Power Balanced Pipelines

John Sartori, Ben Ahrens, and Rakesh Kumar
University of Illinois at Urbana-Champaign

Abstract

Since the onset of pipelined processors, balancing the delay of the microarchitectural pipeline stages such that each microarchitectural pipeline stage has an equal delay has been a primary design objective, as it maximizes instruction throughput. Unfortunately, this causes significant energy inefficiency in processors, as each microarchitectural pipeline stage gets the same amount of time to complete, irrespective of its size or complexity. For power-optimized processors, the inefficiency manifests itself as a significant imbalance in power consumption of different microarchitectural pipestages.

In this paper, rather than balancing processor pipelines for delay, we propose the concept of power balanced pipelines *– i.e., processor pipelines in which different delays are assigned to different microarchitectural pipestages to reduce the power disparity between the stages while guaranteeing the same processor frequency/performance. A specific implementation of the concept uses cycle time stealing [19] to deliberately redistribute cycle time from low-power pipeline stages to power-hungry stages, relaxing their timing constraints and allowing them to operate at reduced voltages or use smaller, less leaky cells. We present several static and dynamic techniques for power balancing and demonstrate that balancing pipeline power rather than delay can result in 46% processor power reduction with no loss in processor throughput for a full FabScalar processor over a power-optimized baseline. Benefits are comparable over a Fabscalar baseline where static cycle time stealing is used to optimize achieved frequency. Power savings increase at lower operating frequencies. To the best of our knowledge, this is the first such work on microarchitecture-level power reduction that guarantees the same performance.*

1 Introduction

Conventional wisdom states that the throughput of a processor pipeline is maximized when the total latency of the pipeline is divided evenly between all the stages. As long as instruction pipelining has been used to increase the throughput of processors, balancing the delay of the microarchitectural pipeline stages has been a primary objective. However, faced with growing concerns about escalating power density and the need for energy efficiency, we challenge the traditional paradigm of delay balanced microarchitectural pipelines. We observe that *pipeline delay balancing is one of the factors that adds to the considerable pressure on processor power*. This is because each microarchitectural pipeline stage in a delay balanced processor gets the same amount of time to complete, irrespective of its size or complexity, causing significant energy inefficiency. For example, a simple data marshaling stage like Dispatch (in FabScalar [5]) has lower area, complexity, and power than a large, complex stage like Issue or Execute. In spite of this natural variance in complexity and area, all stages are expected to finish evaluating in the same amount of time, and more complex logic must be implemented with larger, more leaky cells and expanded topologies that further increase power, thereby exacerbating the energy inefficiency.

For power-optimized processors (i.e., processors where circuit and design-level optimizations reclaim all timing slack to save power), the energy inefficiency of delay balancing manifests itself as a significant imbalance in power consumption of different microarchitectural pipestages – some simple stages consume a small amount of power, while other stages are very power hungry. In fact, a survey of several power-optimized processor pipelines (see Section 2) reveals that the extent of power imbalance can be enormous for such processors (1-2 orders of magnitude between the least power microarchitectural pipeline stage and the highest power pipestage), pointing to significant energy inefficiency.

We propose to abandon the traditional paradigm of delay balanced pipelines in favor of *power balanced pipelines*. We observe that since the ratio of power between pipeline stages is uneven, donating cycle time to a power-hungry stage from a lower power stage will result in processor power savings, even though processor frequency remains the same. Extra cycle time donated to a power-hungry stage enables voltage or area reduction. Although the voltage or area of the low-power, time donor stage increases, the power trade is uneven, resulting in a net power savings for the processor. Thus, by opportunistically creating slack in simple, low-power stages and consuming the slack in complex, high-power stages, we propose to balance the power consumption of pipeline stages (i.e., reduce disparity in their power comsumption) and significantly reduce total processor power for the same guaranteed performance.

In this paper, we demonstrate benefits of pipeline power balancing for a processor baseline that is optimized for power for a given timing target as well as for a processor baseline where cycle time stealing is used to maximize frequency by donating time from fast stages to slow stages. Note that power balancing is not a metric of optimization, it is a conceptual technique with several possible implementations (discussed in Section 3). Power efficiency is indeed the metric of optimization. We use power balancing to reduce power while guaranteeing the same frequency, thereby significantly improving power efficiency.

This work on power balanced pipelines makes the following contributions.

- We observe that there can be significant imbalance in the power consumption of different microarchitectural pipestages of power-optimized processors. The ratio between the power consumption of the least power microarchitectural pipeline stage and the highest power pipestage can be as high as 1-2 orders of magnitude.
- We propose the concept of power balanced microarchitectural pipelines, demonstrating anlytically that deliberately unbalancing delays of microarchitectural pipestages to reduce the disparity in the power consumption of different stages can significantly reduce the total power of a processor without affecting processor throughput. To the best of our knowledge, this is the first such work on micrarchitecture-level power reduction that guarantees the same performance.
- We present static and dynamic techniques based on cycle time stealing for implementing including a new design flow for design-level power balancing, a static voltage assignment technique that balances power at test time, and a dynamic power balancing technique that performs power balancing at runtime.
- We quantify the benefits of different power balancing implementations for a FabScalar processor [5] in terms of power reduction and evaluate the sensitivity of benefits to the number of available voltage domains, the operating frequency, and the workload. We demonstrate 46% power savings for a power balanced pipeline, compared to a conventional delay balanced power-optimized pipeline, when both operate at maximum frequency. Power savings are *higher* when compared against a FabScalar baseline that uses cycle time stealing to maximize operating frequency. Power savings increase at lower frequencies. For processors and workloads that exhibit significant dynamic behavior, dynamic power balancing increases power savings by up to 10% over static power balancing.

2 Motivation

As discussed in the previous section, the power consumption of delay balanced processor pipeline stages can vary significantly, potentially resulting in unnecessary power overheads for the processor. Figure 1 shows the average power breakdown (for a workload of SPEC benchmarks) into the stages of the baseline FabScalar [5] pipeline that we used in our experiments. The power breakdown is based on a full design-level layout and evaluation of processor RTL. The design-flow attempts to minimize the power for the processor for a given timing target. The data demonstrate that power consumption varies from 0.3% of total processor power in the Dispatch stage to almost 40% in the Fetch stage. Such disparate power consumption between stages is due to the fact that all stages are allowed the same evaluation time, irrespective of their complexities. We observed similarly large power imbalance between different microarchitural pipestages for several other cores (OpenSPARC T1, OpenRISC, IVM, Rigel, OpenMIPS, etc.) that we studied. Figure 3 shows the average power breakdown into stages for the power-optimized Rigel [9] (left), OpenMSP430 [17] (middle), and OpenSPARC T1 [14] (right) pipelines. Power imbalance ranges from 1-2 orders of magnitude for these cores.

