UNIVERSITY OF WISCONSIN MADISON

PRELIMINARY PROPOSAL

Accelerating Approximate Programs via Aggressive Slack Recycling

Author:

GOKUL SUBRAMANIAN RAVI

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF WISCONSIN - MADISON

Advisor:

MIKKO LIPASTI

OCTOBER 23, 2017

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 (data) and environment (PVT). In the common non-critical cases, this creates clock cycle *slack* - the fraction of the clock cycle performing no useful work. In prior works, we proposed LAGS and ReDSOC (both under submission) which use transparent latching techniques to effectively recycle multiple forms of slack and improve performance while maintaining required reliability constraints. These mechanisms focused on timing slack reduction for accurate computing - we believe there is significant potential to extend these proposals into the realm of approximate computing.

Allowing computation to be approximate can lead to significant energy savings because it alleviates the correctness tax imposed by the wide safety margins on typical designs. While there have been multiple proposals in the software space to unearth opportunities to leverage approximate computing for speedup or efficiency improvements, we believe that there is much scope for improvement in hardware design. Our proposal is motivated by multiple limitations in current approximate hardware, with specific focus on timing approximation. These limitations include - ① the absence of approximate hardware providing performance speedup as the first-order benefit, ② current hardware often requiring duplicated approximate and approximate resources, ③ lack of timing approximate hardware with fine grained temporal control (of approximation), ④ ignorance of inherent operation precision as well as environmental effects, leading to undisciplined approximation and ⑤ the inability to support multiple approximation levels.

To combat these limitations, our work proposes to extend the notion of timing slack into approximate computation. We propose disciplined but increased aggressiveness in timing slack estimates at an instruction granularity, resulting in improved performance and efficiency. This is performed at the cost of allowing some timing errors to creep into computations, but always staying within application-specific bounds on the error margins, as estimated via circuit timing analysis. We expect to study our proposal's benefits across different error-tolerant applications and across standard OOO and spatial substrates, as well as specialized architectures such as neural network accelerators. Beyond these primary contributions, this work also seeks to explore additional optimizations - selective approximation for effective acceleration and variation aware task scheduling.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Objectives and Contributions . . . 4
1.3 Outline . . . 5

2 Background **5**
2.1 PVT Variations . . . 5
2.2 Data-dependent Variations . . . 6
2.2.1 Operation Type (Opcode-Slack) . . . 6
2.2.2 Data Width of Operands (Width-Slack) . . . 6
2.2.3 Data Type of Operands (Type-Slack) . . . 7
2.2.4 Data Slack Distribution . . . 7
2.3 Prior Solutions for Accurate Computing . . . 8
2.4 Other Related Works . . . 9
2.5 Approximate Computing . . . 9

3 Recycling Slack via LAGS designs **10**
3.1 LAGS for Spatial Fabrics . . . 10
3.1.1 Motivation . . . 10
3.1.2 Locally Asynchronous, Globally Synchronous . . . 11
3.1.3 LAGS via transparent pipelines . . . 12
3.1.4 LAGS for Spatial Fabrics . . . 14
3.1.5 Slack Estimation . . . 16
3.2 ReDSOC for OOO Cores . . . 17
3.2.1 Slack Estimation . . . 18
3.2.2 Optimizing the OOO Scheduler . . . 18

4 Proposed Research **21**
4.1 Accelerating Approximate Programs via Aggressive Slack Recycling . . . 21
4.1.1 First order benefit is power and not performance . . . 22
4.1.2 Require dedicated approximate and accurate resources . . . 23
4.1.3 Lack temporal control at fine granularities . . . 24
4.1.4 Ignore inherent precision and environmental effects . . . 25
4.1.5 Unable to support multiple approximation levels . . . 25
4.1.6 Proposal . . . 26
4.2 Secondary proposals . . . 27
4.2.1 Selective approximation for effective acceleration . . . 27
4.2.2 Variation aware task scheduling . . . 27

5 Conclusion **28**

6 Schedule **28**

1 Introduction

Over the past many decades, through device, circuit, microarchitecture, architecture, and compiler advances, Moores Law, coupled with Dennard scaling, has resulted in commensurate exponential performance increases [19]. However, as transistor scaling trends continue to worsen, causing severe power limitations, improving the performance and energy efficiency of general purpose processors has become ever more challenging. This has made the exploration of non-classical techniques of paramount importance to improving processor efficiency.

One such non-classical technique that is explored in our proposals is the interaction between various abstraction layers of modern computer processing. We believe that the optimization of specific abstraction layers, with inputs about opportunities or limitations in other layers, is key to furthering computer architecture advancement. In this regard, our work proposes innovations to microarchitecture with a clear understanding of both, the software-level application requirements as well as hardware-level circuit/device characteristics.

1.1 Motivation

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: *PVT Slack*, caused due to non-critical environmental (Process, Voltage, Temperature) conditions, and *Data Slack*, caused due to non-triggering of the executional critical path. Scaling to lower technology nodes creates an increasing gap between worst-case and nominal circuit delays, requiring even larger guard bands [16, 40].

Under typical conditions, *PVT Slack* often averages more than 25% of the clock period [70, 29]. Its systematic components generally have relatively low temporal and spatial variability (or are predictable) and can more easily be tackled with traditional solutions [29, 18, 70].

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). 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 [72]. 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. [15]).

Section 2.3 discusses state-of-the-art techniques to reduce or recycle slack along with their limitations. Further, Section 3 discusses our prior proposals to improve the slack recycling and thus improve performance and energy efficiency. These works have predominantly focused on accurate computing. But another key domain with significant

current day processing opportunities is approximate computing.

Workloads from several prevalent and emerging application domains, such as vision, machine learning, data analysis, etc., possess the ability to produce outputs of acceptable quality in the presence of inexactness or approximations in a large fraction of their underlying computations [73]. This forgiving nature of applications has lead to the advent of a new class of design techniques [73], collectively referred to as approximate computing, that leverage the flexibility provided by intrinsic application resilience to realize efficient hardware or software implementations. Allowing computation to be approximate can lead to significant energy savings because it alleviates the correctness tax imposed by the wide safety margins on typical designs [21].

We find tremendous potential in linking the domains of timing slack recycling and approximate computing. More specifically, the fine grained slack recycling mechanism which our prior proposals have established are well suited for fine-grained inter-mingling of approximate and accurate compute operations and the varying levels of approximation among the approximate operations. Towards this goal, this proposal presents multiple opportunities for "Accelerating Approximate Programs via Aggressive Slack Recycling" which are discussed below.

1.2 Objectives and Contributions

The primary contribution of this work is to develop a disciplined but aggressive fine-grained slack recycling mechanism suited for approximate computing, resulting in improved performance and efficiency. Our proposal is motivated by multiple limitations in current approximate hardware (with specific focus on timing approximation).

Current approximate hardware provide first order improvements to power consumption but not to performance; while performance improvements can be achieved subsequently, it is not as effective. Second, most prior proposals on timing approximation require dedicated (duplicated) resources to perform approximate and accurate computing in parallel, due to the fine-grained mixing of approximate and accurate operations in program code. Next, the timing approximate hardware lack the ability to control approximation (via feedback based mechanisms) at fine granularities of time due to other design limitations (eg. voltage control). Moreover, these proposals end up with undisciplined approximation due to ignorance of inherent operation precision and the influence of environmental (PVT) changes. Finally, prior proposals do not have the capability to support multiple approximation levels - which, as shown by analysis in recent software proposals, is important for efficient approximation.

Our proposal is effective in avoiding such limitations, by piggy-backing on our prior work on slack recycling. Converting tolerable approximation into a slack component allows the use of LAGS/ReDSOC slack recycling techniques to accelerate sequences of operations of any length, with no scaling of voltage/frequency etc. Each slack component (PVT, data, approximation) is determined separately, making approximation cognizant of the PVT effects proximate to the compute unit, as well as the precision/data-type of the operands being computed on. Further, without the need for voltage/frequency scaling, the approximation levels in our proposed design can be tuned at very fine temporal granularities based on application requirements. Our design allows each functional unit to work as accurate or approximate, making it suitable for processing application phases with fine-grained inter-mingling of approximate and accurate computations. Moreover, multiple levels of approximation can be integrated seamlessly into each functional unit since approximation is determined as a unique slack component for each computation.

Apart from the above proposal, we also propose secondary contributions. First, we propose to explore dynamic identification of appropriate program slices to approximate. Different program slices contribute differently to output accuracy as well as to the application execution time. Thus, the selection of appropriate slices to approximate (and

thus, accelerate) is important to maximizing performance speedup at best possible accuracy.

Second, we propose exploration of PVT-variation aware task graph scheduling. We will explore the potential of variation-aware scheduling techniques which will schedule critical tasks on *better* compute nodes (in the context of variation). Further, we seek to implement temporal awareness in task scheduling - appropriately spacing out task scheduling over time rather than taking a greedy approach of maximum parallelism, which can influence thermal variations and voltage drops.

1.3 Outline

The rest of this proposal is organized as follows: In Section 2, we discuss background and related work. Section 3 provides an overview of our own prior work towards slack recycling, on top of which we build upon in this proposal. Section 4 details our proposal for "Accelerating Approximate Programs via Aggressive Slack Recycling" across different substrates and other secondary explorations. Finally, Section 5 summarizes this document and Section 6 outlines a proposed schedule for the proposed work.

2 Background

2.1 PVT Variations

As chip designers attempt to reduce supply voltage to meet power targets, parameter variations are a serious problem. Environment induced variations which affect the functioning of a processor fall into three categories: process, voltage and temperature. Process variations are caused due to wafer characteristics, doping fluctuations [4], imperfections in manufacturing process, etc., leading to potentially large variations in device attributes [29]. These variations can be classified into Die-to-die (D2D) and Within-Die (WID) variations. D2D variations are affected by factors such as processing temperature while WID are affected by variations in doping, channel length, threshold voltage, etc. [71].

