Mobile Ecosystem Driven Dynamic Pipeline Adaptation for Low Power

Garo Bournoutian[1,2(✉)] and Alex Orailoglu[2]

[1] Qualcomm Technologies, Inc., San Diego, CA 92121, USA
garo@cs.ucsd.edu
[2] University of California, San Diego, La Jolla, CA 92093, USA
alex@cs.ucsd.edu

Abstract. State-of-the-art mobile smartphone and tablet processors are beginning to employ fully speculative, out-of-order architectures with deep instruction pipelines. These processors often have pipeline lengths of 24 or more stages. Furthermore, to improve high-performance ILP, these processors provide multiple parallel pipeline paths for various instruction types. These architectures provide multiple execution clusters defined by instruction type, each with its own issue queue. Instructions are dispatched to one of the appropriate issue queues, and all issue queues are then scanned in parallel to identify instructions ready for execution. The goal of such a resource-intensive architectural design is to sustain peak processor performance. Unfortunately, applications oftentimes only leverage a small subset of these robust computation resources, and the excess hardware resources still consume power while idle. This paper proposes a novel methodology that leverages the unique characteristics of the mobile ecosystem to drive hardware adaptation for a power-efficient execution pipeline microarchitecture. The proposed architecture will monitor the run-time execution behavior in order to enable only those pipeline resources that are currently needed, allowing the system to rapidly respond to changing resource demands to ensure performance is maintained while reducing power consumption. The simulation results show that processor performance is maintained while achieving a significant reduction in execution pipeline power consumption.

Keywords: Mobile · Low-power · Dynamic · Adaptive hardware · Pipeline

1 Introduction

High-performance mobile processors are beginning to employ heterogeneous processor topologies in order to provide a continuum of computational resources that can handle the wide range of variability that occurs in the mobile domain. These heterogeneous processor topologies typically utilize a cluster of "Little" processor cores optimized for low-power, as well as a cluster of "Big" processor cores targeting higher performance at the cost of higher power dissipation. An example of

L.M. Pinho et al. (Eds): ARCS 2015, LNCS 9017, pp. 83–95, 2015.
DOI: 10.1007/978-3-319-16086-3_7

such a topology is the ARM big.LITTLE architecture, which incorporates high-performance Cortex-A15 "big" processors with low-power Cortex-A7 "LITTLE" processors [6].

In general, the "Little" processor leverages an in-order architecture with a simple pipeline. For example, the ARM Cortex-A7 has a pipeline length of 8–10 stages. On the other hand, the "Big" processor usually employs a fully speculative, out-of-order architecture with a deeper pipeline. For comparison, the ARM Cortex-A15 supports register renaming and has a pipeline length of 15–24 stages. Furthermore, to improve high-performance ILP the Cortex-A15 provides multiple, parallel pipeline paths for various instruction types. These microarchitectural differences are one of the main reasons for the large increase in energy consumption compared to the "Little" processor.

In this paper, we propose reducing the power consumption of these more power-hungry "Big" processor cores by dynamically adapting the instruction pipeline. Leveraging the unique characteristics of high-performance mobile processor architectures, a fine-grained adaptive hardware control mechanism is developed. The microarchitecture will automatically shut down individual pipeline paths during periods of reduced utilization. Upon subsequent demand, these pipeline paths can be re-enabled in a manner that avoids any performance penalties. Furthermore, the aggressiveness of shutting down pipeline paths will be guided by an application-specific code analysis. The result will be a significant reduction of wasted power from idle pipeline paths without any negative impacts to performance.

2 Related Work

A good amount of prior research has been done related to reducing execution pipeline power in general purpose processors while incurring a minor performance degradation. A common approach relies on resizing the issue queue in order to control the rate of execution of the processor pipeline.

The authors in [11] proposed an architecture allowing the sizes of the issue queue, reorder buffer, and the load/store queue to be dynamically adjusted. They employed periodic sampling of occupancy levels to determine when to increase or decrease capacity. Similarly, the authors in [5] present an issue queue design that allows for dynamic configurability of size and speed using transmission gate insertion. The circuit also gathers activity statistics during execution to allow on-the-fly adjustments to improve energy and performance. Both of these approaches rely on costly run-time profiling techniques that consume power to keep track of such statistical information.

