Lazy Pipelines: Enhancing Quality in Approximate Computing

G. Tziantzioulis[†], A. M. Gok[†], S. M. Faisal[‡], N. Hardavellas[†], S. Ogrenci-Memik[†], and S. Parthsarathy[‡]

[†]Department of Electrical Engineering and Computer Science
Northwestern University, Evanston, IL, USA
{georgios, amg}@u.northwestern.edu, {nikos, seda}@northwestern.edu

[‡]Department of Computer Science
The Ohio State University, Columbus, OH, USA
{faisal, srini}@cse.ohio-state.edu

***Abstract*—Approximate computing techniques based on Voltage Over-Scaling (VOS) can provide quadratic improvements in power efficiency. However, voltage scaling is limited by the inherent fault-tolerance of an application, thus preventing VOS schemes from realizing their full potential. To gain further power efficiency a reduction of the error rate experienced in a given voltage level is required. We propose Lazy Pipelines, a micro-architectural technique that utilizes vacant cycles in a VOS functional unit to extend execution and reduce the error rate.**

***Keywords* — approximate computing, power efficiency, micro-architecture**

I. Introduction

Power has become a first-class design constraint [9] for both high-end and mobile systems due to the breakdown of Dennard's scaling [2] and the advent of dark silicon [7]. Several techniques to reduce power have been proposed, including dynamic voltage and frequency scaling, near-threshold computing, and sleep states. However, the impact of these techniques is hampered by the traditional worst-case design approach: as technology scaling leads to probabilistic behavior in CMOS [1], designers add significant voltage and timing margins [10] to overcome the environmental and process variations of the worst-case scenario, no matter how improbable this scenario may be. The resulting highly-conservative guardbands reduce both performance and power efficiency.

One of the most prominent techniques for reducing the power consumption is voltage over-scaling (VOS), where the supply voltage (V_{dd}) of a component is reduced below the safety margin in an attempt to receive quadratic power savings ($P_{dyn.} = C \times V_{dd}^2 \times f$). In contrast to initial VOS schemes with error detection and correction [3], later research identified that allowing faults in the execution while providing acceptable output quality (by limiting the errors to computations that can tolerate them [4]) would allow for voltage scaling well beyond what error-correcting techniques could afford. At the same time, functional units (FUs) in modern computer architectures can stay unoccupied. Tullsen *et al.* [12] quantify the amount of under-utilization and suggest simultaneous hardware multithreading to increase utilization and boost the system's instruction throughput.

Combining the above observations, we propose *Lazy Pipelines*, an architecture that exploits the unutilized execution cycles (i.e., slack) of FUs to improve computational accuracy in VOS approximate computing architectures. Lazy Pipelines utilize the slack of VOS FUs to prolong the computation and reduce the number of timing errors. Our analysis shows that by utilizing slack, we can substantially decrease the bit error rate (BER) in the results of approximate computations. This allows to extract further power benefits by enabling additional reduction of supply voltage, while maintaining the same level of timing errors. To the best of our knowledge, this is the first work that exploits the slack of FUs to improve the accuracy of approximate computations. More specifically, our contributions are:

- We propose Lazy Pipelines, a microarchitectural technique that harnesses multiple contiguous vacant execution cycles (slack) to reduce timing errors on VOS FUs.
- We analyze the impact of slack on reducing BER in a set of FUs covering integer, logic, and FP operations.
- We evaluate the impact of Lazy Pipelines through detailed cycle-accurate architectural simulations.

II. Related Work

There is a large body of work in detecting and correcting the timing errors that stem from VOS FUs. Ernst *et al.* [3] propose Razor, a technique for detecting and correcting timing errors in a VOS circuit. Lazy Pipelines is orthogonal to Razor and could be combined with it to reduce the number of cases where an operation needs to be repeated due to a timing error.

Our work is related to the scheme proposed by Esmaeilzadeh *et al.* [4], which is modified to exploit slack in execution to improve output quality while reducing power. The concept of prolonging the execution of an instruction to improve the quality of the result has been proposed in previous approximate computing studies. Xin *et al.* [14] observed that specific instructions are more likely to stress the circuit's critical paths and produce timing errors. To reduce the effect of re-executing a critical instruction, either an extra cycle is allocated on them, or they are replaced with less critical ones [8]; however, as full accuracy is preserved they can not be directly compared with Lazy Pipelines. Parallel to the work previously mentioned, researchers have proposed abstractions and programming models for instrumenting and confining imprecision to fault-tolerant computations [11].