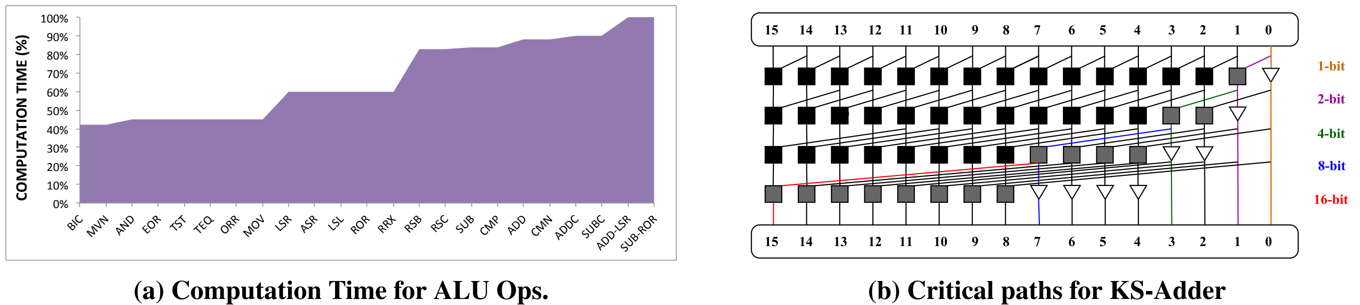
WID variation is further composed of systematic and random components. Systematic variation is characterized by spatial correlation, meaning that adjacent areas on a chip have roughly the same systematic components [66, 67]. Correlation, as per the well-studied spherical model [75], decreases linearly with distance for small distances. Beyond a finite distance (ϕ), correlation converges to zero [67]. Random variations occur at a much finer granularity, at the level of individual transistors, with no correlation characteristics. The required timing margins for random variations is a function of both supply voltage and length of logic paths in the execution unit. Random delay variability increases sharply as supply voltage scales down - scaling from 1.0V to 0.3V increases variability by 6x, making this a large variation component at near-threshold voltages [44]. Similarly, variability increases sharply for path lengths shorter than 20 FO4 delays, 1 FO4 long path has 4x larger delay variability than 20 FO4 [44].

Voltage and temperature variations vary with workload and environment [71]. Voltage variations result in current fluctuations on the order of 10s to 100s of cycles [29] and can also exacerbate thermal hot spots. Thermal variations can cause changes to leakage current and restrict the permissible voltage and TDP in the chip's environment [71].

2.2 Data-dependent Variations

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 [74] 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.

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.



(a) Computation Time for ALU Ops. **(b) Critical paths for KS-Adder**

Figure 1: Data Slack Components

2.2.1 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. But since many of the executing operations do not trigger the critical path of the functional unit, they 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 data path 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.1a shows the critical computation times for different operations on a single-cycle ARM-style ALU, written in 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.

2.2.2 Data Width of Operands (Width-Slack)

High-end processor word widths are usually 32-64 bits while a large fraction of the operations are implcitly 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 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 - 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 [6], among others.

But the problem of low temporal utilization for narrow-width computations has been less explored. Fig.1b 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 [6]. Prediction methods for identification of data-width early in the execution pipeline, have also been very successful [46, 17].

2.2.3 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.1b, 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 [37], often enabling the use of narrow data types, specified directly from the software level.

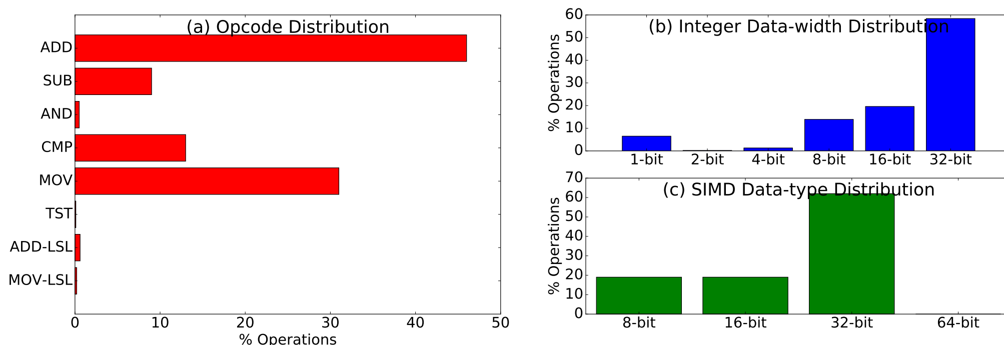


Figure 2: GEMM operation distribution

2.2.4 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 [33] widely usable in Machine Learning applications. Fig.2.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.2.b shows the distribution of actual data-width for 32-bit

integer operations and Fig.2.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.

2.3 Prior Solutions for Accurate Computing

From the above discussion, it is intuitive why traditional designs need to employ large timing guard bands to prevent variation based timing violations, leading to timing slack and thus, less than ideal performance. In order to improve clock-period utilization and thus better performance and efficiency, it is clear that novel solutions are necessary.

There are 4 domains of microarchitecture that have marginally explored this goal in different forms, which are of interest to our proposal. They are discussed below.

The first is *timing speculation*. Timing Speculation is a state-of-the-art mechanism, which cuts into traditional timing guard bands, providing better performance and/or energy efficiency at the risk of timing violations. Prior works in this domain have focused on adaptive variation of the operating points (F,V) by tracking the frequency of timing errors occurrences [18] or by predicting critical instructions [74]. Coupled with error detection and recovery, it presents a functionally correct, efficient, processor design.

Prior synchronous TS solutions suffer two fundamental constraints. First, they are bounded by the possibility of timing errors from *every* computation, in *every* synchronous FU/op-stage, and on *every* clock cycle. Second, the dynamic mechanisms among these are implemented by varying frequency/voltage over time and thus, can only be *reconfigured* at reasonably coarse granularity of time (at best, over epochs of 1000s of cycles).

Thus, ensuring no (or minimal) timing errors over the entire epoch forces these operating points to be set rather conservatively, constrained by timing requirements of each operation in the entire epoch. Otherwise, they run the risk of increased timing violations. While recovery mechanisms [18] maintain reliable operation in the face of timing errors, they impose significant penalties on performance and energy efficiency. Moreover, the design overheads in implementing timing error detection and timing overheads from recovery are significant [42].

The second domain is *specialized data-paths*. When specialized datapaths are built to accelerate certain hot functions, specific circuit logic elements are combined together in sequence and the timing for the datapath can be optimized for the particular chain of operations [64, 77]. But such datapaths 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 [60] and further, rearranging instruction flow to improve the availability of suitable operation sequences to fuse [5]. 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.

The final domain includes *asynchronous compute* solutions which are inherently suited to slack conservation [44]. Varying execution times among operations, which could cause timing errors in an aggressive synchronous TS design, can be avoided by allowing such varied delays to be balanced anywhere within the asynchronous execution window. But pure asynchronous solutions suffer from other functional complexities resulting in low throughput and/or high overhead implementation costs, making them a less popular solution.

2.4 Other Related Works

Multiple prior works have focused on reducing the effects of static variations alone, recognizing that variation affects some pipeline stages more than others and accordingly tune delays per stage at fabrication [70, 43]. Techniques to combat dynamic variations mostly tune V/F on the fly by tracking variations. POWER7 [42] introduced Critical Path Monitors to estimate the delays caused by PVT variations and implemented feedback controllers to adjust V/F. Tribeca [29] uses a last value predictor for V/F settings over discrete time intervals based on dynamically gathered PVT information. TIMBER [14] detect timing errors after the clock edge and borrow slack from the next pipeline stage. Other works have proposed "better than worst case" approaches to improve energy efficiency [26, 2, 1]. Circuit-level speculation [45, 56] and other asynchronous techniques to capture data slack have potential to be incorporated into our work.

Multiple prior works have optimized for narrow data-width based execution in the context of improving utilization of functional units [6], register packing of multiple narrow operands into single registers [17], improving issue width [46] and energy reduction in multiple parts of the core [27]. Some accelerators have provided low-level precision control via bit-serial compute units [39], per-layer precision control [38] and so forth.

Finally, multiple works have studied the complexity of out of order cores [47, 57, 7] 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 [68, 61, 7].

2.5 Approximate Computing

Workloads from several prevalent and emerging application domains possess the ability to produce outputs of acceptable quality in the presence of inexactness or approximations in a large fraction of their underlying computations [73]. This intrinsic resilience can be attributed to several factors [73]: (i) the inputs are drawn from the real world and are therefore noisy and redundant, (ii) the algorithms and computation patterns employed in these applications tend to self-heal or attenuate the errors introduced by approximations, and (iii) users of these applications are conditioned to accept multiple equivalent results, or any result of acceptable quality. This forgiving nature of applications has lead to the advent of a new class of design techniques, collectively referred to as approximate computing, that leverage the flexibility provided by intrinsic application resilience to realize efficient hardware or software implementations. Their results show that it is possible to exploit these properties for a variety of purposes increased performance, reduced energy consumption, increased adaptability, and increased fault tolerance.

