Recycling Data Slack in Out-of-Order Cores

Gokul Subramanian Ravi, Mikko H. Lipasti
ECE Department, University of Wisconsin - Madison
gravi@wisc.edu, mikko@engr.wisc.edu

***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 has exacerbated this problem, requiring 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, to improve performance on Out-Of-Order (OOO) cores. It is implemented via a transparent-flow based data bypass network between the execution units of the core. Further, ReDSOC performs slack-aware OOO instruction scheduling aided by optimizations to the wakeup and select logic, to support this aggressive operation execution mechanism. ReDSOC is implemented atop OOO cores of different sizes and tested on a variety of general purpose and machine learning applications. The implementation achieves average speedups in the range of 5% to 25% across the different cores and application categories. Further, it is shown to be more efficient at improving performance in comparison to prior proposals.

***Keywords*-clock cycle slack; out-of-order; scheduler; transparent dataflow;**

I. 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 execution unit (EU) 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 EU/op-stage and wide environmental (PVT: Process Voltage Temperature) 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 under non-critical PVT 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 [1]–[4]. 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/data-type).

The focus of this work is on *Data Slack*, and as analysis in Sec.II 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. Furthermore, in spite of rich ISA semantics, or perhaps because of them, even optimum compilers are able to use these complex features only some of the time, but these *richer* data paths contribute to the critical timing all the time [5]. This trend is exacerbated by workload pressures, specifically the emergence of machine learning kernels that require only limited-precision fixed-point arithmetic [6].

The end-product of our proposal is to recycle this data slack, to be utilized across multiple operations, and improve system performance. There are three domains of prior work that have explored this goal in different forms, which are discussed below.

The first is *timing speculation* (TS). Prior 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. They can function by tracking the frequency of timing errors occurrences [2] or by predicting critical instructions [7]. TS solutions suffer from the fundamental constraints that they are bounded by the possibility of timing errors from *every* computation, in *every* synchronous EU/op-stage, and on *every* clock cycle. Since data slack has wide variations across operations and since (F,V) operating points can only be altered at reasonably coarse granularity of time, these proposals are forced to be configured conservatively. Moreover, the design overheads in implementing timing error detection and timing overheads from recovery are significant [8].

The second domain is *specialized data-paths*. When specialized data-paths are built to accelerate certain hot code, specific function elements are combined together in sequence and the timing for that data-path can be optimized for the particular chain of operations [9], [10]. 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 easily 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 [11] and further, rearranging instruction flow to improve the availability of suitable operation sequences to fuse [12]. Optimizing the instruction flow is a significant design/programming burden, while unoptimized code provides only limited opportunity for single-cycle fused execution in the context of our work.

Our proposal **ReDSOC**, on the other hand, avoids 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 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 flip-flops (FFs with bypass paths) 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.II).
- Transparent FFs with slack-aware control between execution units, allows slack recycling across multiple operations (Sec.III).
- Slack-Aware instruction scheduling via Eager Grandparent Wakeup and Skewed Selection, which optimizes OOO cores for efficient slack recycling (Sec.IV).

II. Analyzing Data Slack

More often than not, a circuit finishes a computation before the worst-case delay elapses, because the critical paths of the circuit are inactive. Xin et al. [7] analyze timing for ALPHA and OpenRISC ALUs, post synthesis and place-and-route. Their analysis shows that roughly 99% of timing critical paths are triggered by less than 10% of all computations. Similarly, Cherupalli et al. [13] perform data slack analysis for a fully synthesized, placed, and routed openMSP430 processor and show that more than 75% of the timing paths have greater than 30% clock cycle slack.

A. Sources of Data Slack

In order to recycle the considerable data slack it is important to categorize it based on its sources. These sources/categories are discussed below:

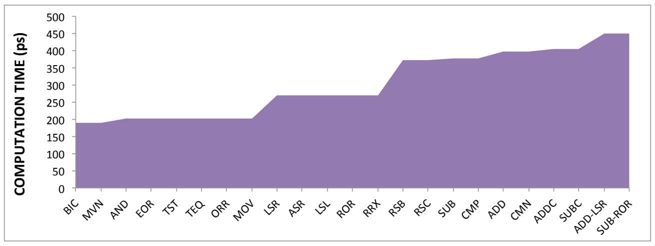


Figure 1: Computation Time for ALU Operations

Operation Type (Opcode-Slack):

In general purpose processors, it is common to have functional units that perform multiple operations, with different opcodes. In a conventional timing conservative design, the functional unit would be timed by the most-critical computation in order to be free of timing violations. Thus, many of the executing opcodes/operations do not trigger the critical path of the functional unit and 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 frequently unused, resulting in even higher data slack.

Fig.1 shows the critical computation times for different operations on a single-cycle ARM-style ALU, coded in RTL and synthesized (2 GHz target) for a TSMC 45nm standard-cell library 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 applications; many synapses have very small weights, a characteristic exploited in multiple prior works to improve spatial utilization [14]. Low spatial utilization (resulting in unnecessary leakage power) has also been attacked in traditional architectures by aggressive power gating of functional units and operand packing [15], 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. The colored paths show increasing critical delays/paths for computations of increasing widths. When only a smaller portion of the total data-width is in use, the length of the critical carry-bit propagation path (and thus, the critical delay) reduces, roughly proportional to $log(datawidth_{eff})$. 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 [15]. Prediction methods for identification of data-width early in the execution pipeline, have also been very successful [16], [17].

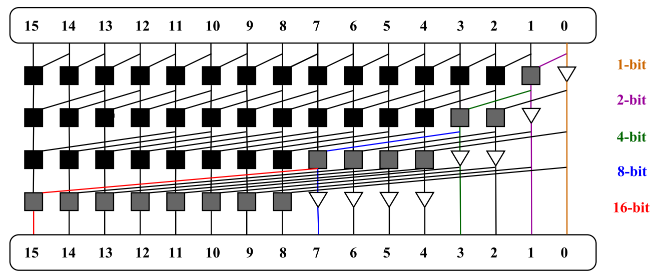


Figure 2: Critical paths for KS-Adder

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, 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 and another case of opportunity to improve temporal utilization. The varying compute latency for different data-widths 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 in machine learning applications over the past few years [18], often enabling the use of narrow data types, specified directly by software.

To summarize, current-day applications often exhibit a diverse distribution of operations with plentiful data slack. Analysis of specific slack breakdowns over multiple applications is discussed in Sec.V. An effective mechanism to recycle this data slack, in order 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 recycling mechanism, to benefit from it.

SIMD	Arith/Logic	Shift	Width/Type
1-bit	1-bit	1-bit	2-bit

Figure 3: 5-bit slack lookup

B. *Design for Slack Estimation*

Slack Look-Up Table:

Static circuit-level timing analysis at design time can measure computation times (i.e. Clock Period - Slack) for different classes of operations. These values are then stored in a slack look up table (LUT). We only break down the computation times into coarse blocks: a) based on operations being arithmetic vs logic, b) based on having a shift component and c) based on 4 different data-widths/types. The 5-bit address to perform a LUT lookup is shown in Fig.3. The Arith/Logic and Shift bits are *don't cares* for sub-word parallel SIMD instructions. The Width/Type bits use predicted data-width for normal instructions and data-type for SIMD. There are a total of 14 possible slack categories/buckets arising from the above. Operations are simplified classified into one of the slack buckets. Details on complexity of above analysis is discussed in Sec.V.