Considering our observation that there is a signifcant imbalance in the power consumption of different pipestages of processors, we propose a new approach to microarchitectural pipeline balancing based on the relative complexity of each stage, whereby cycle time is redistributed to the stages that can benefit from it the most. This can be illustrated with a simple example in which a simple, low-power stage (S_{lo}, with power P_{lo}) donates a fraction of its cycle time to a complex, high-power stage (S_{hi}, with power P_{hi}). As a result of this exchange, the voltage of S_{hi} (V_{hi}) can be decreased by ΔV_{hi}, and the voltage of S_{lo} (V_{lo}) must be increased by ΔV_{lo}. In order for this trade to reduce power, the net change in power for the pipeline must be less than zero, i.e., $\Delta P_{hi} + \Delta P_{lo} < 0$. Equation 2 describes the change in total stage power ($\Delta P_{total} = \Delta P_{leak} + \Delta P_{dyn}$) that results from voltage scaling.

$$P + \Delta P = \left(\frac{V + \Delta V}{V}\right)^n P \implies \Delta P = P\left(\left(\frac{V+\Delta V}{V}\right)^n - 1\right)$$
$$\Delta P_{leak} = \Delta P(n=1) = P_{leak}\left(\frac{\Delta V}{V}\right) \tag{1}$$

$$\Delta P_{dyn} = \Delta P(n=2) = P_{dyn}\left(\frac{\Delta V}{V}\right)\left(2 + \frac{\Delta V}{V}\right)$$
$$\Delta P_{total} = \Delta P_{leak} + \Delta P_{dyn} = \left(\frac{\Delta V}{V}\right)\left(P_{leak} + P_{dyn}\left(2 + \frac{\Delta V}{V}\right)\right) \tag{2}$$

We substitute the expanded expression for ΔP into the inequality describing a trade that reduces total power to obtain Equation 3. For simplicity, let us assume that the initial voltages are equal for each stage ($V_{hi} = V_{lo} = V$), as would be the initial condition for the pipeline.

$$\Delta P_{hi} + \Delta P_{lo} < 0 \Longrightarrow \left|\frac{\Delta V_{hi}}{\Delta V_{lo}}\right| > \frac{P_{lo,leak} + P_{lo,dyn}\left(2 + \frac{\Delta V_{lo}}{V}\right)}{P_{hi,leak} + P_{hi,dyn}\left(2 + \frac{\Delta V_{hi}}{V}\right)} \tag{3}$$

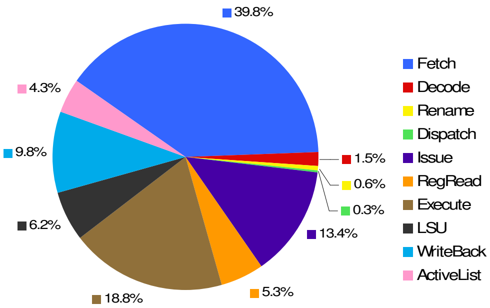

Figure 1: Delay balanced pipeline stages are typically unbalanced in power consumption. On average, the most power-hungry stage in the FabScalar pipeline consumes over two orders of magnitude more power than the least power-hungry stage.

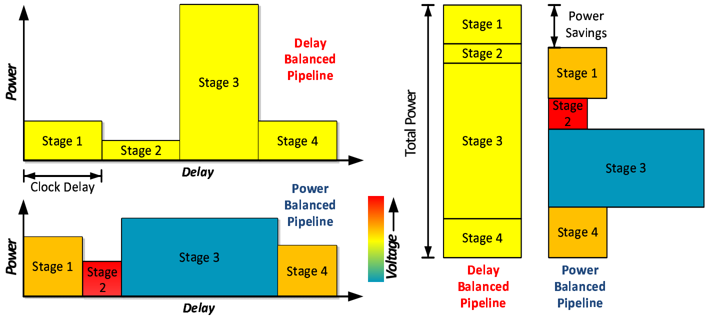

Figure 2: Although both pipelines have the same total delay, the power balanced pipeline consumes less power than the delay balanced pipeline. Note that reducing the height of a tall bar (reducing the power of a high-power stage) has a significantly larger effect on total power than increasing the height of a short bar (increasing the power of a low-power stage).

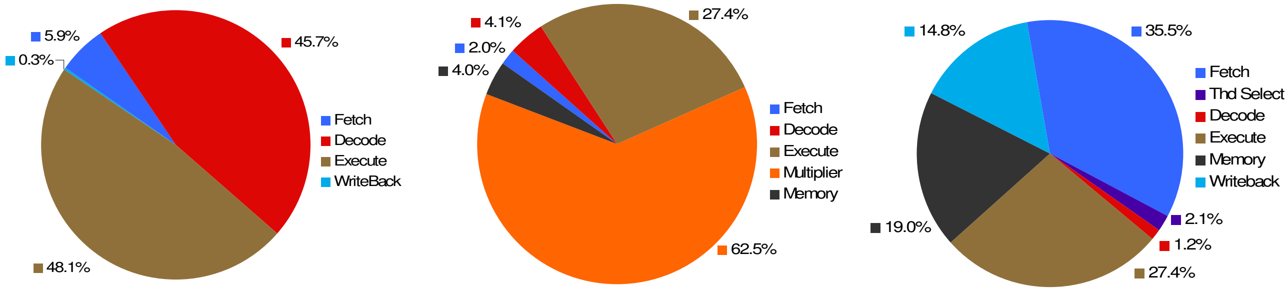

Figure 3: Power imbalance is common in many processor pipelines (even simple, in-order pipelines). On average, the ratio of power consumption between the most power-hungry stage and the least power hungry stage is over 150 for the Rigel pipeline (left) and over 30 for the OpenMSP430 (middle) and OpenSPARC T1 (right) pipelines.

If we assume that $|\Delta V_{hi}| \approx |\Delta V_{lo}|$ (a reasonable assumption if voltage remains in the "linear" region of the delay vs. voltage curve (see Section 4)), the condition for a power-saving trade becomes even simpler (Equation 4).

$$P_{hi,leak} + P_{hi,dyn}\left(2+\frac{\Delta V}{V}\right) > P_{lo,leak} + P_{lo,dyn}\left(2+\frac{\Delta V}{V}\right)$$
$$P_{hi,total} + \kappa \cdot P_{hi,dyn} > P_{lo,total} + \kappa \cdot P_{lo,dyn} \ (\kappa = 1 + \Delta V/V) \tag{4}$$

The simplified condition for a power-saving trade nicely illustrates the main intuition behind microarchitectural power balancing; namely, that power balancing typically results in processor power savings for the same performance when the power of the time stealing pipeline stage is greater than the power of the time donating stage. The greater the power differential between the two pipeline stages, the more power is reduced when cycle time is redistributed from the low-power stage to the high-power stage. Figure 2 illustrates the main insight behind pipeline power balancing.

Note that since delay and power can be traded by adjusting the timing constraints of pipeline stages at design time, multiple voltages are not required in a power balanced pipeline. However, multiple voltage domains increase power savings. Note also that not all delay and power trades are allowed within the framework of microarchitecture-level cycle stealing, as some trades do not preserve correctness or instruction throughput. In general, an allowable trade involves two stages that participate in a common microarchitectural loop and keeps the total latency of all loops the same before and after trading. There are several other practical issues to consider in the implementation of power balancing. We discuss these in detail in Section 4.

3 Power Balancing: Implementations

In the previous section, we motivated and laid the groundwork for power balanced pipelines. In this section, we discuss different static and dynamic power balancing implementations based on cycle time strealing as well as their corresponding design overheads. We formulate the pipeline power balancing problem as an optimization in which cycle time is re-distributed from low-power stages to high-power stages, enabling power reduction in the high-power stages at the expense of power increase in the low-power stages, while preserving processor throughput and minimizing total processor power.

