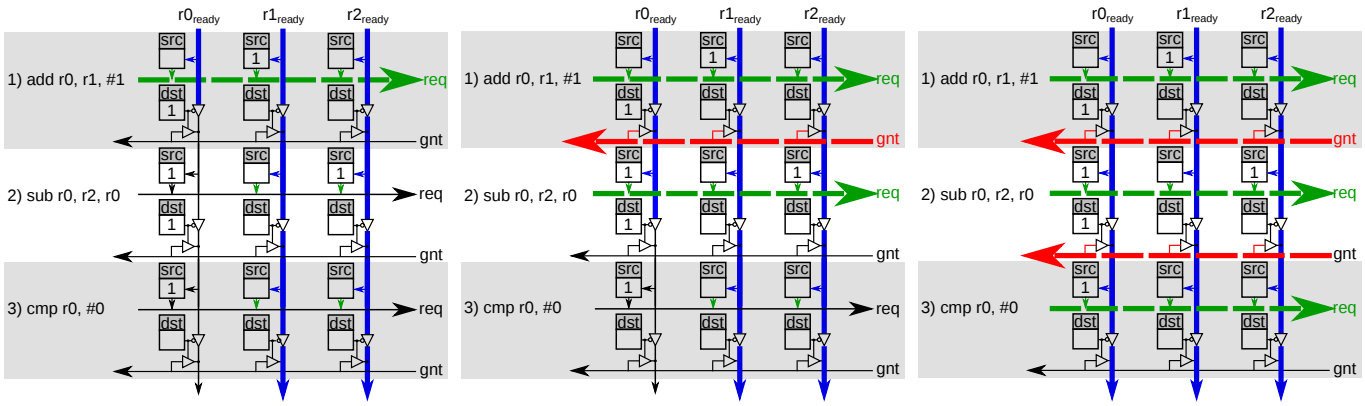

Figure 8: Revolver Wakeup Example

Finally on cycle 3, the grant for *Instruction 2* returns and *r0* is uninhibited. With *r0* now asserted, the final instruction wakes up and assets its request vector.

This example demonstrates how the use of program-based ordering enables Revolver to perform instruction wakeup on logical register identifiers, in contrast to conventional out-of-orders which require frontend renaming and physical register-based wakeup.

4.4. Tag Propagation Unit

Without frontend renaming, Revolver needs a mechanism to properly map logical registers to physical register identifiers. In this section we discuss the reasoning behind the use of the TPU in Revolver and its operation.

4.4.1. Enabler of loop-mode execution The reason Revolver requires a TPU is to enable reuse of physical registers during loop-mode execution. As noted earlier, Revolver does not require any additional resource allocations between loop iterations. The largest obstacle to avoiding allocation is physical register management. This is so because, after committing, a loop instruction should be free to begin speculative execution of the next loop iteration. This is impossible, however, if an instruction only has access to a single physical destination register. After commit, the contents of the physical register may be required by dependent instructions and are part of the architected state. Thus, to begin speculative execution of the next loop iteration, access to an alternative physical register identifier is required.

Revolver solves this issue by providing each result producing loop-mode instruction with two physical destination registers. As loop instructions iterate, they simply alternate writing between their two destination registers. This alternation of writes, known in other literature as double buffering, ensures the previous state is maintained while speculative computation is being performed. To clarify, after iteration $N+1$ commits, an instruction may reuse the destination register from iteration N on iteration $N+2$. This is safe because, upon commit, the $N+1$ destination register holds the architectural

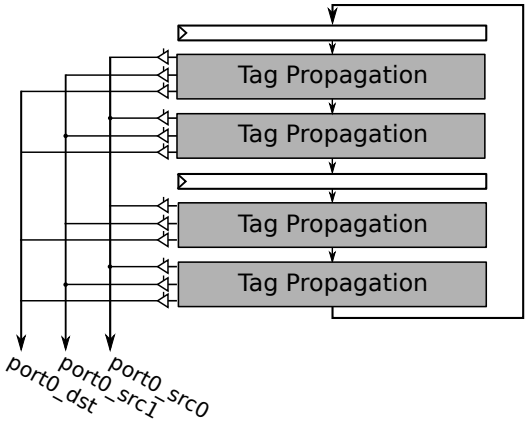

Figure 9: Tag Propagation Unit.

state and no more instructions are dependent on the iteration N destination register. Any instructions dependent on the $N+1$ value continue to source it from the alternative register.

This double buffering technique enables instructions to speculatively write output registers. However, by dynamically changing output registers, additional functionality must be added to properly maintain dependencies between instructions. The TPU's function is to perform this dynamic linkage between dependent instructions and source registers.