The authors in [10] propose a mechanism to disable one or more processor pipelines based on dynamically monitoring the processor's performance. They focus on an Alpha 21264 processor with two integer pipeline clusters and a single floating-point pipeline. In a similar vein, *Pipeline Balancing* was proposed in [2], which dynamically monitors performance and adjusts the issue rate accordingly.

The implementation proposed by the authors necessitates that at least one cluster of functional units for each ISA type remain active, limiting the amount of power savings since the minimum issue rate will be 4 per cycle.

A proposal to power-gate execution units to abate leakage power was proposed in [8]. The authors provide analytical equations for determining break-even points, and then apply this information to put specific execution units to sleep based on elapsed time or branch misprediction events.

A software-assisted approach to dynamically resizing the issue queue was presented in [9]. Compile-time analysis provides information on the required number of issue queue entries. Unfortunately, the proposed static analysis does not handle inter-procedural dependence analysis, limiting the applicability of the algorithm in the presence of function calls.

3 Motivation

The larger, high-performance cores in the heterogeneous processor topology typically consist of an out-of-order architecture with a relatively deep pipeline. The incentive of having an out-of-order processor is to allow execution around data hazards in order to improve performance. The effectiveness of such an architecture is often limited by how far it can "look ahead" by placing decoded instructions into an issue queue used to identify those instructions whose dependencies are completely resolved. For high-end targets, having a window size of 40 or more instructions is often required to meet performance targets. Unfortunately, it is also common knowledge in the mobile industry that the issue queue size is frequency limited to about 8 entries, which severely limits the effectiveness of the architecture. The physical design of issue queues larger than 8 entries incurs longer critical path timing to concurrently scan all entries and route necessary data.

In order to overcome this limitation, it is common practice in mobile microarchitectures to employ multiple, smaller issue queues. The execution is broken down into multiple clusters defined by instruction type, each with its own issue queue. Instructions will be dispatched to the appropriate issue queue, and all

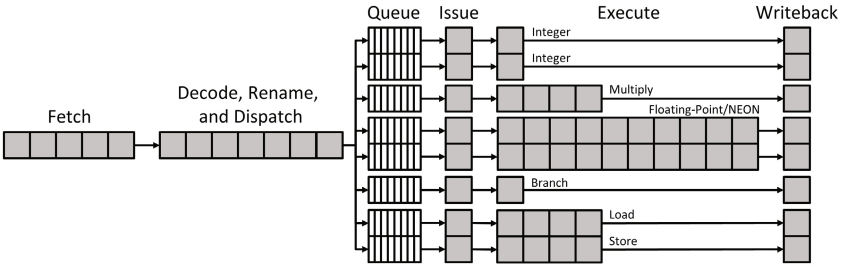

Fig. 1. Typical Mobile Out-of-Order Processor Pipeline

issue queues are then scanned in parallel to identify instructions ready for execution. A typical mobile out-of-order processor pipeline is shown in Figure 1, which is similar to that used in the ARM Cortex-A15 and Cortex-A57, Apple A6 Swift and A7 Cyclone, and Qualcomm Krait 400 and Krait 450 processors.

As one can see, after instructions are decoded, they will be dispatched into an issue queue for the appropriate instruction type. Once the instruction's dependencies are resolved, it will be issued and executed. Each pipeline has its own separate issue queue and can issue independently from the other pipelines. Certain instruction types can have more than one pipeline in order to increase parallelism by ameliorating structural hazards. For example, in the architecture shown in Figure 1, the *Integer* and *Floating-Point* instruction types are provided two parallel pipelines.

