ReDSOC: Recycling Data Slack in Out-of-Order Cores

Abstract

In order to operate reliably and produce expected outputs, modern processors set timing margins conservatively at design time to support extreme variations in workload and environment, imposing a high cost in performance and energy efficiency. The relentless pressure to improve execution bandwidth, especially for emerging workloads like machine learning, have exacerbated this problem, since they require instructions with increasingly diverse semantics, leading to datapaths with a large gap between best-case and worst-case timing. In practice, data slack, the unutilized portion of the clock period due to inactive critical paths in a circuit, can often be as high as half of the clock period.

In this paper we propose ReDSOC, which dynamically identifies data slack and aggressively recycles it. It is implemented via transparent latching, atop the data bypass network among functional in Out-Of-Order (OOO) cores. ReDSOC recycles the data slack from a producer operation, by starting the execution of dependent consumer operations at the exact instant of the producer's completion. Recycling data slack over multiple operations executing on functional units such as ALUs, allows acceleration of these data sequences and improves performance. Further, ReDSOC performs select-free and slack-aware OOO instruction scheduling to support this aggressive operation execution mechanism.

ReDSOC is implemented atop OOO cores of different sizes via the Gem5 simulator, and tested on a variety of general purpose and machine learning benchmarks. The implementation achieves average speedups in the range of 5% to 25% across the different cores and benchmarks. Further, it is shown to be 2-10x times more efficient at improving performance in comparison to other proposals.

1. Introduction

Modern processing architectures are designed to be reliable. They are designed to operate correctly and efficiently on diverse workloads across varying environmental conditions. To achieve this, the work performed by any functional unit (FU) or operational stage in a synchronous design should be completed within its clock period, every clock cycle. Thus, conservative timing guard bands are employed to handle all legitimate workload characteristics that might activate critical paths in any FU/op-stage and wide environmental (PVT) variations that can worsen these paths. Improvements in performance and/or energy efficiency are thus sacrificed for reliability.

In the common non-critical cases, this creates clock cycle *Slack* - the fraction of the clock cycle performing no useful work. Slack can broadly be thought to have two components: ① *PVT Slack*, caused due to non-critical environmental (Process, Voltage, Temperature) conditions, and ② *Data Slack*, caused due to non-triggering of the executional critical path. *PVT Slack*, with its relatively low temporal and spatial variability, can more easily be tackled with traditional solutions [10, 14, 32]. On the other hand, *Data Slack* is strongly data dependent and varies widely and manifests intermittently across different instructions (opcodes), different inputs (operands) and different requirements (precision/datatype).

The focus of this work is on *Data Slack*, and as analysis in Sec.2 shows, its multiple components often cumulatively produce more than half a cycle's worth of slack. The available *data slack* has been increasing, since instruction set architects are under pressure to increase execution bandwidth per fetched instruction, leading to data paths with increasingly rich semantics and large variance from best-case to worst-case logic delay. This trend is exacerbated by workload pressures, specifically the emergence of machine learning kernels that require only limited-precision fixed-point arithmetic [33]. Furthermore, in spite of rich ISA semantics, or perhaps because of them, compilers struggle to generate code that utilizes these complex features effectively (see, e.g. [8]). The end-product of our proposal is to recycle this data slack, to be utilized across multiple operations, and improve system performance. There are 3 domains of microarchitecture that have marginally explored this goal in different forms, which are discussed below.

The first is *timing speculation*. Major proposals focus on raising the frequency or decreasing the voltage to reduce wasted slack, as long as the occurrence of timing violations can be detected and controlled or avoided. Prior works have focused on adaptive variation of the operating points (F,V) by tracking the frequency of timing errors occurrences [10] or by predicting critical instructions [34] and so on. TS solutions suffer from the fundamental constraints that they are bounded by the possibility of timing errors from *every* computation, in *every* synchronous FU/op-stage, and on *every* clock cycle. Since data slack has wide variations across operations, (F,V) reconfiguration in these TS proposals, which can only be altered at reasonably coarse granularity of time (at best, over epochs of 100s of cycles), have to be set very conservatively. Moreover, the design overheads in implementing timing error detection and timing overheads from recovery are significant [22].

The second domain is *specialized data-paths*. When specialized data-paths are built to accelerate certain hot-functions, specific circuit logic elements are combined together in sequence and the timing for the data-path can be optimized for the particular chain of operations [30, 35]. But such data-paths

do not provide flexibility for general-purpose programming and also suffer from low throughput or very large replication overheads. Thus, they cannot be integrated into standard out-of-order(OOO) cores.

The third domain is static and dynamic forms of *Operation Fusion*. These proposals involve identification of sequential operations that can be fit into a single cycle of execution [26] and further, rearranging instruction flow to improve the availability of suitable operation sequences to fuse [4]. Optimizing the instruction flow is a significant design/programming burden while unoptimized code provides very limited opportunity for single-cycle fused execution in the context of our work.

Our proposal **ReDSOC**, on the other hand, avoid all of these issues. ReDSOC aggressively recycles data slack to the maximum extent possible. It identifies the data slack for each operation. It then attempts to cut out (or recycle) the data slack from a producer operation by starting the execution of dependent consumer operations at the exact instant of completion of the producer operation. Further, ReDSOC optimizes the scheduling logic of OOO cores to support this aggressive operation execution mechanism. Recycling data slack in this manner over multiple operations, executing on functional units such as ALUs, allows acceleration of these data sequences. This results in application speedup when such sequences lie on the critical path of execution.

ReDSOC is timing non-speculative, and thus does not need costly error-detection mechanisms. Moreover, it accelerates data operations without altering frequency/voltage, making it suitable for fine-grained data slack. It is implemented in general OOO cores, atop the data bypass network between ALUs via transparent latching and is suitable for all general-purpose execution. Finally, it cumulatively conserves data slack across any naive sequence of execution operations and neither requires adjacent operations to fit into single cycles nor any rearrangement of operations.

Key elements of our proposal are summarized here:

- Classification of execution operations into different slack buckets based on the opcode and input precision (Sec.2).
- Transparent latches with slack-aware control between execution units, allows slack recycling across multiple operations (Sec.3).
- Slack-Aware Select-free instruction scheduling, which optimizes OOO cores for slack recycling (Sec.5).

2. Analyzing Data Slack

More often than not, a circuit finishes a computation before the worst-case delay elapses since the critical paths of the circuit could be inactive. Prior studies [34] show that roughly 99% of all timing errors come from less than 10% of all executions, when the voltage is lowered. Further, the average execution latency over all executions can be as low as less than 60% of the clock period.

2.1. Sources of Data Slack

The three major sources of data slack are the type of operation executing on a common functional unit, the data-width of the operands and the required data types (via software-specified precision) for the computation.

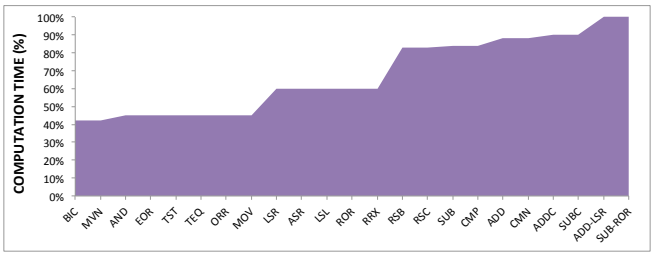

Figure 1: Computation Time for ALU Operations

Operation Type (Opcode-Slack): In general purpose processors, it is common to have functional units (eg.ALU) which perform multiple operations. In a standard non-timing-speculative design, the functional unit would be timed by the most-critical computation in order to be free of timing violation. Thus, Many of the executing operations which do not trigger the critical path of the functional unit, end up producing data slack.