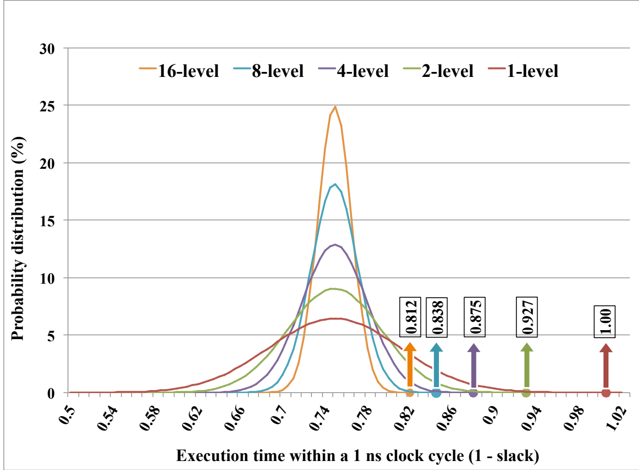
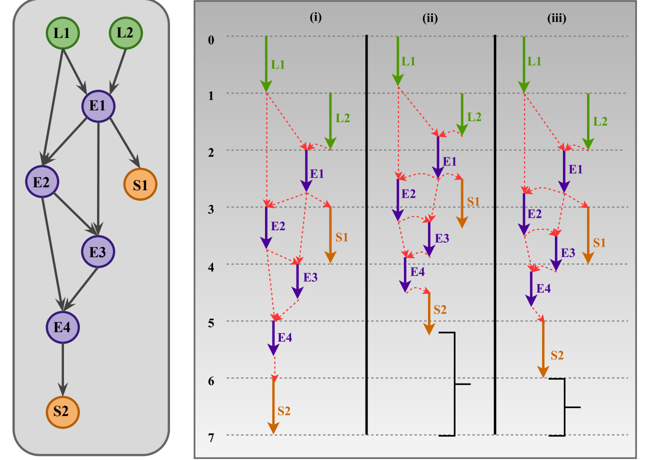


Figure 3: Slack distribution



(a) DFG **(b) (i) PS, (ii) PA, (iii) LAGS**

Figure 4: Comparing the benefits of LAGS

3 Recycling Slack via LAGS designs

In this section we briefly highlight our previous proposals for recycling different forms of slack, targeting accurate computing.

3.1 LAGS for Spatial Fabrics

3.1.1 Motivation

The underlying commonality among Razor and most state-of-art TS approaches is that, they all *focus on reducing slack on a per operation basis* and are *constrained by the possibility that timing errors might be caused by every operation, in every unit and on every cycle.* In order to avoid (or limit) timing errors, the tuning mechanism must ensure reliability for the whole tuned epoch and is limited by the most critical instructions. Therefore, these techniques are relatively conservative - having to cater to the most critical operations or execution stages to prevent mispeculation, or suffer the risk of increasing possibilities of timing errors. The following analysis describes the potential for more aggressive timing speculation and motivates LAGS.

Fig.3 shows different slack distributions. Consider the flattest distribution, the *1-level* curve in red. This represents independent and identically distributed logic delay within an FU - averaging at about 75% of the clock cycle but with a reasonably wide distribution. This curve is equivalent to a design wherein every operation executes synchronously - as is common in prior work. The corresponding red arrow refers to the 99.99% confidence interval mark, which is the clock period that, if set, allows not more than 1 in 10000 operations to hit a timing error. The arrow is roughly at the 1.00 mark and thus, it is evident that in this example, using the 99.99% estimate as guard band is unable to cut out any slack.

On the other hand, assume a design wherein multiple independent operations are executed asynchronously in a sequence i.e. the operations are not separated by clocked elements. The slack accumulates across this sequence and slack estimates are influenced by the number of operations that can be combined together and executed as a single chain. The greater the lengths of these chains ("level" in figure), the higher the slack per operation. This is because,

for independent variation, the available slack is averaged out over the entire sequence predominantly consisting of non-critical operations. Many such operations would tend to lie closer to the mean value (curve peak increases and width narrows), meaning that outliers with a longer critical path can be cushioned by other non-critical operations which consumed less than the high confidence execution time estimate. This allows a lower guard band for the same confidence interval. The higher *N-level* curves in Fig. 3 correspond to the slack distribution when multiple operations are combined together in a multi-cycle sequence. *N-level* corresponds to the average slack estimation for a sequence of length *N*. In this example, combining 16 operations together allows a 20% reduction in the 99.99% guard band estimate.

From this example, it is evident that multi-cycle execution of asynchronous operation sequences provides abundant potential for increased aggressiveness in timing speculation.

3.1.2 Locally Asynchronous, Globally Synchronous

Complete asynchronous architectures which have been proposed in the past have not gained sufficient popularity over the years due to the complexities in their implementation [24]. While asynchronous engines have high potential for timing speculation [44], they are only suited to particular portions of the execution pipeline. Therefore, we believe that potential for incorporation into existing synchronous designs can only be achieved via a LAGS proposal.

We propose designing architectures in which some portions of the data-flow can form an asynchronous engine (A-Engine) surrounded by synchronous boundaries. All external interfaces to the A-Engine are kept synchronous to allow ease of design, verification and programmability *external* to the A-Engine. External stimuli are unaware of the A-Engine's internal asynchronicity. Further, the entire control flow is kept synchronous, which also allows ease of design and verification *internal* to the A-Engine.

A comparison of purely-synchronous (PS), purely-asynchronous (PA) and LAGS designs for general purpose execution is shown in Fig.4. The data dependence graph (DFG) for a sequence of instructions is shown in Fig.4.a. The timing diagram for the sequence is depicted in Fig.4.b. L, E and S represent Load, ALU and Store instructions respectively and red dotted lines represent dependencies. In this example, ALU instructions are considered asynchronously executable while memory instructions are synchronous boundaries - a characteristic of our design space.

① Fig.4b.i represents PS data-flow. The length of the arrows represent the actual execution time for each instruction. Though many computations complete before the clock edge, the dependent instruction can execute only after the clock edge resulting in slack wastage. Such a sequence consumes 7 cycles. ② Fig.4b.ii represents a PA data-flow wherein *every* operation is independent of the clock. Even memory instructions are initiated asynchronously (S_1, S_2) and slack from memory interactions can be used in starting dependent instructions early (E_1). By completely using up all the slack, the sequence completes in 5.25 cycles. While providing maximum benefits, PA leads to higher design complexity and other disadvantages, as discussed earlier. ③ Fig.4b.iii represents LAGS, wherein operations internal to the data-flow engine flow transparently but all interactions with memory and processor state are seen synchronously. This results in losing some opportunities to save slack (E_1 only executes at start of cycle 2) but in comparison to a totally synchronous design, multiple instructions can still execute early (E_2, E_3, E_4). LAGS provides performance benefits akin to true asynchronous designs (1 cycle saving here), without its inherent disadvantages.

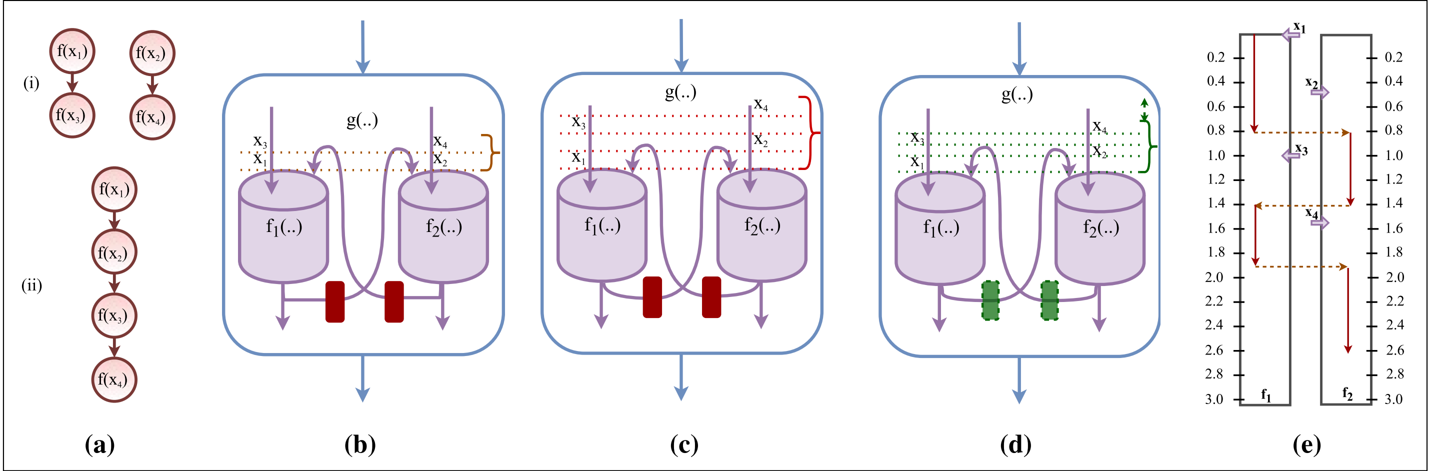


Figure 5: Recycling Data Slack with Transparent Latching

3.1.3 LAGS via transparent pipelines

Background: Having motivated the need for LAGS, we now shift attention to how to implement the local asynchronous engines. Typical asynchronous circuits have been implemented in three ways : ① Purely combinational multi-cycle data paths [64], ② Asynchronous Elastic pipelined logic [10, 55] and ③ Transparent pipelines via intelligent latching [34, 31]. All of these are capable of conserving timing slack by executing a sequence of operations in an asynchronous group but the below discussion motivates the better capabilities of transparent pipelines in comparison to the others.

Purely combinational multi-cycle data paths (MDP) suffer from the problems of limited throughput. Such data paths are not pipelined, meaning that standard synchronous pipeline throughput requirements can only be met via extensive resource replication. MDP for DFG-style execution has been explored in the context of specialized data paths (custom-cores) created for execution of specific basic blocks [64]. In these custom-cores, one basic block executes for each pulse of a *slow clock*, foregoing pipeline registers and saving area/power. Being specialized datapaths, they do not provide sufficient flexibility for general-purpose programming.

Asynchronous elastic pipelines provide asynchronous execution as well as the throughput of a synchronous pipeline [10, 55]. Therefore they can theoretically be wedged between synchronous boundaries and integrated into a synchronous pipeline. But they suffer from some major inefficiencies. Completion detection within pipestages, the handshake mechanisms between pipe-stages (eg. 4-phase signalling) and feedback loops are complex to implement [55]. This restricts high-frequency implementations, causes high area overheads and complicates DFT hardware. Moreover, synchronous boundaries are much harder to implement with these designs, requiring Async-Sync FIFOs, causing added overheads and complexities [11].