4.4.2. Structure and Operation Figure 9 shows the high level structure of the TPU. Revolver's TPU is structured similarly to the wakeup logic discussed in Section 4.3. Like the wakeup logic, the TPU is composed of partitions interconnected along a unidirectional ring interconnect. The ring is composed of multiple channels, where each channel corresponds to a logical register and carries the current physical register mapping. Thus to obtain source register identifiers, all an instruction must do is source the appropriate logical register channel. Instructions present in the scheduler change the logical register mapping of their output register by simply overwriting the appropriate output column. Since instructions are stored in program order, this operation guarantees downstream instructions will obtain proper source register mappings. At

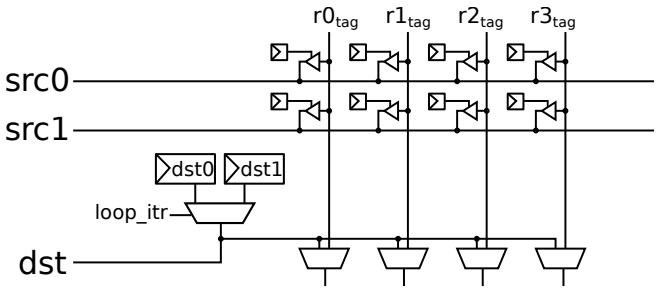

Figure 10: Tag Propagation Unit Cell Design.

all times, one segment is deemed *architected* and retains the architected register mapping in latches. This *architected* latch rotates throughout the TPU as segments commit.

Figure 10 shows the propagation logic cell used in the construction of the TPU. As seen there are multiple logical register columns that carry physical register identifiers. Instructions source their operand tags from corresponding columns and drive their destination onto the appropriate output column. Also within the figure, we show how loop-mode instructions alternate between writing two physical destination registers. Essentially a single bit of state records if the instruction has executed an odd or even number of times, this bit controls a mux that drives the appropriate destination register identifier onto the output.

4.4.3. Checkpoints and Register Reclamation In addition to dynamic dependence linking, Revolver's TPU provides benefits relating to checkpointing and branch misprediction recovery. Every instruction within the TPU has access to a valid physical register mapping for every architectural register. Thus Revolver effectively provides per-instruction renaming checkpoints. If any instruction mispredicts, downstream instructions are simply flushed and the mappings from the branch instruction propagate to all newly scheduled instructions. In comparison, RAT-based renaming encounters significant additional complexity in order to support checkpoints [22]. This checkpoint support is largely a byproduct of Revolver's progam ordering within the issue queue.

In non-loop mode, overwritten registers are reclaimed on commit, as in conventional processors. On branch mispredictions or loop-mode exits, the TPU is walked forward from the terminating branch to reclaim physical registers.

4.5. Load and Store Support

The overall structure of Revolver's LSQ is shown in Figure 11. Other than the additional tracking of loop start and end pointers, few differences exist between Revolver's LSQ and that of a conventional out-of-order. Loads and stores receive their respective LSQ entries prior to dispatch and retain them until the instruction exits the out-of-order backend. As noted in Section 4.1, loop-mode load and store instructions are free to reuse their allocated LSQ entries to execute multiple loop iterations. This due to two factors: 1) The immediate "freeing" of an LSQ entry upon commit and 2) The use of position-based

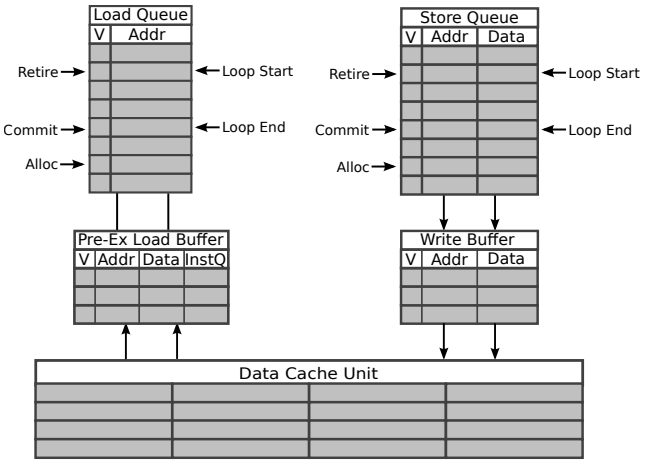

Figure 11: LSQ and Cache Interface.

age logic in modern processor's LSQs.

