Timing Speculation in Multi-Cycle Data Paths

Abstract—Modern processors set timing margins conservatively at design time to support extreme variations in workload and environment, in order to operate reliably and produce expected outputs. Unfortunately, the conservative guard bands set to achieve this reliability are detrimental to processor performance and energy efficiency. In this paper, we propose the use of processors with internal transparent pipelines, which allow data to flow between stages without latching, to maximize timing speculation efficiency as they are inherently suited to slack conservation. We design a synchronous tracking mechanism which runs in parallel with the multi-cycle data path to estimate the accumulated slack across instructions/pipeline stages and then appropriately clock synchronous boundaries early to minimize wasted slack and achieve maximum clock cycle savings. Preliminary evaluations atop the CRIB processor show performance improvements of greater than 10% on average and as high as 30% for an assumed 25% slack per clock cycle.

1 Introduction

Microprocessors are designed to operate on diverse workloads across varying environmental conditions. In most current day processors, timing margins (operating frequency and voltage) are set conservatively at design time using corner analyses to support rare cases of workload (data-dependent) and environmental criticality, allowing them to operate reliably and produce expected outputs. Unfortunately, the conservative guard bands set to achieve this reliability are detrimental to performance and efficiency. Under typical conditions and workload characteristics, each clock cycle is often seen to produce slack averaging more than 25% of the clock period and sometimes even as much as 40% [1], [2]. Raising the frequency or decreasing the voltage to reduce wasted slack can increase performance or decrease power respectively as long as timing errors can be detected and controlled or avoided. This realm of *timing speculation* is an important current area of research towards improving the efficiency of future processors.

In this paper, we propose the use of processors with internal multi-cycle data paths (MDP) [3], [4], which allow data flow between stages/instructions without latching, to maximize timing speculation efficiency as they are inherently suited to slack conservation. We advocate a novel microarchitecture design which allows internal operations of the core to be grouped into multi-cycle asynchronous executions while maintaining synchronous interaction with external interfaces such as memory and architectural state. Our work, therefore, improves performance and/or energy efficiency with an internal MDP pipeline while providing the ease of programmability and verification (outside the MDP) with a synchronous interface. In our model, while the {frequency, voltage} nodes are still set conservatively at design time, the clock cycles to perform executions are adaptively reduced, to provide just sufficient time for computations to complete (effectively cutting slack down to near 0).

Our contributions: ① Present the inherent advantages of MDP towards slack conservation and timing speculation. ② Analyze the accumulation of slack over MDP and its dependency on the depth of these paths. ③ Design a novel *synchronous* slack tracking mechanism to track *asynchronous* slack accumulation and appropriately latching data early at synchronous boundaries. ④ Evaluate performance speedups and overheads of timing speculation using our mechanism.

2 Background and Related Work

Design targets for clock frequency and operating voltage are set based on expected PVT variations - manufacturing variability, supply power variations, workload induced voltage droops, thermal variations from external or internal influences and much more. Apart from PVT variations, guard bands are decided based on the timing of the *most critical* instructions. Data dependent variabilities result in worst-case impact on timing only under certain instruction and data sequences. As long as the occurrences of these sequences are a possibility, conservative guard bands must account for them and extend the guard band accordingly. As a result, timing guard bands in most current day processors are very conservative, often resulting in up to half a clock period slack when executing common sequences.

2.1 Tackling PVT variations

To reduce the guard band width needed to accommodate PVT variations, prior works attempt to predict or detect the variations themselves or their effects on timing/voltage. ReCycle [1] targets process variation alone. It analyzes the property that variation affects some pipeline stages more than others and propose a one-time post-fabrication tuning of delays per stage. Tribeca [2] uses a last value predictor for voltage and frequency settings over discrete time intervals based on dynamically gathered PVT information. In industry, POWER7 [5] introduced Critical Path Monitors to estimate the delays of critical paths caused by PVT variations and implemented feedback controllers to adjust frequency/voltage. These works utilize different, mostly static, mechanisms to accommodate PVT variations but are less concerned about dynamic workload dependent variations.