Our work therefore explores the third option - the use of transparent pipelines via intelligent latching. 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 [34, 28] as well as interlocked synchronous pipelines which reduce stall related overheads [35].

We propose a synchronous slack tracking and opportunistic early clocking mechanism implemented atop a trans-

parent 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.5.b. We assume that the FUs have forwarding paths to each other (shown in figure) and back to themselves (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.5.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.5.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.5.a.ii. In this scenario all 4 operations are dependent and need to be executed in sequence. The functional flow is depicted in Fig.5.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.5.d depicts how the use of transparent latches brings the 4 execution operations *closer* together, reducing execution latency (compared to 5.c).

Example: Fig.5.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, which are discussed in more details later on, and in the LAGS paper. Execution latencies are assumed to be obtained from operation details - opcode, data-width and precision. Latches can be made transparent or opaque for half clock cycles. New inputs can be brought to the input of a $f()$ at every half clock cycle.

➀ At t=0ns, x_1 is brought to the input of f_1. This begins computation and would complete at t=0.8ns. ➁ 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. ➂ 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. ➃ 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. ➄ 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.5.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.

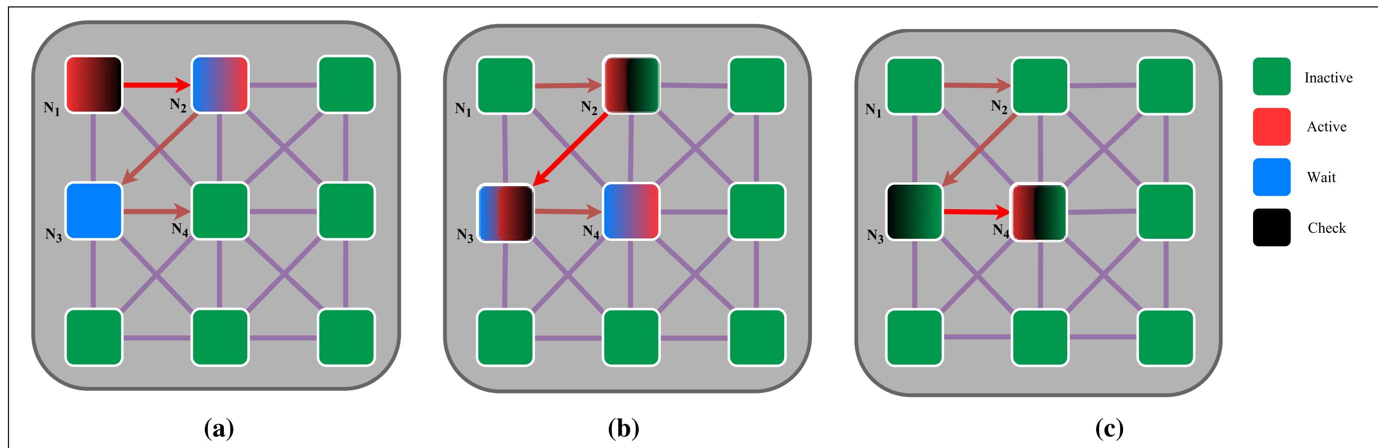


Figure 6: Execution of a DFG on a Spatial Fabric over 3 cycles. (a) - (c): Cycle 1 - 3.

3.1.4 LAGS for Spatial Fabrics

Background: In spatial architectures, an algorithm's DFG is broken into regions, which are connected by producer-consumer relations, and are mapped on to multiple processing elements (PEs). A typical spatial architecture could consist of 10s or 100s of PEs, connected via an on-chip network which can be fixed [28] or reconfigurable [65, 25]. Execution Data-Flow Graphs (DFGs) are mapped on to these PEs, allowing functional parallelism while respecting producer-consumer relations. External interactions outside the boundaries of the spatial fabric are limited to mapping of operation blocks from the front end, configuring the routing network (if reconfigurable), architectural state updates (eg. Arch. Register File), and special operations that cannot be executed internal to the spatial fabric, such as memory operations and complex functions. Data routing between PEs can use non-autonomous control (from outside the fabric) or autonomous control internal to the PEs [58]. Depending on the compute capabilities of the PEs, spatial architectures are widespread ranging from tiled microprocessors [65, 28] to special-function neural network accelerators [12, 21] to heterogeneous general-purpose accelerators [25, 53].

Why Spatial Fabrics?: ① Spatial fabrics are generally over-provisioned with enough compute resources for high throughput when sufficient application parallelism exists. It is intuitive to imagine that when opportunities for parallel execution are reduced leading to low resource utilization, *idling/waiting resources can instead be put to use to perform slack conservation*, simply by allowing transparent data-flow.

② Availability of idling/waiting PEs also eases transparent latch control. In traditional transparent pipelines [34, 31], latch management is more complex due to concurrently executing tasks in both the producer stage and the consumer stage, resulting in the need for scheduling bubbles [31], etc. But this is avoided in spatial frameworks which allocate tasks only to free PEs, often far ahead of actual execution.

③ Spatial architectures reduce scheduling complexity for timing speculation. In standard state-of-art OOO designs, changes in delay at any particular part of the pipeline percolate throughout the pipeline through bypass networks. If the latency of a single instruction changes, all dependents must be ”rescheduled” to pickup their operands off the bypass at the correct time. Generally, this implies global control across large regions of the pipeline resulting

in significant wire delay and overall reductions in clock frequency. In the context of spatial architectures performing in-place execution (due to the luxury of multiple PEs), there are no such complexities from data forwarding. As brought up earlier, dependent younger operations (consumers) already sit at FUs ahead of execution time and can directly begin execution after the parent operation (producer) is deemed to be complete.

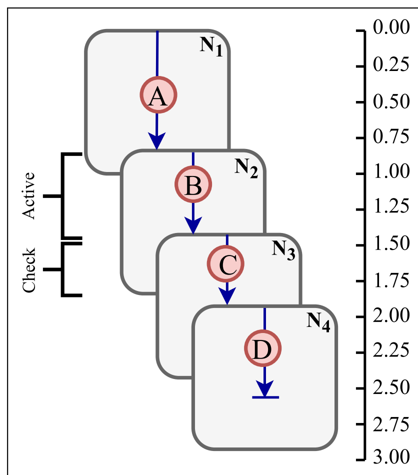


Figure 7: Timing Diagram for Spatial LAGS

Baseline: Consider Fig.6.a which shows a spatial fabric with 9 PEs with interconnect. To understand the functioning, the PEs can be thought to have 3 primary logical states - *Inactive*, *Active* and *Wait* (The fourth, *Check* is discussed later this section, in the context of timing speculation). An *Inactive* PE has no DFG node mapped onto it or its useful work is complete. An *Active* PE has a DFG node mapped onto it: all of its producer-consumer (P-C) constraints have been satisfied (or ”triggered” [58]) and it is in the process of execution. A *Wait* PE has a DFG node mapped onto it, but its P-C constraints are yet to be satisfied and until then it is idle. Similar to TIA [58], CRIB [28] and others, P-C completion/trigger bits flow between PEs, and when all its trigger bits are set, a PE transitions from *Wait* to *Active*. Note, these are not actual physical states in the design.

At regular fine-grained intervals of time, new DFG nodes are mapped onto the *Inactive* PEs from the Front-End. At the same time *Active* PEs and *Wait* PEs can be transitioning to *Inactive* and *Active* respectively. PE selection for optimal compute and routing is orthogonal to this work [54]. As is common to all synchronous designs, these PEs are bounded by clocked elements (eg. flip-flops). The PEs themselves are assumed to execute combinationally and PE-PE latency is considered negligible relative to ALU latency.

Adding LAGS: Transparent latches with slack-aware control are added between the PEs to allow transparent data-flow at appropriate times. Further, a *Check* state is introduced, which is the period of time between the estimated computation completion of a PE (via timing speculation) and the worst-case computation time. Any output transitions within the *Check* phase triggers a timing violation.

Fig.7 shows the mapping of a 4-node DFG (A-D) onto single-cycle PEs N_1 to N_4 of the spatial fabric. It also shows the actual execution timing for the DFG atop the PEs. N_1 - N_4 (and other PEs) are also shown in Fig.6. In this figure, illustrations (a)-(c) show the *state* changes among the PEs in 3 sequential cycles of execution of the same DFG. Here, we assume a clock period of 1ns and also assume the presence of slack (PVT/data) in the execution of each node. The execution latencies are $A : 0.8ns$, $B : 0.6ns$, $C : 0.5ns$ and $D : 0.7ns$. The transitioning colors within some nodes in Fig.6.a-c implies changing state within that cycle of execution.

The initial state of each PE is shown as the left-most color of each PE in Fig.6.a. N_1 is in *Active* (red) state,

executing $f(A)$. N_2 and N_3 are in *Wait* (blue) state, with B and C already mapped to them but not in execution since $f(A)$ is yet to complete. N_4 is in *Inactive* (green) state - D has not yet been mapped from the Front End. All other PEs are *Inactive*. The change of PE states over the 3 cycles in Fig.6 in accordance with the DFG timing in Fig.7, is discussed below:

1. Over the course of the 1st cycle, N_1 executes $f(A)$ from 0-0.8ns and holds the values from 0.8-1.0ns for error detection (N_1 transitions from *Active* to *Check*). As discussed prior, the error detection extends until the worst-case execution time (which is 1ns here). Similarly, N_2 transitions from *Wait* to *Active* since $f(B)$ begins timing speculative execution at 0.8ns (and will eventually complete at 1.4ns).