LSQ entries are immediately freed upon commit by two means depending on whether the committing instruction is a load or store. In the event of a load, the load queue entry is simply reset to allow future load execution. Stores however must be written back to memory. To enable immediate freeing of store queue entries, stores are drained into a write-combining buffer that sits between the store queue and the L1 cache interface. If a store cannot drain into the write buffer, commit stalls. This is however a rare occurrence as due to the impact of write-combining and the fact we target an ISA with a relaxed consistency model that places very few ordering restrictions on write-combining.

Existing age-based ordering techniques work in Revolver as they are based upon the relative position of a commit pointer. The only difference during loop-mode operation is that Revolver's commit pointer wraps from *loop end* to *loop start*. Whereas a normal LSQ only wraps the commit pointer based upon the physical end and start of the queue.

The final portion of Revolver's LSQ, the pre-executed load buffer, is an enhancement enabled by loop mode execution and will be discussed in the following section.

5. Load Pre-Execution

In this section we detail an extension to loop-mode that enables the pre-execution of future loop iteration loads. Pre-executing future loads realizes parallelism beyond the processor's instruction window and can be used to enable zero-latency loads. The remainder of this section covers the insight behind load pre-execution, the conditions where load pre-execution is possible, and why load pre-execution would be untenable in a conventional out-of-order.

5.1. Optimization Insight

During loop execution in an out-of-order processor, loads from within the loop body are repeatedly executed until all necessary iterations complete. Due to the recurrent nature

of loops, these loads often have highly predictable address patterns. Our load pre-execution mechanism aims to exploit these predictable loads.

Revisiting the example from Figure 5, the string copy loop simply strides through memory copying bytes from a source array into a destination array. Thus, the load addresses in consecutive iterations are perfectly predictable. In a conventional processor, the dynamic instances of each load receive unique issue queue and load queue entries. In Revolver however, a load dispatched by the frontend is statically bound to fixed entries for all loop iterations. This static binding makes it easy to observe when an entry is performing loads that follow a simple pattern. In Figure 5, since the string copy loop was unrolled by a factor of two, the first load queue entry will be observed striding through memory, reading consecutive even-addressed bytes from memory.

The insight behind load pre-execution is that, when these patterns are recognized, it is possible to speculatively initiate future iteration loads. On the next iteration, if a load was pre-executed, it will not pay the L1 cache access latency and will complete after verifying the pre-executed load address. This technique yields a performance benefit when the out-of-order execution window is insufficient to hide a load's latency. The next subsections detail the supported access patterns and hardware implementation.

5.2. Supported Address Patterns

In Revolver we support three primary access patterns for load pre-execution: stride, constant, and pointer-based addressing. For each of these access patterns we place simple pattern identification hardware alongside the pre-executed load buffer.

Striding memory accesses are the most common addressing pattern, as many loops iterate over arrays of data. To identify stride-based addressing we simply compute the address delta between two consecutive loads. If a third load from the same load queue entry matches the predicted stride, the stride is verified and the next load will be pre-executed. Constant loads, the second most common pattern, occur when loads continuously read from the same address. Constant loads exist primarily due to stack-allocated variables and pointer aliasing. The stride-based prediction hardware also handles constant loads, as they are a special case of a zero-sized stride. Finally we support pointer-based addressing, where the value returned by the current load is used as the next address. This captures many simple linked list traversals.

Once a pattern is recognized, the pre-executed load buffer speculatively initiates the next iteration memory access. This buffer, shown in Figure 11, sits between the load queue and the L1 cache interface. Once the next iteration load executes, the value is claimed and the buffer may initiate the next iteration load. If any store aliases with the pre-executed load, the entry is invalidated to maintain coherency.

5.3. Scheduler Modification

With the pre-execution buffer and supported access patterns defined, we now detail how Revolver's out-of-order scheduler can take advantage of pre-execution.

In many out-of-order designs, operations dependent on loads are speculatively scheduled assuming an L1 cache hit latency. When a pre-executed load returns from memory, the corresponding issue queue entry is notified that the load has been pre-executed. Once scheduled, this load will speculatively wake dependent operations on the next cycle instead of waiting for the L1 cache access latency. If the predicted address is incorrect, the scheduler must perform a cancel and re-issue operation. The design could be more aggressive than as described here and wake dependent operations before the load is capable of being scheduled, however our evaluated implementation does not support this.

Finally, it should be noted that performing this scheduler optimization in a conventional out-of-order is untenable, as there is no relation between static load instructions and issue queue entries.