Under ideal circumstances, the instruction mix of an application will be well-balanced and essentially match the physically available pipeline functional units. In this case, the pipeline will deliver a substantial amount of ILP and hardware resources will be well utilized. Unfortunately, it is rare for an application to follow this ideal. For example, one may have an application that completely avoids using any floating-point instructions. Yet, the physical hardware for two entire floating-point pipelines is present. During the course of executing this application, a large portion of the hardware will be idle and wasting precious battery life. Even while nothing is executing within these pipelines a substantial amount of power is consumed by the issue queue logic checking for valid entries to issue, and the functional units themselves consuming power as they idle.

To demonstrate the possible skewed distribution of instruction types, the instruction mix of all SPEC CPU2000 [13] integer and floating-point benchmarks is shown in Figure 2 and Figure 3, respectively. As one would expect, the integer benchmarks very rarely make use of any floating-point operations. Thus, having some mechanism to disable the floating-point pipelines will clearly

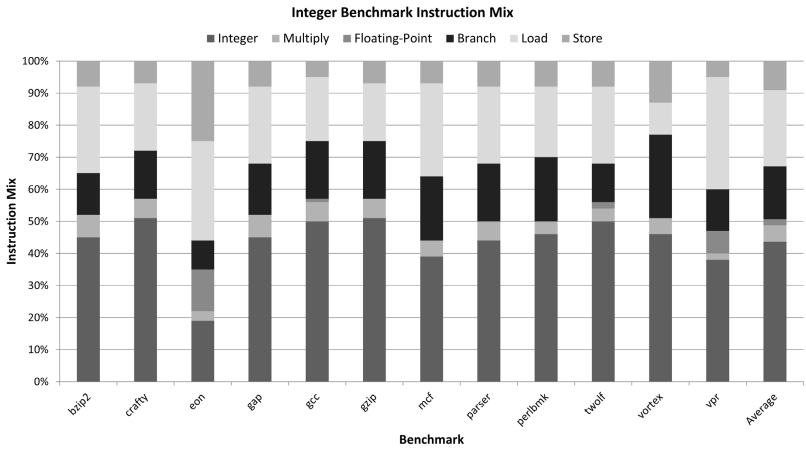

Fig. 2. SPEC Integer Benchmark Instruction Mix

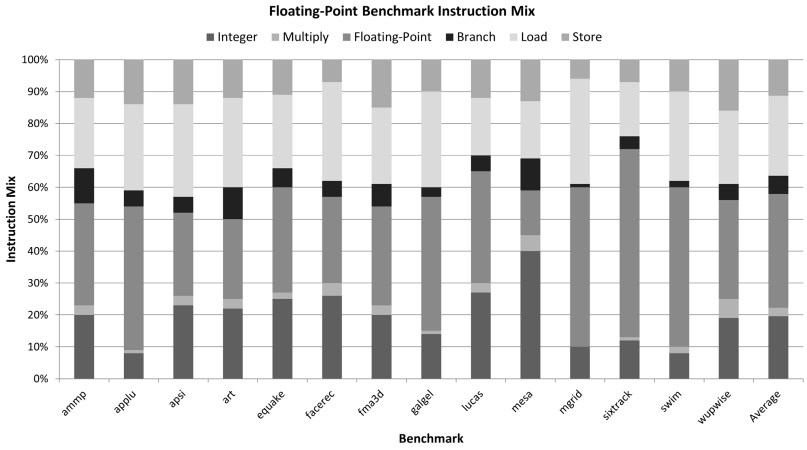

Fig. 3. SPEC Floating-Point Benchmark Instruction Mix

help to conserve power. Furthermore, for those instruction types with multiple parallel pipelines, a given application may be unable to actually exploit any ILP benefit of this additional hardware unless a sufficiently large amount of those instructions is present within the execution window.

Given these observations, it becomes clear that an adaptive approach is necessary to help tune the microarchitecture in order to conserve power. Each individual instruction pipeline should be monitored to determine if there is a sufficient demand for keeping it enabled. If these structures remain idle, an automated mechanism should exist to shut down the issue queue and execution logic in order to avoid both dynamic and static power dissipation.