2. In the 2nd cycle, N_1 is *Inactive* and available for new inputs. N_2 transitions from *Active* (until 1.4ns) to *Check* (1.4-1.8ns) and finally to *Inactive* (from 1.8ns). N_3 starts the second cycle in *Wait* state, and then begins to execute $f(C)$ (as soon as $f(B)$ completes) at 1.4ns and completes at 1.9ns (*Active* state). It then transitions to the *Check* state at 1.9ns (and remains there until 2.4ns). N_4 has been mapped with D and is in *Wait* state until 1.9ns. The completion of its parent $f(C)$ at 1.9ns allows it to move into the *Active* state.

3. In the third cycle, N_1 and N_2 are *Inactive* from before. N_3 completes its *Check* state and then transitions to *Inactive*. Lastly, N_4 stays in *Active* until 2.6ns, stays in *Check* state from 2.6-2.9ns and then transitions to *Inactive*.

Summary: Thus, this example shows how a 4-node DFG which would consume 4 cycles in a synchronous substrate is complete within 3 cycles. A subsequent synchronous boundary (eg. Store instruction) can therefore be *Early Clocked* at 3.0ns.

Apart from accumulating slack via asynchronous execution, a key difference in our proposal in comparison to prior works of timing speculation is that we obtain benefits without dynamic frequency or voltage scaling. To the best of our knowledge, all prior works have focused on gains via DVFS alone. Such adjustments can only occur at reasonably coarse granularity, which forces these mechanisms to be conservative. On the other hand, our work performs *early clocking of synchronous boundaries rather than faster clocking between synchronous boundaries*. This translates to higher performance and better energy efficiency via those accelerated sequences that lie on the critical path of program execution. Note that the boundaries of the spatial fabric will correspond directly to the synchronous clock boundaries of our earlier described mechanism.

3.1.5 Slack Estimation

In the earlier discussion, we assumed knowledge of exact slack for each operation, which is not practical. The complexities in accurately estimating the available slack for every operation are tremendously high and therefore exact measurements on a cycle-by-cycle basis are only possible with e.g. bit-level current sensing [78]. On the other hand it is reasonable to make accurate measurements of easily measurable components and use approximate estimates for other components using a statistical model. The components of slack that we consider are - ① systematic process variations, ② systematic temperature and voltage variations, ③ random process variations and ④ data-based variations. 3 & 4 are modeled as independent and identically distributed (IID) across each operation while 1 & 2 are correlated across operations.

Critical Path Monitors: We assume the presence of industry standard Critical Path Monitors [42] to capture slack components from PVT variations. Conservative estimates of average PVT-based slack is employed by IBM POWER7 by utilizing Critical Path Monitors (CPM) [42]. CPMs are on-chip sensors that measure the timing margin available to circuits on the chip. In the Power7, CPM measurements are accurate to 1/12th of a clock period. By locating the CPMs in the power-dense regions of the microprocessor, they can experience operating points similar to worst chip-wide critical paths and therefore, are used as a conservative estimate.

Statistical Model: Slack components from localized PVT variation effects (especially in a large spatial architectures) and data-based variation cannot be captured from the above. Therefore, to capture slack more aggressively we further use statistical modelling. Prior works [40, 67] model slack from the probability density function (PDF) of the logic delay as a Gaussian normal distribution with calculated σ and μ values. These parameters are calculated based on estimates of the effects of different data inputs as well as PVT variations on logic delay. Statistical slack modelling is especially useful for multi-cycle coalescing based slack estimates, which was discussed earlier in Sec.3.1.1.

The slack estimate at a particular confidence interval mark (we use 99.99%) is estimated for different sequence lengths and written into a look-up table (LUT). Based on the position of an operation within its asynchronous sequence, its slack estimate is obtained from the LUT and used appropriately.

Slack Correction: While the model described above can *theoretically* maintain error rates at the required confidence intervals, it is possible to go over or fall significantly below under some extreme stimuli that are local to the PEs but not to the CPMs. CPMs are located at some specific area on the chip, and very high WID (Within-Die) process variations, for instance, can make some other parts of the chip experience significant difference in variations effects and slack. While this could be combated by replicating the CPMs sufficiently, that would lead to high area/power overheads. Similarly, application phases within some PEs can function at extreme ends of the data-slack model.

Both these scenarios can lead to higher error rates, leading to low performance and energy inefficiency, or might result in very low error rates and might therefore allow for more aggressive timing speculation. To account for this, we make use of a *feedback based slack-correction fraction* which is multiplied with the values read from the LUTs. The fraction jumps higher or lower in proportion to the number of errors (measured over some timing intervals), in comparison to the required confidence error rate.

Also note: First, the non-IID components are modeled just as in a synchronous design, with no asynchronous coalescing. Second, this work focuses only on within-FU slack and ignores slack in the FU-FU interconnect.

3.2 ReDSOC for OOO Cores

Our second proposal, ReDSOC, aggressively recycles data slack in OOO cores. 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.

ReDSOC is a follow-up to the LAGS work discussed above. Hence, in the below sections, we only discuss details of ReDSOC which are vastly different from the earlier LAGS description.

3.2.1 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.3.2.2). We therefore assume the use of a data-width predictor as proposed by Loh [46] and also used by others for optimizations such as Register packing [17].

We utilize a resetting counter based predictor as proposed by Loh [46]. 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 $2k-1$, 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 [46, 17] 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. In this work, we focus on single-cycle execution, but this can be extended to multi-cycle operations as well.

3.2.2 Optimizing the OOO Scheduler

The baseline processor is a conventional super-scalar out-of-order processor - we assume a simple 7-stage pipeline that is shown in Fig.8. Instructions are fetched and decoded in the first two stages and the rename stage translates architectural register identifiers into physical register identifiers. We assume a reservation station based model for scheduling. After instructions are renamed, they wait in reservation stations for their source operands and a functional unit to become available. More details of this model can be found in prior works [68].

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 8: 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 instructions 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.

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.

The second challenge is especially important - instruction select-broadcast-wakeup is already a critical timing loop [57] and cannot naively be made faster (though double pumped scheduling and execution had been implemented in the Intel Pentium 4 [63], for instance). The loop can't be naively pipelined either as this disallows dependent instructions from executing in consecutive cycles [68, 8].

Brown et al. [8] 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 [57], 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.9.

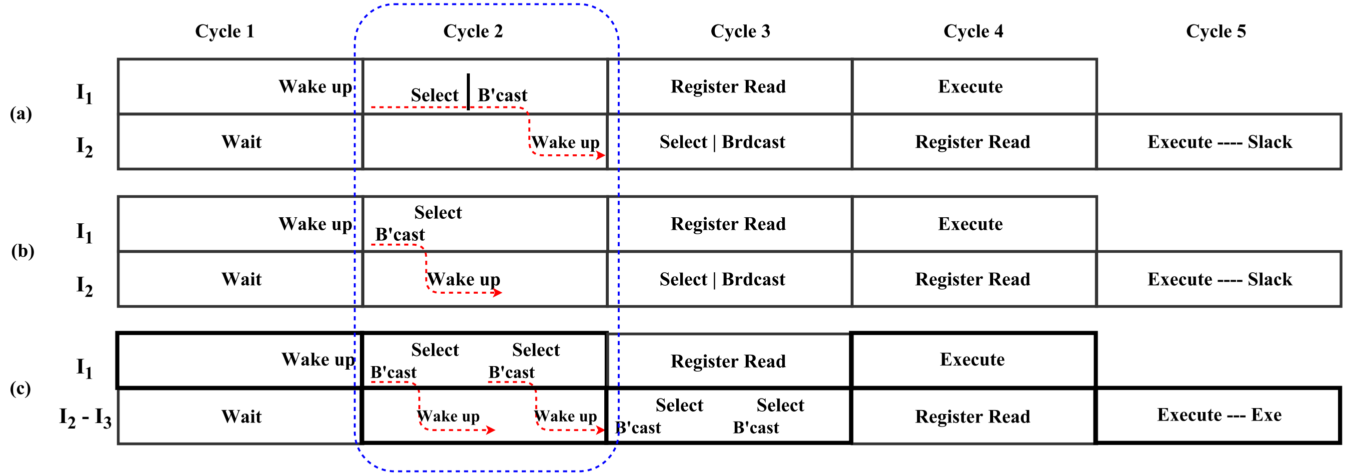


Figure 9: Timing Diagram of Execution Pipeline

Conventional Scheduling: Fig.9.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 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.

Select-Free Scheduling: Select-Free Scheduling [8] is explained via Fig.9.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 [57]. Thus, in select-free scheduling, the broadcast-wakeup path can complete in roughly half a cycle, while the selection logic works in parallel. 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.

Double Scheduling for Slack Recycling: Double Scheduling is shown in Fig.9.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.9.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.

In the context of the earlier example discussed in Fig.5.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.1a. Scheduling can be made even more aggressive by additional techniques such as speculative wake-up [68] and circuit optimizations such as resizing buffers and clustering [47], which are not explored here.