6. Evaluation Methodology

To evaluate Revolver, we use a combination of cycle accurate simulation and power modeling. For performance simulation, a custom, cycle-accurate out-of-order core model was implemented within the gem5 simulator infrastructure [7].

Two baseline out-of-order configurations, shown in Table 1, were used in our evaluation. The *OO2* configuration is a small 2-wide out-of-order processor configured similarly to the recently announced Intel Silvermont architecture [18]. The *OO4* design represents a more aggressive 4-wide architecture with the window size and execution resources scaled up from the OO2 configuration. Two Revolver designs are compared against these baselines, a 2-wide (*Rev2*) and 4-wide (*Rev4*) configuration. All designs utilize aggressive memory systems with prefetchers at every cache level.

For competitive baselines, each configuration can optionally use a 32-μop loop buffer (*LB*) or a 1.5K μop cache (μC), similar to recent Intel and ARM designs [18, 25, 17, 20].

Power modeling was performed through a correlated and extended version of McPAT [22]. Models for loop buffers and μop caches were added for proper energy accounting. All energy numbers represent core energy, including the L1 caches.

We simulated a wide variety of applications from the San Diego Vision Benchmark Suite (SD-VBS), MiBench, and SPEC2006 [27, 12, 14]. All applications were compiled for the ARMv7 ISA on gcc 4.7.2 with full optimizations (*-O3*), vectorization, and link-time optimization(*-flto*) enabled. On SD-VBS, simulation was limited to the instrumented regions of interest. For MiBench, entire applications were simulated. Finally for SPEC2006, a SimPoint simulation methodology was employed, resulting in the suite being represented by 177

	OO2, Small Out-of-Order	OO4, Large Out-of-Order
Branch Predictor	Combined bimodal (16k entry) / gshare (16k entry), selector (16k entry), 32 entry RAS, 2k BTB	
Out-of-Order Core	2GHz, 2-wide fetch/commit, 6-wide issue, 32 ROB/IQ, 12 LQ, 8 SQ, 8 WB, 48 Int PRF, 64 FP PRF, aggressive memory speculation, speculative scheduling, 13-stage pipeline	2GHz, 4-wide fetch/commit, 8-wide issue, 64 ROB/IQ, 24 LQ, 16 SQ, 8 WB, 80 Int PRF 96 FP PRF, aggressive memory speculation, speculative scheduling, 13-stage pipeline
Functional Units	2 Int ALU (1-cycle), 1 Int Mult/Div (3-cycle/20-cycle), 1 LD (1-cycle AGU), 1 ST (1-cycle), 2 SIMD units (1-cycle), 2 FP Add/Mult (5-cycle), 1 FP Div/Sqrt (10-cycle)	3 Int ALU (1-cycle), 1 Int Mult/Div (3-cycle/20-cycle), 2 LD (1-cycle AGU), 1 ST (1-cycle), 2 SIMD units (1-cycle), 2 FP Add/Mult (5-cycle), 1 FP Div/Sqrt (10-cycle)
Memory System	L1 ICache 32KB, 2-way, 64B line size (2-cycle), 2-ahead sequential prefetcher L1 DCache 32KB, 4-way, 64B line size (3-cycle), 2-ahead stride prefetcher L2 Unified 256KB, 8-way, 64B line size (12-cycle), 2-ahead combined stride/sequential prefetcher L3 Unified 4MB, 16-way, 64B line size (24-cycle), 4-ahead combined stride/sequential prefetcher Off-Chip Memory: 2GB DDR3-1600	

Table 1: Common Processor Configurations.

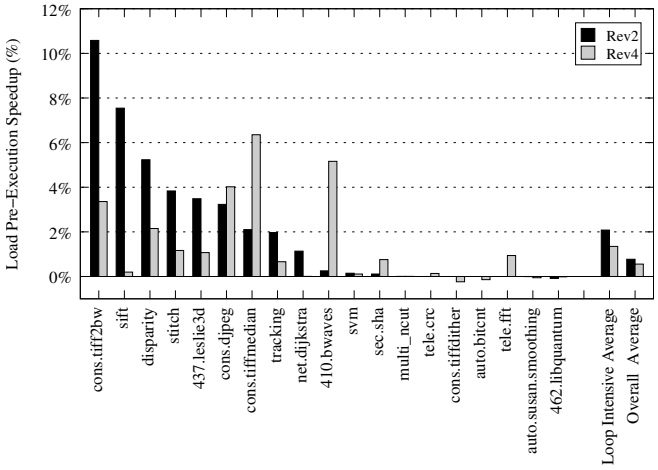

Figure 12: Loop Intensive Load Pre-Execution Speedup.