As one would expect, there are trade-offs to such a dynamically reconfigurable microarchitecture. For instance, monitoring the pipeline activity patterns during run time to detect idleness requires additional hardware and power. Similarly, the process of disabling and re-enabling a pipeline path can incur both power and performance penalties. In particular, if the logic to re-enable a path is not accurately predicted, the pipeline will stall and waste even more power. Our goal is to intelligently exploit the unique characteristics of mobile processor architectures to minimize or completely eliminate these overheads.

4 Implementation

In order to conserve power, one would like to dynamically adapt the hardware pipeline to best match the computational needs of a specific application. This application-specific tailoring of the hardware microarchitecture needs to be fine-grained and able to efficiently respond to changes in the application's execution patterns. Furthermore, it is essential that any proposed additional hardware logic itself be frugal so as not to countermand the reductions in power we are trying

to achieve. The following subsections will describe the hardware architecture to enable a fine-grained, adaptive instruction pipeline as well as a software-driven approach to determining the aggressiveness of the hardware's adaptation.

4.1 Hardware Architecture

The proposed hardware architecture to enable a fine-grained, adaptive instruction pipeline is shown in Figure 4. This figure illustrates the various hardware mechanisms that will be added to each individual instruction pipeline. As shown in Figure 1, there are typically 8 separate instruction pipelines, each of which will follow this same approach.

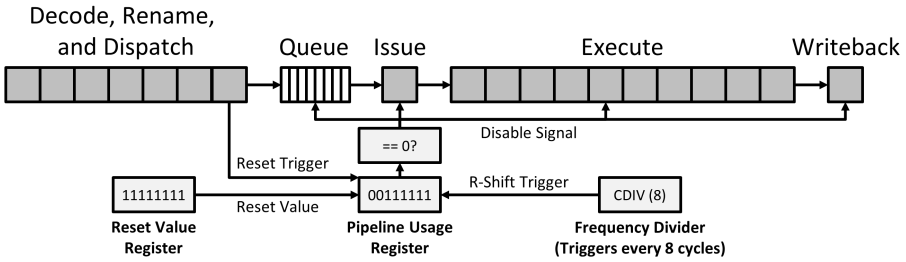

Fig. 4. Proposed Adaptive Pipeline Architecture

The first new hardware structure is a *Pipeline Usage Register*. The purpose of this register is to indicate the temporal utilization of a given pipeline path. This register will initially be populated with a non-zero value, and over time that value will decrease towards zero if the pipeline is idle. Once this register becomes zero, a *Disable Signal* will be asserted high, which will disable the issue queue, issue stage, execution stage, and writeback stage of the pipeline. These disabled hardware structures will have their supply voltage gated as described in [12], obviating both dynamic switching power and any static leakage power.

The *Pipeline Usage Register* is implemented as a shift register. Upon receiving the *R-Shift Trigger* signal, the *Pipeline Usage Register* is right shifted and fed a most-significant-bit (MSB) value of 0. The *R-Shift Trigger* signal will be generated by a *Frequency Divider* that will take the pipeline clock and divide it by 8. In this fashion, the *R-Shift Trigger* signal will occur every 8 pipeline cycles. This mechanism will be what determines that a given pipeline has been idle for a period of time and causes the *Pipeline Usage Register* to become zero, in turn shutting down the pipeline hardware structures.

The reciprocal logic that marks that a pipeline is actively in use is controlled by the *Reset Trigger* signal. This reset signal will be asserted during the decode stage four cycles before the instruction is dispatched into an issue queue. Upon the *Reset Trigger* signal going high, the *Pipeline Usage Register* will be reset to the value specified in the *Reset Value Register*. This logic not only handles the

case of ensuring that an active pipeline is not turned off by keeping the *Pipeline Usage Register* non-zero, but it also handles re-enabling a disabled pipeline without causing any performance penalty. The decode stage typically takes multiple cycles (7 in our example architecture), since the instruction is not only decoded, but other logic like register renaming is done as well. Given this, the instruction type (i.e. integer, floating-point, branch, etc.) is determined rather early in the decode stage (based on the instruction opcode), and this information can be conveyed to the *Pipeline Usage Register* while things like register renaming are being done. Based on this observation, the *Reset Trigger* signal can be generated four cycles prior to the instruction being dispatched into an issue queue. This allows a possibly disabled issue queue to be fully re-enabled before a dispatched instruction is sent to it.