III. Background

In this section we establish the needed background knowledge and set down assumptions we made to design our scheme.

A. Precision Marking

To implement approximate-computing-aware hardware, a mechanism to inform the hardware about the precision levels of individual operations is required. Sampson *et al.* [11] propose a programming language framework that facilitates the marking of data and computation with precision levels. The

978-3-9815370-6-2/DATE16/©2016 EDAA

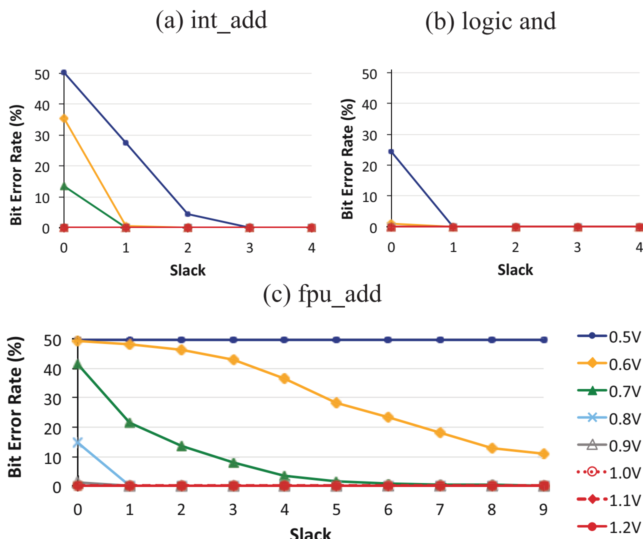


Fig. 1. BER as a function of slack and V_{dd} for integer and fp functional units.

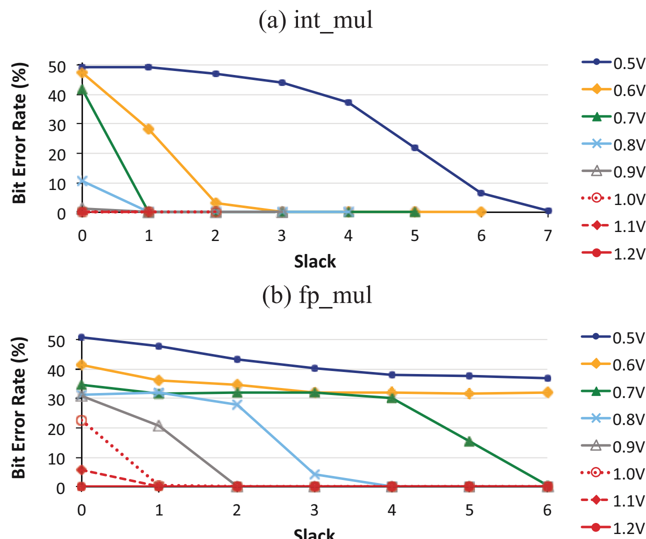


Fig. 2. BER as a function of slack and V_{dd} for integer and fp multipliers.

precision information is conveyed to the hardware by extending the ISA with imprecise variants of existing operations. However, adding new instructions can be difficult in a fixed instruction-length ISA (e.g., ARM) where existing operations occupy the majority of the available opcode space. Furthermore, including precision information in the bit encoding of an operation hinders scaling to multiple precision-levels; to support N levels, $log(N)$ bits per operation are required. This approach allows for fine-grain marking, but the decoder logic requires significant modifications and the opcode length increases and occupies significant bitwidth from the ISA, which may not be feasible for some ISAs.

A more flexible approach is to introduce a single marking instruction in the ISA that marks the precision level of the subsequent arithmetic instructions (see Listing 1). The arithmetic instructions that follow a marking instruction will be issued to the FUs at the specified precision level. Non-arithmetic instructions (e.g., ld, st) always execute accurately. This approach, which we adopt, requires minimal changes to the decoder and results in small code size increase when there are large regions of same-precision instructions.

