COMPUTER ARCHITECTURE TECHNIQUES FOR POWER-EFFICIENCY

Computer Architecture Techniques for Power-Efficiency

Stefanos Kaxiras and Margaret Martonosi

www.morganclaypool.com

ISBN: 9781598292084 paper
ISBN: 9781598292091 ebook

DOI: 10.2200/S00119ED1V01Y200805CAC004

A Publication in the Morgan & Claypool Publishers series

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #4

Lecture #4

Series Editor: Mark D. Hill, University of Wisconsin, Madison

Library of Congress Cataloging-in-Publication Data

Series ISSN: 1935-3235 print
Series ISSN: 1935-3243 electronic

COMPUTER ARCHITECTURE TECHNIQUES FOR POWER-EFFICIENCY

Stefanos Kaxiras
University of Patras, Greece
Kaxiras@ece.upatras.gr

Margaret Martonosi
Princeton University
mrm@princeton.edu

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

In the last few years, power dissipation has become an important design constraint, on par with performance, in the design of new computer systems. Whereas in the past, the primary job of the computer architect was to translate improvements in operating frequency and transistor count into performance, now power efficiency must be taken into account at every step of the design process.

While for some time, architects have been successful in delivering 40% to 50% annual improvement in processor performance, costs that were previously brushed aside eventually caught up. The most critical of these costs is the inexorable increase in power dissipation and power density in processors. Power dissipation issues have catalyzed new topic areas in computer architecture, resulting in a substantial body of work on more power-efficient architectures. Power dissipation coupled with diminishing performance gains, was also the main cause for the switch from single-core to multi-core architectures and a slowdown in frequency increase.

This book aims to document some of the most important *architectural* techniques that were invented, proposed, and applied to reduce both dynamic power and static power dissipation in processors and memory hierarchies. A significant number of techniques have been proposed for a wide range of situations and this book synthesizes those techniques by focusing on their common characteristics.

KEYWORDS

Computer power consumption, computer energy consumption, low power computer design, computer power efficiency, dynamic power, static power, leakage power, dynamic voltage/frequency scaling, computer architecture, computer hardware.

Contents

In this equation, V refers to the supply voltage, while V_{th} refers to the threshold voltage. The exponential link between leakage power and threshold voltage is immediately obvious.[1] Lowering the threshold voltage brings a tremendous increase in leakage power. Unfortunately, lowering the threshold voltage is what we have to do to maintain the switching speed in the face of lower supply voltages. Temperature, T, is also an important factor in the equation: leakage power depends exponentially on temperature. The remaining parameters, q, a, and k_a, summarize logic design and fabrication characteristics. The exponential dependence of leakage on temperature, and the interplay between leakage and dynamic energy will be discussed in more detail in Chapter 2.

1.2.3 Other Forms of CMOS Power Dissipation

While dynamic and leakage power dominate the landscape, other forms of power dissipation do exist. For example, short circuit or "glitching" power refers to the power dissipated during the brief transitional period when both the n and p transistors of a CMOS gate are "on," thus forming a short-circuit path from power to ground. This is distinguished from dynamic power because dynamic power typically refers to power dissipated due to discharging charged capacitors; it would be dissipated even if transitions occurred instantaneously. In contrast, glitching power refers to transitional power that occurs because of non-ideal transition times.

1.3 POWER-AWARE COMPUTING TODAY

From the early 1990s to today, power consumption has transitioned into a primary design constraint for nearly all computer systems. In mobile and embedded computing, the connection from power consumption to battery lifetime has made the motivation for power-aware computing very clear. Here, it is typically low energy that is stressed, although obviously power/performance is also important for some embedded systems with high computational requirements.

In desktop systems, the key constraint has become thermal issues. Excessive power consumption is one of the prevailing reasons for the abrupt halt of clock frequency increases. Currently, high-performance processor clocks have hit a "power wall" and are pegged at or below 4 GHz. This is contrary to 2001 ITRS projections which predicted clocks in excess of 6 GHz by roughly 2006. Power consumption is also one important factor driving the adoption of chip multiprocessors (CMPs) since they allow high-throughput computing to be performed within cost-effective power and thermal envelopes.

[1]What is not shown in this simplified equation is the—also exponential—dependence of leakage power to the supply voltage. This is discussed in Chapter 5.