3.1 Static Power Balancing

The best power balancing strategy for a pipeline is determined by the relative amount of power consumed in each pipeline stage. Stages with high power relative to others steal cycle time from stages with relatively low power consumption. Dynamic fluctuations in the absolute power consumption of the stages do not matter, as long as the relative power breakdown remains roughly the same. For processors in which the relative power breakdown between stages remains fairly constant for different programs and program phases, a static power balancing approach does well at reducing total power, while keeping implementation overhead low. For example, processors in which the utilization of each stage depends on the utilization of the other stages (e.g., in-order processors) are well-suited for static power balancing. Static power balancing can be performed either at design time, or after manufacturing, at test time. The implementations and design considerations for these approaches are discussed below.

3.1.1 Design-Level Power Balancing

Pipeline power balancing can be incorporated into the design-level implementation of a processor. We propose a novel design flow for *design-level power balancing* that takes as input the hardware description (RTL) for a pipeline, the desired operating frequency (f), and the number of allowable voltage domains (N_V), chooses the implementation and voltage for each microarchitectural pipeline stage such that processor power is minimized and the throughput target is met, and performs synthesis, placement, and routing (SPR) to implement the power balanced pipeline. Since design-level power balancing changes how the processor is synthesized, benefits over a delay balanced pipeline can be in terms of both power and area reduction.

To arrive at the minimum-power implementation for a power balanced processor pipeline, the optimization heuristic first implements each processor pipeline stage for a range of possible timing constraints, and selects the minimum-power implementation of each stage that meets timing ($1/f$). Then, the heuristic performs cycle stealing between the stages to reduce power by selecting lower power versions of high-power stages and selecting higher power versions of low-power stages to satisfy cycle stealing constraints. The power and delay of a stage can be varied either by choosing an implementation with a different timing constraint or choosing a different voltage for the currently selected implementation (or both). Since design-level power balancing is performed only once at design time, an exhaustive algorithm (Algorithm 1) can be used to evaluate all possible design choices and select the pipeline implementation with minimum power. After implementations are chosen for each stage, final layout is performed for the processor, including clock tree synthesis for the optimized cycle stealing strategy.

Algorithm 1 begins by defining a range of datapoints ($implementation, voltage$ pairs) that are valid for each stage, based on the constraints of the loops that the stage participates in. Cycle time stealing constraints can be boiled down to a requirement that as long as all loops have non-negative slack, the design can be implemented. Each recursive call in the algorithm is associated with a stage in the pipeline. A call to the $recurse$ function passes on the stage configurations that have already been selected, along with the amount of slack available to each loop, given the choices that have been made for this specific implementation path. A path is pruned when no combination of voltage and/or timing constraints can be chosen for the current stage such that all loops still have non-negative slack. If a path reaches the final stage and is able to choose a datapoint that satisfies all loops, it calculates power savings and saves the data, if it is the best implementation found so far. The algorithm is complete when all paths have been completed or pruned. We were able to reduce runtime significantly by initially using a coarse voltage step granularity to identify the ranges of datapoints that allow for the most savings, then focusing in on the datapoints at successively finer granularities in subsequent calls to the algorithm.

Algorithm 1 Exhaustive Power Balancing Algorthm

```
$find\_valid\_datapoints(stage, loop\_data)$;
for each $datapoint \in valid\_datapoints$ do
    $update\_stages(stage\_data\_copy, datapoint)$;
    $update\_loops(loop\_data\_copy, datapoint)$;
    if $stage = NUM\_STAGES$ then
        $calculate\_power\_and\_save(stage\_data\_copy)$;
    else
        $recurse(stage + 1, stage\_data\_copy, loop\_data\_copy)$;
    end if
end for
```

An interesting case for design-level power balancing is demonstrated when $N_V = 1$. In this case, the design requires no additional hardware support for multiple voltage domains or post-silicon cycle time adjustment. All cycle stealing is performed by optimizing the timing constraints of the stages during SPR and adjusting the evaluation times of the stages in clock tree synthesis. Thus, power is reduced with respect to a delay balanced pipeline without any significant implementation overheads. This is especially beneficial when the number of allowable voltage domains is limited.

3.1.2 Post-Silicon Static Voltage Assignment

Rather than using a new design flow to create a power balanced pipeline, power balancing can also be achieved through post-silicon static voltage assignment. With this approach, the processor is designed as normal, and the voltages and delays of the stages are selected at test time to balance the power of the pipeline and reduce total power. To enable post-silicon voltage assignment and delay adaptation, support for dynamic voltage scaling (DVS) or multiple voltage domains and tunable delay buffers [10] is incorporated into the design. Once the power balancing strategy is chosen, inputs to delay and voltage select lines are set or fuses are burned to finalize the cycle time stealing strategy for the chip.

To determine the most efficient power balancing configuration for a chip, the power and delay of each stage is characterized over the range of possible voltages during testing. An optimization similar to the one described in Algorithm 1 (with only one implementation per stage) is performed to select the cycle stealing and voltage assignment strategy that minimizes total power.

Note that post-silicon static power balancing can also be used to overcome inefficiencies caused by process variations. While previous works used cycle stealing to rebalance delay and minimize throughput degradation caused by process variations [19, 11], we seek to intentionally *unbalance* delay in order to balance power consumption and reduce total power for the same guaranteed performance.

The hardware overheads associated with post-silicon static voltage assignment can be reduced by limiting the number of voltage domains, or even by implementing the cycle stealing and voltage assignment strategy at design time. In this scenario, static voltage assignment could be viewed as a limited case of design-level power balancing that only considers a single implementation of each stage, optimized for the target frequency of the processor.

Post-silicon static voltage assignment may increase testing time if an exhaustive power balancing algorithm is used. However, the time required to find a suitable power balancing strategy can be reduced to negligible levels by using an optimization heuristic. For example, Algorithm 2 describes a fast power balancing heuristic that performs gradient descent to approach the minimum power configuration. First, all stages are set to the maximum voltage, such that delay is minimized. Then, for each stage, we calculate the potential power savings of reducing the voltage to the minimum value such that all loops constraints are met. We follow the direction of steepest descent by reducing the voltage by a small amount ($v_{step=0.01V}$) on the stage that has the highest potential power savings. Gradient descent continues until no stage can reduce its voltage without breaking a loop constraint. This heuristic avoids local minima by computing the total potential power savings for a stage, rather than the savings for a small change in voltage. This prevents the heuristic from choosing stages that present significant savings in the short run but consume too much delay in the process. This also prevents the heuristic from getting stuck due to noise in the characterization data. The power savings achieved by this fast heuristic are typically within 3-5% of the exhaustive algorithm's savings, and runtime is reduced significantly (to less than $1ms$).

3.2 Dynamic Power Balancing

For processors in which the relative power breakdown between pipeline stages may change due to changes in the workload, *dynamic power balancing* may afford additional power reduction over static power balancing. This is because the optimal power balancing strategy depends on which stages consume the most power. A processor that contains units for which utilization depends strongly on the program or program phase (e.g., a FPU) can potentially benefit from adapting the power balancing strategy during runtime. The mechanisms used to adapt stage power and delay dynamically can be the same as those used for post-silicon voltage assignment. However, to allow dynamic adaptation, the select lines for tunable delay buffers and DVS are controlled by the operating system (OS) or a simple hardware controller. Also, an additional mechanism is needed to determine when to re-balance power. We describe the implementation of this mechanism for the case of the FPU, as we observed that the fraction of power consumed in other stages stays relatively constant (Section 6.3).