B. Relation between V_{dd} and Delay

We perform analog simulations to analyze the relationship between slack and BER. Figure 1(a) presents the BER for a VOS integer adder and the impact of slack execution cycles after the nominal end. Figure 1(b) presents the same relation for the logic AND operation, and is representative of the

```
startImprecise ; precision_l=0b111 (0.5V)
mul   r8, ip, r8
add   r1, r2, r3
start Precise ; precision l=0b000 (1.2V)
cmp   r2, #16
startImprecise ; precision l=0b111 (0.5V)
lsl   r6, r8, #2
str   r6, [r3]
...
rsb   r6, r2, r3
startPrecise ; precision l=0b000 (1.2V)
add   r0, r0, #4
```

Listing 1. Assembly example with precision marking.

behavior of other logic operations (OR, XOR, MOVE; omitted due to space constrains). Figure 1(c) presents the same relation for the floating point add operation. We observe that integer operations eventually converge to correct results, but the FP-add converges much slower due to its complexity.

C. Functional Unit Design

The design of a FU has a direct impact on the slack vs. BER relationship. The most favorable FU design for Lazy Pipelines is a combinational circuit. In such a scheme the amount of available slack is directly applied to the whole execution, thus maximizing the gain. The integer ALU we model is a single-cycle combinational FU that belongs to this category. Another FU design amenable to Lazy Pipelines is a multi-stage pipelined FU without any feedback. Given enough slack, it will eventually be free of all timing errors. The FP adder unit we model belongs to this category.

Finally, the third kind of design includes pipelined FUs with feedback in the circuit, and thus the current output is a function of the previous output. The integer multiplier, FP multiplier and FP divider we model belong to this category. Because the current output is a function of the previous output, the output can only be read at the exact cycle it was supposed to be generated; the output is useless even one cycle later, even at nominal V_{dd}. This makes exploiting additional cycles non-trivial. However, circuit designs that perform these arithmetic operations without a feedback loop exist. While a comprehensive exploration of such circuits is beyond the scope of this paper, we model FUs in this category that have this property, by approximating the behavior using clock division (Figure 2).

In order to exploit slack, an FU requires the input operands to be held constant throughout the whole operation. Thus, the FUs we model employ input buffers for their operands. These buffers are written with new values only when a new operation is issued to the FU.

IV. Architecture

In this section we describe the modifications required to incorporate Lazy Pipelines in a typical micro-architecture.

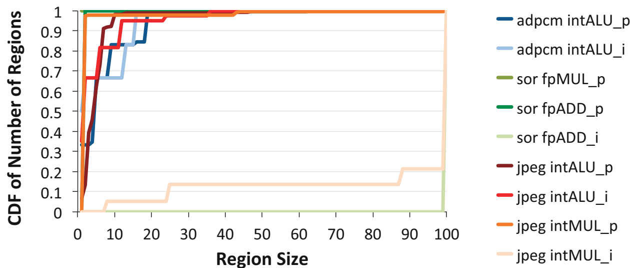


Fig. 3. Region size CDF. Legend key: <app> <FU>_<precise/imprecise i/p>

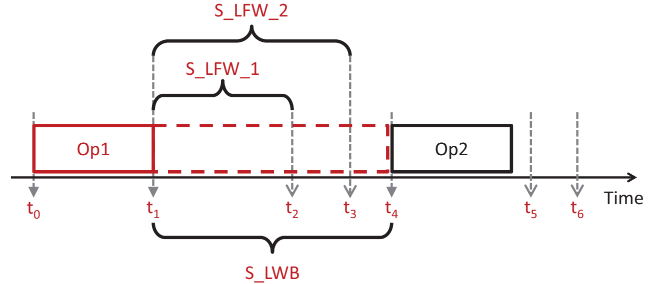


Fig. 4. Lazy Forwarding and Lazy Writeback.

A. Approximate Computing Architecture

Our base approximate computing architecture scheme resembles the one proposed by Esmaeilzadeh *et al.* [4]. For every FU in the baseline system, we add one more FU that runs at lower V_{dd} and executes imprecise computations. The user sets the precision level based on the quality requirements of the application, and the application upon loading sets the V_{dd} of the imprecise FUs to the corresponding level. The precision marking instructions in the code simply denote which of the two precision levels (precise, imprecise) will be used for subsequent arithmetic operations. In contrast to [4], the issue logic in our scheme differentiates between the precise and imprecise version of a FU and allows independent instruction issue and execute.