100M instruction simulation points [13].

7. Experimental Results

Our evaluation is divided into three subsections. In subsection 7.1, we detail the performance benefit obtained from pre-execution of loads during loop execution. Subsection 7.2 provides detailed per-benchmark analysis of Revolvers impact on the total number of frontend instruction dispatches. Finally, subsection 7.3 evaluates the overall energy-delay impact of Revolver.

7.1. Load Pre-Execution

Although Revolver is targeted primarily towards energy conservation, load pre-execution enables Revolver to extract memory level parallelism beyond the currently active instruction window. In Figure 12, we show the performance benefit from load pre-execution obtained by the *Rev2* and *Rev4* configurations on loop intensive benchmarks [6]. On loop intensive code, load pre-execution benefits *Rev2* and *Rev4* by 2.1% and 1.4% respectively. Across all benchmarks, including non-loop intensive codes, the overall benefit is 0.8% and 0.6% for *Rev2* and *Rev4*.

In general, *Rev2* sees more benefit from load pre-execution than *Rev4*. This occurs because the larger out-of-order window of *Rev4* often hides the latencies of loads captured by pre-execution on *Rev2*. *Rev4* obtains more benfit on select benchmarks, such as *410.bwaves*, because its larger instruction window is capable of capturing certain loops that *Rev2* cannot. Finally, the reduced benefit observed on non-loop intensive code, relative to loop-intensive code, is expected as load pre-execution is only triggered during loop-mode.

It should be noted that many design parameters can impact the benefit obtained from load pre-execution. In particular, our moderate L1 cache latency and aggressive L1 prefetching reduce the observed benefit from load pre-execution.

7.2. Frontend Dispatch Impact

Through loop execution mode, Revolver is capable of eliminating many frontend instruction dispatches. Removing frontend dispatches allows Revolver to save energy, even beyond loop buffers and μop caches, as multiple pipeline stages between decode and execute are elided.

To demonstrate this benefit, Figure 13 details the fraction of instructions dispatched by Revolver, in comparison to a traditional out-of-order core, across all three benchmark suites. Each configuration is normalized against the equivalent width out-of-order baseline. In general we observe Revolver is capable of eliminating 84%, 55%, and 20% of all instruction dispatches across the SD-VBS, MiBench, and SPEC2006 benchmark suites respectively.

For SD-VBS, shown in Figure 13a, Revolver performs exceptionally well. This benchmark suite contains many data intensive processing loops that, even with vectorization enabled, iterate numerous times. Image segmentation (*multi_ncut*) re-

[6] Defined as executing more than 50% of all instructions in loop-mode.