Further, the semantic richness of current-day ISAs means that multiple modes of operations are supported via the same data paths. For example, the ARM ISA based designs support a *flexible second operand* input to the ALU to perform complex operations such as a shift-and-add instruction. Supporting such complex operations via the enhancements to the standard datapath further elongates the critical path of execution. These rich/complex semantics are often less-exploited by standard compilers, resulting in even higher data slack.

Fig.1 shows the critical computation times for different operations on a single-cycle ARM-style ALU, written for RTL and synthesised using Synopsys Design Compiler. It is evident that a large number of ALU operations (eg. logical) have significantly lower computation times than more critical arithmetic operations. And even these arithmetic operations produce some data slack in the absence of modifications to the second operand. It is, therefore, intuitive that ALUs would produce considerable data slack across common applications and that this data slack can be intermittently distributed depending on the application characteristics. This form of slack is easily identifiable for the operations, simply by means of the instruction opcode.

Data Width of Operands (Width-Slack): High-end processor word widths are usually 32-64 bits while a large fraction of the operations are narrow-width (large number of leading zeros). The execution of such operation on a wide(r) compute unit means that there is low spatial and temporal utilization of the compute unit. Low spatial utilization refers to the higher-bit wires and logic-gates which are not performing useful work, while low temporal utilization refers to data slack from non-triggering of the entire critical propagation paths.

Computations with a significant number of higher-order zero bits are especially common in machine learning applica-

tions - a large number of features often have very low weights, a characteristic exploited in multiple prior work, in terms of improving spatial utilization. Low spatial utilization (resulting in unnecessary leakage power) has also been attacked in traditional architectures in prior work by aggressive power gating of functional units and operand packing [5], among others.

But the problem of low temporal utilization for narrow-width computations has not been explored. Fig.2 shows the varying length of the critical path on a 16-bit Kogge Stone adder for different bit-widths. When only a smaller portion of the data-width is in use, the length of the critical carry-bit propagation path (and thus, the critical delay) reduces, roughly proportional to $log(datawidth)$. This form of slack can be estimated via data-width identification. Data-width identification at the time of execution can be performed via simple logical operations at the input ports to the functional units [5]. Prediction methods for identification of data-width early in the execution pipeline, have also been very successful [9, 23].

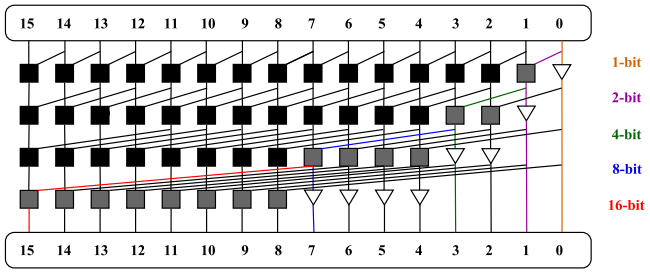

Figure 2: Critical paths for KS-Adder

Data Type of Operands (Type-Slack): Sub-word parallelism, in which multiple 8/16/32/64-bit operations (i.e. sub-word operations of *smaller* precision/data types) are performed in parallel by a 128-bit SIMD unit (for example), is supported in current processors via instruction set and organizational extensions (eg. ARM NEON, Intel MMX). This is yet another form of improving spatial utilization (described earlier) and another case of opportunity to improve temporal utilization. The varying compute latency is similar to Fig.2, but the method of identification is from the ISA itself, rather than from observing the bits of the inputs. Low-precision computation has especially gained popularity on the Machine Learning forefront over the past few of years [19], often enabling the use of narrow data types, specified directly from the software level.

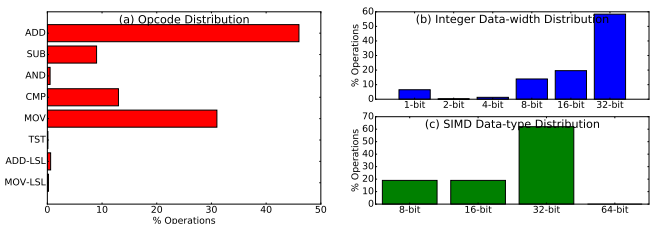

Figure 3: GEMM operation distribution

2.2. Data Slack Distribution

To understand the distribution of operations causing the 3 kinds of data-slack discussed above, we analyze a small self-contained low-precision GEMM library - gemmlowp [16] widely usable in Machine Learning applications. Analysis of other benchmark suites is performed in Sec.6. Fig.3.a shows the operation-type/opcode distribution, clearly indicating that opportunity to recycle data slack out from some of these operations like *cmp* and *mov*. Fig.3.b shows the distribution of actual data-width for 32-bit integer operations and Fig.3.c. shows the sub-word data-type distribution for NEON SIMD operations. Both of these indicate opportunities for significant slack recycling.

Thus, we can see that there can be a significant amount of data slack among operations and current-day applications are often made up of large distributions of such operations with low data slack. An effective mechanism to remove data slack from these operations and recycle this slack to speed up sequences of operations, can therefore have substantial opportunity to accelerate these applications. Further, conventional epoch-based voltage and frequency scaling is not effective for capturing this type of slack, since it isn't pervasive, but only manifests intermittently in ALU operations. Hence, we need a scheme to track slack on an instruction-by-instruction basis, and a very fine-grained mechanism (early clocking) to benefit from it.

3. Recycling Slack via Transparent Latching

The previous section observed the presence of considerable data slack in executing operations. In order to execute consumer operations immediately after the producer completes (i.e. recycling the data slack), we make use of transparent pipelines via intelligent latching.

3.1. Transparent Data-flow

A transparent latch is a storage element with an input, an output, and an enable. When the enable is active, the output transparently follows the input. When the enable becomes inactive, the latch becomes opaque and the output freezes. In our work, latches between FUs are made transparent at appropriate times to allow data to flow through at non-clock boundaries. This allows varied delays across operations to be balanced anywhere within the transparent execution window. The primary benefits explored earlier from transparent pipelines include reducing clocking power [13, 17] as well as interlocked synchronous pipelines which reduce stall related overheads [18].

We propose a synchronous slack tracking and opportunistic early clocking mechanism implemented atop a transparent execution pipeline. Our proposed mechanism *reduces the execution latency in the absence of peak throughput*. We introduce the concept with a simple discussion on applying transparent latching to a generic pair of functional units (FUs) as shown in Fig.4.b. We assume that the FUs have forwarding paths to each other (shown in figure) and back to themselves

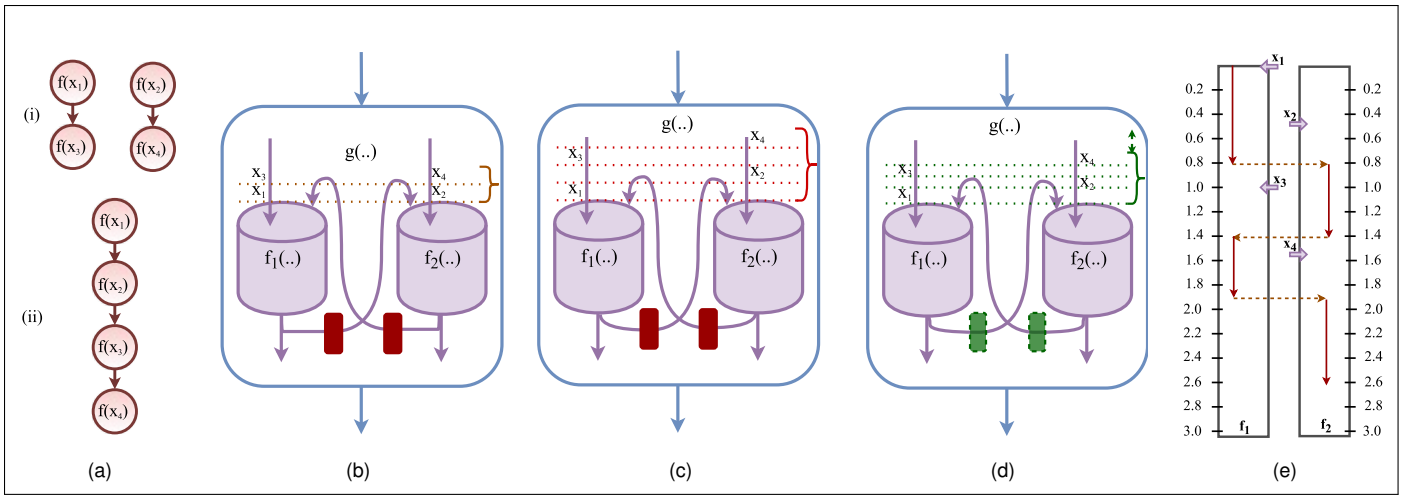

Figure 4: Recycling Data Slack with Transparent Latching

(not shown). In the context of this discussion, we also assume single-cycle combinational execution. These FUs could be thought of as the ALUs in standard OOO processors.