Having a set of FUs for each precision level is preferable over one set of adjustable-V_{dd} FUs because precise and imprecise computations are finely interleaved (Figure 3). As a result, if a single adjustable-V_{dd} FU is used, it will have to perform switches between precision levels much faster than it is possible to do. Switching and stabilizing the V_{dd} of a unit takes 10's to 100's of cycles [6], and would result in a significant performance degradation, eliminating any potential power benefits. For our work we assume two levels of precision for all FUs. However, this scheme can be extended to multiple levels.

B. Vacant Cycles and Lazy Pipelines

A Lazy Pipeline exploits the naturally occurring under-utilization of FUs to extend execution, thus reducing the BER in the result. Despite substantive effort to maximize FU utilization in modern processors (out-of-order execution, hyper-threading) there are still vacant cycles in FUs. The under-utilization further increases in architectures such as the one proposed by Esmaeilzadeh *et al.* [4], where additional FUs are introduced to allow for multiple precision levels. With Lazy Pipelines we present an extension of the out-of-order micro-architecture to exploit this phenomenon for improving output quality. This is achieved by allowing operations to continue execution after the nominal end cycle until another operation is issued to the occupied FU. This slack allows signals that missed timing to correctly propagate through the circuit, thus improving output correctness. Section VI discusses the relationship of slack vs. BER in more detail.

Lazy Pipelines do not change the issue time of an operations, but exploit the slack that naturally occurs during runtime due to dependencies and memory accesses. In the base architecture, the writeback (WB) occurs exactly after the nominal execution cycles have passed. In Lazy Pipelines, however, the WB of the output is delayed past the nominal execution time, until sufficient vacant cycles are utilized to obtain the correct result for that voltage level, or until another instruction is ready to issue to the same FU, whichever happens first. In both cases the operation is evicted. We call this delayed WB "*Lazy Write-back*" (LWB). LWB utilizes slack and probably results in lower BER. While the operation executes on the FU, it is possible that its output is required by another operation. In the base architecture, the value will be communicated through the forwarding logic. Forwarding can only occur after the nominal execution cycles have passed, but before the operation reaches the write-back stage. In Lazy Pipelines, forwarding can occur after nominal execution cycles have passed, until an eviction occurs, which is a larger window of time. If the output is used by another operation through forwarding, it will be calculated utilizing slack, thus having a probably lower BER than the base architecture. We call this improved forwarding method "*Lazy Forwarding*" (LFW). Lazy Pipelines implement support for these two techniques: Lazy Forwarding and Lazy Write-back.

If the FU is allowed to continue the execution of Operation 1 after t_1, later outputs will have lower BER. But in the base architecture, only the output at t_1 would be used by all future reads, resulting in higher BER. In Lazy Pipelines, reads at t_2 and t_3 will read the output values of the FU at t_2 and t_3 through the use of LFW. On the other hand, reads at t_5 and t_6 would read the value at t_4 through LWB. The slack utilized by Lazy Pipelines for LFW is indicated as S_LWF_1 for t_2 and S_LFW_2 for t_3 in Figure 4. All LWB cases will utilize the same slack, indicated as S_LWB.

The architectural changes required to support Lazy Pipelines are as follows:

1) *Decode Stage*: Since our precision marking is done through specific marking instructions, the decoder stage needs to keep track of the current precision level by having a *log(N)*-bits register (see Section III-A), and then pass this information to pipeline registers.

2) *Datapath Pipeline Registers*: They require an additional *log(N)* bits for precision level information, and 1 bit per imprecise FU to mask its Write Enable (see LWB below)

3) *Issue Logic*: Should support additional FUs with different precision levels. A 2-bit register per FU keeps track of the status of imprecise FUs (occupied, free, freeOnDemand).

4) *LWB*: It is possible that multiple FUs may want to write their outputs to registers at the same time, but this issue is already addressed in most modern architectures through the OoO execution. A LWB is not done immediately after the nominal execution cycles have passed, but delayed until an eviction occurs. This can be done by using a 1-bit WB masking signal per imprecise FU.

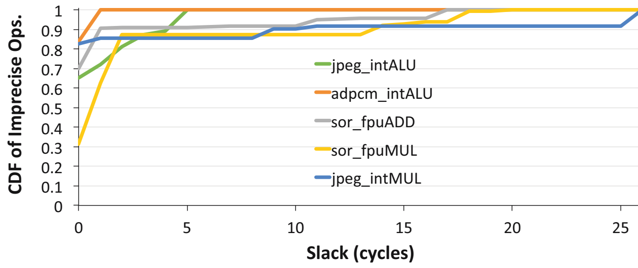


Fig. 5. CDF of slack after nominal end of execution for various FU types.