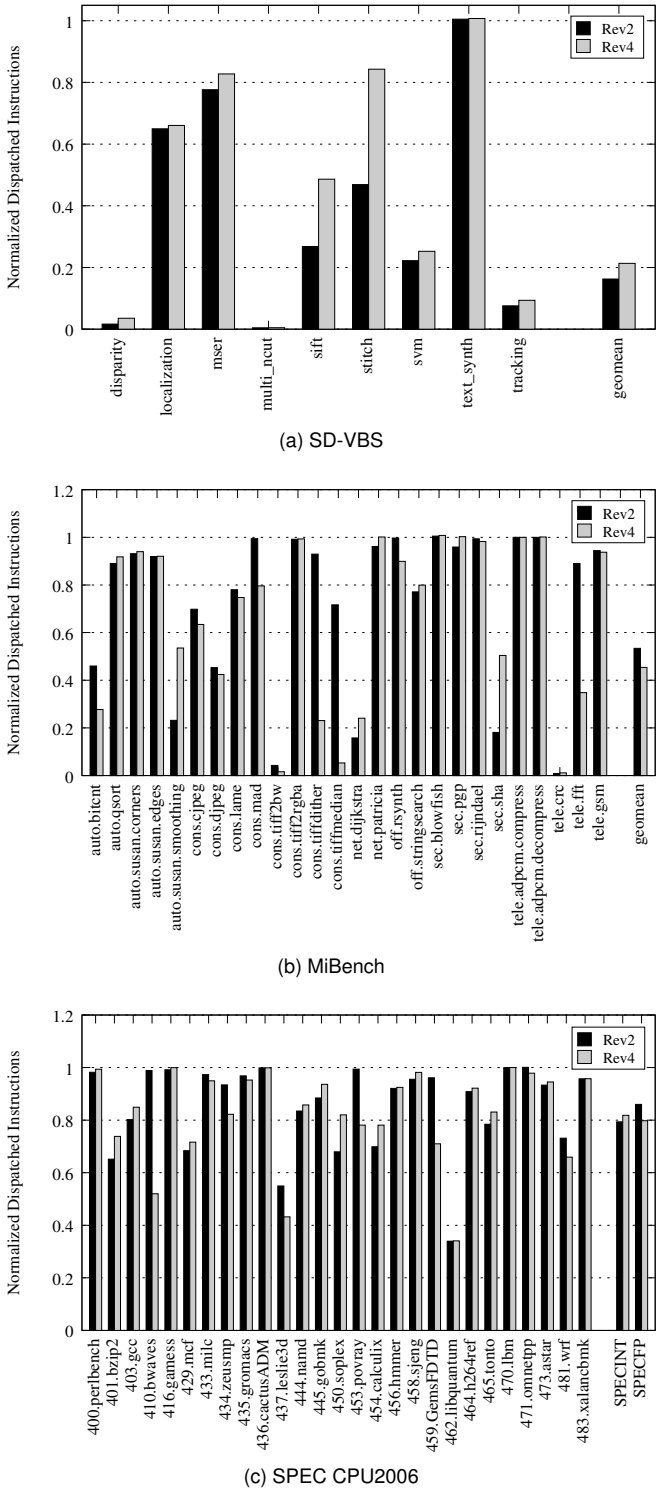

Figure 13: Normalized Number of Dispatched Instructions. Designs normalized against equivalent baseline.

ceives the most benefit from Revolver, successfully eliminating 99.6% of all frontend instruction dispatches. The only benchmark that receives no benefit from Revolver is texture synthesis (*text_synth*). The lack of benefit is due to unstable, nested control flow within inner-most loops that lead to loop-mode being disabled. Overall, *Rev2* and *Rev4* configurations reduce instruction dispatches by 84% and 79% against their respective baselines.

In Figure 13b the instruction dispatch results for MiBench are presented. Revolver successfully eliminates 47% and 55% of all dispatches on *Rev2* and *Rev4* respectively. *Rev4* eliminates more dispatches than *Rev2* because it captures more loops with its larger available instruction window. In general Revolver eliminates many instruction dispatches on MiBench, though its ability to eliminate dispatches is hindered on some benchmarks by the presence of serializing instructions within inner-most loops.

Finally, Figure 13c shows the normalized instruction dispatches for the SPEC2006 benchmark suite. Breakdowns for the SPECFP and SPECINT subsets are also shown, with Revolver eliminating approximately 20% (geomean) of all dispatches on each. Amongst our benchmark suites, the SPEC2006 suite receives the least benefit from Revolver. This is expected since, as shown in Figure 1, SPEC2006 has the fewest capturable loops. From SPECINT, *462.libquantum* receives the most benefit from Revolover, eliminating 66% of all instruction dispatches. SPECFP, as shown in Figure 1, is dominated by the execution of very large loops. Regardless, Revolver is able to eliminate 20% of instruction dispatches, with the larger instruction window of *Rev4* capturing more loops than that of *Rev2*.

Overall, we find Revolver to be quite successful on eliminating many frontend dispatches across the three benchmark suites.

7.3. Overall Energy and Performance

To evaluate Revolver in terms of energy and performance, we compare our two Revolver (*Rev2/Rev4*) configurations against out-of-order baselines (*OO2/OO4*) with 32-μop loop buffers (*LB*) and 1.5K μop caches (μC). Additionally, we evaluate the energy-delay of Revolver when equipped with a 1.5K μop cache.

Figure 14 presents the energy-delay product for each configuration normalized against a conventional out-of-order without loop buffers or μop caches. Figure 14a presents results for the smaller out-of-order configurations, while Figure 14b presents results for the large out-of-order designs. Energy numbers represent core and L1 cache energy inclusive. We have omitted presenting delay numbers separately as the difference was negligible between Revolver and the traditional out-of-order designs ($<$1% geomean).