Discussion: Consider the data flow graph (DFG) shown in Fig.4.a.i. It shows two parallel execution paths and therefore, the FUs can execute in parallel on the independent operation sequences as shown in Fig.4.b. The peak throughput of 2 (parallel) operations per cycle is maintained throughout and the entire DFG completes in 2 cycles.

Next, consider the data flow graph Fig.4.a.ii. In this scenario all 4 operations are dependent and need to be executed in sequence. The functional flow is depicted in Fig.4.c where the stream of inputs x_i are distributed in sequence over the two $f()$. This system is entirely executing at a throughput of 1 operation per cycle i.e. not executing at peak throughput, and consumers 4 clock cycles to complete. Note that the operations could have any other distribution across the 2 $f()$ units - but the throughput is limited to 1 operation per cycle. In other words, in each cycle one $f()$ is always idle.

At this point, note that the actual compute time of each $f(x_i)$ varies for different x_i and varying PVT conditions. But the clock period would be set for the most conservative execution. This creates slack on each execution - sacrificing performance to maintain reliability. In standard synchronous design, $f()$ is lodged between opaque flip-flops and inputs and outputs pass through only at clock edges, causing this slack to go wasted. But our proposed mechanism cuts out this slack by introducing transparent latches between the $f()$ and using intelligent slack-aware control mechanism to make the latches transparent or opaque at the appropriate times. This allows transparent data-flow into idle $f()$s over appropriate time windows. Fig.4.d depicts how the use of transparent latches brings the 4 execution operations *closer* together, reducing execution latency (compared to 4.c).

Example: Fig.4.e describes the functional flow over the 2 FUs, in more detail, via an example. Consider that the four operations (x_i) described earlier, can execute on $f()$ with latencies of 0.8ns, 0.6ns, 0.5ns and 0.7ns respectively. The red solid arrow indicates estimated execution time and yellow arrows show dependencies. Assumptions for this example (but not for the actual design) are listed below, along with sections that discuss these in more detail: Execution latencies are assumed to be obtained from operation details - opcode, data-width and precision (Sec.2, Sec.5.1). Latches can be made transparent or opaque for half clock cycles (latch control - Sec.5.3). New inputs can be brought to the input of a $f()$ at every half clock cycle (OOO issue - Sec.5).

(1) At t=0ns, x_1 is brought to the input of f_1. This begins computation and would complete at t=0.8ns. (2) At t=0.5ns, x_2 is brought to input of f_2. $f(x_2)$ isn't ready for computation yet, since x_1 is yet to complete on f_1 but is brought in early so that $f(x_1)$'s slack can be completely utilized. (3) Also at t=0.5ns, f_1's latch is made transparent for half clock cycle as it is estimated that $f(x_1)$ completes in this half. The value passes through and stabilizes to the correct $f(x_1)$ value at t=0.8ns. Further, $f(x_2)$ starts correct computation at t=0.8ns and finishes at t=1.4ns. (4) x_3 is brought early to f_1 at t=1ns and f_2's latch is made transparent for half a cycle beginning at t=1ns to allow $f(x_2)$ to propagate through. $f(x_3)$ computes correct data from t=1.4ns to t=1.9ns. (5) Similarly, x_4 is brought in at f_2 at t=1.5ns and computes from t=1.9ns to t=2.6ns, meaning that a subsequent synchronous boundary (eg. Store instruction) can clock at t=3.0ns. Some slack is lost but the computation is still 1 cycle faster than the pure synchronous baseline (Fig.4.c) which took 4 cycles.

Summary: It is important to understand that this mechanism *does not require per-operation slack to be so significant that multiple operations can execute within a single cycle*. It only requires *one or more cycles worth of slack to accumulate over an entire sequence of operations*. This translates to higher performance and better energy efficiency via those accelerated sequences that lie on the critical path of program execution.

4. Slack-Aware OOO Scheduling

The previous two sections have described how data-dependent slack is estimated and how dependent operations can flow between functional units by appropriately turning latches transparent or opaque if a slack aware scheduling mechanism is in place. The baseline processor is a conventional super=scalar out-of-order processor - we assume a simple 7-stage pipeline that is shown in Fig.5. Instructions are fetched and decoded in the first two stages and the rename stage translates architectural register identifiers into physical register identifiers. Next, they are written into the reservation station entries (RSE) where they wait for their source operands and a functional unit to become available. The scheduler is responsible for issuing instructions, based on some priority scheme, to the execution units when all required resources (source operands and execution units) are available. The remainder of the pipeline consists of the register file read, execute/bypass, and retirement stages. We focus on the scheduling portion of the pipeline in this work, and in specific, the wakeup and select logic.

Figure 5: Processor Pipeline

The **wakeup logic** is responsible for waking up the instructions that are waiting for their source operands and execution resources to become available. This is accomplished by monitoring each instruction's parents as well as the available resources. Monitoring resources for wake-up is especially useful for long-latency functional units, so that instructions are not awoken too early when parents are (estimated to be) ready but resources are not (estimated to be) available for its scheduled execution - which could otherwise unnecessarily burdening the selection logic.

The **selection logic** chooses instructions for execution from the pool of ready instructions waiting in its reservation station entries (RSEs). Priority-based scheduling (eg. oldest-first) is required when the number of ready instructions are greater than the number of available functional units. This could happen when tags from parents of multiple instructions become available; and when the particular functional unit type is available, these instructions are all awoken and sent to the select-logic. It is evident from this discussion that the selection-logic is important only when there are more instructions than available functional units - which is critical in our discussion of select-free scheduling.

Implementing slack-aware scheduling in OOO processors has 2 challenges. *First*, the operation timing schedule is decoupled from actual execution. With assumptions on the latency of the executing instruction, appropriate dependents are scheduled to wake up and pickup their operands off the bypass at the correct time. Accounting for data slack means that the scheduling logic has to be made aware of the early completion of operations within their clock-cycle (eg. logical instructions complete in roughly half the clock cycle). This requires augmenting the scheduler with the data-slack information (as explained in Sec.5.2).

Second, the scheduling logic needs to run fast enough to be able to issue adjacent dependent instructions without wasting data slack. For example, if two logical instructions (each with half-cycle slack) are to be executed in a single cycle, then the scheduler needs to be providing operations for execution at twice the baseline execution frequency, so that the two operations can be issued back to back (and thus, executed) in a single cycle. Note, we restrict ourselves to double-speed scheduling. More on this in Sec.5.1 and 5.2.

The second challenge is especially important - instruction select-broadcast-wakeup is already a critical timing loop [25] and cannot naively be made faster (though double pumped scheduling and execution had been implemented in the Intel Pentium 4 [29], for instance).. The loop can't be naively pipelined either as this disallows dependent instructions from executing in consecutive cycles [6, 31].