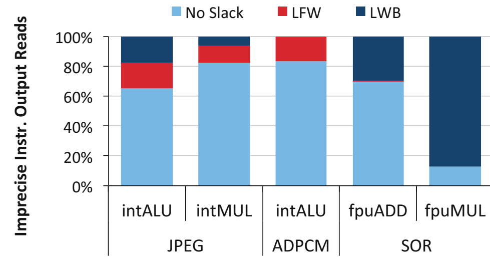


Fig. 6. Breakdown of imprecise instruction uses (i.e., reads of the imprecise instruction's output by another instruction).

5) *LFW*: Most modern architectures already come with forwarding mechanisms. The only changes required for LFW support are limited to the issue/select logic that keeps the states of the FUs. Thus, LFW can be requested (by issuing the appropriate control signals) from an occupied FU any time between its nominal execution end, and its eviction.

V. Methodology

We extract the relationship between slack and BER for different types of FUs through analog simulation. We fully synthesized the integer and floating point FUs of an OpenSPARC T1 core using Synopsys Design Compiler and the SAED 90nm standard cell library. We simulate all modules using Synopsys VCS and HSIM+ tools with the SPICE-level models of standard cells.

Using semi-randomly generated operand values we collect statistics for the BER-slack-power trade-off. Based on the collected data we create a simple model for computing the total power consumption for the execution of a Lazy operation. We estimate the total power consumption as follows: First, to compute the static power consumption we calculate the maximum static power consumption (P_s^{max}) of the FU assuming the entire circuit is idle. Based on that, the static power consumption for slack cycle n is given by $P_s(n) = P_s^{max} \times (1 - \mathrm{BER}(n-1))$ where $BER(n)$ is the BER at cycle n. This equation captures the contribution to the static power consumption coming from the parts of the circuit that are no longer switching. The dynamic power consumption is proportional to the switching activity of the circuit. If $P_d^{nominal}$ is the power consumption of the instruction at nominal V_{dd}, then the dynamic power consumption is computed based on the additional correct bits of each cycle, $P_d(n) = P_d^{nominal} \times [(1 - BER(n)) - (1 - BER(n-1))]$.

To evaluate the BER and power consumption in real workloads when using Lazy Pipelines, we extend the ISA with precision marking instructions and model the extended ISA in the GEM5 simulator, along with the functionality to support LFW and LWB. To emulate the errors in computations we incorporate in GEM5 a fault injection library that implements the b-HiVE error models [13]. We evaluate our design on a simulated single-core processor running at 2GHz with Fetch, Decode and Rename width of 3, Issue and Commit width of 8, IQ Entry size of 32 and ROB size of 40. To calculate the power consumption of the functional units in our proposed scheme and in the base architecture we use McPat v1.3. We model Low Stand-by Transistors (LSTP) at 32nm and 340 Kelvin, with a nominal voltage of 1.2V. We also evaluated designs with power-gated FUs, but power-gating offers negligible benefits due to the low static power of LSTP transistors. To obtain the final power values we post-process McPat's estimates through the BER-power-slack relationship described above. Extended execution is limited to the point that no additional quality benefit can be harvested; e.g., after one cycle for logic operations.

We evaluate Lazy Pipelines using JPEG and ADPCM from the mediabench benchmark suite and SOR from the scimark2 benchmark suite. All applications were compiled using GCC v4.7.3 at -O3 optimization level targeting the ARM ISA. After compilation, we manually identify and mark the imprecise regions in the assembly listing. Marking can be automated using a compiler without much overhead and we are currently developing a version of LLVM for that purpose.

VI. Experimental Results

A. Slack Distribution

To demonstrate the potential of utilizing Lazy Pipelines in approximate computing designs we collect for all imprecise instructions of our test applications the available slack that could be used to extend the execution of each instruction. Figure 5 presents the CDF of the available slack for each type of imprecise FU for each application. Our results show that for all three applications and utilized FUs there exist more than 10% of imprecise instructions that have at least 1 cycle of slack. Figure 6 collects all the imprecise instruction uses, i.e., the reads of the imprecise instruction's output by another instruction, and categorizes them based on whether the value was read through LFW, or LWB, or the value read was produced at exactly the nominal end of the producing imprecise instruction (i.e., used no slack). The figure indicates that the potential for LWB and LFW is largely application dependent.