For the small out-of-order designs shown in Figure 14a, we note multiple trends. First, the benchmark suites perform as expected with Revolver extracting the most energy benefit

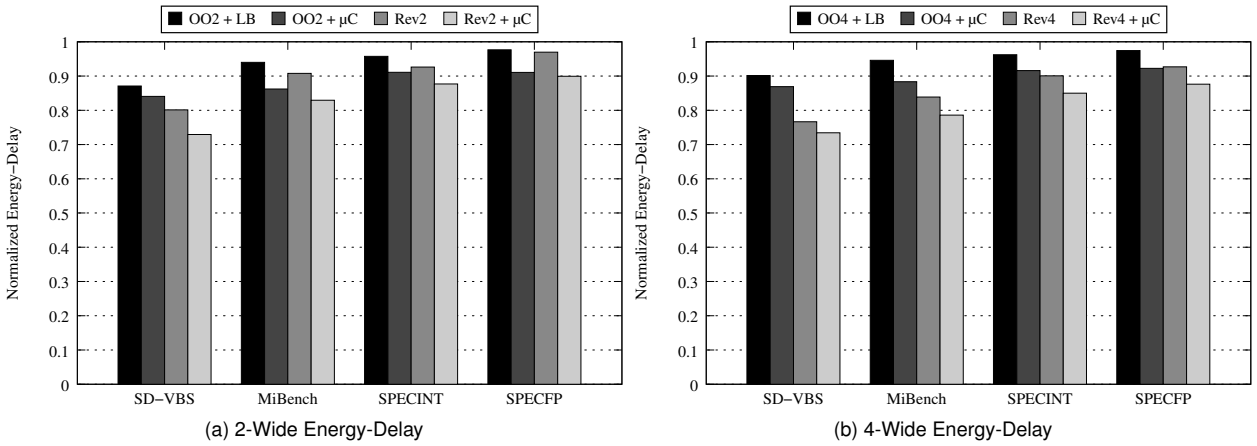

Figure 14: Overall Energy-Delay Comparison. Normalized against OO2 and OO4 baselines.

from SD-VBS, followed by MiBench and SPEC. Secondly, Revolver always outperforms the loop buffer equipped out-of-order, due to reduced energy costs while capturing similarly sized loops. Third, the μop cache always outperforms the loop buffer with respect to entire benchmark suites. However, for some specific benchmarks[7] that spend the majority of time in simple loops, energy-delay performance is superior for the loop buffer due to its reduced access energy. Fourth, for benchmark suites with fewer capturable loops (MiBench and SPEC), the μop cache outperforms the *Rev2* configuration. Lastly, *Rev2* with a μop cache exhibits the best energy-delay performance across all benchmarks.

On the large out-of-order designs in Figure 14b, Revolver demonstrates even greater energy-delay benefit due to its ability to capture larger loops. Across the benchmark suites, *Rev4* outperforms traditional out-of-orders with loop buffers or μop caches on all benchmark suites except SPECFP. Again, this is expected due to the numerous large loops that cannot be captured by Revolver in SPECFP. However, when combined with a μop cache, Revolver outperforms all other configurations.

Results from Figure 14 are summarized in Table 2 to show exact differences between the alternative configurations. Overall, Revolver can result in up to 18.3% energy-delay improvement over comparable baselines.

8. Related Work

In addition to the industrial works presented earlier, multiple academic works have investigated methods to improve loop execution performance or energy efficiency.

The most related works to our own are [16] by Hu et al. and [23] by Pratas et. al. These works also attempt to buffer loops within the out-of-order backend, thus eliminating instruction dispatch overheads. However, both works require backend communication with the frontend in order to obtain new renaming information, destination registers, and LSQ allocations during loop execution. This communication is performed serially on in-order instruction issue [16] or loop commit [23]. As designed, these works primarily save energy related to opcode movement into the out-of-order backend. Thus, Revolver's primary benefit over these designs is the complete removal of additional allocations during loop execution and any potential overhead from serialized frontend communication.

To further performance and energy efficiency on loop execution, Clark et al. [8] propose VEAL, a custom accelerator targeted towards offloading loop execution. As with all accelerators, the subset of kernels acceptable for execution on VEAL is determined by the overheads of offloading, algorithm suitability, and quantity of work available. Thus the subsets of loops preferable for execution on VEAL and Revolver differ.

Finally, in terms of loop buffers and tightly integrated caches for energy reduction, many academic works exist [19, 6, 21, 15, 5, 4]. Of the loop buffer works, each use an algorithm similar to Revolver in order to detect and initiate loop dispatch.

9. Conclusion

In this paper we have presented the Revolver architecture, an aggressive out-of-order design targeted towards minimizing energy during the execution of loops. Revolver achieves energy-efficiency during loop execution by enabling in-place execution within the processor's out-of-order backend. Through in-place loop execution, Revolver eliminates frontend energy overheads originating from pipeline activity and resource allocation. These energy benefits exceed those traditionally achieved through loop buffers or μop caches. Additionally, we propose load pre-execution, a novel mechanism to increase performance during loop execution by hiding L1 cache access latencies. Overall, we observe a 5.3%-18.3% energy-delay benefit beyond a traditional out-of-order with loop buffers or μop caches.