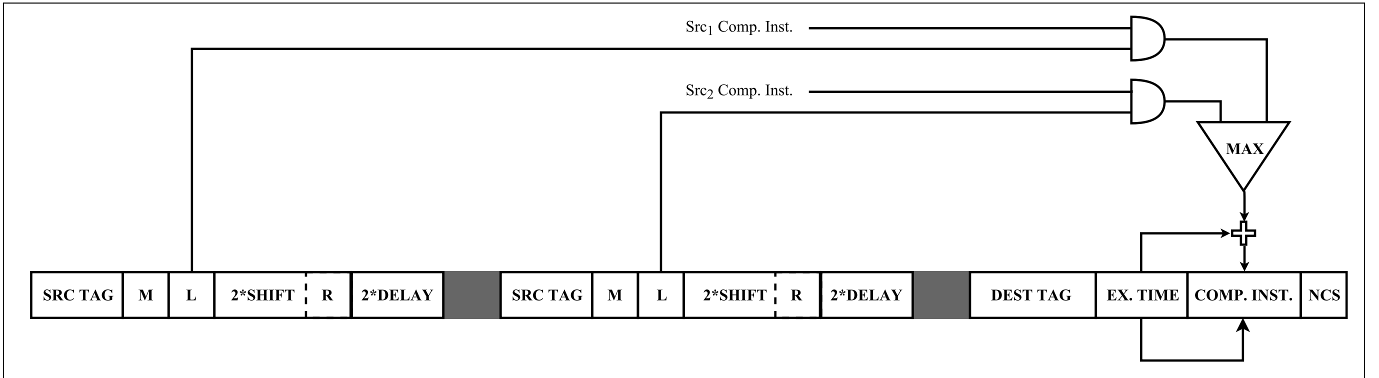


Figure 10: RSE modified for slack-aware scheduling

Slack-Aware Scheduling: Our design goal is augmenting this baseline design with slack-awareness. The resultant optimized RSE entry is shown in Fig.10. 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. When the entry is ready for (speculative) execution i.e. when the R-bits are set, the completion time of the current entry is calculated and stored into the COMP-INST field. The value is dependent on the last completing parent instruction and the execution time of the particular instruction. The computation logic is shown in the figure. Further details can be found in the ReDSOC paper.

4 Proposed Research

The goal of this research is to extend our prior work on analysing different forms of compute time variation and slack recycling designs, into new domains. Our work primarily focuses on the approximate computing domain but also extends to task graph scheduling, as discussed below.

4.1 Accelerating Approximate Programs via Aggressive Slack Recycling

The mainstream adoption of approximate computing, and its use in a broader range of applications, are predicated upon the creation of programmable platforms for approximate computing [73]. A key requirement for efficient approximate computing is an amalgamation of (i) programs that can naturally expose inherent application resilience, and (ii) hardware designs flexible enough to leverage the appropriate amount of approximation that the applications engender. While there have been multiple proposals in the software space to unearth opportunities to leverage

Prior Proposal	Type	Substrate	H/W Description	Prim. Benefit
Truffle [20]	Compute	GPP	Duplicated h/w at low vdd	Power
Quality Prog. Vector Processors [73]	Compute	Vector Processor	Bit-level precision control (power)	Power
Scalable Effort Hardware Design [13]	Compute	SVM accelerator	Low precision MAC, voltage reduction	Power
Dynamic Bit-Width Adaptation [59]	Compute	DCT accelerator	Bit widths per frequency components	Power
Soft Digital Signal Processing [30]	Compute	DSP	Voltage reduction	Power
Stripes [39]	Compute	NN accelerator	Per-layer precision, Bit-serial compute	T'put, Power
Concise Loads and Stores [36]	Memory	GPP	concise loads/stores, accurate compute	Power, Perf.
Load Value Approximation [50]	Memory	GPP	Use approximate value on cache misses	Power, Perf.
The Bunker Cache [48]	Memory	GPP	Proximate addresses use similar values	Power, Perf.
Doppelgnger [49]	Memory	GPP	Similarity between data elements	Power, Area

Table 1: Prior proposals and their primary benefits

approximation for speedup or efficiency improvements, we believe that there is much scope for improvement in hardware design. Our proposal is motivated by multiple limitations in current approximate hardware (with specific focus on timing approximation), which are listed below and then discussed in more detail further in this section.

1. First order benefit is power and not performance.
2. Require dedicated approximate and accurate resources.
3. Lack temporal control at fine granularities.
4. Ignore inherent precision and environmental effects.
5. Unable to support multiple approximation levels.

4.1.1 First order benefit is power and not performance

Challenge: While proposals for approximate compute hardware are numerous, most of the primary benefits from prior proposals is power savings. Power savings are usually achieved via power gating portions of the compute units in standard approximate hardware [73, 59] or via provision of additional low voltage rails in timing approximate hardware [20, 30, 13]. While saving power is an important goal, so is performance speedup - accelerating approximate computations can, for example, help meet QoS requirements.

Table.1 lists multiple prior proposals implement on varying substrates along with their primary benefits. Note, it is necessary to point out the multiple works related to approximation within memory provide both power savings and performance speedup from reducing cache misses and DRAM accesses etc. [50, 49, 48, 36], but our focus is on approximate compute alone.

While some of these proposals can increase frequency (eg. via DVFS) in order to improve performance, frequency scaling is applicable only across the entire compute hardware and not specific to the approximate hardware blocks and thus have limited potential with a strict power budget. It should be noted that GALS (Globally Asynchronous Locally Synchronous) solutions can be applicable but prior studies have shown they have multiple limitations [32].

Proposal: Our work, instead piggy-backs approximate computing on top of slack recycling. As discussed in prior sections, our slack recycling proposals can accelerate sequences of operations with no scaling of voltage/frequency. Meaning that there is potential for performance speedup as and when sufficient slack exists. While our prior

works were limited by timing error recovery overheads (LAGS) or only recycled accurate-compute slack (ReD-SOC), our current proposal includes approximation slack, which is the extra slack available when timing errors are acceptable in approximate applications. Fig.11 illustrates the total slack available for an example computation - broken down into PVT slack, data slack and approximate slack. The fraction of approximate slack varies depending on the amount of approximation that can be borne by the computation. The greater the slack, the more potential for speedup.

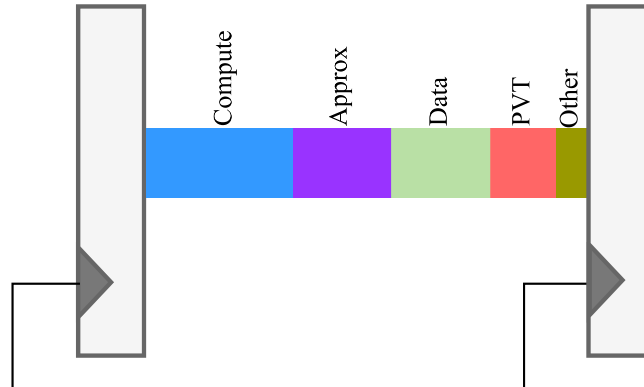


Figure 11: Slack breakdown

4.1.2 Require dedicated approximate and accurate resources

Challenge: Most applications amenable to energy-accuracy trade-offs (e.g., vision, machine learning, data analysis, games, etc.) have approximate components, where energy savings are possible, and precise components, whose correctness is critical for application invariants [20]. These applications are almost always composed of a fine-grained inter-mingling of both accurate and approximate operations.

For example, consider software level approximation provided by DECAF [3], a type-based approach to controlling quality in approximate programs. DECAF's goal is to let programmers specify important quality constraints while leaving the details to the compiler. A code snippet for Euclidean-distance is shown below.

The programmer can specify only the reliability of the output (here, the return value). For other values, where the right reliability levels are less obvious, the programmer can leave the probability inferred. The programmer decides only which variables may tolerate some degree of approximation and which must remain fully reliable. The programmer may write, for example, @Approx(0.9) float for the return type to specify that the computed value should have at least a 90% probability of being correct. The intermediate value *d* and the accumulator *total* can be given the un-parameterized type @Approx float to have its reliability inferred. And the loop induction variable *i* can be left reliable to avoid compromising control flow.

```
@Approx(0.9) float distance(float[] v1, float[] v2) {
  @Approx float total = 0.0;
  for (int i = 0; i < v1.length; ++i) {
    @Approx float d = v1[i] - v2[i];
    total += d * d;
  }
  return sqrt(check(total));
}
```

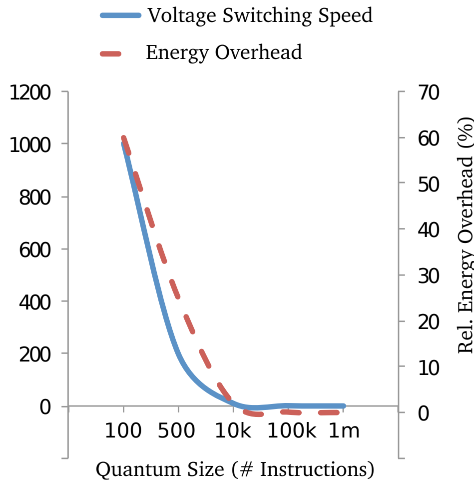
Listing 1: Dynamic approximation analysis by DECAF [3]

From the code snippet above, it is evident that approximate code, more often than not, would contain a mixture of accurate and approximate operations. In light of this potential fine-grained mix of accurate and approximate operations, prior solutions broadly fall into multiple categories: ① They can either perform voltage lowering only for large approximate blocks (with only approximate operations and no accurate operations) which has very limited potential due to lack of such blocks. ② Others employ specific approximate hardware resources, in order to be able to perform specific operations approximately. Proposals such as [13], perform approximate compute only for operations such as multiply-and-accumulate. The obvious limitation is that there might be multiple other operations worthy of approximation which cannot be approximated and further, the approximate hardware can go under-utilized if those specific operations are few in number. ③ More general-purpose solutions are suited to wider range of approximate operations. For instance, Truffle [20] duplicates general purpose functional units and runs one set at lower voltage. But such solutions have limitations as well. Thy incur significant design overheads - extra resources, multiple voltage rails, scheduling logic overheads. Further, resource specialization (i.e. separate dedicated resources to approximate and accurate compute) means under-utilization of one type of resource during sparse code regions of its particular computation type.