2.2 Tackling data-dependent variations

The best known technique to adaptively control voltage guard bands in the presence of data dependent variation is Razor [6]. Razor performs DVS of the entire core - tuning voltage based solely on the frequency of timing error occurrences without analyzing the instructions themselves. On the other hand, Xin et al. [7] build a mechanism to predict likely error causing instructions. Such prior works have focused on reducing slack on a per clock cycle basis and are bounded by the possibility of timing errors on every clock cycle. They often have to conservatively cater to the most critical instructions or pipeline stages to prevent misspeculation, or suffer the risk of increasing possibilities of timing errors. Our work, on the other hand, speculates across sequences of stages or instructions by utilizing multi-cycle data paths thereby cushioning most timing criticalities.

2.3 Multi-Cycle Data Path Processors

Processors with multi-cycle data paths [3], [4] allow data to flow through the pipeline from one stage to another without latching. They are inherently suited to variations, allowing

delays to be balanced anywhere within the transparent window. [4] keeps its pipeline latch stages transparent by default, reducing clocking power. The CRIB processor [3], our baseline, propagates data between multiple *entries* combinationally. More details on CRIB are found in Section 4.

In our timing speculation implementation, we exploit the inherently simple idea behind multi-cycle paths: that slack accumulates over sequences of operations that can be executed asynchronously relative to the outside world. Within a few stages in such a sequence, sufficient slack can accumulate allowing synchronous boundaries to be latched one or more cycles early. At best, the entire slack can be eliminated, resulting in early instruction completions, higher performance and better energy efficiency.

3 Proposal

We propose a synchronous slack tracking and opportunistic early clocking mechanism implemented atop the multi-cycle data path execution pipeline. An MDP pipeline in which data flows between stages/instructions without latching is bracketed by synchronous boundaries - the boundaries themselves might vary between different architectures. These synchronous boundaries are the interface to external resources which are designed to be synchronous. They could also be certain tasks which are required to be synchronous for multiple reasons. For example, we enforce all updates to architectural state be kept synchronous to avoid programming and verifiability complications, hence all instruction commits and related writebacks are synchronous. All memory operations are also kept synchronous as we do not modify the design of the memory system. This internal MDP design suits timing flexibility while presenting a synchronous exterior.

In such an implementation, performance speedup can be achieved if it is possible to clock synchronous boundaries early i.e. if instructions could be fetched, committed and allowed to interact with memory, on an earlier clock cycle than the baseline. Opportunities to clock synchronous boundaries early are brought about by slack accumulation within the chain of instructions that can be executed asynchronously between these boundaries. We design a *synchronous* tracking mechanism which runs in parallel with the MDP to estimate the accumulated slack across instructions/pipeline stages. When the accumulated slack estimate crosses integral values this means that instructions have completed execution one or more clock cycles earlier than the synchronous baseline. The synchronous boundaries are then latched on appropriate early clock cycles, effectively eliminating wasted slack and increasing performance.

This work observes memory interactions, register updates, instruction dispatch and commit synchronously. To maintain clarity, examples in this paper focus only on the memory instruction synchronous boundaries.

3.1 Motivating Example

We motivate the description above with an illustrative example shown in Fig.1. The data dependence graph (DDG) for a sequence of instructions bounded by memory instructions is shown in Fig.1.a. The timing diagram for the sequence is depicted in Fig.1.b. L, E and S represent Load, ALU and Store instructions respectively and red dotted lines represent dependencies.