Data-Width Predictor:

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 (more on this in Sec.IV-C). We therefore use a data-width predictor as proposed by Loh [16] and also used by others for optimizations such as Register packing [17].

We utilize a resetting counter based predictor as proposed by Loh [16]. The predictor is addressed by the instruction PC 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 $2^k - 1$, 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 $2^k - 1$. We use 4 possible prediction outputs indicating high to low data-width.

Inaccuracy in prediction is detected at execute stage when the operands are available for execution by simply checking the higher order bits. Incorrect predictions are of two kinds - aggressive and conservative. Conservative incorrect predictions result in lost opportunity to recycle data-width slack but do not result in functional errors. Aggressive incorrect predictions would result in correctness violations if allowed and therefore such instructions need to be conservatively re-executed. Recovery is performed similar to cache miss replays via selective reissue of instructions.

Overheads/Accuracy:

Prior analyses [16] have shown that a resetting predictor allows aggressive errors in the range of only 0.1-0.6%. We use a 4K-entry prediction table which results in an aggressive misprediction of around 0.3-0.4%. Such a predictor requires a total state of 1.5KB. In comparison, current day branch predictors use prediction tables with as much as 64KB of state [19].

Considering the very small sizes of the LUT and predictor (in comparison to, say, register file and branch predictor) their overheads in terms of area and access energy are only 0.52% and 0.5% of the OOO core.

III. Recycling Slack via Transparent Flip-flops

The previous section highlighted the prevalence of considerable data slack in executing operations. In order to execute consumer operations immediately after the producer completes (i.e. to recycle this data slack), we make use of transparent dataflow via intelligent FF control. Note that we incorporate transparent datatflow only within data bypass network between execution units. Via this design, ALU operation sequences can execute ”transparently”. Other operations such as multi-cycle, FP and memory operations are still ”true synchronous” operations and do not themselves benefit from transparent execution.

A transparent mode FF design is a simple implementation consisting of a standard FF but with a bypass path [20]. A

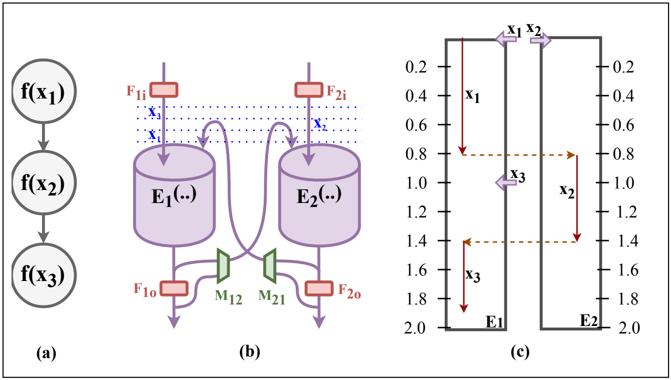


Figure 4: Data Slack Recycling

mux at the end of the 2 paths can select the ”opaque” stored FF value or the bypassed ”transparent” value, based on an enable input. In our work, transparent mode is enabled in the bypass path between ALUs whenever data is required to flow through at non-clock boundaries. This allows varied delays across operations to be balanced anywhere within the transparent execution window. Note that such a design can also be implemented via latches [21], which is prevalent in Intel designs [22].

We propose a synchronous slack tracking and opportunistic early clocking mechanism implemented atop a transparent execution pipeline. Our proposed mechanism *reduces the execution latency at the cost of increased EU utilization*. We introduce the concept with a simple discussion on applying transparent dataflow to a generic pair of execution units (E_i) as shown in Fig.4.b. We assume that the units have forwarding paths to each other (shown in figure) and back to themselves (not shown). Also, forwarding logic is simplified to only show forwarding to a particular input of the execution units (i.e. right input of E_1 and left of E_2) but actual design would support both inputs of each, and would extend to more execution units as well. Moreover, we focus on single-cycle combinational execution. These units could be thought of as the ALUs in standard OOO processors.

Consider the data flow graph Fig.4.a. It shows a sequence of 3 dependent operations and need to be executed in sequence. The functional flow atop a pair of execution units (EUs) is depicted in Fig.4.b, wherein the stream of inputs x_i are distributed in sequence over the two E_i. In conventional design, this system is entirely executing at a throughput of 1 operation per cycle i.e. not executing at peak throughput, and consumes 3 clock cycles to complete. Note that the operations could have any other distribution across the 2 EUs, but the throughput is always limited to 1 operation per cycle. In other words, in each cycle one execution unit is always idle in this system.

Assume the presence of data slack for each $f(x_i)$, i.e. for each operation's computation on an execution unit. In standard synchronous design, EUs are lodged between opaque FFs and inputs and outputs pass through only at clock edges,

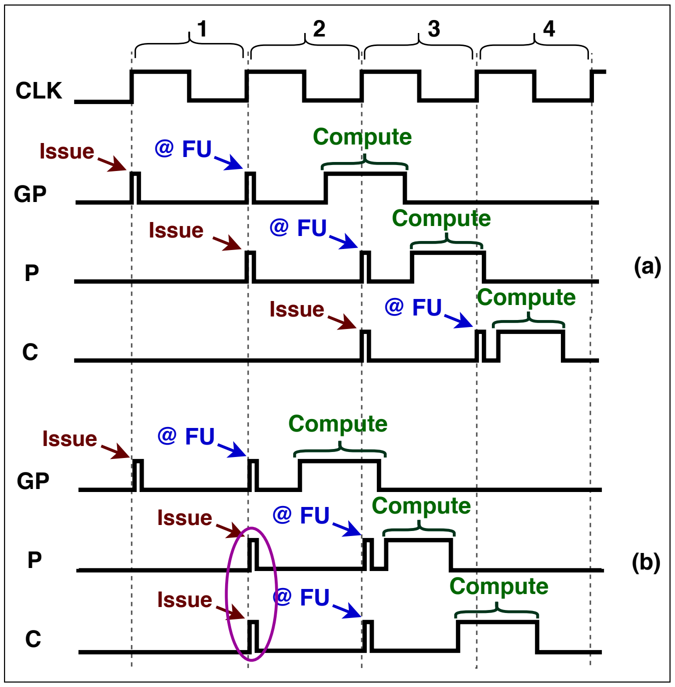


Figure 5: Timing Diagram of Execution Pipeline

causing this slack to be wasted. This is indicated in the figure by F_{1i} and F_{1o} bounding E_1 and similarly for E_2. Our proposed mechanism cuts out this slack by introducing ”transparent FF” based data bypass between the execution units. The ability to bypass the data around the output FF is also shown in Fig.4.b. Mux M_{12} can enable bypassing of F_{1o} when forwarding E_1's output to E_2. Similar bypassed dataflow is possible from E_2 to E_1.

Our proposal performs 3 distinct intelligent tasks (ITs):

- IT_1: For a producer operation with slack, a consumer operation is brought early to an idling EU (if available).
- IT_2: The FFs are made transparent (via mux-control) in the bypass paths between the EUs holding the producer and consumer operations respectively, for the period of time that the producer is available at its functional unit.
- IT_3: An operation is held to a EU for two cycles or one cycle depending on whether its execution (via the above mechanism) might cross a clock boundary or not, respectively.