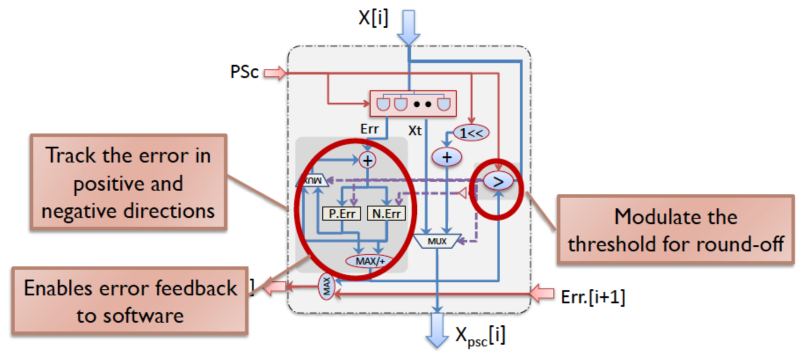
Proposal: In our proposal, each functional unit can work as accurate compute or as approximate compute. In the latter scenario, the FU can only exploit slack from PVT and data-type (as in our prior LAGS and ReDSOC proposals) but in the latter scenario, it can estimate slack more aggressively estimating for the required level of approximation as well (as was shown earlier in Fig.11). This allows flexible execution in regions of fine-grained inter-mingling of approximate and accurate computations - both compute forms can be executed without additional type constraints. (Note: the transparent data flow mechanism, as described in prior sections, does require less than peak utilization.)

4.1.3 Lack temporal control at fine granularities

Challenge: Multiple prior works performing some dynamic form of approximate compute employ a feedback mechanism [73]. Analysis of outputs (via comparison against some form of a golden standard) is used to correct the among of approximation applied at the approximate compute. One such instance is shown in Fig.12.b where feedback is used to optimize precision scaling. In the case of timing approximation hardware (via voltage/frequency scaling) the time period of this feedback loop can be limited by granularity at which voltage or frequency can be changed. Overheads from voltage switching can become pretty significant at finer temporal granularities, as illustrated in Fig.12.a. For example, even if a canary output [41] obtained in say, 1us, is sufficient to judge the quality of the approximation, the voltage can be scaled only at coarse temporal granularity (eg. 1 ms) in order to avoid high overhead fine-grained voltage regulators. This can result in poor temporal adaptability of feedback based timing approximation hardware.



(a) Voltage Switching Overheads



(b) Precision Scaling with feedback [73]

Figure 12: Fine grained temporal control

Proposal: On the other hand, our proposal can perform timing approximation at very fine temporal granularities, due to the absence of voltage and/or frequency scaling. A feedback mechanism, similar to that employed in LAGS, only needs to add a correction factor to the slack estimation mechanism. This feedback makes the aggressive or conservative correction to the slack estimation, and is immediately reflected in the approximate computation - affecting speed/accuracy QOS accordingly.

4.1.4 Ignore inherent precision and environmental effects

Challenge: Traditional timing approximation solutions also lack the ability to adapt to local characteristics such as random PVT variations and inherent operation precision/data-type/data-width. When voltage scaling is applied in prior timing approximation hardware, the voltage is set according to average error rates across phases of execution. As discussed in our prior proposal LAGS, random and local variations are ineffectively tracked by such mechanisms. Similarly, as discussed in ReDSOC, data slack varies widely across compute operations. This means that setting a constant low voltage for all approximate computations has limitations, because the actual effect of the approximation is dependent on the data slack and PVT slack experienced by the computation. This leads to undisciplined approximation across the approximate operations.

Proposal: In our work, approximate slack is applied atop other forms of slack. Slack is modelled separately for separate compute units. Once PVT slack and operand/operator based data slack is computed, the approximate slack for that compute unit (depending on the level of approximation) is added. Thus, multiple operations on the same compute unit, with the same level of approximation might have different total slack depending on the operands.

4.1.5 Unable to support multiple approximation levels

Challenge: In practice, no computation can tolerate an unbounded accumulation of soft errors to execute acceptably, even the approximate regions must execute correctly with some minimum reliability. Recent software advancements in approximate computing [9, 3] have made significant improvements over existing approaches, which supported only a binary distinction between critical and approximate regions. Such quantitative approximability proposals, both at application [9, 3] and ISA [73] levels allow for fluid precision or reliability control.

In the DECAF based code snippet shown earlier, not only are their approximate and accurate operations, but the level of approximability (or reliability) of operations can be different as well. For instance, in that example, the reliability for the accumulator *total* will be different from that for *d* and the function output. Similarly, in the presence of nested loops and multiple variables used within these loops or elsewhere, multiple approximation levels might exist. Prior analysis has shown breakdown of different reliability levels of operations within standard approximate applications and a few are shown in Fig.13 (as obtained from prior work [3]).

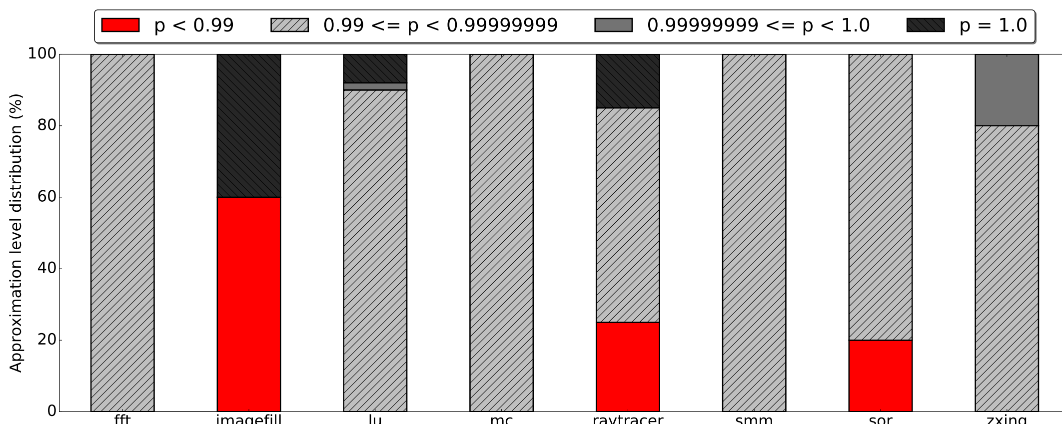


Figure 13: Approximate operation probabilities on an ideal continuous machine [3]

Thus approximate programs often will consist of accurate variables and multiple levels of approximation - and the capacity to handle multiple approximate levels is important for hitting the sweet-spot between accuracy and energy efficiency. Despite these developments for fluid precision control at the higher abstraction layers, current approximate hardware proposals are often inefficient to take advantage of it - especially in the case of timing approximate hardware.

Proposal: In our work, multiple levels of approximation can be integrated seamlessly. In the presence of software and ISA support, each instruction carries information on the level of approximation (if any) that it requires. This is interpreted at the time of slack estimation for the operation. Similar to the assignment of data slack based on opcode, operand widths etc, the approximation slack is assigned based on this approximation level.

4.1.6 Proposal

Different highlights of our proposal were discussed previously as solutions to challenges/limitations of current timing approximate hardware. The design itself is summarized below.

1. Simple transparent latch based execution modules with synchronous control are integrated seamlessly into standard synchronous pipelines.

2. Application-level approximability is translated into operation-level approximability, to allow fine-grained control of approximation.

3. Potential slack for each executing operation is identified - accounting for PVT Slack (from PVT variations), Data Slack (from operation type, and operand data widths/types) and Approximate Slack (from amount of approximation the computation can handle, if any).

4. Transparent latching with accurate slack estimation allows slack from a producer operation to be cut out (or recycled) by starting the execution of dependent consumer operations at the exact instant of completion of the producer operation, if possible.
5. Speedup is obtained by accelerated execution of potential operation sequences (atop standard functional units) in this manner, rather than increasing frequency or decreasing voltage.
6. We expect to study our proposals benefits across different error-tolerant applications and across standard OOO and spatial substrates, as well as specialized architectures such as neural network accelerators.

4.2 Secondary proposals

Below, we briefly describe other related proposals which we will explore.

4.2.1 Selective approximation for effective acceleration

Our primary proposal (above) has motivated the use of timing approximation for program acceleration. With effective error modeling, different levels of approximation can be assigned to each variable and the output of the application deteriorates accordingly. But approximation might not always lead to performance speedup. Only those sequences which are on the critical path of the program, when accelerated (via approximation) will speedup the application.

Prior work has studied the identification of critical program slices, instrumental to application performance [23] as well as the identification of program slices which have architectural slack (not circuit slack - i.e. sequences which when delayed, do not affect program execution time) [22]. Avoiding timing approximation (or reducing approximation levels) of these sequences will not impact program speedup but will improve application accuracy. Alternatively, additional approximation can instead be performed on slices which are on the critical path. We seek to explore the designing of dynamic mechanisms (heuristic based, history based, or otherwise) which can control the approximation appropriately.

4.2.2 Variation aware task scheduling

PVT variation can speedup or slow down different parts of a chip - its effects were explored in LAGS, one of our prior proposals. In a spatial architecture with a wide spread of compute nodes, PVT variations introduces heterogeneity even within a homogeneous cluster. For example, if the architecture works on one single global voltage rail, different compute nodes will run at different peak frequencies depending on the proximate effects of PVT variations.

Prior work has studies the effects of CMPs under PVT variations - with different cores functioning at different voltage-frequency levels [69]. These CMP proposals have explored mapping threads according to variation characteristics [51], migrating voltage-violent threads to temperature-mild cores [76], lock/barrier management to avoid voltage emergencies [52] etc.

We propose to study variation affected spatial architectures in the context of task scheduling. We will explore the potential of variation-aware scheduling techniques which will schedule critical tasks on ”better” compute nodes (in the context of variation). Our work also seeks to analyze the variation-aware scheduling under other constraints such as communication overheads, thermal hotspots, potential for effective gating [62, 54] etc.

Further, we seek to implement temporal awareness in task scheduling. While spatial awareness (eg. spreading out vs co-locating tasks) has been reasonably explored prior, temporal awareness has not. Temporal awareness refers to appropriately spacing out task scheduling over time rather than taking a greedy approach of maximum parallelism. Temporal awareness has potential for efficient execution because the temporal spreading out of tasks will cause lower temperatures and lesser/lower voltage drops.