1) b.i represents standard pipelines with synchronous data flow from one instruction to another. The length of the arrows represent the actual time required for each instruction. It is evident that though many instructions complete before the clock edge, the dependent instruction can execute only after the clock edge resulting in slack wastage. In this example, such a sequence consumes 7 cycles.
2) b.ii represents a completely asynchronous design wherein *every* operation is independent of the clock. This means that in this ideal design, memory instructions can be asynchronous (eg. S_1 and S_2) as well and slack from memory interactions can be used in starting dependent instructions early (eg. E_1). By completely using up all the slack, the sequence completes in 5.25 cycles. While this benefit is the ideal case, it is not suited for implementation since completely asynchronous designs lead to higher design complexity and other disadvantages, discussed earlier.
3) b.iii represents our proposal, wherein operations internal to the data-flow engine are asynchronous but all interactions with memory and processor state are seen synchronously. This results in losing some opportunities to save slack (eg. E_1 only executes at start of cycle 2) but in comparison to a totally synchronous design, multiple instructions can still execute early (eg. E_2, E_3, E_4). This provides performance benefits akin to true asynchronous designs (1 cycle saving here), without its inherent disadvantages.

Fig.1.c shows this sequence of instructions executing on a CRIB partition in the manner discussed in b.iii. In this example, only the interactions with the LSQ and RF occur at synchronous boundaries (denoted by clock symbols).

4 Design

We implement our proposed mechanism atop the CRIB architecture [3]. CRIB performs data flow execution and is inherently suited to our proposal of MDP with synchronous boundaries. Analysis for standard OOO or InO pipelines is beyond the scope of the current work.

4.1 CRIB Baseline Architecture

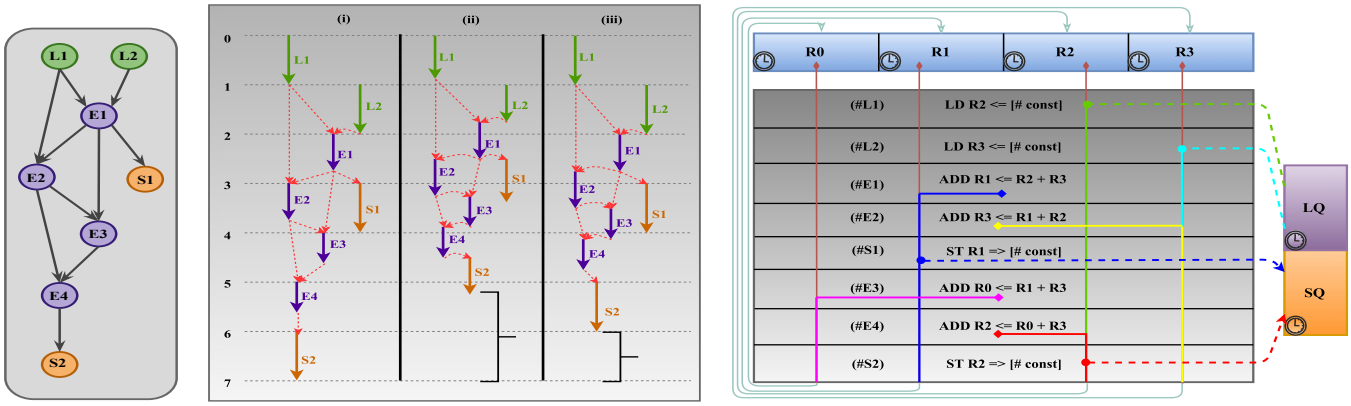
CRIB achieves dramatic power savings by avoiding pipeline latches, register files, complex scheduling logic, and conventional register renaming. In the CRIB processor, the RAT, the RS, and the ROB are consolidated into one structure, called the consolidated rename/issue/bypass block, or CRIB. Figure 1.c shows a simplified example of a single-partition CRIB, with eight entries and four architected registers depicting the flow of instructions through it, avoiding dependency hazards.

Instructions from the front end are placed into the CRIB in program order, starting from the bottom. Each CRIB entry contains routing logic that connects logical register columns, which are spaced horizontally, to an ALU. Each instruction in the CRIB taps its source operands from the register columns. It then overwrites its destination register column accordingly. Data propagation inside the CRIB is done combinationally without latching. This is possible since an ALU serves the instruction until it leaves the CRIB, at which point the result is latched into the ARF. A completion bit is added to each register column and each CRIB entry to maintain synchronous wake-up and completion. When the completion bits in all entries are set, the ARF is clocked and new instructions are inserted into the CRIB. Further details are found in [3].