Fig.4.c describes this functional flow over the 2 EUs, in more detail, via an example. Consider that the three operations (x_i) described earlier, can execute on the EUs with latencies of 0.8ns, 0.6ns, 0.5ns respectively. The red solid arrow indicates estimated execution time and the yellow arrows show dependencies.

1 At t=0ns, x_1 is brought to the input of E_1. This begins computation and would complete at t=0.8ns. 2 In parallel with x_1, x_2 is brought to input of E_2 (an IT_1). $f(x_2)$ isn't ready for computation yet, since x_1 is yet to complete on E_1 but is brought in early so that $f(x_1)$'s slack can be completely utilized. 3 Also at t=0ns, the transparent bypass path from E_1 to E_2 is selected via mux M_{12} as it is estimated that $f(x_1)$ completes in this cycle (IT_2). 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 Note that x_1 is held at E_1 for one cycle while x_2 is held at E_2 for 2 cycles. This is because computation time estimates indicated that x_1's execution does not cross a clock boundary while x_2's does (IT_3). 5 At t=1.0ns, x_3 is brought early to E_1 (IT_1) and bypass path $E_2 - E_1$ is made transparent while bypass path $E_1 - E_2$ is made opaque (IT_2). Further x_3 will hold the unit only for 1 cycle since it computes correct data from t=1.4ns to t=1.9ns (IT_3). 6 A true-synchronous operation after x_3 (eg. Store instruction) can clock at t=2.0ns. Some slack is lost but the computation is still 1 cycle faster than the pure synchronous baseline which took 3 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.

IV. Slack-Aware OOO Scheduling

The transparent dataflow of dependent operations between functional units can be used to recycle data slack *IF* a slack aware scheduling mechanism is in place. The scheduler is responsible for issuing instructions to execution units, based on some priority scheme, when all required resources (source operands and execution units) are available. Our slack-aware optimization focuses on two components of the scheduler: the wakeup logic and the select logic.

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 parent (producer) instructions as well as the available resources. The **selection logic** chooses instructions for execution from the pool of ready instructions waiting in its reservation station entries (RSEs). Priority-based scheduling (e.g. oldest-first) is required when the number of ready instructions are greater than the number of available functional units. This happens when tags from parents of multiple instructions become available; these instructions are all awakened and send requests to the select logic.

Note that our slack aware scheduling mechanism is *focused only on single-cycle operations*. We do not attempt to recycle slack in multi-cycle operations, which reduces some potential overheads which are beyond the scope of this paper.

A. Motivation

Implementing slack-aware scheduling in OOO processors requires some challenges to be addressed. In state-of-art deeply pipelined processors, the instruction scheduler is decoupled from actual execution. Using a fixed latency assumption for each 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 potential early completion of operations within their clock cycle. This requires augmenting the scheduler with data-slack information. Moreover, when a producer operation is expected to produce slack, the scheduler needs to schedule a consumer operation early enough (onto an idle functional unit), so that the consumer can begin evaluating immediately after the producer's completion.

An illustration of how the timing of instruction issue is integral to recycle slack via transparent dataflow is shown in Fig.5. The figures show 3 instructions (named GP: grandparent, P: parent, C: child) being executed in a processor pipeline. This simple illustration shows the pipeline issuing instructions one cycle before they arrive at the functional unit and become available for compute to begin. (Note that this is not a design assumption and is only for illustrative purpose.) In Fig.5.a, GP is issued at the beginning of cycle 1, and becomes available for execution on an FU at the beginning of cycle 2. Assuming it is made to wait for some previous producer operation (aka great grandparent) to complete in cycle 2, (as described earlier), it then begins evaluating immediately within cycle 2, and completes at some instant in cycle 3. Even via conventional single-cycle tag broadcast, GP's tags can be broadcast in cycle 1 and can wake up instruction P to issue (if selected) at the beginning of cycle 2. P then becomes available at the FU at the start of cycle 3, and begins evaluating after GP is complete and then evaluates into cycle 4. Similarly, C is woken up and selected at the beginning of cycle 3 and is prepared for execution. In this example, operations GP, P and C only need to be issued on consecutive cycles as enabled by conventional scheduling logic.

On the other hand, a different scenario is shown in Fig.5.b. While GP and P are issued as was discussed in the first scenario, a difference arises here because P finishes evaluating within cycle 3 (due to high data slack). To recycle P's slack, C needs to begin evaluating in cycle 3 as well, so it needs to arrive at its FU (note: a different FU from the one P is computing on) at the beginning of cycle 3. To achieve this it needs to issue at the beginning of cycle 2, i.e. at the same time as its parent, P. This scenario is not possible with existing scheduling logic as the scheduling loop requires one cycle; this motivates our modifications to the scheduler, which are discussed below.

1) **Eager Grandparent Wakeup**: speculative wakeup based on grandparent operations (a modified design based on [23]) so that child operations can be issued in parallel with parent operations.
2) **Slack Tracking**: Calculating and tracking an operation's completion time based on execution times and producers' completion times.
3) **Skewed Selection**: Select logic which prioritizes non-speculative operations over speculative (grandparent-awoken) operations.

B. Eager Grandparent Wakeup

Grandparent wakeup (GPW) is a speculative wakeup technique used to wake up a child operation based on the broadcasted tags of its grandparent operations [23]. In the original proposal by Stark et al. [23], GPW is used to prevent pipeline bubbles when the scheduling loop (wakeup-select-broadcast) is pipelined. The goal behind their work was that as pipeline stages and clock frequency grow, it is imperative to break down the timing critical scheduling loop into multiple stages. Pipelining this scheduling loop naively would result in inefficiency: not being able to execute dependent operations in consecutive cycles. But if tags of the grandparent(s) are used to wake up the child instruction, this inefficiency can be avoided. The idea was motivated by the notion that if the grandparents of a child instruction have been selected for execution, then it is likely that the parent will be selected in the following cycle (considering single-cycle operations). The child can then be executed in the cycle following its parent. More details can be found in the original proposal.

Clock frequency and pipeline depth have stabilized in the last decade, so current day schedulers can support single cycle scheduling without the use of grandparent wakeup. The conventional pipeline schedule for a 3-operation dependency graph is shown in Fig.6.a. The single cycle scheduling loop is performed in cycle one for waking up the XOR operation based on the OR operation. Similarly, in cycle two, XOR broadcasts and wakes up the AND.