The *Reset Value Register* will possess the property of having a continuous run of 1's of some length L starting from the LSB. The longer the length L is, the longer the pipeline must remain idle before it will cause an automatic shutdown. This structuring of the *Reset Value Register* will minimize bit-flipping transitions within the *Pipeline Usage Register* (avoiding needless dynamic power consumption). Since the *Pipeline Usage Register* is right-shifted over and over, each right-shift will only incur a single bit-flip.

A further power reducing optimization is the circuitry to determine when the *Pipeline Usage Register* is zero. Instead of using a relatively expensive comparator circuit, all the bits within the *Pipeline Usage Register* can simply be fed into a *NOR* gate, which will become 1 only when all the input bits are 0.

Lastly, we can further optimize the instruction types with multiple pipelines (e.g. *Integer* and *Floating-Point*). In these cases, there may not be sufficient instructions to merit having two parallel pipeline paths. In order to gracefully account for this situation, the dispatch logic will be slightly updated. Instead of randomly selecting one of the issue queues to dispatch the instruction to, the multiple issue queues will be prioritized. Only when the first issue queue is full will the instruction be dispatched into the next queue. In this fashion, if one issue queue is able to support the instruction stream without becoming inundated, it will cause the second issue queue to remain idle, causing it to turn off. If the instruction demand increases and spills over the first issue queue, then the second issue queue can service the instructions as before. Performance will remain unaffected by this change. Rather, this prioritization of issue queues helps essentially defragment and compress the instructions into one queue before needing to expand into another, helping the second pipeline remain idle and thus be possibly turned off to conserve power.

To ensure any active executions have sufficient time to complete and exit before we shut down the pipeline, the *Reset Value Register* will be required to have at least two 1's in the LSBs. This will ensure at least two right-shift intervals occur before the pipeline is shut down, where each interval occurs after 8 pipeline cycles based on the *Pipeline Usage Register*'s clock divider. Given that the longest pipeline stage in our design is the *Floating-Point* pipeline, taking up to 12 cycles once leaving the issue queue, having two intervals of 8 pipeline

cycles ensures that the last instruction in the pipeline has completed before the pipeline is disabled. This helps greatly simplify the shutdown logic, since no querying within the execution stages needs to occur before we turn off the pipeline. For pipeline stalls, the same stalling mechanism will cause the clock divider to also not move forward ensuring consistency in the timing.

4.2 Software-Driven Reset Thresholds

The prior section described the microarchitectural design to enable the adaptive instruction pipeline. However, instead of arbitrarily selecting the value for the *Reset Value Register* and hard-coding it across all the pipelines and even across different applications, one would like to make a more intelligent selection. Looking back at the instruction mixes shown for the benchmarks in Figure 2 and Figure 3, a logical extension would be to leverage this information to help guide the selection of the reset threshold value.

The general observation is that if the quantity of a particular instruction type is quite low, there is a higher probability that the pipeline for that instruction type will be idle. Furthermore, when the rare instruction type does occur, it most likely will be sporadic and shutting down the pipeline sooner rather than later can help maximize power savings. Thus, it is proposed to set the *Reset Value Register* to the following values based on the relative percentage of the instruction type (IT), where N is the size of the *Reset Value Register* in bits:

- If $IT < 5\%$, *Reset Value Register* $= \{\{(N-2)\{0\}\}, 1, 1\}$
- Else if $IT < 20\%$, *Reset Value Register* $= \left\{\left\{\frac{(N-2)}{2}\{0\}\right\}, \left\{\frac{(N-2)}{2}\{1\}\right\}, 1, 1\right\}$
- Else, *Reset Value Register* $= \{\{(N-2)\{1\}\}, 1, 1\}$