Algorithm 2 A Fast Gradient Descent-based Power Balancing Heuristic for Reducing Time Overhead

```
for each $stage$ do
    $stage\_data[stage].voltage = MAX\_VOLTAGE$;
end for
while $(stage = max\_savings\_stage(stage\_data)) \neq -1$ do
    $stage\_data[stage].voltage = stage\_data[stage].voltage - v_{step}$;
    $update\_loops(stage\_data)$;
end while
```

To identify a FP-intensive program phase, we use a performance counter to count the number of FP instructions committed within a fixed period of time. Since the number of committed FP instructions provides an estimate of FPU energy, measuring the count over a fixed period of time (say, an OS timeslice) gives an estimate of FPU power. Based on the number of FP instructions in the time window (FPU power), the dynamic power balancing mechanism adapts the power balancing strategy to shift power and delay into or out of the FPU. We assume that the FPU is part of architectural loops that contain other pipeline stages – the same assumption made in prior work [19, 11]. We observed that the finest granularity of adaptation required for our test workloads is on the order of hundreds of ms (Figure 11). Therefore, we model an OS-based power balancing mechanism that counts the number of FP instructions committed (e.g., $PAPI\_FP\_OPS$) in an OS timeslice (5 ms), and decides whether power should be re-balanced. The exact mechanism involves using the FP instruction count to reference into a lookup table that stores the voltage and delay assignments for each stage in each configuration. When rebalancing is needed, the OS assigns the stage voltages and delays loaded from the table. As we show in Section 6.3, the number of required configurations is low in practice (only two are needed – FP and non-FP). Thus, we simply use a single comparator to select the appropriate power balancing configuration to load from a two-entry lookup table. In our evaluation of dynamic power balancing (Section 6.3), we also consider the time required to adapt stage voltages to their new levels (at 10 $mV/\mu s$) when power is re-balanced. Note that in practice, the runtime overheads of dynamic power balancing can be piggybacked on the context switch overhead that occurs every OS timeslice. Note also that mechanisms to dynamically adapt voltage and cycle time are only needed for loops containing the FPU. Thus, hardware overhead for dynamic adaptation mechanisms can be kept relatively low.

4 Practical Considerations

The previous section describes techniques for balancing power in a pipeline to reduce total power. To make power balancing work correctly and efficiently, several practical issues should be considered. In this section, we discuss in further detail the relationship between delay and voltage, and the mechanisms that we use to perform power balancing.

4.1 Cycle Time Stealing

We use cycle time stealing as the core mechanism to perform power balancing. Cycle time stealing [19, 11, 10] allows a pipeline stage to donate a fraction of its evaluation period (cycle time) to another stage, without affecting the operating frequency of the pipeline. Cycle time stealing re-distributes cycle time from a donating stage (S_D) to a receiving stage (S_R) by delaying the clock signal at the input flip-flop (FF) of S_D (allowing less time to evaluate) and the output FF of S_R (allowing more time to evaluate) by the same amount (δ). This delay is propagated between S_R and S_D by delaying the clock signals to all intervening FFs from S_R, up to and including the FF preceding S_D. Since clock signals at both the input and the output FFs of these stages are delayed by the same amount, their cycle times are unaffected. However, since the clock signal at the input FF of S_R is unchanged, S_R now has an evaluation period of $T_{cp} + \delta$. Similarly, since the clock signal at the output FF of S_D is unchanged, S_D now has an evaluation period of $T_{cp} - \delta$.

For design-level power balancing, we implement cycle time stealing statically in the clock network during clock tree synthesis. For techniques that require post-silicon adaptation, we assume the use of tunable delay buffers [10].

To avoid effects on throughput or correctness, we observe certain cycle time stealing constraints [19, 11]. To perform a trade between two stages, the stages must participate in a common loop. Most pipelines have different execution paths, to accommodate different instructions. If two stages do not share a loop, delay cannot be traded between the stages. On the other hand, a stage may participate in multiple loops. Thus, time stealing within one loop may alter the latency of other loops. To avoid throughput implications, the total latency of each loop ($s * T_{cp}$ for an s-stage loop) must remain constant before and after cycle time stealing.

For example, consider a stage (S) that participates in two loops – L_1 and L_2 – where L_2 is a feedback loop containing only S. If the delay of S decreases due to a trade within L_1, the delay of the feedback loop, L_2, must be increased to restore its original delay. We can think of the feedback path in L_2, which contains an output FF feeding into an input FF, as a second stage. Therefore, the stage, S, and the feedback loop make up a 2-stage loop that must be balanced like any other loop. This implies that a stage with a feedback path into itself can only participate in cycle time stealing when its feedback path is from output FF to input FF. Even then, trading with the "feedback stage" is limited to a maximum of T_{cp}. Again, the important constraint to keep in mind is that after cycle time stealing, all loops must have the same delay as before. Feedback paths that end up shorter than before can be corrected with the addition of delay buffers. However, increased delay in a feedback loop can only be corrected by adding in a dummy stage [19] to take up the negative slack. Dummy stage insertion can degrade performance (IPC), and we have not allowed them in our power balancing heuristics.

Additional practical issues were considered when donating a large amount of cycle time to a single stage. First, we recognize that voltage cannot be decreased indefinitely. Also, voltages of donating stages cannot be increased indefinitely, due to aging considerations, since circuit aging is accelerated at higher voltages. On a more subtle note, when a stage steals cycle time, its critical paths are allowed longer than the clock period (T_{cp}) to evaluate. However, the input FF is still clocked every clock period. Thus, a stage may be evaluating multiple instructions during the same evaluation period. If a fast path were to cause a stage output to change before the output of the previous instruction had been sampled, data would be corrupted. Therefore, all paths in a stage must satisfy a short path or hold time constraint ($D_{min} \geq \delta_f - \delta_i + T_{hold}$) [19]. The constraint on the minimum path delay allowable in a stage (D_{min}) depends on the amount by which the evaluation time of the stage has been extended ($\delta_f - \delta_i$) and the FF hold time (T_{hold}). Since the delay of the most power hungry stages could potentially be extended significantly, fast path buffering may become necessary for stages that steal cycle time.

4.2 Local Voltage Scaling

While power balancing does not require multiple voltage domains, benefits may improve with local voltage scaling. Per-stage local voltage scaling [10] requires that each stage has its own voltage regulator. This allows voltage and power reduction for high-power stages, enabled by voltage increase and slack creation in low-power stages. While local voltage scaling is conceptually simple, there are a few practical issues that must be addressed. First, routing a unique voltage to each stage can be costly [10]. Rather than a single voltage network feeding the pipeline, a separate network is needed for each stage. In practice, the overhead of creating separate voltage domains can be limited to acceptable levels by restricting the number of allowable voltage domains.

Voltage level conversion between stages may also be a concern. When a low-voltage stage feeds into a high-voltage stage, the signal from the low-voltage stage may not completely turn off an input transistor in the high-voltage stage, potentially creating a short circuit path. This issue was analyzed by the authors of [11], who concluded that as

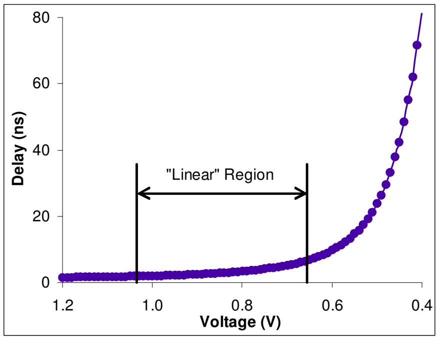

Figure 4: Trades are efficient when voltage remains within the "linear" domain of the voltage-delay curve.

long as $\Delta V < V_t$, the static power increase due to short circuit paths is negligible. Thus, during optimization, we take care to avoid larger voltage differentials between adjacent stages to avoid excessive leakage.