4.2 Synchronous tracking of asynchronous slack

In the baseline CRIB design, data flows from one entry to another without latching but completion bits, which are indicative of instruction completion, are synchronous. While the transparency in data flow between entries reduces clock power–a worth objective–the slack from each operation goes untapped unless the completion bit latching mechanism is somehow made aware of it.

Our slack tracking mechanism is based on two key factors that influence any particular instruction's available slack



(a) DDG

(b) Timing flow diagram of (i) Fully Synchronous, (ii) Fully Asynchronous and (iii) Our Implementation.

(c) Data flow in a CRIB partition

Fig. 1: Illustrating the Proposal

- a) PVT variations and b) depth of that instruction in the DDG formed from the asynchronously executable instruction sequence of which it is a part. Mathematical models have been developed in literature [8] to estimate the slack of instructions based on PVT variations, modeled as a Gaussian distribution. But such models are limited to recapturing slack within single clock cycles (appropriate for synchronous pipelines). Thus, coarse grained DFS-like mechanisms to recover slack are bounded by the minimal slack available from the most critical instructions or pipeline stages. On the other hand, in our MDP architecture, slack accumulates asynchronously and thus, slack estimates are influenced by the number of instructions in an MDP sequence that can be clubbed together and executed as a single chain. Deeper the DDG of these chains, the higher the slack per instruction (Fig.3) because the available slack is averaged out over the entire sequence predominantly consisting of non-critical instructions. Accordingly, synchronous boundaries can be clocked in early if sufficient slack accumulates.

Currently, we estimate slack at a 99.99% confidence level, though this can be reduced to obtain higher performance at increased risk of timing failures. We envision the use of a Razor [6] like double clocking mechanism for error detection and standard local/global recovery mechanisms. We do not discuss these mechanisms further in this work.

The slack tracking and early instruction clocking mechanism is implemented in CRIB by modifying the implementation of the CRIB entries (Fig.2.a). Each instruction/entry is provided with additional data (bits) - the instruction's level in its DDG (L) and its estimated instant of completion within a clock cycle (D). L is 4 bits and saturates at the max value since experiments show that the slack difference between levels becomes negligible beyond 16 levels (Fig.3). D is 7 bits, allowing 128 different values so as to track time of completion to within of 1% of the cycle period. For eg. a synchronous instruction starts with $L = 0$ and $D = 127$ and a following dependent instruction which takes 75% of a clock cycle would have $L = 0+1 = 1$ and $D = (127 + 0.75 * 128)|128 = 95$.

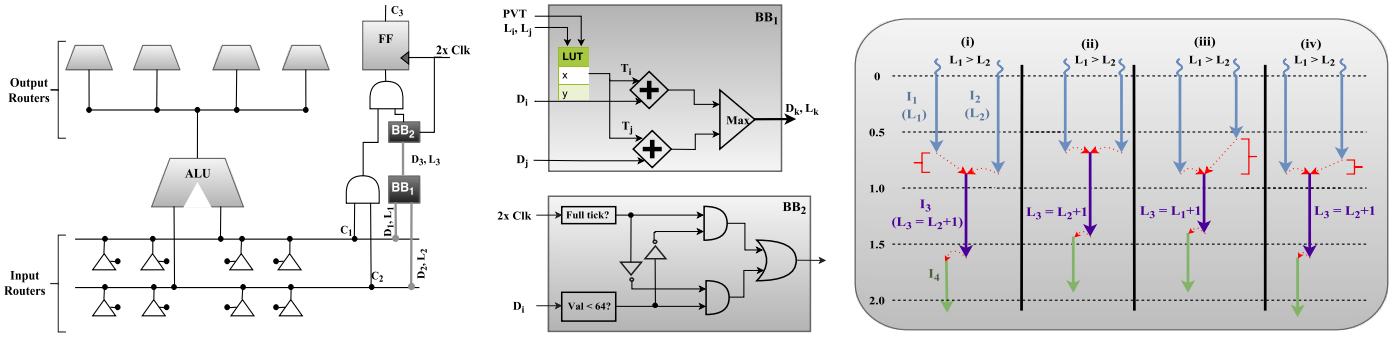
The completion time of an instruction I_3 (Fig.2.c) is dependent on its inputs (I_1, I_2) and their D and L values. Fig.2.c shows all possible relations between I_1, I_2 and I_3. I_3 starts immediately after the last input arrives but its execution time 'T_3' (arrow length) varies in different scenarios. Note, in all four scenarios $L_1 > L_2$ i.e. I_2 is shallower in its DDG than I_1.