Brown et al. [6] proposed "Select Free Instruction Scheduling Logic" to be able to break down the critical scheduling timing loop into multiple cycles - one, critical, single-cycle loop for wakeup; and one, non-critical, potentially multi-cycle loop for select. They do so without sacrificing back-to-back dependent executions. This is accomplished by speculating that all waking instructions are immediately selected for execution. The rationale is that usually the number of instructions woken up every cycle is never greater than the available slots for execution, as shown in their work. Select-free scheduling logic exploits this fact by removing the select logic from the critical scheduling loop and scheduling instructions speculatively. The select logic is only used to confirm that the schedule is correct. Select is known to account for more than half the scheduling latency [25], and so this technique provides opportunity for high-speed scheduling. The goal behind this work was that as pipeline stages and clock frequency grow, it is imperative to break down such critical loop into multiple stages. While frequency and number of stages has somewhat saturated in the last decade, the need for high speed scheduling to recycle data-slack provides perfect opportunity to take advantage of this proposed mechanism.

We implement slack-aware mechanisms atop this select-free optimized scheduler. In this section we explain conventional scheduling, select-free scheduling and double-scheduling for slack recycling via the timing diagram in Fig.6.

4.1. Conventional Scheduling

Fig.6.a is an example of the conventional scheduling operation. It shows the pipeline diagram for the execution of the two dependent instructions (I_2 is dependent solely on I_1) assuming each instruction has a 1-cycle latency. In Cycle 1, I_1 is woken-up for scheduling execution, due to its parent operation's and resource tags being available i.e. parent data and resource are expected to be available for I_1's execution in Cycle 4. If

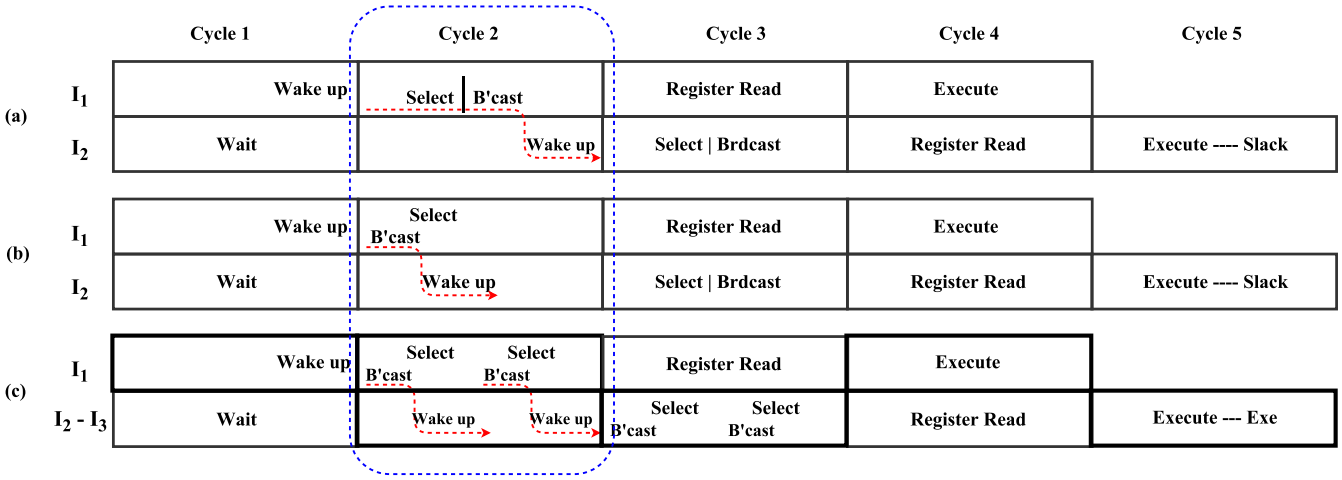

Figure 6: Timing Diagram of Execution Pipeline

I_1 is selected for execution in Cycle 2, then the scheduling logic wakes I_2, also in Cycle 2 so that it can execute in Cycle 5, if it is selected. Observe the work performed in Cycle 2. It involves: the **selection** logic to select I_1, the **broadcast** of tags to dependent operations (i.e. I_2) and the **wake-up** of I_2. As explained earlier, prior work has shown that this can be a critical path for single-cycle execution.

4.2. Select-Free Scheduling

Select-Free Scheduling [6] is explained via Fig.6.b. In this example, assume functional unit is available for execution. Also assume that I_1, I_2 are the only instructions in execution. This means that the selection logic to select I_1 in Cycle 2 is inconsequential. Thus, a speculative broadcast can be performed right at the beginning of the Cycle 2 and this eases the timing of Cycle 2 - i.e. the critical path is only broadcast (of I_1's tags) and wake-up (of I_2). Prior work has shown that selection logic (especially in wide cores) takes up half cycle's worth of timing, if not more [25]. Thus, in select-free scheduling, the broadcast-wakeup path can complete in roughly half a cycle, while the selection logic works in parallel. We discuss mispeculation in the design section (Sec.5.2). The end result of select-free scheduling is that there is opportunity here to do more useful work in Cycle 2, since broadcast-wakeup completes in half clock cycle. This is discussed below.

4.3. Double Scheduling for Slack Recycling

Double Scheduling is shown in Fig.6.c. From the previous discussion, observe that half a cycle is sufficient for the speculative broadcast-wakeup with parallel selection path. This means that a follow up broadcast-wakeup (again, with parallel selection) can be performed in the same cycle. Assume that there is a 3rd dependent instruction I_3 to be executed as well, and that I_2 has some data slack - i.e. I_3 can start executing at some instant in the second half of Cycle 5, if scheduling supports this. As shown in Fig.6.c, the double scheduling allows 2 dependent wakeups to occur back-to-back in Cycle 2. This allows I_3 to be ready for execution in the second half of Cycle 5, thus recycling I_2's data slack. Incorporating the slack information for the scheduling mechanism to be aware that I_3 *can* start execution in I_2's slack period is discussed in Sec.5.2.

In the context of the earlier example discussed in Fig.4.e, it is the double scheduling capability described here, that allows x_2 and x_4 to reach the functional unit at t=0.5ns and t=1.5ns respectively. Note that double scheduling (and double execution) allows a maximum of 2 dependent instructions to execute in a single cycle and can thus recycle a maximum of a half cycle of slack. We restrict our design to this maximum slack of a half cycle, which is reasonably in tune with the maximum available slack estimates shown in Fig.1. Scheduling can be made even more aggressive by additional techniques such as speculative wake-up [31] and circuit optimizations such as resizing buffers and clustering [24], which are not explored here.

5. Designing ReDSOC

This section discusses the implementation of ReDSOC atop OOO cores in 3 parts - data slack identification, slack-aware scheduling and transparent latching among execution units.

5.1. Design for Slack Estimation

Both *opcode slack* and *type slack* can be found out as early as the decode stage in the processor pipeline since the opcode and data type (for SIMD) are encoded with the instruction. *Width slack* (via data-width), on the other hand, is often not available until the execution stage itself. This is because register values or data bypass values are often not available until just prior to execution. For prior work on partial power gating of functional units or combining multiple operations into a single execution on the functional unit, it is sufficient to identify data-width at the time of execution. But in our work, the data-width/operand-slack information is required in the scheduling stage (Sec.5.2). We therefore assume the use of a data-width predictor as proposed by Loh [23] and also used by others for optimizations such as Register packing [9].