However, the need for eager scheduling to recycle data slack (as motivated in Fig.5) creates a need for a grandparent scheduling-like mechanism. We modify GPW to create Eager GPW (EGPW), to wake up the child operation in the same cycle as the parent operation. While this is unnecessary in standard pipelines, it is useful for slack recycling: consumer operations can be sent to idle functional units early (in the same cycle that the producer operation completes) so that the slack from the parent operation is recycled. For the same DDG, assume that the XOR operation has data slack which can be exploited by the AND if it can start execution on the same cycle as the XOR. The corresponding pipeline schedule via EGPW is shown in Fig.6. The OR instruction wakes up its child (XOR) and its grandchild (AND) in the same cycle. XOR wakeup is conventional, while AND wakeup is achieved by the speculative EGPW

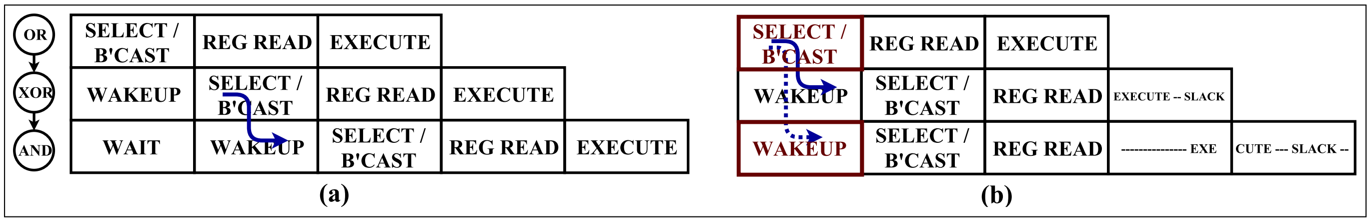


Figure 6: Eager Grandparent Wakeup

mechanism. This allows the AND instruction to arrive in parallel with the XOR at a functional unit and wait for the XOR output to transparently flow through. It then begins useful computation in the same cycle and effectively recycles the XOR operation's slack. As also seen in figure, if the AND reads a second operand from the register file, this also happens early (in parallel with the XOR) based on conventional RF port availability.

In the original implementation [23], GP-mispeculation can potentially occur when the grandchild instruction is woken up with the grandparents tags, but the parent does not get selected. This is verified by checking for the eventual broadcast of parent tags. They show that these mispeculation rates are very low when sufficient functional units are available. Our skewed selection mechanism deprioritizes GP-wakeups and can largely (or even completely) eliminate GP-mispeculation. This is discussed further in Sec.IV-D.

Note that EGPW only wakes up the grandchild instruction. The conditions for this grandchild to be selected for issue are explained in the following two sections on Slack Tracking and Skewed Selection.

C. Slack Tracking

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. In a conventional design, each reservation station entry (RSE) has 2 parent (or source) tags which are identifiers for the source operands. Once the tag matches occur, the instruction is woken up. A request is placed to the Select logic and if selected, it receives a grant. If selected, its destination tag is then broadcast on the tag bus. A more detailed description can be found in prior works [23]. Our goal is to augment this baseline design with slack-awareness.

The following discussion will put forth two designs for slack-aware scheduling. The first is *Illustrative*: its discussion aids in explaining our technique in a step-by-step manner. The second is *Operational*: it is the actual design we employ, suitable for practical implementation.

Illustrative Design for Slack-Aware Scheduling:

Our proposed augmented RSE entry is shown in Fig.7. In the RSE, slack is tracked with a 3-bit fractional representation i.e. slack precision of 1/8th of the clock period (details in Sec.V). The timestamp within a clock cycle at which an instruction completes is its 3-bit Completion Instant (CI). The CI is calculated for a given instruction based on its parent/grand-parent CIs and slack information, and is written into the COMP-INST field of the RSE. This is explained as part of the mechanism below.

(1) Conventional parent tags (P_1, P_2) which are identifiers for source operands, are shown. If both tags comparisons hit (i.e. a match with tags broadcast on the destination bus), the instruction is awoken and a request is sent to the select logic for selection. (2) Similar to the grandparent scheduler design by Stark et al. [23], we add grandparent tags (GP_1 - GP_4) to enable grandparent based instruction wakeup. If all tags hit, a *speculative* request is sent to the select logic. Differentiating between a normal select and a speculative select by the *skewed* select logic will be explained in Sec.IV-D. (3) In case of a parent based wakeup, the estimated CI of the parents are used to determine the starting instant of the child (or current) instruction. $P_iC.I.$ are the CIs, which are broadcast along with the tags (obtained from the CI bus). (4) Similarly, in case of a grandparent based wakeup, estimated CI of the grandparents are obtained off the CI bus. The Max logic estimates the later CI (i.e. last completing) from each set of grandparents. (5) The 3 EX-TIME fields in RSE indicate the estimated execution time for this particular instruction and its parents respectively, each of which is a 3-bit value. These values are calculated at decode (read out of the slack LUT: described in Sec.II-B) and are written into the RSE and the Register Alias Table (RAT). The child instruction obtains the EX-TIMEs for the parents from the RAT during register renaming. (6) In case of Parent-based wakeup, the current instruction can start executing immediately after the last completing parent. In case of GP-based wakeup, the execution time of the Parent instructions should be accounted for. For GP-based wakeup, each parent's EX-Time is added to the latest CI of the corresponding grandparent set, thus producing the parent CI. (7) Based on the instruction wakeup being parent-based or GP-based, the appropriate CI is selected (via a mux) for the two source operands to the current instruction. (8) Among the 2 parent CIs, the later one is selected via the MAX operation (as the child would start executing after this). (9) The completion instant of the child is then calculated by adding its EX-TIME to the last completing parent's CI. (10) A child operation would issue in the current cycle only if a) slack recycling is enabled, b) the completion instant of the last parent is expected within the current cycle (like operation P in Fig.5.b) and further, is within some *slack threshold* (discussed later) and c) a grant

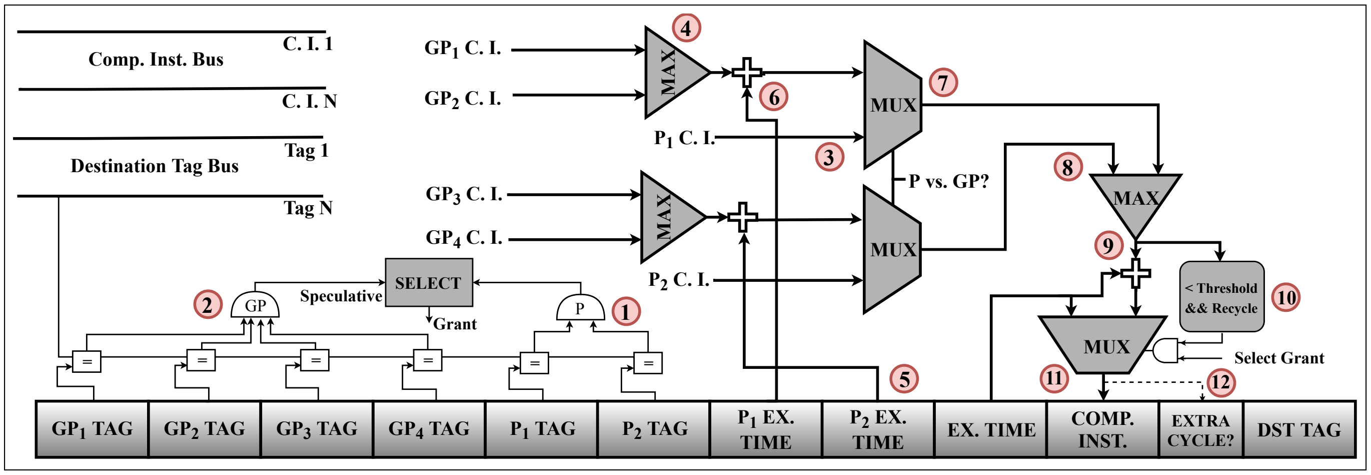


Figure 7: Illustrative design for Slack aware RSE (Note: steps 3-10 occur in parallel with selection)

is obtained from the select logic. (11) The appropriate CI is written into the current instruction's CI field, and then broadcast along with the tag. This could either be the CI, as calculated in (9), or, in a scenario where slack recycling does not happen and the current instruction is executed from a clock cycle boundary (of a later cycle), the value written in would be the operation's EX-TIME itself. (12) Further, if the execution of the operation is expected to cross a clock boundary (such as GP and C, but not P, in Fig.5.b), the execution unit is allocated for an extra cycle (i.e. 2 cycles for traditional single cycle operations).