CHAPTER 3

Using Voltage and Frequency Adjustments to Manage Dynamic Power

Issues addressing dynamic power have predominated the power-aware architecture landscape. Amongst these dynamic power techniques, methods for addressing voltage and frequency have dominated in turn. Most of these methods have focused on dynamic adjustments to supply voltage, clock frequency, of both, and they go under the broad title of *Dynamic Voltage and Frequency Scaling*, or *DVFS*. This chapter discusses the motivation for these techniques overall, and gives examples drawn from different categories of techniques.

Chapter Structure: Decisions to engage voltage and frequency scaling are made at various levels. The decision level, the level of the control policy, defines the structure of this chapter. Starting from the top, the system (or operating system) level, the chapter unfolds to progressively more focused levels: program (or program phase) level and the hardware (flip-flop) level. The following section (Section 3.1) gives an overview of voltage/frequency scaling and discusses a number of issues pertaining to the corresponding techniques.

3.1 DYNAMIC VOLTAGE AND FREQUENCY SCALING: MOTIVATION AND OVERVIEW

The basic dynamic power equation: $P = CV^2Af$ clearly shows the significant leverage possible by adjusting voltage and frequency [47, 101]. If we can reduce voltage by some small factor, we can reduce power by the square of that factor. Reducing supply voltage, however, might possibly reduce the performance of systems as well. In particular, reducing supply voltage often slows transistors such that reducing the clock frequency is also required. The benefit of this is that within a given system, scaling supply voltage down now offers the potential of a cubic reduction in power dissipation. The downside of this is that it may also linearly degrade performance. If the program runs at lower power dissipation levels, but for longer durations, then the benefit in terms of total energy will not be cubic. It is interesting to note that while voltage/frequency

scaling improves EDP (because the reduction in power outpaces the reduction in performance), it can do no better than break even on the ED^2P metric (and this, only when the scaling factors for frequency and voltage are the same).

Nonetheless, DVFS is appealing first because max-power limits may welcome max-power reductions even if the energy is not reduced much. In addition, DVFS is appealing because often we can discern ways, as this chapter will discuss, to reduce clock frequency without having the workload experience a proportional reduction in performance.

3.1.1 Design Issues and Overview

From an architect's perspective, key design issues for DVFS include the following:

(1) At what level should the DVFS control policies operate? Fundamentally, DVFS approaches exploit *slack*. Slack can appear at different levels and various DVFS approaches have been proposed for each level. Approaches operating at the same level share a similar set of mechanisms, constraints, and available information. We can discern three major levels where DVFS decisions can be made:

- *System-level based on system slack*: At this level, the *idleness* of the whole system is the factor that drives DVFS decisions (Section 3.2). In many cases, decisions are taken according to system load. The *whole* processor (or embedded system, wireless system, etc.) is typically voltage/frequency scaled to eliminate idle periods.
- *Program- or program-phase-level based on instruction slack*: Here, decisions are taken according to program (or program phase) behavior (Section 3.3 for a single clock domain and Section 3.4 for multiple clock domains). *Instruction Slack* due to long-latency memory operations is typically exploited at this level for DVFS in single-threaded programs. In multi-core processors, the ability to run parallel (multi-threaded) programs opens up the possibility for the *parallel* program behavior to drive voltage/frequency decisions.
- *Hardware-level based on hardware slack*: Finally, a recent approach, called *Razor*, goes below the program level, right to the hardware (Section 3.5). Razor tries to exploit *slack* hidden in hardware operation. This slack exists because of *margins* needed to isolate each hardware abstraction layer from variations in lower levels. This slack is exploited similarly to the way idle time is exploited at the system level.

(2) How will the DVFS settings be selected and orchestrated? In some cases, DVFS approaches may allow software to adjust a register which encodes the desired (V, f) setting. In other cases, the choices will be made dynamically "under the covers" by hardware mechanisms

alone. In either scenario, research questions arise regarding whether to make offline (e.g., compile-time) decisions about DVFS settings, versus online, reactive, approaches.

(3) What is the hardware granularity at which voltage and frequency can be controlled? This question is closely related to the question above. The bulk of the DVFS research has focused on cases in which the entire processor core operates at the same (V, f) setting but is asynchronous to the "outside" work, such as main memory. In such scenarios, the main goal of DVFS is to capitalize on cases in which the processor's workload is heavily memory-bound. In these cases, the processor is often stalled waiting on memory, so reducing its supply voltage and clock frequency will reduce power and energy without having significant impact on performance.