We utilize a resetting counter based predictor as proposed by Loh [23]. The predictor is addressed by the instruction PC

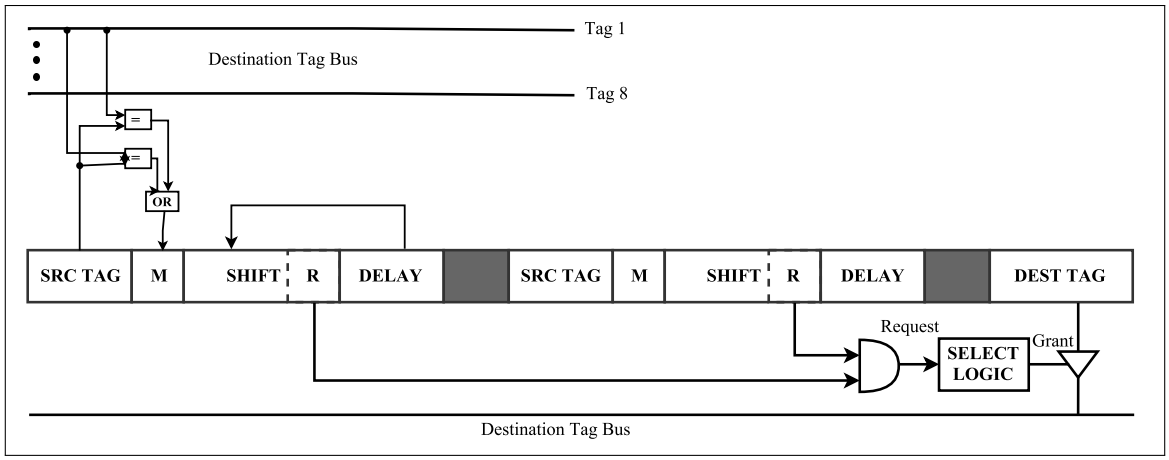

Figure 7: Baseline RSE with select/wakeup logic

and two pieces of information for each instruction - the most recent data-width of the instruction and a k-bit confidence counter that indicates how many consecutive times the stored data-width has been repeated. On a lookup, if the confidence counter is less than the maximum value of $2k-1$, then the predictor makes a conservative prediction that the instruction is of maximum size. Otherwise, the prediction is made according to the stored value of the most recent data-width. If there was a data-width misprediction, the data-width field is updated and the counter is reset to zero. On a match, the counter is incremented, saturating at the maximum value of $2k-1$. Prior analyses [9, 23] of such a predictor have shown prediction accuracies of 96% and better.

Estimated completion time (i.e. Clock Period - Slack) is obtained from a lookup table addressed by the instruction bits corresponding to opcode and data width (for standard instructions) and data type (for SIMD, eg. NEON). We utilize data slack precise to 1% of the clock period and therefore use a 7-bit completion time (T) value which is passed on to the instruction scheduler (Sec. 5.3, Sec. 5.2). In this work, we focus on single-cycle execution, but this can be extended to multi-cycle operations as well.

5.2. Design for Slack-Aware Scheduling

Baseline: We assume a reservation station based model for scheduling, as described below. After instructions are renamed, they wait in reservation stations for their sources to become ready. The reservation station entry (RSE) is shown in Fig.7. The two "SRC TAG"s are identifiers for the source operands. The number of cycles between the time the tags are broadcast and the time the results are available is encoded via the DELAY fields. For a source whose producer has an N-cycle execution latency, the DELAY field contains N-1 zeros in the least significant bits of the field. The remaining bits are all set to 1. The M (MATCH) bit is set when a tag match occurs, and DELAY is loaded into SHIFT field. By means of a right shift (on the SHIFT value) enabled by the MATCH bit, the R (READY) bit gets set N-1 cycles after the match. Since the tag match occurs one cycle after the parent scheduling and R is set N-1 cycles after match, this means that R is set N cycles after the parent's instruction's scheduling i.e. which matches the N-cycle execution latency.

As an example, the DELAY field for an instruction dependent on a 3-cycle instruction would contain the value '1100'. When the tag match for the parent instruction occurs, this value will be loaded into the SHIFT field, and the MATCH bit will be set. After two more cycles, the SHIFT field will contain the value '1111', and, this source will be ready. For a source operand with a 1-cycle latency, the DELAY field will simply be '1111'.

Once all the R bits are set, the instruction is ready for selection and is selected if it gets the grant from the selection logic. If selected, its destination tag is broadcast on the tag bus. Fig.7 also shows the wake-up and select mechanism described above. More details of this model can be found in prior works [31].

Slack-Aware Scheduling: Our design goal is augmenting this baseline design with slack-awareness. The resultant optimized RSE entry is shown in Fig.8. For illustration simplicity, Select/Wakeup logic from Fig.7 are not shown here. The L-bit indicates the last arriving input among the 2 sources. The L-bit is set in the cycle when the tag match occurs and is cleared in the subsequent cycle. The EX-TIME field indicates the estimated execution time for this particular instruction which is a 8-bit value that flows through from decode (described in Sec.5.1). When the entry is ready for (speculative) execution i.e. when the R-bits are set, the completion time of the current entry is to be calculated. This is done in the following 3 steps:

(1) First, the latest completion instant among the sources arriving in the current cycle are compared and the maximum value is obtained (Max. logic). Older sources are ignored via their L-bit. (2) This value is added to the EX-TIME field - thus providing the estimated completion instant for this execution/entry IF it is executed immediately and slack is conserved. This value is written into the 8-bit COMP-INST field and available to be used by future instructions for their own time estimations. (3) If the instruction does not execute (i.e. if a

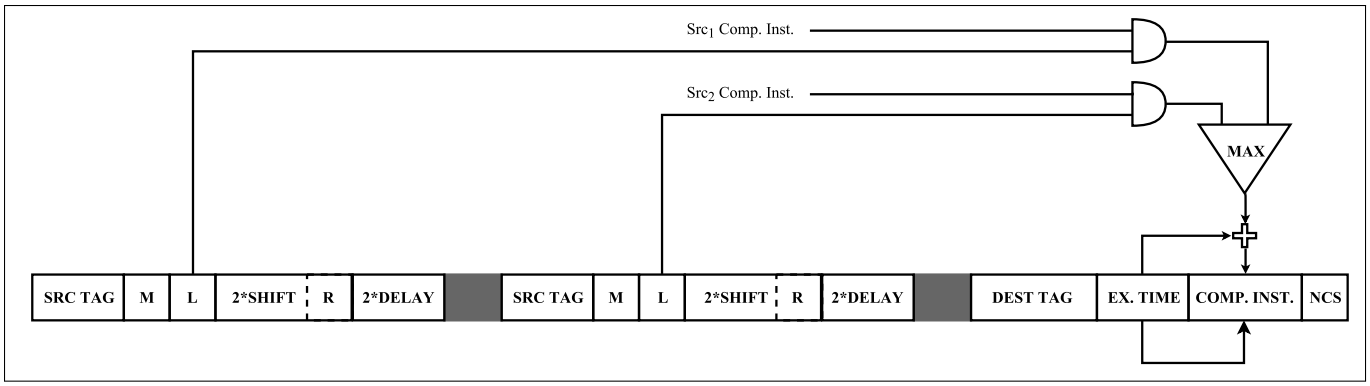

Figure 8: RSE modified for slack-aware scheduling

collision is found to occur), then the collision and pileup victims are first cleared, and then the collision victim is executed from a proper base clock cycle boundary (i.e. the slack is lost). ④ In this case, the EX-TIME field is directly written into COMP-INST field, since the instruction started at a base cycle boundary and future instructions will again continue the slack recycling process. ⑤ Further if the COMP-INST field and the output of the Max. logic (MAX-OUTPUT) are on adjacent half cycles, then the NEXT-CYCLE-SCHEDULE (NCS) bit is set. The significance of this bit is discussed further in this section.

First note that NCS bit is not always set. For example, say I_1 and I_2 and I_3 are a sequence of dependent instructions. If I_1 finishes at 0.7ns and I_2 has a latency of 0.7ns, then I_2 would finish at 1.4ns and thus I_3 could be scheduled in the very next scheduling cycle - thus the NCS bit is set. But if I_1 finished at 0.9ns instead, I_2 would complete only at 1.6ns. In this scenario I_3 or any other dependent instruction on I_2 cannot be scheduled in the immediate next scheduling cycle after I_2 and so the NCS bit is not set. The use of the NCS bit will become clear later in this section.