We examine two different software-based approaches to estimating instruction type density. The first approach is a pure compile-time code analysis to get static instruction type counts. In order to determine the instruction type distribution of a mobile smartphone application including all foundation libraries, an on-device code analysis framework is employed [3]. A simple post-processing script can then analyze and identify the opcodes for each instruction type. The second approach instead relies on profiling an actual execution of the application in order to garner dynamic instruction type counts. This latter approach will help identify hotspot patterns wherein loops may greatly increase the overall predominance of a small number of static instructions. In both of these approaches, the

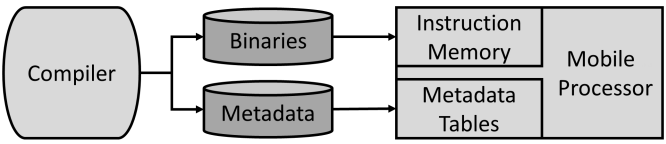

Fig. 5. Overview of Compiler and Hardware Interaction

pipeline-specific *Reset Value Register* number computed based on instruction type will be passed to the underlying hardware microarchitecture via metadata embedded within the software executable. Upon the application being loaded, the metadata will populate the values for the various hardware *Reset Value Registers* associated with each instruction type. An overview of this architecture is shown in Figure 5.

5 Experimental Results

In order to assess the benefit from this proposed architectural design, we utilized the SimpleScalar framework [1]. The stock code initially utilized a basic register update unit (RUU) structure, combining the reorder buffer (ROB) and reservation stations and provided no register renaming. In order to match the target architecture and fully exploit possible parallelism and instruction throughput, the default *sim-outorder* simulator was modified to implement a full speculative Tomasulo architecture [7], including register renaming and decentralized issue queues stations. Furthermore, the simulator is augmented with the adaptive instruction pipeline logic proposed in this paper and also incorporates a heavily customized version of the Wattch power analysis framework [4]. The size N of the *Pipeline Usage Register* and *Reset Value Register* was chosen to be 8 bits.

Table 1 summarizes the system configuration parameters, reflecting a typical high-performance mobile processor.

Table 1. Hardware Configuration Parameters

Fetch Stages	5
Decode Stages	7
Issue Stages	1
Execution Stages, INT	1
Execution Stages, MULT	4
Execution Stages, FP ADD/SUB	2
Execution Stages, FP MUL	6
Execution Stages, FP DIV	10
Execution Stages, BRANCH	1
Execution Stages, LD/ST	4
Writeback Stages	1
Issue Queue Entries	8
Instruction L1 cache	64 KB, 4-way set-associative
Data L1 cache	64 KB, 4-way set-associative
Unified L2 cache	2MB, 8-way set-associative
Number of Pipelines	2 INT, 1 MULT, 2 FP 1 BRANCH, 1 LD, 1 ST

The complete SPEC CPU2000 benchmark suite [13] is used, providing 12 integer and 14 floating-point real-world applications. The benchmarks are cross-compiled for the PISA instruction set using the highest level of optimization available for the language-specific compiler. The reference inputs are used for each benchmark, with each benchmark executed in its entirety from start to

finish. For the approach of using profiled application information, the training inputs for each benchmark are used.

In order to assess the proposed architecture, the complete pipeline power utilization is analyzed, including both dynamic and static power. The additional power overhead incurred for enabling the adaptive pipeline, such as from registers, control logic, and transitioning pipelines off and on, are also incorporated into the results.

Figure 6 shows the power reduction across the pipeline logic observed for the integer benchmarks. The average power savings for integer benchmarks using pure static analysis is 27.11%, while execution profiling yields a slightly better average of 27.41%. In a majority of the integer benchmarks, the *Reset Value Register* threshold selected in both pure static analysis and execution profiling turns out to be the same. Additionally, while the execution profiling approach does better on average, in the *crafty* benchmark the improvement was actually worse than pure static analysis. In this case, it is likely that the training inputs used during profiling had a significantly different dynamic instruction distribution compared to the reference inputs used during actual benchmarking.