The slack threshold discussed in (10) is used to achieve a balance between potential benefit from slack recycling vs. potentially excessive FU utilization caused by the 2-cycle allocation requirement. A higher threshold would recycle slack more aggressively, starting consumer operations in the producer's completion cycle even when if there is very low slack in that cycle. The potential benefit then depends on enough small slack increments accumulate to cross a clock cycle boundary, reducing exposed latency in the dataflow graph. The potential detriment is that the FU underutilization caused by 2-cycle allocation might cause slowdowns under high FU demand. Ideally, a simple but intelligent dynamic mechanism can be used to increase or decrease this threshold based on overall observed benefits. In this initial work, we tuned this value via a design sweep for each set of applications (refer Sec.VI-C).

Operational Design Slack-Aware Scheduling:

From the physical design perspective, increasing the number of tags in the RSE is quite expensive because all wakeup buses should be connected to all source tag comparators in all entries. This can significantly increase the load capacitance on the bus and the wakeup logic drivers [24].

In order to reduce potentially detrimental energy/area overheads from the Illustrative design and for the implementation to be practical, we propose an Operational design which closely matches (within 1%) the former's performance. It is based on two key observations: 1) a significant fraction of arithmetic computations have only a single source operand [24], and 2) even within the fraction of operations with multiple source operands, the last arriving source operand (tag) is predictable with high accuracy [25].

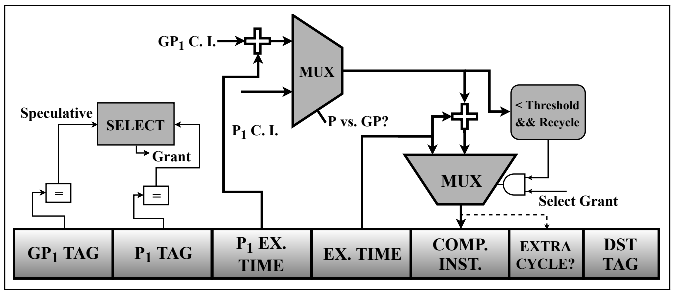


Figure 8: Operational design for Slack Aware RSE

Based on the above observations, we predict the last arriving parent for each single-cycle arithmetic operation. Further, this information is passed from a parent to its child operation during rename (via the RAT), meaning that a child operation uses a prediction for both its last arriving parent and its last arriving grandparent.

This last-arrival prediction mechanism tremendously reduces design complexity and is shown in Fig.8. Only 2 tags are now required in each RSE, one each for the last arriving parent and grandparent respectively. The RSE will require only 2 EX-TIME fields one being its own execution time, and the other being that of the last arriving parent. Their usage was described in the earlier Illustrative design. Moreover, the slack calculation logic gets significantly simplified, as there is no requirement to compare and estimate the last arriving source operands.

The prediction of the last-arriving tags must be validated to ensure that the instruction did not execute before all of its operands are available. The prediction is correct if the operand predicted to be not arriving last is already available when the instruction enters the register read stage of the pipeline. We utilize a small register scoreboard mechanism

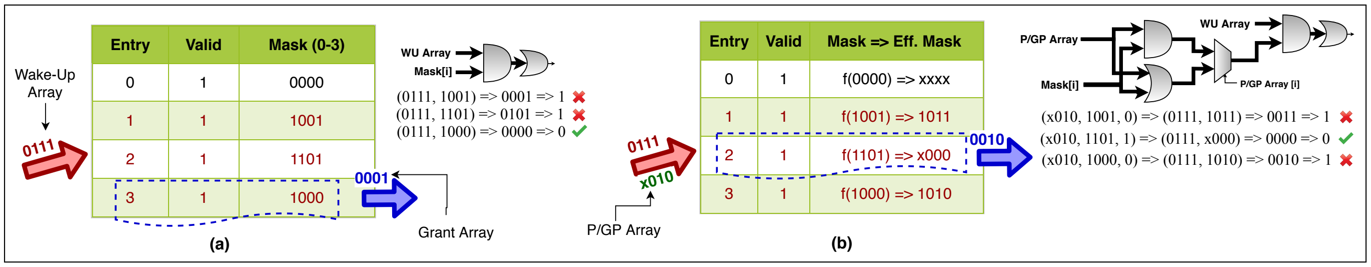


Figure 9: Skewed Select Logic (Note: gate-level design is illustrative)

from prior work [25] to achieve the same. If the prediction is incorrect, error recovery is required, in a fashion identical to latency mispredictions (but with lower penalty). Considering the almost perfect prediction accuracy (Sec.VI-B), the performance impact is nearly zero.

D. Skewing the Select Arbiter

We *skew* the selection logic to prioritize conventional requests over speculative GP-requests. Only if there are remaining FUs after allotment to conventional requests will they be allotted to GP-requests. Thus, no conventional request suffers from not being serviced due to *other* selections. The mechanism to skew the selection logic is shown in Fig.9.

Conventional Selection:

Fig.9.a shows conventional N:1 selection logic (representative of standard processors) implementing an oldest first priority mechanism. Valid entries are filled up into the selection table in parallel with the reservation station. For an N-entry table, an N-bit mask is used to indicate the priority order. In any entry's mask, a 1 at the i_{th} bit from the left indicates that the i_{th} entry is older (i.e. has higher priority). For example, the 0_{th} entry has highest priority (mask is all 0s) while 1_{st} entry has a lower priority than entries 0 and 3. Ready instructions send requests to the select arbiter - indicated via the wake-up array. Here, instructions corresponding to entries 1, 2 and 3 have woken up and are requesting grants. The circuit shown adjacent to the table, decides which entry gets the grant, i.e. which among the woken up entries has the highest priority. The 3rd (producing a 0 through the circuit) is found to be the highest priority awake entry and is given the grant. In the figure, the grant array represents the output from the selection logic indicating which instruction gets the grant.

Skewed Selection:

Fig.9.b shows our proposed skewed selection logic. Skewed selection prioritizes non-speculative requests (from parent based wake-ups) over speculative requests (from GP based wake-ups) while respecting the original priority scheme among each group of requests. The P/GP array is an additional input to the selection logic, which indicates which requests are speculative (GP) and which are non-speculative (P). P requests are shown as a 1 and GP requests show up as a 0 in our design. Again considering the same example, entries 1-3 are woken up. Further, the example assumes entry 2 wakes up non-speculatively while entries 1, 3 wake up speculatively. Since entry 2 is the non-speculative request, it has priority over the other 2 speculative requests. This is implemented by calculating the ”effective mask”. The circuit implementation is shown adjacent to the table. In the example, entry 1 has its mask altered from 1001 to 1011, since the 3rd entry is a non-speculative wakeup. Similar alteration occurs for entry 4. Conversely, entry 2 has its mask altered from 1101 to x000 i.e. bits corresponding to speculative entries are made 0. The 'x' indicates a don't care since entry 0 is not woken up. After this, the selection circuit from earlier calculates the appropriate entry for selection, which is the 2nd entry in this scenario. It should be noted that the skewed logic is laid out as a simple (but inefficient) sequence of gates, simply for illustrative purposes. The actual (negligible) increase in delay is discussed in Sec.IV-E.