The naive choice for T_3 is expecting it to be constrained by the shallower DDG (I_2) as the shallower DDG would mean a higher (and thus, safer) estimate of T_3. This is intuitive and seen in cases (i), (ii) and (iv), wherein $L_3 = (1+L_2)$ and correspondingly D_3 can be obtained as a sum of T_3 ($= f(PVT, L_3)$) and the instant of I_3's last arriving input. Interestingly in (iii), L_3 (and D_3) are constrained by I_1 instead. This is because, since I_2 finished significantly earlier than I_1, it does not constrain the slack/execution time of I_3. On the other hand, in (iv), I_2 finishes only marginally before I_1. Therefore, the more aggressive T_3 estimate based on I_1 might not be supported by the low slack accumulated over I_2's shallow DDG, leading to timing errors - hence T_3 is constrained by I_2. In summary, the correct way to estimate the execution time of the dependent instruction is to estimate completion times based on both the input instructions' DDGs and take the conservative (max) estimate. This is implemented in hardware as BB_1 (Fig.2.b), which evaluates $D_3 = (max(D_1+T_1, D_2+T_2))|128)$. Additionally, $L_3 = (1+L_1)$ or $L_3 = (1 + L_2)$ in accordance to D_3. Note that, $f()$ is implemented as a small LUT providing slack estimates based on PVT and DDG length.

4.3 Clocking the instructions

Once L and D for the current instruction have been set, the instruction's completion bit (C) needs to be set when the instruction is scheduled to complete. In the synchronous baseline implementation, the completion bit for a single cycle instruction is set on the cycle after its last arriving input's completion bit is set. In order to capture accurate slack (at say, 1% of a clock cycle), a naive implementation of synchronous completion bit capture would require a clocking mechanism running at 100x the baseline frequency - which is impractical to implement. Instead, we utilize the fact that the maximum available slack to an instruction can be reasonably bounded. We assume this bound to be 50% in our design i.e. no single cycle instruction completes in less than half a clock cycle. Then it is intuitive that only a maximum of 2 dependent instructions can complete in a single cycle. For the completion bits of both those instructions to be set at the end of the base clock cycle, we clock the capture mechanism simply at twice the base frequency. In this scenario, the *half-tick* would capture the completion of the first instruction and the *full-tick* (coinciding with the base tick) would capture the completion of the dependent second instruction.

The hardware implementation is shown as BB_2 in Fig.2.b. If an instruction's estimated time of completion is in the first half of the cycle, then $D < 64$. Now, when the 2x clock is at a *half-tick*, since $D < 64$ the output is set. Conversely, if $D > 64$ and the 2x clock is at *full-tick*, the output is set. If BB_2 output, as well as the input completion bits are set, the current instruction



(a) Modified CRIB entry (b) Implementing special functions (c) Design Example

Fig. 2: Design Implementation

has completed within that half cycle and its completion bit is set.

A working example is shown in Fig.2.c with a base clock period of 1.0 (note C_i is completion bit of I_i) -

1) In (i), C_3 is set at time 2.0 and C_4 is set at time 2.5, meaning that instruction I_4 (and any older ones) can be committed at time 3.0.
2) In (ii), C_3 is set at 1.5 and C_4 is set at 2.0 itself, meaning that instruction I_4 and older can be committed at time 2.0, saving one cycle compared to (i).
3) If the completion bit capturing happened only at the frequency of core clock, (i) would be unaffected, but in (ii) C_3 would be set only at 2.0 and therefore C_4 is set only at 3.0, meaning that the extra cycle saving is lost.