4.3 Voltage-Delay Relationship

Circuit delay is inversely related to the voltage supplied to the circuit ($Delay \propto \frac{V}{(V - V_t)^{\epsilon}}$) [12]. In the delay equation, V is drain voltage, V_t is threshold voltage, and ϵ is a technology-dependent constant. Figure 4 shows the delay vs. voltage curve for the Issue stage of the FabScalar pipeline. As V approaches V_t, there exists a voltage after which delay begins to rise sharply, even for a small reduction in voltage. Similarly, there exists a voltage after which increasing the voltage, even by a large amount, only results in a small decrease in delay. We observe that our power balancing heuristics (which select the most power-efficient voltage and delay for each stage) avoid these regions and choose voltages in the "linear" region of the delay vs. voltage curve. We have taken all the practical considerations and constraints mentioned in this section into account in our power balanced pipeline implementations.

5 Methodology

Since an accurate evaluation of power balancing requires detailed analysis of processor power, delay, and hardware characteristics, we use a detailed design flow to implement and evaluate processor hardware. We use the FabScalar [5] framework for our evaluation of power balanced pipelines. FabScalar is a parameterizable, synthesizable processor specification that allows for the generation and simulation of RTL descriptions for arbitrarily configured scalar and superscalar processor architectures. For our evaluations, we execute benchmarks from the SPEC benchmark suite (INT: bzip, crafty, gap, mcf, parser, twolf, vortex; FP: ammp, art, equake, swim, wupwise) for 3 billion cycles. As described in Section 6.3, we use FP benchmarks primarily to evaluate dynamic power balancing, since the FabScalar architecture does not contain a FPU. Benchmarks are executed on a synthesized, placed, and routed FabScalar processor after fast-forwarding the benchmarks to their Simpoints [6]. Table 1 gives the microarchitectural parameters of the FabScalar pipeline that we implemented. We also evaluated pipeline power balancing for other microarchitectural configurations but did not observe a significant difference in results. Thus, we chose a simpler core in the design space for faster implementation and characterization.

Table 1: Processor Microarchitecture Parameters.

Fetch Width	ALU	IQ Size	ROB Size
1	1	16	64
Phys Regs	LSQ Size	Dcache	Icache
64	16	32 kB	32 kB

We compare power balanced pipelines against two different baselines. Our first baseline is a conventional design that has been leakage optimized by our CAD flow for minimum power. Since we use cycle stealing as a mechanism for power balancing, we also compare power balanced pipelines against a second baseline that takes the original synthesized, placed, and routed design and performs cycle stealing to maximize the frequency of the processor. When comparing against the second baseline, we evaluate power balancing at the highest frequency achievable by the cycle stealing performance-maximized baseline.

Designs are implemented with the *TSMC* 65GP standard cell library (65nm), using *Synopsys Design Compiler* [15] for synthesis and *Cadence SoC Encounter* [4] for layout. In order to evaluate the power and performance of designs at different voltages and to provide V_{th} sizing options for synthesis, *Cadence Library Characterizer* [2] was used to generate low, nominal, and high V_{th} cell libraries at each voltage (V_{dd}) between $1.2V$ and $0.4V$, at $0.01V$ intervals. Power, area, and timing analyses are performed using *Synopsys PrimeTime* [16]. Gate-level benchmark simulation is performed with *Cadence NC-Verilog* [3] to gather activity information for the design in the form of a value change dump (VCD) file, which is subsequently used for dynamic power estimation.

Since design-level power balancing may require characterization of pipeline stages for multiple timing constraints, implementation time may increase proportionally with the number of additional design points. However, design time overhead can be reduced by limiting the number of timing constraints for which each stage is characterized or by performing characterization after synthesis rather than after layout.

We implement our designs using cell libraries that guardband for worst case process, voltage, and temperature (PVT) variations (V=0.9V [$V_{nominal}$=1.0V], T=125C, process=SS). This is standard practice in industry to ensure that designs operate correctly, even in the presence of variations. We also evaluate the benefits of power balanced pipelining assuming worst case variations. This is a fairly conservative methodology, since it minimizes any additional slack that might have been advantageous for cycle stealing. Note that post-silicon voltage assignment and dynamic power balancing could potentially achieve more power savings by adapting to process variations. However, we choose not to evaluate this potential, since it is not a main contribution of our work, and previous works have already explored process variation-related optimizations [19, 11].

Since SRAM structures are already typically optimized for and operated at their lowest possible voltages on a separate voltage rail, we do not target SRAM power reduction with our techniques. Consequently, processor-wide power savings that consider core logic and SRAMs must be derated by the fraction of processor power consumed in SRAMs. We use CACTI [18] with smtsim [20] and Wattch [1] to estimate the fraction of processor power consumed in SRAMs.

6 Results

6.1 Design-Level Power Balancing

Figure 5 shows total processor power savings achieved by design-level power balancing with respect to a delay balanced power-optimized pipeline for the same operating frequency. Results are provided for different operating frequencies (clock periods). At nominal voltage, the fastest attainable clock period for the processor is $1.4ns$. We also compare power savings for an unlimited number of voltage domains (one per stage) against cases where only one or two voltage domains are allowed.

As Figure 5 demonstrates, the power savings afforded by balancing pipeline power rather than delay can be significant, even when only a single voltage domain is allowed. Power savings increase for higher clock periods because designs are less tightly constrained at higher clock periods, allowing more flexibility to perform cycle time stealing. This is especially helpful for design-level power balancing, because the added flexibility allows more options for trading power and delay by changing the design implementation, which may be more efficient in some scenarios than changing the voltage.

For example, we observe that in several instances, low-power stages donate cycle time by taking advantage of design implementations with tighter timing constraints, rather than operating at an increased voltage. When possible, tightening the timing constraint can result in less power overhead than increasing the voltage, because tightening the timing constraint mostly increases leakage on the critical paths of a design, while increasing the voltage increases power for the entire design.

Power savings also increase as more voltage domains are allowed, since each stage operates closer to its optimal voltage. Still, we observe that reducing the number of voltage domains does not significantly hinder power savings. On average, allowing only two voltage domains reduces power savings by only 3% from the per-stage voltage domain case. Even for a single voltage domain design, power savings are only reduced by 8%, on average. This is an encouraging result, since it means that power balancing has the potential to achieve significant power savings without design overheads for additional voltage domains. Also, since cycle time stealing can be accomplished in clock tree synthesis, design-level power balancing with a single voltage domain has no appreciable hardware overheads compared to a conventional delay balanced pipeline, other than hold buffering, which increases area and power by less than 2%.

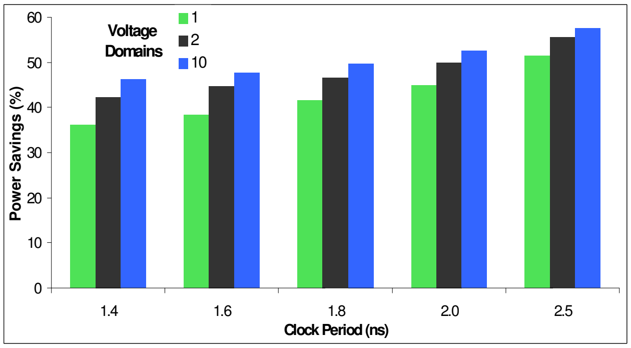

Figure 5: Design-level power balancing achieves significant power savings over delay balanced pipelining. Benefits increase with the clock period and the number of voltage domains.