Discussion:

There are two key reasons to skew the selection policy of the select arbiter. They are motivated below.

The first motivation is to improve FU utilization. Previously, we had discussed how speculative GP-wakeups and non-speculative conventional parent-based wakeups both send requests to select logic to obtain grants. We also discussed that the grandparent based early wakeup is useful only when the child instruction needs to be issued in the same cycle as the parent (i.e. when there is slack beyond the completion instant of the parent, refer Sec.IV-A). This means that any grants provided to the grandparent-based wakeup are unutilized if there is no slack to recycle in that particular cycle. This is indicated by ANDing the select grant with the recycling decision in figure 7 and 8. In such a scenario, execution units go under-utilized in those cycles when the select logic selects a GP-wakeup request instead of a conventional parent-wakeup and there is no slack to recycle.

The second motivation is to prevent (or reduce) mispeculation from grandparent scheduling. GP-mispeculation occurs when a child operation woken up by a grandparent is selected for issue without the parent also being selected (Sec.IV-B). Skewed selection prioritizes conventional (parent-wakeup based) requests over speculative GP-wakeup based requests. This means that within an arbitration window, a GP-wakeup can never race ahead of a conventional wakeup. Therefore, a child would never be selected for execution ahead of its parent as long as they are a part of the same select arbitration window.

The arbitration window depends on the design of the select logic. Assume that the processor selects M instructions for execution (on M units) from N requests. This can be implemented as a) a global arbitration window performing N:M selection [26] or b) M/K local arbitration windows, each performing N:K selection. In the first scenario, there would be no GP-mispeculation thanks to the skewed selection logic. In the second scenario there would be no GP-mispeculation within each window but there could be GP-mispeculation across windows. In this work we assume global arbitration.

E. Summary of Overheads

It is key to note that the entire slack aware mechanism described above happens in parallel with select logic. Select requests are issued at wakeup oblivious to slack, and select grants are returned at the end of the cycle. The instruction's execution is then finally determined by the grant as well as the slack/CI calculation described above. Moreover, the slack-aware computations are only 3 bits wide, resulting in the critical path of slack-computation being significantly shorter than select logic arbitration. Thus this primary design component of slack based scheduling does not increase the critical paths in scheduling logic.

Area/power overhead of the proposed Operational design is negligible, the main additions only being 10 extra bits per RSE, two 3-bit adders (with overflow) and muxes, and a comparator, contributing to an area overhead of 0.3% and an energy overhead of 0.8%. Note: the adder overflow determines if the computation's execution crosses a clock boundary, the use of which was explained in Sec.IV-C.

Synthesis of the skewed selection logic shows that the additional delay in select logic amounts to only 3 ps additional delay over the baseline 100 ps select logic. Further, considering the significant wire delay that exists in the select arbitration tree [26], this increase in delay would be reflected negligibly in real design.

The marginal increase in critical path delay via skewed selection and the absence of any additional critical timing component in the slack tracking mechanism described earlier means that there is hardly any change to the timing of the scheduling loop (which can be a near timing critical, sometimes dominated by load-store unit and the fetch loop [3]).

V. Methodology

Simulation: We extended the Gem5 [27] simulator to support Slack Recycling atop standard out-of-order cores. We model 3 cores labeled *Big*, *Medium* and *Small*. The description of the cores can be found in Table.I.

Benchmarks: The benchmarks for analyzing results are 2-fold. The first set encompass relatively compute intensive applications from the SPEC CPU 2006 [28] and the MiBench benchmark [29] suites. They are run via multiple Simpoints [30], each of length 100 million instructions. The second set consists of kernels from ARM Compute Library [31] for computer vision and machine learning with support for ARM NEON SIMD (brief details in Table.II). Benchmarks are all compiled for the ARM ISA; NEON vectorization flags are turned on for the ML kernels.

Table I: Processor Baselines

Parameter	Small	Medium	Big
Frequency	2 GHz		
Front-End Width	3	4	8
ROB/LSQ/RSE	40/16/32	80/32/64	160/64/128
ALU/SIMD/FP	3/2/2	4/3/3	6/4/4
L1/L2 Cache	64kB/2MB w/ prefetch		

Table II: Kernels for Machine Learning

Kernel	Description
CONV	Convolution: Gaussian 3x3
ACT	Activation: ReLU
POOL 0/1	Pooling: 2x2 Max/Average
SOFTMAX	Softmax function

The benchmarks and their operation characteristics are shown in Fig.10. The characteristics shown are: memory operations with high/low latency (MEM-HL/MEM-LL; HL refers to L1 cache misses), NEON SIMD operations, other multi-cycle operations (eg. fp) and high/low slack single-cycle ALU operations (ALU-HS/ALU-LS, HS refers to data slack greater than 20% of the clock cycle). While many SIMD operations are pipelined and multi-cycle, accumulate, multiply-accumulate etc. support late-forwarding of accumulate operands from similar ops, allowing sequential single-cycle execution [32].

Influence of PVT variation: Pure data slack estimates correspond to worst-case design corner to isolate it from PVT (process, voltage, temperature) variations. Thus, the data slack estimates expect to stand true across all PVT conditions. Executing under nominal PVT conditions provides some exploitable guard band [1], [8] and adds a small additional component to the total slack.

In real design, guard band variations from PVT can be measured with Critical Path Monitors (CPMs) [8]. To exploit PVT guard band, our design only requires *localized CPMs in the proximity of the ALUs and bypass network*. Conventional CPM based guard band estimates (eg. Power7 [8]) are more conservative since they are located in the most timing/power critical regions of the entire chip.

To account for PVT variation, slack LUTs are recalibrated on-the-fly, thus supporting changes to voltage, temperature, aging etc. We adhere to a tuning granularity of 10,000 cycles as is prescribed in Tribeca [1]. There are no design-time/testing overheads for PVT based slack calibration, which is simply tracked dynamically with CPMs.