The scheduling process is itself controlled by the SHIFT field, which when shifted sufficiently writes into the R-bit (as explained earlier). The DELAY field (which writes into SHIFT) and the SHIFT field are doubled in number of bits, since the scheduling mechanism is clocked at double speed, which was motivated and described in Sec.4. The encoding of the DELAY field is changes to reflect the double speed scheduling. For a source whose producer has an N-cycle execution latency, the DELAY field contains 2N-1 zeros in the least significant bits of the field. An instruction dependent on a 3-cycle latency instruction would now have its DELAY field value as '11100000'. An instruction dependent on a 1-cycle latency instruction would DELAY field as '11111110'.

In order to schedule instructions on consecutive scheduling cycles (i.e. half a base cycle apart) we use the NCS bit. If the NCS bit of the parent operation is set, the DELAY field value is set to '11111111'. This would mean that the R (READY) bit is set immediately on tag match, on the first half-cycle after the parent is scheduled and it can be speculatively selected immediately. This is similar to a 1-cycle latency instruction in the baseline design.

Note that NCS-bit based implementation suffices for half-cycle scheduling that is required for recycling slack produced only by single cycle instructions. It can be extended for recycling slack for multi-cycle instructions (via intelligent conjunction with the DELAY field), but this is not discussed here.

Mispeculation: There are two forms of mispeculation possible - one from speculative wakeup and the other from incorrect slack prediction. We refer to the former as collision. It is the scenario where more instructions wake up than can be selected. Collision victims (instructions not selected by the select logic) are identified at the same time an instruction is selected. Dependents of the collision victims may wake up before they are really ready to be scheduled, thus entering the scheduling pipeline too early. More information on collision and dependent victims can be found in the original Select-Free Scheduling work.

Mispeculation from incorrect slack prediction due to incorrect prediction from the data-width predictor is found out at the time of execution. Both forms of suffer penalty that is similar to that in normal scheduling mispeculation - when, for instance, the instructions after a load are scheduled assuming a L1 cache hit for the load, but the load actually misses in the L1.

5.3. Execute-Stage Redesign

In order to perform the transparent data-flow mechanism described above, the execution units need to be equipped with latches in their data bypass paths. When data flows from a producer execution unit to a consumer functional unit, the output latch for the producer is transparent, while the output latch for the consumer is opaque. The ability to bring inputs early to the execution unit at double the base-clock rate, in order to facilitate slack recycling (akin to Fig.4.e) was discussed earlier. This is performed via scheduling optimization - Sec.4, Sec.5.2.

This also requires support from latch transparency control in the execution stage of the pipeline. The latches controlling data bypass between the functional units, need to be made opaque or transparent over certain clock periods with an intel-

Parameter	Small	Medium	Big
Frequency	2 Ghz		
Width	3	4	8
ROB/LSQ	40/16	80/32	160/64
RSE/ALUs	32/3	64/4	128/6
L1/L2 Cache	64kB/2MB w/ prefetch		

Table 1: Processor Baselines

Kernel	Description
GEMM	Low-precision matrix multiply (GoogLeNet)
CONV	Convolution Layer: Gaussian 3x3, 5x5 filters
ACT	Activation Layer: Rectified Linear Unit (ReLU)
POOL	Pooling Layer: 2x2 pool, Function: Average
FC	Fully Connected Layer

Table 2: Kernels for Machine Learning

ligent control mechanism. The latch control mechanism also works at double the base-clock rate, in conjunction with the double-scheduling technique. The latch is kept transparent in the *half-cycle* that the operation is to complete and is made opaque in the *half-cycle* that new inputs are brought in. This is achieved by again utilizing the 8-bit COMP-INST (completion instant) that was created in the scheduling stage, and is passed on to the execute stage.

Let us call the cycles of the fast (2x) scheduling clock alternating *high cycles* and *low cycles*. If an operation has an estimated time of completion within the first half of a base cycle, then COMP-INST < 64. In this scenario the latch is kept transparent in the *high cycle*. Since new inputs can be brought in by the scheduler at the half-tick, the latch is made opaque for the *low cycle* Conversely, if COMP-INST > 64, the operation is expected to complete in the latter half of the base cycle and thus, the latch is transparent for the *low cycle*.

6. Methodology

We extended the Gem5 [3] simulator to support Slack Recycling atop standard out-of-order cores. We model 3 cores labelled *Big*, *Medium* and *Small* and results atop these 3 cores are described in Sec.7. The description of the cores can be found in Table.1.

The benchmarks for analyzing results are 2-fold. The first set encompass subsets of the SPEC CPU 2006 benchmark suite and the MiBench benchmark suites. SPEC benchmarks are run via multiple Simpoints [28], each of length 100 million instructions, while MiBench benchmarks are run in their entirety. The second set consists of kernels written for a low precision GEMM library [16] (described in Sec.2) and the ARM Compute Library [15]. Both of these are suitable software libraries for computer vision and machine learning and are chosen because of the recent popularity for machine learning in the computer architecture domain, as well as their support for ARM NEON SIMD. Benchmarks are all compiled for the ARM ISA; NEON vectorization flags are turned on for the ML kernels. While descriptions of the SPEC/MiBench benchmarks can be found at their respective sources, brief details of the ML kernels are provided in Table.2.

The set of chosen benchmarks are listed along with their operation characteristics are shown in Fig.9. Means for the 3 sets of benchmarks are also shown - *SPEC-MEAN*, *MiB-MEAN*, *ML-MEAN*. The characteristics shown are memory operations with high or low latency (MEM-HL/MEM-LL; HL refers to L1 cache misses), NEON SIMD operations, other multi-cycle operations and high and low slack single-cycle ALU operations (ALU-HS/ALU-LS, HS refers to <20% data slack). MiBench benchmarks see high percentage of single-cycle data slack and low percentage of memory operations, allowing them to get significant benefits from ReDSOC. On the other hand, SPEC benchmarks are more memory-intensive (even if they are predominantly MEM-LL), resulting in fewer opportunities for slack recycling. ML kernels have sizable portions of low-precision SIMD operations, though some of the kernels (eg. FC, POOL) have high MEM-HL fractions, resulting in less significance of slack recycling, even if large opportunities exist.

In this work, we target single cycle data slack for the integer ALU as well as data slack in integer SIMD operations. Slack is modeled at nominal PVT conditions [14] via RTL design in verilog and synthesis using the Synopsys Design compiler, supported by gate-level C models for more extensive characterization, for integer ALU, Kogge-Stone Adder and Wallace-Tree multiplier.

7. Results

In this section we evaluate the impact of ReDSOC based slack recycling across the benchmark suites discussed above for different types of OOO cores.

7.1. Performance Speedup

Fig.10 shows the performance speedup obtained over a standard baseline without slack conservation for the BIG, MEDIUM and SMALL cores described previously. Speedups are generally good across most benchmarks and all cores - more specific details below.

First, speedups are relatively lower for SPEC benchmarks which, as was discussed previously, is a direct result of having a high percentage of memory operations. Moreover, the average high slack ALU operations in SPEC are only around 25% while it is close to 50% in MiBench. MiBench applications, on the other hand, show significant speedups (25% average on the BIG cores). The *bitcount* application sees as much as an 84% speedup over the baseline. This is not surprising, considering that the earlier benchmark characterization (Fig.9) shows that this application has less than 5% of memory operations and close to 60% of high slack single-cycle ALU operations.