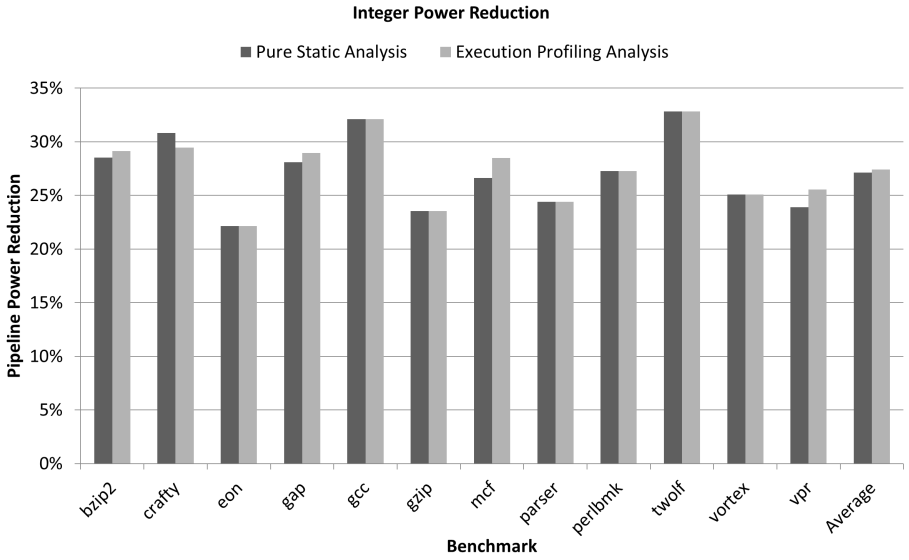

Fig. 6. Integer Benchmarks Pipeline Power Reduction

Similarly, the power savings for floating-point benchmarks are shown in Figure 7. The average power savings using pure static analysis is approximately 18.80%, while using execution profiling garners a 19.12% average reduction. Again, a majority of the benchmarks ended up having the same *Reset Value Register* threshold based on static analysis and execution profiling. The *lucas*

benchmark was another exception to the execution profiling approach outperforming the pure static analysis approach. Additionally, as one would expect, the benefit for floating-point benchmarks is slightly less than the integer case, since the floating-point pipeline will need to be enabled much more often. However, the adaptive logic is still able to identify excess hardware resources that may occur throughout the execution lifetime and intelligently turn them off to save power.