B. Impact of LFW and LWB to Bit Error Rate

Through simulations using our modified cycle-accurate simulator (Section V) we quantify the BER reduction that slack utilization provides for a range of different voltage levels. Figures 7, 8, and 9 present the BER for operations that could benefit from slack (i.e., operations with zero slack at Figure 6 are excluded). Figure 7 presents the BER for the integer ALU

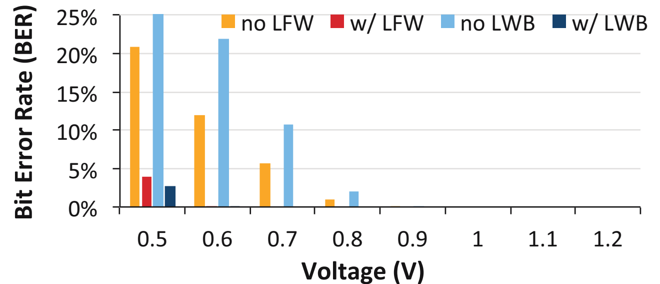


Fig. 7. BER reduction due to LFW and LWB in intALU operations (JPEG).

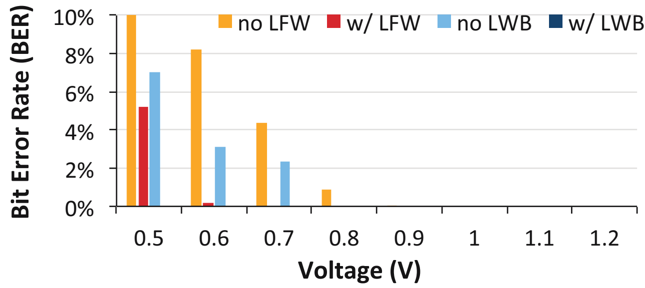


Fig. 8. BER reduction due to LFW and LWB in intALU operations (ADPCM)

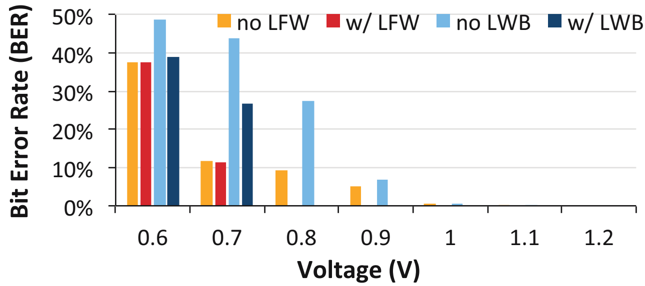


Fig. 9. BER reduction due to LFW and LWB in fpuADD operations (SOR).

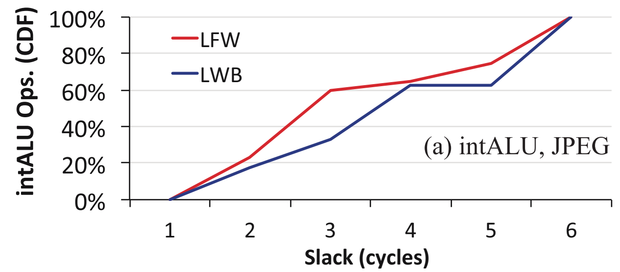
operations of the JPEG application. In the base approximate computing architecture we see that errors occur even up to a supply voltage of 1V. However, utilizing the available slack through Lazy Pipelines achieves a 0% BER for the affected operations at a supply voltage as low as 0.7V. Moreover, even at voltage levels below 0.7V the Lazy architecture provides significant BER reduction.

A similar behavior is observed for the ADPCM imprecise integer operations that utilize slack (see Figure 8). However, as the amount of slack is smaller, we observe smaller BER reductions than the ones observed at JPEG. Finally, Figure 9 presents the BER reduction for SOR's floating point additions.

In all figures above we see that LFW provides consistently higher BER reduction compared to LWB. This counter-intuitive result can be understood by examining the CDF of the slack for each one of the two techniques. In Figure 10(a) and (b) we observe that LFW predominantly occurs in operations that exhibit a larger slack (i.e., more vacant cycles), whereas LWB occurs in cases where the operation is evicted from the FU shortly after the nominal end of execution.