Second, note that benefits generally increase with size of the core. A larger core provides more vacant functional units for data to transparently flow into (as described in Sec.3) which is a requirement for slack recycling. Further, the larger number of reservation stations in the big cores allow for more dependent waiting operations in the RS to be scheduled aggressively,

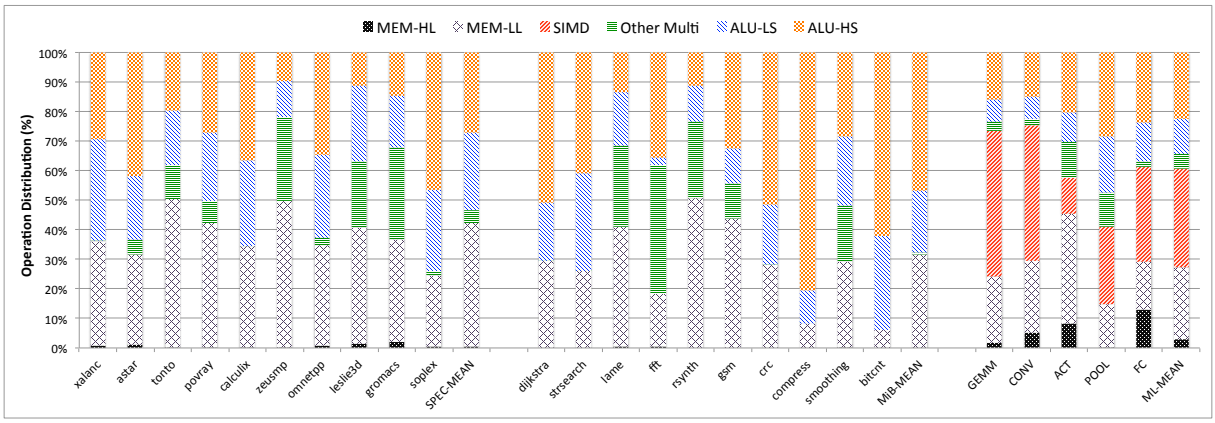

Figure 9: Benchmark Operation Characteristics

allowing multiple dependency chains within the application to perform slack conservation. Moreover, the larger cores also exhibit better memory parallelism, allowing a higher impact from compute optimizations. A good example of these characteristics is the *compress* benchmark from MiBench. Though it is similar to *bitcount* with very high percentage of high slack single-cycle ALU operations (80%), the speedups are almost zero in the MEDIUM and SMALL cores. This is because, being a high IPC application, a) there is not enough vacancy in the ALUs in for MEDIUM/SMALL cores to exploit data-slack, and b) the application becomes front-end limited for lower front-end widths.

Finally, speedup is similar on all core sizes for the ML kernels and is pretty significant for most of the kernels. This is because of a large fraction of low-precision NEON SIMD operations along with reasonable fractions of high slack single-cycle operations. Due to their working sets, some of these kernels spend a significant portion of time waiting for long-latency memory operations to complete, and this cuts down gains completely in FC and to some extent in other kernels. Tuning the prefetchers and blocking the matrices could increase opportunities for slack, so these results as shown might be pessimistic.

7.2. Comparison with other proposals

In the introductory section, we had discussed the benefits of our proposal compared to specialized data-paths [30], timing speculation [10] and operation fusion [26]. Specialized data-paths suffer from the lack of flexibility for general-purpose programming and therefore not suited to our goal of recycling slack in OOO cores. In this section, we quantitatively compare the efficiency of our mechanism in comparison to our own implementations of timing speculation and operation fusion. We describe our implementations below.

TS is our timing speculation mechanism wherein frequency is controlled depending on the error rate in the application. The temporal granularity for frequency control is 1ms. We utilize an oracular mechanism which maintains error rates between 1% and 0.01% across each epoch of frequency control. Due to the low error rates, we do not model recovery, thus, the performance numbers shown are better than realistic expectations.

MOS is Multiple Operations in Single-cycle - i.e. the implemented operation fusion mechanism. The mechanism dynamically combines multiple operations within a single cycle, if they are capable of fitting within a single cycle. For example, 2 consecutive logical operations (roughly 50-55% data slack) can executed in a single cycle.

Comparison of these two mechanisms against ReDSOC atop the three different core types are shown in Fig.11. It is clear that ReDSOC significantly outperforms both mechanisms. *MOS* opportunity is limited in most applications, due to inability to find many sequential operations to combine into a single cycle. *TS* performs better than *MOS* but is limited by the fact that frequency changes can happen only at a coarse temporal granularity, while data-dependent slack varies from operation to operation. Hence, the *TS* setting for each setting has to be set rather conservatively to maintain low error rates.

7.3. Impact of select-free scheduling

Section.4 discussed the need for a instruction scheduling mechanism that runs at double the speed of the rest of the core and how select-free scheduling [6] allows such an implementation. Select-free scheduling speculatively schedules all woken-up instructions, and in some scenarios this can lead to collisions - when there are not enough resources to execute all the woken instructions. Collision and pile-up results in instructions having to be rescheduled.

Fig.12 analyzes the fraction of collision victims across the benchmarks atop the three core types. It is evident that the fraction of collision is very low across the 3 suites, roughly around 1%, confirming that this form of speculation is not a significant overhead. The probability of collision can further be reduced with techniques such as wakeup prediction [6], but these are not explored here.

8. Related Work

The best known technique to adaptively control voltage guard bands in the presence of data dependent variation is Razor [10]. Razor performs DVS of the entire core, tuning voltage based solely on the frequency of timing errors without

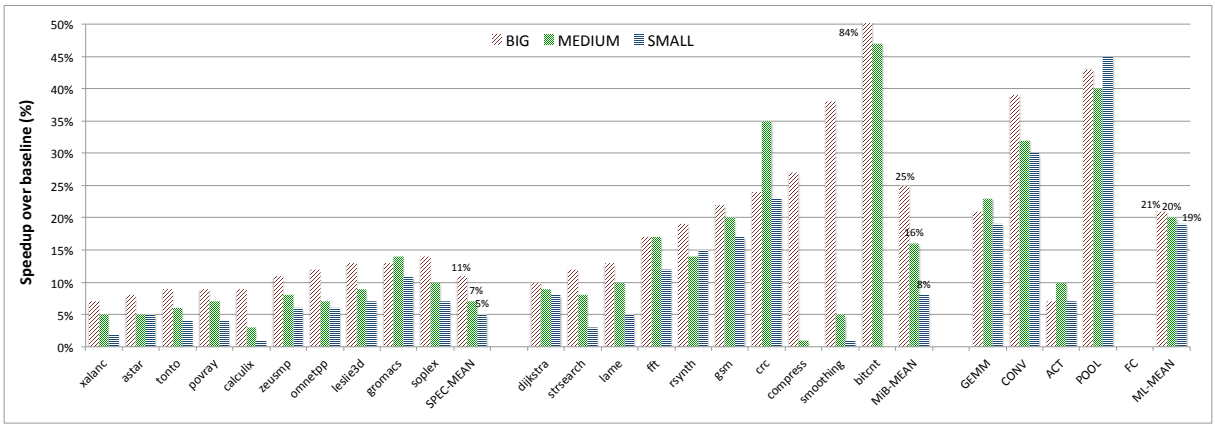

Figure 10: Speedup for different cores

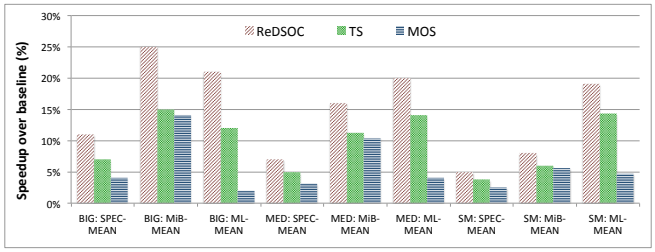

Figure 11: Comparison with other proposals

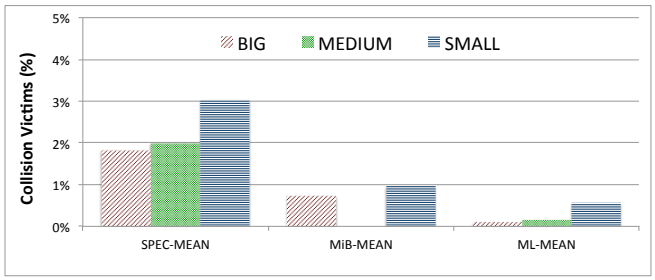

Figure 12: Collisions from select-free scheduling

analyzing the instructions themselves. Xin et al. [34] build a mechanism to predict likely error causing static instructions. Predicted instructions are allowed an extra cycle of execution under the reasonable assumption that an extra cycle is sufficient. Moreover, instructions that are expected to cause timing errors might exceed the base clock time by a small fraction of the period - but giving them an entire extra cycle might be inefficient. Other works have proposed similar "better than worst case" approaches to improve energy efficiency [1, 2, 11].