Note, if the assumption of maximum 50% slack is increased to, say, 75%, the BB_2 clock can be increased to 4x the base frequency. Under the 50% assumption, a 2x clock is *necessary and sufficient* to capture *every* opportunistic early clocking.

4.4 Overheads

Additional data per entry is 7-bit D and 4-bit L values. BB_2 involves only single bit operations with almost no overheads. BB_1 makes use of two 7-bit adders, one 7-bit comparator and an LUT which could hold up to 16 7-bit entries (one entry per level) which can be updated based on PVT variations. The 2x clock only clocks one FF, so the overhead is negligible. Total overheads are reasonable in comparison to the contents of the baseline CRIB entry. Overheads in all structures can be reduced if the granularities of D and L are made coarser (fewer D,L bits).

5 Evaluation

Fig.3 shows execution time (period – slack) as a function of MDP dependency chain lengths. The highlighted points are the value at 99.99% confidence. It is evident that slack increases with longer lengths but begins to saturate at deeper levels.

Bench	Speedup
bzip2	1.08
bwaves	1.10
mcf	1.13
milc	1.01
zeus	1.15
cactus	1.00
soplex	1.16
povray	1.11
hmmer	1.16
libq	1.26
astar	1.30
Mean	1.11

TABLE 1: Speedup

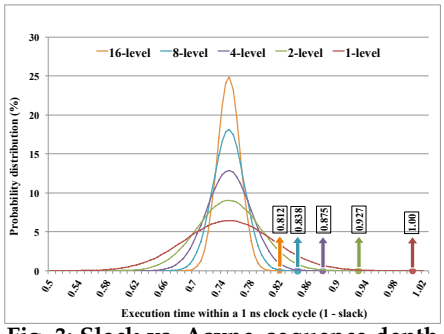


Fig. 3: Slack vs. Async. sequence depth

We implement our proposed mechanism atop CRIB modeled in Gem5 and evaluate results by running the entire SPEC CPU2006 benchmark suite on Simpoints of 100 million instructions. Table.1 shows speedups for a subset of benchmarks at an assumed mean slack of 25%. We see maximum speedups of 30% and 11% on average highlighting the potential of this work. Future work targets simulating environment variabilities and real system slack distributions and implementing optimized error detection and recovery.

6 Conclusion

In this paper, we analyzed the advantages of timing speculation on processors with multi-cycle data paths with synchronous interactions with external interfaces. We analyze the dependence of slack on PVT variations as well as length of instruction sequences between synchronous boundaries. We implement our design on the CRIB processor and design a synchronous slack tracking mechanism which runs in parallel with the MDP. It tracks the accumulated slack across instructions/pipeline stages and then appropriately clock synchronous boundaries early to keep the slack minimal and achieve maximum clock cycle savings. Finally, we evaluate preliminary performance benefits and assess overheads under reasonable assumptions.

References

[1] A. Tiwari, S. R. Sarangi, and J. Torrellas, "Recycle:: Pipeline adaptation to tolerate process variation," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07, 2007, pp. 323–334.

[2] 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 *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 435–446.

[3] E. Gunadi and M. H. Lipasti, "Crib: Consolidated rename, issue, and bypass," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11, 2011, pp. 23–32.

[4] H. M. Jacobson, "Improved clock-gating through transparent pipelining," in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ser. ISLPED '04, 2004, pp. 26–31.

[5] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, pp. 1–11.

[6] 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 *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 7–.

[7] J. Xin and R. Joseph, "Identifying and predicting timing-critical instructions to boost timing speculation," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, 2011, pp. 128–139.

[8] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "Eval: Utilizing processors with variation-induced timing errors," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 41, 2008, pp. 423–434.