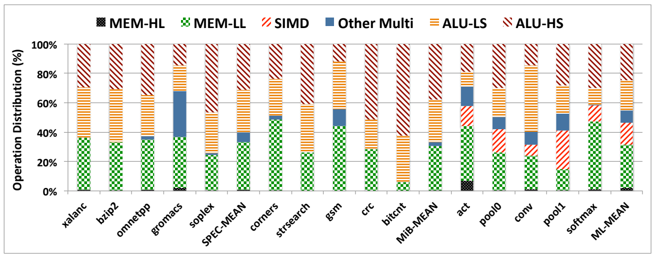
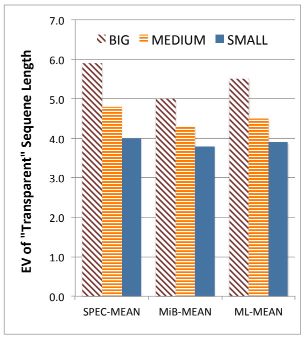


Figure 10: Benchmark Operation Characteristics



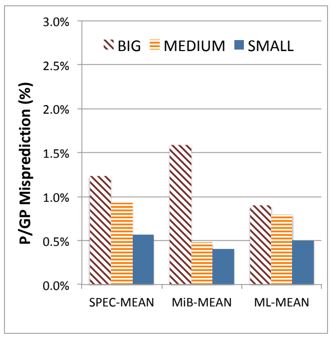


Figure 11: Seq. Length Figure 12: Tag Prediction

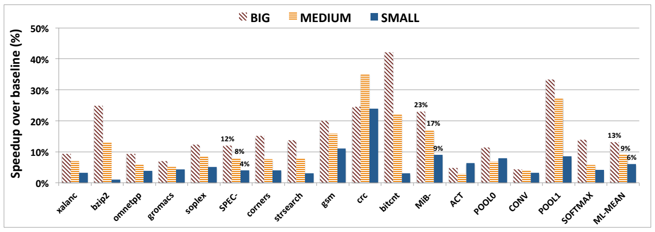


Figure 13: Speedup for different cores

Data slack measurements: In this work, we only target single cycle data slack for the integer ALU as well as in specific integer SIMD operations. We do not target data slack from other multi-cycle operations such as FP ops. Slack is modeled via RTL design in Verilog and synthesis using the Synopsys Design compiler. We synthesize the execution pipeline stage with a 0.5 ns cycle time (i.e. 2 GHz) constraint. As explained in Sec.II-B, ReDSOC only requires to categorize operations into 14 different slack buckets - we do not need accurate slack estimations of each operation. This completely simplifies CAD timing analysis.

Our slack analysis agrees with estimations from prior work [33] as well characterization via gate-level C-models. Considering the low effort, we expect state-of-the-art CAD tools to be capable of (or extendable to) such analysis. During processor execution, appropriate slack bucket is selected simply based on opcodes and operands: no dynamic timing analysis is involved.

Timing Closure: The introduction of transparent FFs (i.e. selective FF bypassing) adds some complexity to timing analysis/closure of the execution unit and data bypass network. For the simplified 2-EU system shown in Fig.4, traditional timing paths (in a standard FF design) to analyze for timing closure would be $(F_{1i}-F_{1o})$, $(F_{2i}-F_{2o})$, $(F_{1o}-F_{2o})$ and $(F_{2o}-F_{1o})$. These would require to be single cycle timing paths. The introduction of FF bypassing introduces 2-cycle timing paths which also need to be verified. In the same example, these would be $(F_{1i}-F_{2o})$ and $(F_{2o}-F_{2o})$ when M_{12} is enabled for transparent dataflow. Similarly, there would be $(F_{2i}-F_{1o})$ and $(F_{1o}-F_{1o})$ when M_{21} is enabled for transparent dataflow. These paths are marked as 2-cycle paths during timing analysis, the rest of the design's timing remains traditional. Note, M_{12} and M_{21} are never transparent at the same time - this precludes combinational loops.

Slack Tracking Precision in the RSE: We quantized slack / timing corresponding to different precisions (up to 8-bits) in our architecture simulator and analyzed performance. Performance saturated at 3-bits (or 1/8th of a clock cycle). Hence, 3-bit values are sufficient for slack reycling.

VI. Results

A. Potential for Sequence Acceleration

As discussed in Sec.III, slack accumulates over a sequence of operations which can be executed in a transparent manner. Fig.11 shows the expected value (i.e. weighted mean) of the length of all such sequences. The values shown are averaged across each benchmark class and evaluated for each core type. It can be seen that the observed average length of these transparent sequences is between 4-6 operations. Slack per operation can usually vary between 10%-60% to of the clock-cycle. Thus, the average length of these transparent sequences is sufficient to accumulate one or more cycles of slack, resulting in early clocking of sequence-ending ”true synchronous” operations, providing speedup.

B. Last parent / grand parent prediction

Fig.12 analyzes accuracy of tag prediction in the *Operational* design, using a prediction table with 1K entries. The table is addressed by PC-bits and uses 1 bit for prediction per entry to indicate if the particular operand is last to arrive. High accuracy keeps mispredictions to around 1%. Accuracy is lower for larger cores due to higher scheduling traffic.

C. Performance Speedup

Fig.13 shows the speedup obtained over a standard baseline without slack conservation for different core sizes.

The first observation is that speedups are lower for SPEC benchmarks compared to MiBench. This is partially due to SPEC having a significant percentage of high dependence memory operations. Moreover, the average percentage of high slack ALU operations in SPEC is only around 30% while it is close to 60% in MiBench. MiBench applications, on the other hand, show significant speedups (23% average on the BIG cores). The *bitcount* application sees over 40% speedup over the baseline. This is not surprising, considering that benchmark characterization (Fig.10) 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 idle functional units for data to transparently flow into, 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, allowing multiple dependency chains within the application to perform slack conservation. Fig.14 illustrates how the pipeline stalls from busy FUs increases from the baseline to REDSOC. For smaller cores, this has some limiting effect on the maximum speedup from slack recycling.

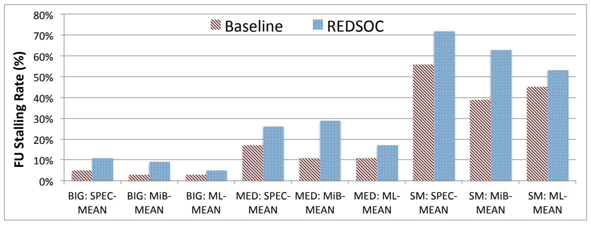


Figure 14: Pipeline stall rates from busy FUs

Finally, speedup on the ML kernels is from both low-precision NEON SIMD operations and reasonable fractions of high slack single-cycle operations. Due to their working sets, some of these kernels (e.g. ACT) spend a significant portion of time waiting for long-latency memory operations to complete, and this cuts down gains to some extent. Efficient prefetcher tuning and blocking the matrices could increase slack opportunities, so these results might be pessimistic.

To estimate power efficiency at baseline performance, we convert speedup into power savings via application-level V/F scaling. Scaling is modeled on ARM A57 [34]. Mean power savings on chosen SPEC, MiBench and ML benchmarks range from 8-15%, 12-36% and 8-18% respectively.

D. Comparison with other proposals

We quantitatively compare ReDSOC against our own implementations of timing speculation and operation fusion. **TS** is our timing speculation mechanism (similar to Razor) wherein frequency is controlled depending on the error rate in the application. Frequency is statically fixed so as to maintain an error rate between 1% and 0.01% across application execution. Note, we do not model recovery for timing errors; thus, the performance numbers shown for TS can be considered as optimistic. **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.

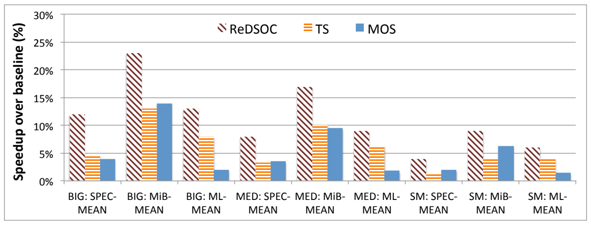


Figure 15: Comparison with other proposals

Comparison of these two mechanisms against ReDSOC atop the three different core types are shown in Fig.15. It is clear that ReDSOC significantly outperforms both mechanisms by 2x or more. *MOS* opportunity is limited in most applications, due to an inability to find many sequential operations to combine into a single cycle. It achieves reasonable speedup in MiBench due to higher data slack averages. *TS* is limited by the fact that frequency control can happen only at a coarse temporal granularity, while data-dependent slack varies from operation to operation. Hence, the *TS* setting has to be set rather conservatively to maintain low error rates.