Optimizations via analyzing DFGs in the synchronous domain are performed in [4, 35]. Park et al. accelerated sequential code regions by executing multiple operations in a single cycle [26]. Such proposals depend on very high fractions of slack in consecutive operations. On the other hand, ours can accelerate even with low slack, **only requiring a cycle (or more) worth across an entire sequence**. DFG-style execution has also been explored in the context of specialized data paths (custom-cores) created for execution of specific basic blocks [30].

Multiple prior works have optimized for narrow data-width based execution in the context of improving utilization of functional units [5], register packing of multiple narrow operands into single registers [9], improving issue width [23] and energy reduction in multiple parts of the core [12]. Some accelerators have provided low-level precision control via bit-serial compute units [20], per-layer precision control [21] and so forth.

Finally, multiple works have studied the complexity of out of order cores [7, 24, 25] and many techniques have been proposed over the past two decades to optimize scheduling and break-down critical loops in the scheduling logic to be able to speed it up or make it more efficient in utilizing issue queues, multiple functional units and so on [7, 27, 31].

9. Conclusion

This paper showed that data slack, the unutilized portion of the clock period due to inactive critical paths in a circuit, can often be as high as half of the clock period, and cutting out this data slack provides tremendous opportunity to improver performance. With the increasing popularity of low-precision computing, data-slack is becoming even more significant in terms of non-utilization of a portion of the clock cycle, every clock cycle.

Our proposal, ReDSOC recycles the data slack from a producer operation, by starting the execution of dependent consumer operations at the exact instant of the producer's completion. Recycling data slack over multiple operations executing on functional units such as ALUs, allows acceleration of these data sequences and improves performance.

ReDSOC is particularly beneficial for compute-intensive benchmarks with long data-dependency chains. In the absence of very high ILP due to strict data-dependency, but at the same time when memory is not a bottleneck, ReDSOC provides an ideal mechanism to improve performance in an energy-efficient manner, without having to increase processor voltage/frequency. Moreover, its suitability to general purpose processors and its non-speculative nature for circuit timing makes it a very reasonable solution for better clock-period utilization in standard OOO cores.

References

[1] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[2] Todd Austin, Valeria Bertacco, David Blaauw, and Trevor Mudge. Opportunities and challenges for better than worst-case design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 2–7, New York, NY, USA, 2005. ACM.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[4] Anne Bracy, Prashant Prahlad, and Amir Roth. Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 18–29, Washington, DC, USA, 2004. IEEE Computer Society.

[5] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA '99, pages 13–, Washington, DC, USA, 1999. IEEE Computer Society.

[6] Mary D. Brown, Jared Stark, and Yale N. Patt. Select-free instruction scheduling logic. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 204–213, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Mary Douglass Brown. *Reducing Critical Path Execution Time by Breaking Critical Loops*. PhD thesis, Austin, TX, USA, 2005. AAI3187660.

[8] R. P. Colwell, C. Y. I. Hitchcock, E. D. Jensen, H. M. Brinkley Sprunt, and C. P. Kollar. Instruction sets and beyond: Computers, complexity, and controversy. *Computer*, 18(9):8–19, Sept 1985.

[9] Oguz Ergin, Deniz Balkan, Kanad Ghose, and Dmitry Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 304–315, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO 36*, pages 7–, 2003.

[11] Brian Greskamp and Josep Torrellas. Paceline: Improving single-thread performance in nanoscale cmps through core overclocking. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 213–224, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Erika Gunadi and Mikko H Lipasti. Narrow width dynamic scheduling. *Journal of Instruction-Level Parallelism*, 9:1–23, 2007.

[13] Erika Gunadi and Mikko H. Lipasti. Crib: Consolidated rename, issue, and bypass. In *ISCA '11*, pages 23–32, 2011.

[14] Meeta S. Gupta, Jude A. Rivers, Pradip Bose, Gu-Yeon Wei, and David Brooks. Tribeca: Design for pvt variations with local recovery and fine-grained adaptation. In *MICRO 42*, pages 435–446, 2009.

[15] ARM Inc. Arm compute library. *https://developer.arm.com/compute-library/*, 2017.

[16] Jacob.B. gemmlowp: a small selfcontained low-precision gemm library. *github.com/google/gemmlowp*, 2015.

[17] Hans M. Jacobson. Improved clock-gating through transparent pipelining. In *ISLPED '04*, pages 26–31, 2004.

[18] Hans M Jacobson, Prabhakar N Kudva, Pradip Bose, Peter W Cook, Stanley E Schuster, Eric G Mercer, and Chris J Myers. Synchronous interlocked pipelines. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 3–12. IEEE, 2002.

[19] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[20] P. Judd, J. Albericio, and A. Moshovos. Stripes: Bit-serial deep neural network computing. *IEEE Computer Architecture Letters*, 16(1):80–83, Jan 2017.

[21] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 23:1–23:12, New York, NY, USA, 2016. ACM.

[22] Charles R. Lefurgy, Alan J. Drake, Michael S. Floyd, Malcolm S. Allen-Ware, Bishop Brock, Jose A. Tierno, and John B. Carter. Active management of timing guardband to save energy in power7. In *MICRO-44*, pages 1–11, 2011.

[23] Gabriel H. Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 395–405, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[24] Pierre Michaud, Andrea Mondelli, and André Seznec. Revisiting clustered microarchitecture for future superscalar cores: A case for wide issue clusters. *ACM Trans. Archit. Code Optim.*, 12(3):28:1–28:22, August 2015.

[25] Subbarao Palacharla, Norman P Jouppi, and James E Smith. *Complexity-effective superscalar processors*, volume 25. ACM, 1997.

[26] Yongjun Park, Hyunchul Park, and Scott Mahlke. Cgra express: Accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 271–280, New York, NY, USA, 2009. ACM.

[27] Arthur Perais, André Seznec, Pierre Michaud, Andreas Sembrant, and Erik Hagersten. Cost-effective speculative scheduling in high performance processors. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 247–259, New York, NY, USA, 2015. ACM.

[28] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[29] Dave Sager, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[30] J. Sampson, G. Venkatesh, N. Goulding-Hotta, S. Garcia, S. Swanson, and M. B. Taylor. Efficient complex operators for irregular codes. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 491–502, Feb 2011.

[31] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 57–66, New York, NY, USA, 2000. ACM.

[32] Abhishek Tiwari, Smruti R. Sarangi, and Josep Torrellas. Recycle:: Pipeline adaptation to tolerate process variation. In *ISCA '07*, pages 323–334, 2007.

[33] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[34] Jing Xin and Russ Joseph. Identifying and predicting timing-critical instructions to boost timing speculation. In *MICRO-44*, pages 128–139, 2011.

[35] Sami Yehia and Olivier Temam. From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 238–, Washington, DC, USA, 2004. IEEE Computer Society.