C. Overall Quality Improvement

The cumulative benefit (reduction of BER) from executing on a processor with Lazy Pipelines is presented in Figure 11. Figure 11(a) shows the overall BER of operations executed in the integer ALU for the JPEG application. The Lazy architecture consistently provides a BER reduction of at least 10%, and as high as 41% across all voltages. This provides a direct comparison against Truffle [4], as their design's output quality would be the same as our base approximate architecture's. Figure 11(b) shows the benefits for the imprecise integer ALU operations of the ADPCM application. We observe that Lazy Pipelines can provide substantial improvements only in the lowest range of voltages. This is due to the nature of the application: ADPCM has only a small amount and a low number of vacant cycles that its imprecise operations can utilize. More



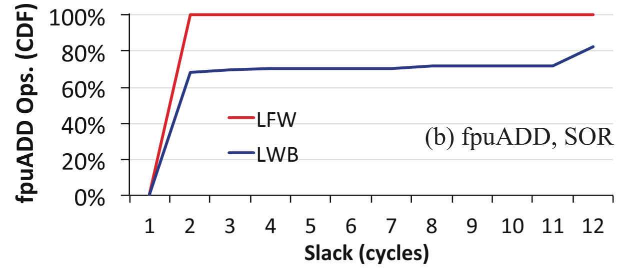
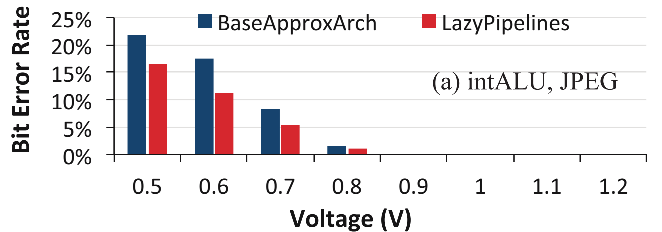
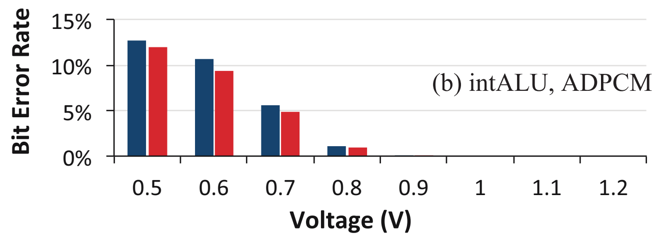


Fig. 10. CDF of slack after the nominal end of execution of (a) integer ALU (JPEG), and (b) FP add (SOR).

specifically, the reduced benefits come from the high number of similar and inter-dependent imprecise operations of the application. These cause a large number of dependent instructions to execute back-to-back, thereby preventing the use of slack.

Overall, the impact of Lazy Pipelines in the quality of an application depends on the granularity of interleaving between precision levels and the level of instruction level parallelism.





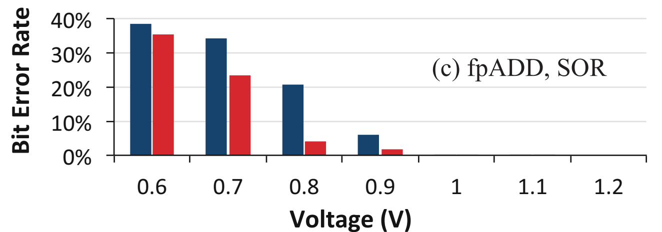


Fig. 11. Overall improvement in BER of (a) integer ALU for JPEG, (b) integer ALU for ADPCM, and (c) FP add for SOR.

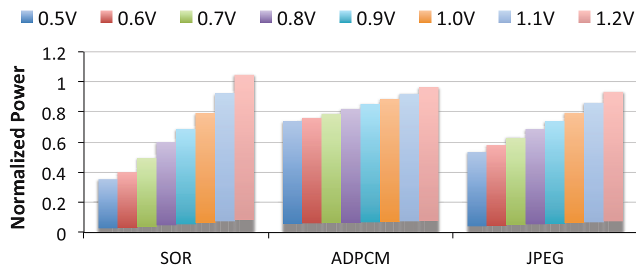


Fig. 12. Power consumption of Lazy Pipelines at various V_{dd} against a conventional (precise) architecture for ADPCM, SOR, and JPEG (idct kernel).