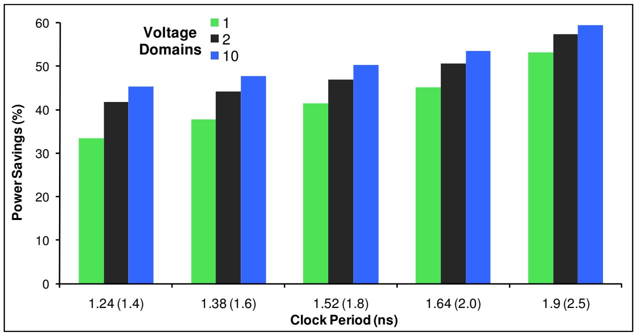

Figure 6: Results for the alternative baseline that uses cycle stealing to maximize performance (frequency). The numbers in parentheses are the corresponding clock periods for the power-optimized baseline.

We observe that design-level power balancing does not significantly affect area. On average, the area of a design-level power balanced pipeline is within 2% of that of a delay balanced pipeline.

We also evaluated the benefits of power balancing over a FabScalar baseline that takes the original synthesized, placed, and routed design and performs cycle stealing to maximize the frequency of the processor. Figure 6 shows the results. As can be seen, the benefits of power balancing *increase* for the cycle stealing-based performance-maximized baseline. This is because while cycle time stealing does indeed improve the frequency of the baseline processor (for example, the minimum clock period of the baseline decreased by 12% – 1.4ns to 1.24ns), resulting in increased power for the corresponding power balanced pipeline (since microarchitectural loops are now tighter), the performance maximized baseline consumed 15% more power, on average, than the power-optimized baseline, for

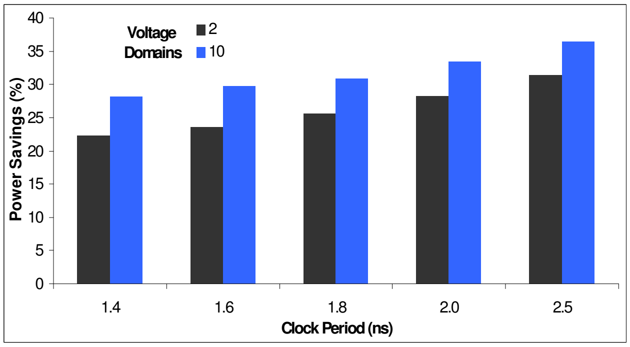

Figure 7: Power balancing through post-silicon voltage assignment reduces processor power significantly. Benefits are less than those of design-level power balancing, because the design implementation is fixed, so power can only be traded through voltage.

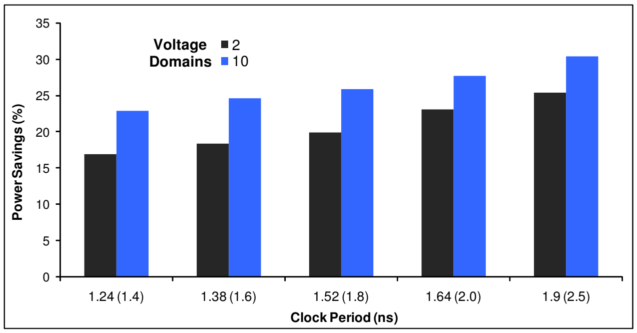

Figure 8: Results for the alternative baseline that uses cycle stealing to maximize performance (frequency). The numbers in parentheses are the corresponding clock periods for the power-optimized baseline.

clock periods between 1.4-2.5ns. This led to higher relative benefits from power balancing. On an ancillary note, the results also show that the power-optimized baseline is more power efficient than a cycle time stealing-based performance maximized baseline for our Fabscalar design. Note also that power balancing can potentially save power even when cycle stealing cannot increase performance. Consider an example processor with 2 pipeline stages where both stages have equivalent delay, but Stage1 consumes 10X more power than Stage2. Although the performance of this design cannot be increased with cycle stealing, power balancing can significantly reduce the power.

6.2 Post-Silicon Static Voltage Assignment

Figure 7 shows total processor power savings achieved by a power balanced pipeline that employs post-silicon static voltage assignment, with respect to a delay balanced pipeline. Again, results are shown for different operating frequencies and numbers of voltage domains.

As in the design-level case, benefits increase with the clock period and the number of voltage domains. On average, allowing per-stage voltage domains increases power savings by 5.5% compared to the dual voltage rail case. Power savings are lower (20%, on average) than those of design-level power balancing, because the implementation for a given frequency is fixed. As discussed in the previous section, adapting the design-level implementation is especially beneficial for several low-power stages that donate cycle time. The main benefits of post-silicon voltage assignment over design-level power balancing are reduced design time and the potential to achieve additional benefits by adapting to process variations, if they are significant. As discussed in Section 5, we choose to present conservative results that do not account for adaptation to process variations, since it is not a main contribution of our work, and previous works have already explored process variation-related optimizations [19, 11].

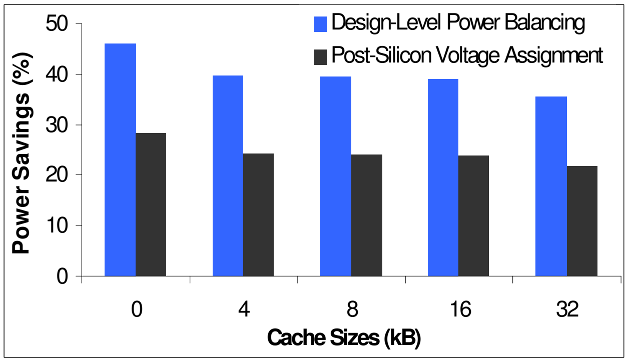

Figure 9: Since our power balancing techniques do not target SRAM power, we also show power savings that account for the power consumed by SRAMS and core logic for different cache sizes (e.g., $x = 4$ means $Icache = Dcache = 4kB$).

Figure 8 shows the corresponding results for the cycle stealing-based performance maximized baseline. Again, the benefits from power balancing are higher in spite of tighter microarchitectural loops due to significantly increased power consumption of the performance-maximized baseline.

As explained in Section 5, our power balancing techniques do not target SRAM power reduction. Figure 9 shows processor-wide power savings, averaged over all benchmarks, that account for the power of core logic and SRAMs. Results are shown for different Icache and Dcache sizes at maximum operating frequency ($T = 1.4ns$). The data for cache size 0 represents power savings for core logic alone.

6.3 Dynamic Power Balancing

In our evaluation of power balancing for a FabScalar pipeline, which does not support a FPU, we did not observe a significant difference in the optimization strategy or benefits for different benchmarks (results omitted due to page limit). This is not surprising, since the optimization strategy depends not on the absolute power consumption of each stage (which can vary significantly between benchmarks) but the relative power breakdown between stages

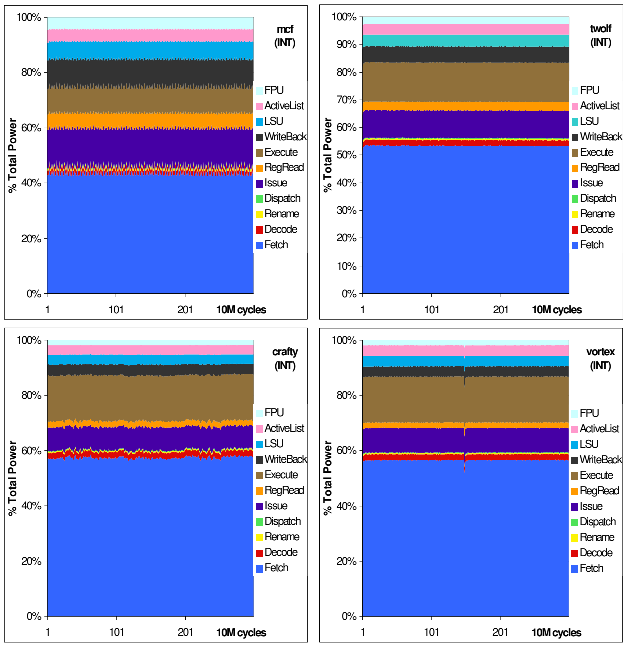

Figure 10: Pipeline stages show little variation in power consumption while executing INT benchmarks.