VII. Related Work

Multiple "Better than worst case" approaches have been proposed in prior work [2], [35]–[37], especially targeting PVT variation [1], [3], [8]. Fast ALU computations are implemented in some Intel processors [38].

Prior works optimize narrow data-width based execution to improve EU utilization [15], effective register capacity [17], issue width [16] and energy reduction in multiple parts of the core [39].

Finally, multiple works optimize scheduling and breakdown its critical loop [23], [26], [40]–[42].

VIII. Conclusion

This paper showed that data slack can often be a significant portion of the clock period, and cutting out this slack provides tremendous opportunity to improve performance. With the increasing popularity of applications that utilize low-precision arithmetic, data slack is becoming even more prevalent.

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 over multiple operations executing on 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 reasonable solution for better clock-period utilization in standard OOO cores.

Acknowledgment

The authors would like to thank anonymous reviewers for their insights and comments. This work was supported in part by the National Science Foundation under award CCF-1615014.

References

[1] M. S. Gupta, J. A. Rivers *et al.*, “Tribeca: Design for pvt variations with local recovery and fine-grained adaptation,” in *MICRO*, 2009.

[2] D. Ernst, N. S. Kim *et al.*, “Razor: A low-power pipeline based on circuit-level timing speculation,” in *MICRO*, 2003.

[3] A. Tiwari, S. R. Sarangi *et al.*, “Recycle:: Pipeline adaptation to tolerate process variation,” in *ISCA*, 2007.

[4] G. S. Ravi and M. H. Lipasti, “Aggressive slack recycling via transparent pipelines,” ser. ISLPED, 2018.

[5] R. P. Colwell, C. Y. I. Hitchcock *et al.*, “Instruction sets and beyond: Computers, complexity, and controversy,” *Computer*, 1985.

[6] V. Vanhoucke, A. Senior *et al.*, “Improving the speed of neural networks on cpus,” in *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[7] J. Xin and R. Joseph, “Identifying and predicting timing-critical instructions to boost timing speculation,” in *MICRO*, 2011.

[8] C. R. Lefurgy, A. J. Drake *et al.*, “Active management of timing guardband to save energy in power7,” in *MICRO*, 2011.

[9] J. Sampson, G. Venkatesh *et al.*, “Efficient complex operators for irregular codes,” in *HPCA*, 2011.

[10] S. Yehia and O. Temam, “From sequences of dependent instructions to functions: An approach for improving performance without ilp or speculation,” ser. ISCA, 2004.

[11] Y. Park, H. Park *et al.*, “Cgra express: Accelerating execution using dynamic operation fusion,” ser. CASES, 2009.

[12] A. Bracy, P. Prahlad *et al.*, “Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth,” ser. MICRO, 2004.

[13] H. Cherupalli, R. Kumar *et al.*, “Exploiting dynamic timing slack for energy efficiency in ultra-low-power embedded systems,” ser. ISCA, 2016.

[14] P. Judd, J. Albericio *et al.*, “Proteus: Exploiting numerical precision variability in deep neural networks,” ser. ICS, 2016.

[15] D. Brooks and M. Martonosi, “Dynamically exploiting narrow width operands to improve processor power and performance,” ser. HPCA, 1999.

[16] G. H. Loh, “Exploiting data-width locality to increase superscalar execution bandwidth,” ser. MICRO, 2002.

[17] O. Ergin, D. Balkan *et al.*, “Register packing: Exploiting narrow-width operands for reducing register file pressure,” ser. MICRO, 2004.

[18] N. P. Jouppi, C. Young *et al.*, “In-datacenter performance analysis of a tensor processing unit,” ser. ISCA, 2017.

[19] D. J. Schlais and M. H. Lipasti, “Badgr: A practical ghr implementation for tage branch predictors,” in *ICCD*, 2016.

[20] E. L. Hill and M. H. Lipasti, “Transparent mode flip-flops for collapsible pipelines,” in *ICCD 2007*.

[21] M. Fojtik, D. Fick *et al.*, “Bubble razor: Eliminating timing margins in an arm cortex-m3 processor in 45 nm cmos using architecturally independent error detection and correction,” *IEEE Journal of Solid-State Circuits*, 2013.

[22] D. M. Wu, M. Lin *et al.*, “An optimized dft and test pattern generation strategy for an intel high performance microprocessor,” in *ITC*, 2004.

[23] J. Stark, M. D. Brown *et al.*, “On pipelining dynamic instruction scheduling logic,” ser. MICRO, 2000.

[24] I. Kim and M. H. Lipasti, “Half-price architecture,” ser. ISCA, 2003.

[25] D. Ernst and T. Austin, “Efficient dynamic scheduling through tag elimination,” ser. ISCA, 2002.

[26] S. Palacharla, N. P. Jouppi *et al.*, *Complexity-effective superscalar processors*. ACM, 1997.

[27] N. Binkert, B. Beckmann *et al.*, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, 2011.

[28] J. L. Henning, “Spec cpu2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, 2006.

[29] M. R. Guthaus, J. S. Ringenberg *et al.*, “Mibench: A free, commercially representative embedded benchmark suite,” ser. WWC, 2001.

[30] E. Perelman, G. Hamerly *et al.*, “Using simpoint for accurate and efficient simulation,” in *International Conference on Measurement and Modeling of Computer Systems*, 2003.

[31] A. Inc., “Arm compute library,” *https://developer.arm.com/compute-library/*, 2017.

[32] A. Inc., “Cortex-a57 software optimization guide,” *https://infocenter.arm.com/*, 2016.

[33] G. Tziantzioulis, A. M. Gok *et al.*, “b-hive: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units,” ser. DAC, 2015.

[34] A. Frumusanu and R. Smith, “Arm a53/a57/t760 investigated: Samsung galaxy note 4 exynos review,” *https://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review*, 2017.

[35] B. Greskamp and J. Torrellas, “Paceline: Improving single-thread performance in nanoscale cmps through core overclocking,” ser. PACT, 2007.

[36] T. M. Austin, “Diva: a reliable substrate for deep submicron microarchitecture design,” in *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, 1999.

[37] T. Austin, V. Bertacco *et al.*, “Opportunities and challenges for better than worst-case design,” ser. ASP-DAC, 2005.

[38] G. Hinton, D. Sager *et al.*, “The microarchitecture of the pentium 4 processor,” *Intel Technology Journal*, 2001.

[39] E. Gunadi and M. H. Lipasti, “Narrow width dynamic scheduling,” *Journal of Instruction-Level Parallelism*, 2007.

[40] P. Michaud, A. Mondelli *et al.*, “Revisiting clustered microarchitecture for future superscalar cores: A case for wide issue clusters,” *TACO*, 2015.

[41] M. D. Brown, “Reducing critical path execution time by breaking critical loops,” Ph.D. dissertation, 2005.

[42] A. Perais, A. Seznec *et al.*, “Cost-effective speculative scheduling in high performance processors,” ser. ISCA, 2015.