5 Conclusion

Allowing computation to be approximate can lead to significant energy savings because it alleviates the correctness tax imposed by the wide safety margins on typical designs. However, the mainstream adoption of approximate computing, and its use in a broader range of applications, are predicated upon the creation of programmable platforms for approximate computing. While there have been multiple proposals in the software space to unearth opportunities to leverage approximability for speedup or efficiency improvements, we believe that there is much scope for improvement in hardware design.

Our primary contribution is effective in avoiding such limitations, by developing a disciplined but aggressive fine-grained slack recycling mechanism suited for approximate computing, resulting in improved performance and efficiency. Converting tolerable approximation into a slack component allows the use of LAGS/ReDSOC slack recycling techniques to accelerate sequences of operations of any length. Simple transparent latch based execution modules with synchronous control are integrated seamlessly into standard synchronous pipelines. Transparent latching with accurate slack estimation allows slack from a producer operation to be cut out (or recycled) by starting the execution of dependent consumer operations at the exact instant of completion of the producer operation, if possible. We expect to study our proposals benefits across different error-tolerant applications and across standard OOO and spatial substrates, as well as specialized architectures such as neural network accelerators.

For secondary contributions, first, we propose to explore dynamic identification of appropriate program slices to approximate, as this is important to maximizing performance speedup at best possible accuracy. Second, we propose exploration of PVT-variation aware task graph scheduling - spatial variation aware scheduling will schedule critical tasks on better compute nodes and temporal variation aware scheduling will appropriately space out task scheduling over time rather than taking a greedy approach of maximum parallelism.

6 Schedule

The following table highlights future publication plans and expected date of dissertation.

Focus of Research	To be Published in	Tentative Date
Accelerating Approximate Programs via Slack Recycling (Part 1)	ISCA	November 2017
Accelerating Approximate Programs via Slack Recycling (Part 2)	MICRO	April 2018
Selective Approximation for Effective Acceleration	HPCA	August 2018
Variation Based Task Scheduling	ISCA	November 2018
Ph.D. Thesis	UW - Madison	March 2019

References

[1] T. Austin, V. Bertacco, D. Blaauw, and T. 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.

[2] 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.

[3] B. Boston, A. Sampson, D. Grossman, and L. Ceze. Probability type inference for flexible approximate programming. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 470–487, New York, NY, USA, 2015. ACM.

[4] K. A. Bowman, S. G. Duvall, and J. D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, Feb 2002.

[5] A. Bracy, P. Prahlad, and A. 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.

[6] D. Brooks and M. 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.

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

[8] M. D. Brown, J. Stark, and Y. 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.

[9] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 33–52, New York, NY, USA, 2013. ACM.

[10] J. Carmona, J. Cortadella, M. Kishinevsky, and A. Taubin. Elastic circuits. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(10):1437–1455, Oct. 2009.

[11] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(8):857–873, Aug 2004.

[12] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[13] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Design Automation Conference*, pages 555–560, June 2010.

[14] M. Choudhury, V. Chandra, K. Mohanram, and R. Aitken. Timber: Time borrowing and error relaying for online timing error resilience. In *DATE '10*, pages 1554–1559, 2010.

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

[16] J. Cortadella, L. Lavagno, P. López, M. Lupon, A. Moreno, A. Roca, and S. S. Sapatnekar. Reactive clocks with variability-tracking jitter. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 511–518. IEEE, 2015.

[17] O. Ergin, D. Balkan, K. Ghose, and D. 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.

[18] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO 36*, pages 7–, 2003.

[19] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[20] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312, New York, NY, USA, 2012. ACM.

[21] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[22] B. Fields, R. Bodík, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 47–58, Washington, DC, USA, 2002. IEEE Computer Society.

[23] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 74–85, New York, NY, USA, 2001. ACM.

[24] S. B. Furber, D. A. Edwards, and J. D. Garside. Amulet3: a 100 mips asynchronous embedded processor. In *Proceedings 2000 International Conference on Computer Design*, pages 329–334, 2000.

[25] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, Sept 2012.

[26] B. Greskamp and J. 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.

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

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

[29] M. S. Gupta, J. A. Rivers, P. Bose, G.-Y. Wei, and D. Brooks. Tribeca: Design for pvt variations with local recovery and fine-grained adaptation. In *MICRO 42*, pages 435–446, 2009.

[30] R. Hegde and N. R. Shanbhag. Soft digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(6):813–823, Dec 2001.

[31] E. L. Hill and M. H. Lipasti. Stall cycle redistribution in a transparent fetch pipeline. In *ISLPED'06 Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, pages 31–36, Oct 2006.

[32] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 158–168, Washington, DC, USA, 2002. IEEE Computer Society.

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

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

[35] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers. Synchronous interlocked pipelines. In *Asynchronous Circuits and Systems, 2002. Proceedings. Eighth International Symposium on*, pages 3–12. IEEE, 2002.

[36] A. Jain, P. Hill, S. C. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[37] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. 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.

[38] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. 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.

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

[40] S. K. Khatamifard, M. Resch, N. S. Kim, and U. R. Karpuzcu. Varius-tc: A modular architecture-level model of parametric variation for thin-channel switches. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 654–661, Oct 2016.

[41] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang. Input responsiveness: Using canary inputs to dynamically steer approximation. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 161–176, New York, NY, USA, 2016. ACM.

[42] C. R. Lefurgy, A. J. Drake, M. S. Floyd, M. S. Allen-Ware, B. Brock, J. A. Tierno, and J. B. Carter. Active management of timing guardband to save energy in power7. In *MICRO-44*, pages 1–11, 2011.

[43] X. Liang, G.-Y. Wei, and D. Brooks. Revival: A variation-tolerant architecture using voltage interpolation and variable latency. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 191–202, Washington, DC, USA, 2008. IEEE Computer Society.

[44] J. Liu, S. M. Nowick, and M. Seok. Soft mousetrap: A bundled-data asynchronous pipeline scheme tolerant to random variations at ultra-low supply voltages. In *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*, pages 1–7, May 2013.

[45] T. Liu and S.-L. Lu. Performance improvement with circuit-level speculation. In *Microarchitecture, 2000. MICRO-33. Proceedings. 33rd Annual IEEE/ACM International Symposium on*, pages 348–355. IEEE, 2000.

[46] G. 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.

[47] P. Michaud, A. Mondelli, and A. 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, Aug. 2015.

[48] J. S. Miguel, J. Albericio, N. E. Jerger, and A. Jaleel. The bunker cache for spatio-value approximation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[49] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. DoppelgÄnger: A cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 50–61, New York, NY, USA, 2015. ACM.

[50] J. S. Miguel, M. Badr, and N. E. Jerger. Load value approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 127–139, Washington, DC, USA, 2014. IEEE Computer Society.

[51] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012.

[52] T. N. Miller, R. Thomas, X. Pan, and R. Teodorescu. Vrsync: Characterizing and eliminating synchronization-induced voltage emergencies in many-core processors. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 249–260, June 2012.

[53] T. Nowatzki, V. Gangadhan, K. Sankaralingam, and G. Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 27–39, March 2016.

[54] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 495–506, New York, NY, USA, 2013. ACM.

[55] S. M. Nowick and M. Singh. High-performance asynchronous pipelines: An overview. *IEEE Design Test of Computers*, 28(5):8–22, Sept 2011.

[56] S. M. Nowick, K. Y. Yun, P. A. Beerel, and A. E. Dooply. Speculative completion for the design of high-performance asynchronous dynamic adders. In *Advanced Research in Asynchronous Circuits and Systems, 1997. Proceedings., Third International Symposium on*, pages 210–223. IEEE, 1997.

[57] S. Palacharla, N. P. Jouppi, and J. E. Smith. *Complexity-effective superscalar processors*, volume 25. ACM, 1997.

[58] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM.

[59] J. Park, J. H. Choi, and K. Roy. Dynamic bit-width adaptation in dct: An approach to trade off image quality and computation energy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(5):787–793, May 2010.

[60] Y. Park, H. Park, and S. 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.

[61] A. Perais, A. Seznec, P. Michaud, A. Sembrant, and E. 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.

[62] G. S. Ravi and M. H. Lipasti. Charstar: Clock hierarchy aware resource scaling in tiled architectures. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 147–160, New York, NY, USA, 2017. ACM.

[63] D. Sager, D. P. Group, and I. Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 1:2001, 2001.

[64] 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.

[65] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. *ACM Trans. Archit. Code Optim.*, 1(1):62–93, Mar. 2004.

[66] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas. Eval: Utilizing processors with variation-induced timing errors. In *MICRO 41*, pages 423–434, 2008.

[67] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. *IEEE Transactions on Semiconductor Manufacturing*, 21(1):3–13, Feb 2008.

[68] J. Stark, M. D. Brown, and Y. 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.

[69] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 363–374, Washington, DC, USA, 2008. IEEE Computer Society.

[70] A. Tiwari, S. R. Sarangi, and J. Torrellas. Recycle:: Pipeline adaptation to tolerate process variation. In *ISCA '07*, pages 323–334, 2007.

[71] O. S. Unsal, J. W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. *IEEE Micro*, 26(6):30–39, Nov 2006.

[72] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[73] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 1–12, New York, NY, USA, 2013. ACM.

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

[75] J. Xiong, V. Zolotov, and L. He. Robust extraction of spatial correlation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(4):619–631, April 2007.

[76] G. Yan, X. Liang, Y. Han, and X. Li. Leveraging the core-level complementary effects of pvt variations to reduce timing emergencies in multi-core processors. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 485–496, New York, NY, USA, 2010. ACM.

[77] S. Yehia and O. 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.

[78] Y. Zhang, M. Khayatzadeh, K. Yang, M. Saligane, N. Pinckney, M. Alioto, D. Blaauw, and D. Sylvester. 8.8 irazor: 3-transistor current-based error detection and correction in an arm cortex-r4 processor. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 160–162, Jan 2016.