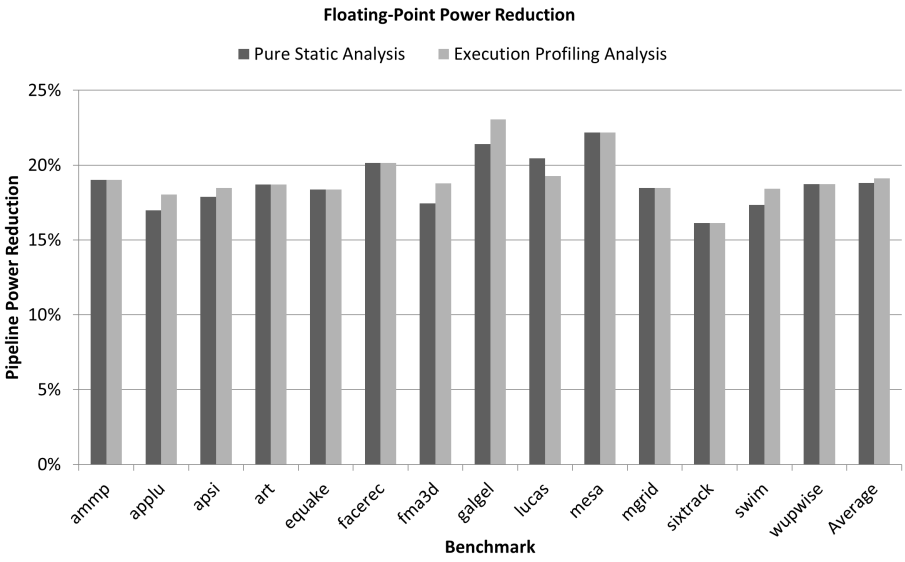

Fig. 7. Floating-Point Benchmarks Pipeline Power Reduction

As one can see, the proposed architecture is able to eliminate a substantial amount of power consumption by adaptively disabling portions of the pipeline when they are not actively needed. In particular, most of the integer benchmarks rarely, if ever, use the floating-point pipeline, which can account for approximately 20% of the energy in the pipeline logic. Furthermore, these power savings come at no cost to processor performance. Instruction throughput is not degraded using this proposal, since any time disabled pipeline resources are needed they will be immediately re-enabled four cycles ahead of time.

6 Conclusions

High-performance mobile processors typically employ more power-hungry out-of-order processors with deep pipelines in order to meet peak demands. However, not all applications possess the exact same instruction mix, leading to uneven physical resource allocations within the process pipeline. Given that power is a

critical concern for mobile devices, conserving power without negatively impacting performance is a top priority.

A novel adaptive instruction pipeline architecture for mobile processors has been presented. This adaptive architecture leverages the unique characteristics of high-performance mobile processor microarchitecture design to propose a frugal dynamically adaptive mechanism to enable fine-grained pipeline gating. Based on run time utilization, idle pipelines and associated issue queues are turned off to reduce dynamic and leakage power. The proposed microarchitecture automatically shuts down individual pipeline paths during periods of reduced utilization. Upon subsequent demand, these pipeline paths are preemptively re-enabled to completely avoid any performance penalties. Application-specific code analysis is also leveraged to guide the aggressiveness of the pipeline gating. The results demonstrate a substantial amount of power savings can be achieved without any impact to performance.

References

1. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. Computer **35**(2), 59–67 (2002)
2. Bahar, R., Manne, S.: Power and energy reduction via pipeline balancing. In: ISCA 2001: Proceedings of the 28th Annual International Symposium on Computer Architecture, pp. 218–229 (2001)
3. Bournoutian, G., Orailoglu, A.: On-device objective-C application optimization framework for high-performance mobile processors. In: DATE 2014: Proceedings of the 2014 Design, Automation Test in Europe Conference Exhibition, pp. 85:1–85:6 (2014)
4. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. In: ISCA 2000: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 83–94 (2000)
5. Buyuktosunoglu, A., Albonesi, D., Schuster, S., Brooks, D., Bose, P., Cook, P.: A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In: GLSVLSI 2001: Proceedings of the 11th Great Lakes Symposium on VLSI, pp. 73–78 (2001)
6. Greenhalgh, P.: big.LITTLE processing with ARM Cortex-A15 & Cortex-A7, May 2013. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf
7. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Fifth edn (2011)
8. Hu, Z., Buyuktosunoglu, A., Srinivasan, V., Zyuban, V., Jacobson, H., Bose, P.: Microarchitectural techniques for power gating of execution units. In: ISLPED 2004: Proceedings of the 2004 International Symposium on Low Power Electronics and Design, pp. 32–37 (2004)
9. Jones, T., O'Boyle, M., Abella, J., Gonzalez, A.: Software directed issue queue power reduction. In: HPCA 2005: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 144–153 (2005)
10. Maro, R., Bai, Y., Bahar, R.I.: Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In: Falsafi, B., VijayKumar, T.N. (eds.) PACS 2000. LNCS, vol. 2008, p. 97. Springer, Heidelberg (2001)

11. Ponomarev, D., Kucuk, G., Ghose, K.: Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In: MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 90–101 (2001)
12. Powell, M., Yang, S., Falsafi, B., Roy, K., Vijaykumar, T.: Gated-V_{dd}: a circuit technique to reduce leakage in deep-submicron cache memories. In: ISLPED 2000: Proceedings of the 2000 International Symposium on Low Power Electronics and Design, pp. 90–95 (2000)
13. SPEC: SPEC CPU2000 Benchmarks (2000). http://www.spec.org/cpu/