(which remains fairly constant). Since all the stages in a pipeline operate synchronously, when the utilization of one stage changes, the utilization of all other stages tends to follow suit. Thus, the fraction of power consumed by each stage does not vary significantly, and static power balancing performs well in most scenarios. Figure 10 shows the percentage of total power consumed by each pipeline stage throughout the execution of sample integer (INT) benchmarks (results are similar for others), demonstrating that the power breakdown remains fairly stable.

The situation may be different, however, in a processor that contains a FPU. In order for a stage to require dynamic power balancing, the *fraction* of power consumed by that stage must vary dynamically. This requires that (1) the stage consumes a significant fraction of total processor power and (2) the utilization of the stage varies significantly, and somewhat independently, from the rest of the pipeline. These conditions may be true for a FPU, since a FPU can consume a significant fraction of total processor power, and FP benchmarks typically contain phases of intense FPU utilization that integer benchmarks do not.

In order to test our intuitions about the FPU, we characterized the activity factor of the FPU over time for different benchmarks using smtsim [20]. We then performed SPR for the FPU from the OpenSPARC T1 processor [14] to allow accurate design-level power and delay characterization. To characterize FPU power vs. time for different benchmarks, we propagated the activity profiles captured from smtsim on the OpenSPARC FPU, using PrimeTime [16].

Figure 11 shows the percentage of total processor power consumed in the FPU over the execution of several benchmarks, and Figure 12 shows the average pipeline power breakdown. (We show results for all FP benchmarks but only three INT benchmarks (mcf, parser, twolf), as results are similar for all INT benchmarks.) Figures 11 and 12 confirm that the pipeline power breakdown does not vary significantly within or between INT benchmarks, even for a processor with a FPU. Also, the fraction of power consumed by the FPU for INT benchmarks is small. Thus, there is no potential for benefits from dynamic power balancing within INT benchmarks. The figures also show that the difference in the pipeline power breakdown between INT and FP benchmarks can be significant due to the change in FPU power consumption. Thus, dynamic power balancing may achieve benefits by identifying and adapting to FP and non-FP workloads. We did observe significant variations in FPU power within FP benchmarks (Figure 11). However, the benefits of adaptation within a FP benchmark will be limited, as these variations do not significantly affect the pipeline power breakdown (Figure 13).

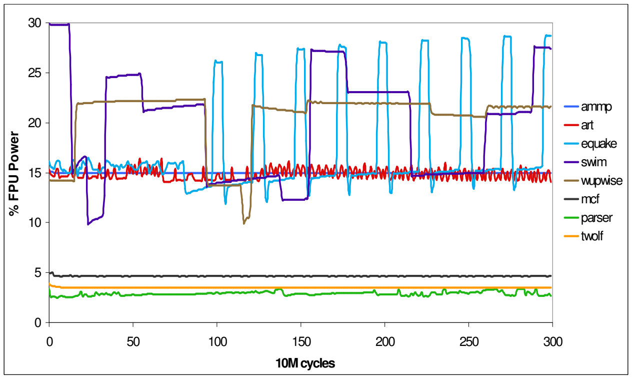

Figure 11: The fraction of processor power consumed by the FPU is significantly different for INT benchmarks and FP benchmarks.

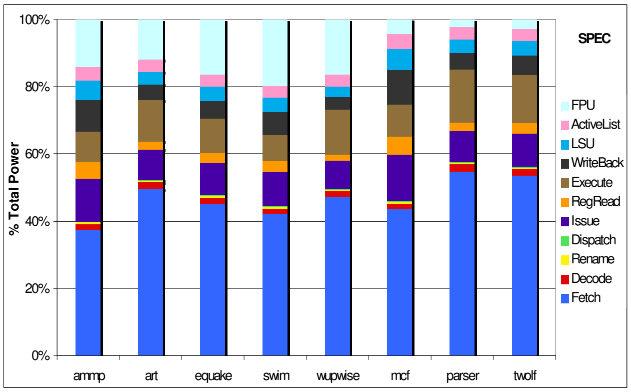

Figure 12: The breakdown of power consumption into pipeline stages is different for INT benchmarks and FP benchmarks.

Figure 14 evaluates dynamic power balancing by comparing the energy required to execute several benchmarks for three scenarios. In the *static* case, the power balancing strategy is the same for all benchmarks, based on the average power consumption of each stage, including the FPU. In the *dynamic* cases, we adapt the power balancing strategy as the FPU utilization changes. The *dynamic oracle* represents an ideal policy where the processor always uses

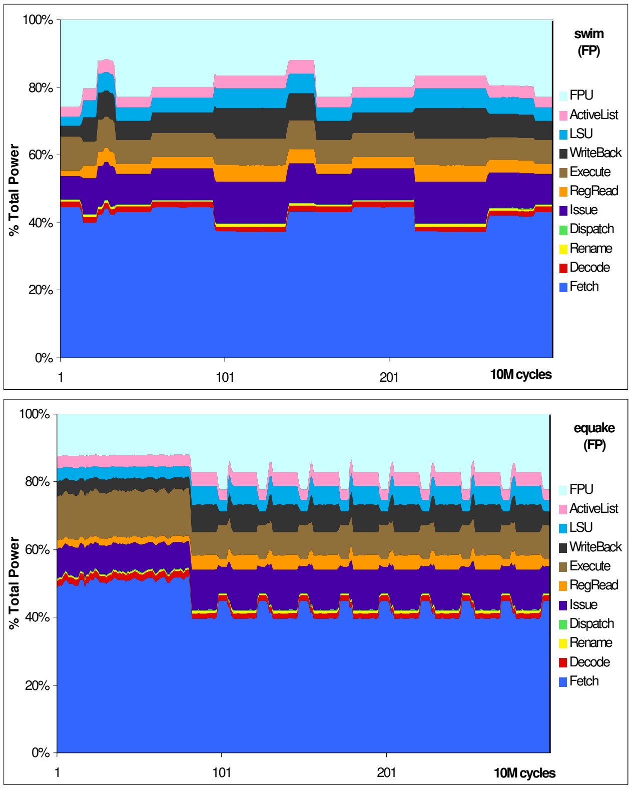

Figure 13: Dynamic changes in the power breakdown of a processor while running a FP benchmark do not deviate significantly from the average. Results are shown for the FP benchmarks that exhibit the most dynamic variation.

the optimal power balancing strategy for a given program phase. Plain *dynamic* represents our realistic implementation that includes all overheads of dynamic adaptation, required to recognize and adapt to the program phase dynamically.