[7] disparity, multi_ncut, tracking, etc.

	SD-VBS	MiBench	SPECINT	SPECFP
Rev2 vs. *OO2+LB*	8.7%	3.5%	3.4%	0.7%
Rev2+μC vs. *OO2+μC*	15.3%	4.0%	3.9%	1.3%
Rev4 vs. *OO4+LB*	17.6%	12.8%	6.9%	5.1%
Rev4+μC vs. *OO4+μC*	18.3%	12.4%	7.8%	5.3%

Table 2: Revolver Energy-Delay Improvement.

References

[1] AMD Jaguar Software Optimization Guide. [Online]. Available: http://support.amd.com/us/Processor_TechDocs/52128_16h_Software_Opt_Guide.zip

[2] "NVIDIA Tegra 4 Family CPU Architecture," NVIDIA, Tech. Rep., 2013. [Online]. Available: http://www.nvidia.com/docs/IO/116757/NVIDIA_Quad_a15_whitepaper_FINALv2.pdf

[3] M. Alidina, G. Burns, C. Holmqvist, E. Morgan, D. Rhodes, S. Simanapalli, and M. Thierbach, "DSP16000: A High Performance, Low-Power Dual-MAC DSP Core for Communications Applications," in *CICC'98*, 1998, pp. 119–122.

[4] T. Anderson and S. Agarwala, "Effective Hardware-Based Two-Way Loop Cache for High Performance Low Power Processors," in *ICCD-18*, 2000, pp. 403–407.

[5] R. Bajwa, M. Hiraki, H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki, "Instruction Buffering to Reduce Power in Processors for Signal Processing," *VLSI Systems, IEEE Transactions on*, vol. 5, no. 4, pp. 417–424, 1997.

[6] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis, "Energy and Performance Improvements in Microprocessor Design Using a Loop Cache," in *ICCD-17*, 1999, pp. 378–383.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[8] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *ISCA-35*, 2008, pp. 389–400.

[9] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, 1974.

[10] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *ISCA-38*, 2011, pp. 365–376.

[11] D. J. Everitt, "Inexpensive Performance Using the Am29000," *Microprocessors and Microsystems*, vol. 14, no. 6, pp. 397 – 406, 1990.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.

[13] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and More Flexible Program Phase Analysis," in *Journal of Instruction Level Parallelism*, 2005.

[14] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[15] M. Hiraki, R. Bajwa, H. Kojima, D. Gorny, K. Nitta, and A. Shri, "Stage-Skip Pipeline: A Low Power Processor Architecture Using a Decoded Instruction Buffer," in *ISLPED'96*, 1996, pp. 353–358.

[16] J. Hu, N. Vijaykrishnan, S. Kim, M. Kandemir, and M. Irwin, "Scheduling Reusable Instructions for Power Reduction," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 1, 2004, pp. 148–153 Vol.1.

[17] D. Kanter. (2010, September) Intel's Sandy Bridge Microarchitecture. Available: http://www.realworldtech.com/sandy-bridge

[18] D. Kanter. (2013, May) Silvermont, Intel's Low Power Architecture. Available: http://www.realworldtech.com/silvermont

[19] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," in *MICRO-30*, 1997, pp. 184–193.

[20] T. Lanier, "Exploring the Design of the Cortex-A15 Processor," ARM, Tech. Rep., 2011.

[21] L. H. Lee, B. Moyer, and J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches for Embedded Applications with Small Tight Loops," in *ISLPED'99*, 1999, pp. 267–269.

[22] S. Li, J.-H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO-42*, 2009, pp. 469–480.

[23] F. Pratas, G. Gaydadjiev, M. Berekovic, L. Sousa, and S. Kaxiras, "Low Power Microarchitecture with Instruction Reuse," in *CF-5*. New York, NY, USA: ACM, 2008, pp. 149–158.

[24] P. G. Sassone, J. Rupley, II, E. Brekelbaum, G. H. Loh, and B. Black, "Matrix Scheduler Reloaded," in *ISCA-34*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 335–346.

[25] R. Singhal, "Inside Intel Next Generation Nehalem Microarchitecture," in *Hot Chips*, vol. 20, 2008.

[26] A. Sodani, "Race to Exascale: Challenges and Opportunities." Presented at the 44th International Symposium on Microarchitecture, Porto Alegre, Brazil, 2011.

[27] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "SD-VBS: The San Diego Vision Benchmark Suite," in *IISWC'09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 55–64.