This is because higher interleaving of precision levels spreads the utilization across functional units and increases slack.

While we do not employ this in our work, we believe that there is the potential for the processor to achieve even higher accuracy for a fixed power budget: it may be possible to speculatively prolong the execution of operations that exhibit zero slack and delay the subsequent instructions with minimal impact on the performance of the processor. We base this belief on the observation of Fields *et al*. in [5] that 75% of dynamic instructions can be delayed by five or more cycles with no impact on program execution time.

D. Power Consumption

Lazy Pipelines exhibit a similar level of dynamic power consumption as Truffle [4]. Both techniques hold the input to the imprecise FU stable while its precise counterpart is executing a different instruction, and vice versa. However, as we demonstrated, the additional cycles of execution (while maintaining the input stable) allow for the progressive correction of timing errors, thus the circuit switches and consumes dynamic power. As both techniques utilize FUs in a similar manner, they will encounter an instruction eviction after approximately the same number of cycles and their dynamic power consumption is similar. Their static power consumption is respectively similar. Overall, Lazy Pipelines can provide the power benefits of an approximate computing architecture while providing significantly more accurate results. Figure 12 presents the normalized power consumption of FUs in Lazy Pipelines over a conventional (fully-precise) architecture.

VII. Conclusion

Approximate computing techniques that utilize voltage over-scaling can significantly reduce the processor's power consumption. Hence, expanding their application can help tackle the "Power Wall" that current designs face. We have shown that prolonging the execution of operations in voltage over-scaled functional units can result in a significant reduction of the operations' bit error rate and improve the accuracy of approximate computations. We demonstrate that this can be achieved by utilizing the vacant execution cycles (slack) of functional units. The amount of available slack is application- and microarchitecture-specific. The available slack is higher in simple architectures with a small super-scalar width and no hardware multi-threading, and it increases as the number of functional units increases. Processor architectures that target low-power design are typically based on simple RISC processors with relatively narrower superscalar width and weaker out-of-order execution than their high-performance counterparts. Thus, such designs harness less of the available instruction-level parallelism and lead to lower utilization of the available functional units. These architectures are good matches to be coupled with Lazy Pipelines and can further improve their power efficiency by utilizing the available slack to operate a subset of their functional units at a lower supply voltage. Compared to a state-of-the-art approximate architecture (Truffle [4]), Lazy Pipelines exhibit similar power savings, but can provide up to 41% lower bit error rate, leading to significant quality improvement.

VIII. Acknowledgements

This work is partially supported by NSF award CCF-1218768, NSF CAREER award CCF-1453853, and the Intel Parallel Computing Center at Northwestern.

References

[1] A. Asenov, G. Slavcheva, A. Brown, J. Davies, and S. Saini. Increase in the random dopant induced threshold fluctuations and lowering in sub-100 nm mosfets due to quantum effects: a 3-d density-gradient simulation study. *IEEE Transactions on Electron Devices*, 48(4):722- 729, Apr 2001.

[2] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted mosfet s with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256- 268, Oct 1974.

[3] 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 *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[5] B. Fields, R. Bodik, and M. Hill. Slack: maximizing performance under technological constraints. In *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 47- 58, 2002.

[6] W. Godycki, C. Torng, I. Bukreyev, A. B. Apsel, and C. Batten. Enabling realistic fine-grain voltage scaling with reconfigurable power distribution networks. In *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 381- 393, 2014.

[7] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6- 15, July 2011.

[8] G. Hoang, R. B. Findler, and R. Joseph. Exploring circuit timing-aware language and compilation. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 345- 356, 2011.

[9] T. Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4):52- 58, Apr. 2001.

[10] V. J. Reddi and M. S. Gupta. *Resilient architecture design for voltage variation*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publ., San Rafael, 2013.

[11] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. *SIGPLAN Not.*, 46(6):164- 174, June 2011.

[12] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of 22nd Annual International Symposium on Computer Architecture*, pages 392- 403, 1995.

[13] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy. b-HiVE: A bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *Proc. of the 52nd Annual Design Automation Conference*, 2015.

[14] J. Xin and R. Joseph. Identifying and predicting timing-critical instructions to boost timing speculation. In *Proc. of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.