The most substantial difference between *static* and *dynamic* is for INT benchmarks. Since *static* is optimized for average case FPU activity, INT benchmarks – which have almost no FPU activity – exhibit 10% higher energy, on average. For FP benchmarks, the difference between optimizing for average FPU activity (*static*) and full dynamic adaptation is small (1 – 2%), since variation in FPU activity does not cause the relative power breakdown to deviate significantly from the average. Therefore, the potential benefit of dynamic adaptation is mainly in adapting to the differences between INT and FP benchmarks, but not the differences between phases within a FP benchmark. This is somewhat beneficial, because it allows for a very simple adaptation mechanism. The mechanism only needs to recognize the difference between FP and non-FP phases and adapt the power balancing strategy accordingly. This mainly involves shifting power between the FPU and the stages of one architectural loop. Thus, the overhead for adaptation circuitry (TDBs and voltage scaling) can be confined to this loop of the processor.

The dynamic power balancing results in this section assume the availability of per-stage voltage adaptation. Nevertheless, energy increases by less than 5% if only two voltage domains are allowed.

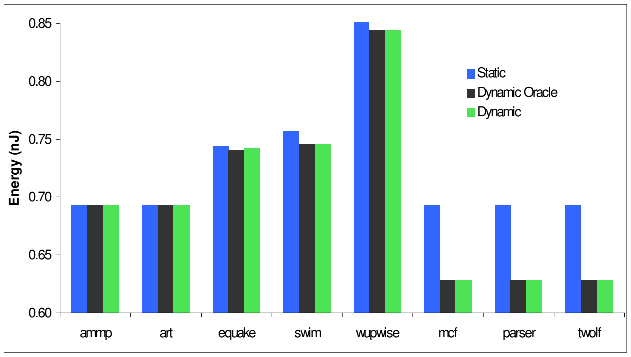

Figure 14: When the relative power breakdown of the processor changes substantially (e.g., between INT and FP benchmarks), dynamically adapting the power balancing configuration can increase power savings.

7 Related Work

In this paper, we propose the concept of power balanced pipelines in which delay is delberately unbalanced to reduce the disparity in power consumption of different processor pipestages to minimize total processor power without affecting throughput. Below, we characterize previous work related to pipeline power and delay adaptation.

Cycle time stealing has been used in the past as a means of post-silicon tuning to increase the performance of a pipeline affected by process variations. ReCycle [19] uses post-silicon cycle time stealing to re-balance delay in a pipeline – re-distributing cycle time from fast stages to slow stages, allowing a pipeline to operate at a clock period closer to the average stage delay. ReVIVaL [11] proposes a form of latch-based cycle time stealing in which an empty "donor stage" can add an extra cycle of latency to a timing-critical pipeline loop in case manufacturing variations prevent the processor from meeting timing. A version of Razor [10] proposes to reclaim per-stage slack caused by factors such as process variation by allowing cycle stealing and per-stage voltage scaling to re-balance the delay of the pipeline stages. We use cycle time stealing in a novel context – rather than re-balancing the delay of the pipeline stages to cope with manufacturing variations, we deliberately unbalance the stage delays in favor of power balancing to reduce power in energy-constrained processors while guaranteeing the same performance. In fact, we can take any design where delay has been balanced using any of the above techniques (or combinations thereof) and get power benefits through power balancing. Section 6 presents the benefits of power balancing for a baseline that uses cycle time stealing to maximize operating frequency.

Multiple voltage and clock domain (MVCD) designs [13, 8, 21] aim to exploit application characteristics to manipulate the power and delay of regions within a design in order to save energy. Voltage and clock frequency adaptations can be applied at the granularity of cores in a multi-core system or even within a single processor core. Performance loss in MVCD depends on the application. On the other hand, power balanced pipelines guarantee the same performance. For this reason, we can perform power balancing using the peak power of low power stages and the minimum power of high power stages even if we don't know the application behavior. Also, unlike MVCD designs, pipeline power balancing does not require multiple frequency or voltage domains.

Our goals are somewhat similar to those of asynchronous design [7]. We minimize energy by adapting the delay of each pipeline stage, allowing large, complex blocks more time to evaluate at reduced power. An asynchronous design is even more adaptable than a power balanced pipeline. Asynchronous design uses additional circuitry to signal completion of a logic block for every applied input. In our case, we only manipulate the path delays of a logic block, and the adjustment is static for a given input. There is no additional circuitry to signal that a logic block is ready for the next input, should the current input fail to excite the critical path of the logic block. The added adaptability of asynchronous design may allow higher throughput or lower energy in several cases, however, practical implications of asynchronous design tend to incur considerable overheads and design difficulties [7].

8 Conclusion

In this paper, we make a case for power balanced processor pipelines by demonstrating that deliberately unbalancing the delay of different pipeline stages to reduce the disparity in the power consumption of the different stages of the processor pipeline can result in significant processor power savings for the same guaranteed performance. We propose and evaluate several static and dynamic implementation techniques based on cycle time stealing for pipeline power balancing, and demonstrate that even low-overhead, static approaches have potential to reduce processor power significantly. We demonstrate 46% power savings at maximum frequency for a power balanced pipeline, compared to a delay balanced power-optimized pipeline with the same frequency. Benefits are comparable over a baseline where static cycle time stealing is used to optimize frequency. Power savings increase at lower frequencies. To the best of our knowledge, this is the first such work on microarchitecture-level power reduction that guarantees the same performance.

Acknowledgments

This work was supported in part by SRC. We would like to thank Janak Patel and the anonymous reviewers for providing feedback that helped improve this work.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[2] Cadence. *Cadence LC User's Manual*.

[3] Cadence. *Cadence NC-Verilog User's Manual*.

[4] Cadence. *Cadence SOCEncounter User's Manual*.

[5] N. Choudhary, S. Wadhavkar, T. Shah, H. Mayukh, J. Gandhi, B. Dwiel, S. Navada, H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *ISCA*, 2011.

[6] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *JILP*, 2005.

[7] S. Hauck. Asynchronous design methodologies: an overview. *Proc. of the IEEE*, 83(1):69–93, 1995.

[8] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous locally synchronous processors. In *ISCA*, pages 158–168, 2002.

[9] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. Lumetta, M. Frank, and S. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA*, pages 140–151, 2009.

[10] S. Lee, S. Das, T. Pham, T. Austin, D. Blaauw, and T. Mudge. Reducing pipeline energy demands with local dvs and dynamic retiming. In *ISLPED*, pages 319–324, 2004.

[11] X. Liang, G. Wei, and D. Brooks. Revival: A variation-tolerant architecture using voltage interpolation and variable latency. In *ISCA*, pages 191–202, 2008.

[12] W. Liao, L. He, and K. Lepak. Temperature and supply voltage aware performance and power modeling at microarchitecture level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7):1042–1053, 2005.

[13] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA*, pages 29–40, 2002.

[14] Sun. *Sun OpenSPARC Project*.

[15] Synopsys. *Synopsys Design Compiler User's Manual*.

[16] Synopsys. *Synopsys PrimeTime User's Manual*.

[17] Texas Instruments. *16-Bit Ultra-Low Power MSP430 Microcontrollers*.

[18] S. Thozlyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. Cacti 5.1. Technical report, HP Labs, 2008.

[19] A. Tiwari, S. Sarangi, and J. Torrellas. Recycle: pipeline adaptation to tolerate process variation. In *ISCA*, pages 323–334, 2007.

[20] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, 1996.

[21] Q. Wu, P. Juang, M. Martonosi, and D. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS*, pages 248–259, 2004.