Other work has considered cases in which multiple clock domains may exist on a chip. These so-called MCD scenarios might either be multiple clock domains within a single processor core [199, 200, 216, 227, 228] or chip multiprocessors in which each on-chip processor core has a different voltage/clock domain [67]. This dimension is explored in Section 3.4.

(4) How do the implementation characteristics of the DVFS approach being used affect the strategies to employ? Some of the implementation characteristics for DVFS can have significant influence on the strategies an architect might choose, and the likely payoffs they might offer. For example, what is the delay required to engage a new setting of (V, f)? (And, can the processor continue to execute during the transition from one (V, f) pair to another?) If the delay is very short, then simple reactive techniques may offer high payoff. If the delay is quite long, however, then techniques based on more intelligent or offline analysis might make more sense.

(5) How does the DVFS landscape change when considering parallel applications on multiple-core processors? When considering one, single-threaded application in isolation, one need only consider the possible asynchrony between compute and memory. In other regards, reducing the clock frequency proportionately degrades the performance. In a parallel scenario, however, reducing the clock frequency of one thread may impact other dependent threads that are waiting for a result to be produced. Thus, when considering DVFS for parallel applications, some notion of critical path analysis may be helpful.

Another similar question regards whether continuous settings of (V, f) pairs are possible, or whether these values can only be changed in fixed, discrete steps. If only discrete step-wise adjustments of (V, f) are possible, then the optimization space becomes difficult to navigate because it is "non-convex." As a result, simple online techniques might have difficulty finding global optima, and more complicated or offline analysis again becomes warranted.

Because DVFS is available for experimentation on real systems [111, 112, 2], and because it offers such high leverage in power/energy savings, it has been widely studied in a variety of communities. Our discussion only touches on some of the key observations from the *architectural*

micro-operation," and used this as an indicator of the memory boundedness indicative of likely DVFS effectiveness. For each pattern of past behavior stored in a history entry, a different prediction of next-step behavior can be made. For each next-step prediction, there is a one-to-one mapping to an appropriate DVFS setting. If the DVFS setting is different from the current setting, then the V, f are adjusted accordingly. When guided by the GPHT, DVFS was found to achieve EDP improvements as high as 34% for the highly variable benchmarks that this approach targets.

3.4 PROGRAM-LEVEL DVFS FOR MULTIPLE-CLOCK DOMAINS

Some of the early architectural work on DVFS actually focused on opportunities within multiple-clock-domain (MCD) processors. The rationale for MCD processors is that as feature sizes get smaller, it becomes more difficult and expensive to distribute a global clock signal with low skew through the processor die. Thus, researchers have explored globally-asynchronous locally-synchronous (GALS) techniques.

Scaling voltage/frequency independently for each clock domain within a processor can be done dynamically (Section 3.4.1) or statically (Section 3.4.2); both cases aim to exploit *slack* in the execution of individual instructions.

Finally, the emerging architectural paradigm for deep sub-micron technologies, the multi-core paradigm, can be considered as an MCD design where synchronous cores operate asynchronously to each other. DVFS techniques for multi-cores are discussed in Section 3.4.3.

3.4.1 DVFS for MCD Processors

In GALS approaches, a processor core is divided into synchronous islands, each of which is then interconnected asynchronously but with added circuitry to avoid metastability. The islands are typically intended to correspond to different functional units, such as the instruction fetch unit, the ALUs, the load-store unit, and so forth. A typical division is shown in Figure 3.5.

In early GALS DVFS work, Marculescu and her students considered the performance and power implications of GALS designs [216, 117]. In [117], they first predicted that going from a synchronous to a GALS design caused a drop in performance, but that elimination of the global clock would not single-handedly lead to drastic power reductions. In fact, from a power perspective, GALS designs are initially less efficient when compared to synchronous architectures. Their potential, however, lies in the flexibility offered by having several independently controllable clocks. As with other DVFS opportunities, the key lies in finding inter-domain slack that one can exploit. For example, in some MCD designs, the floating point unit could be clocked much more slowly than the instruction fetch unit, because its throughput and latency demands are lower. Iyer and Marculescu's results show that for a GALS processor with five

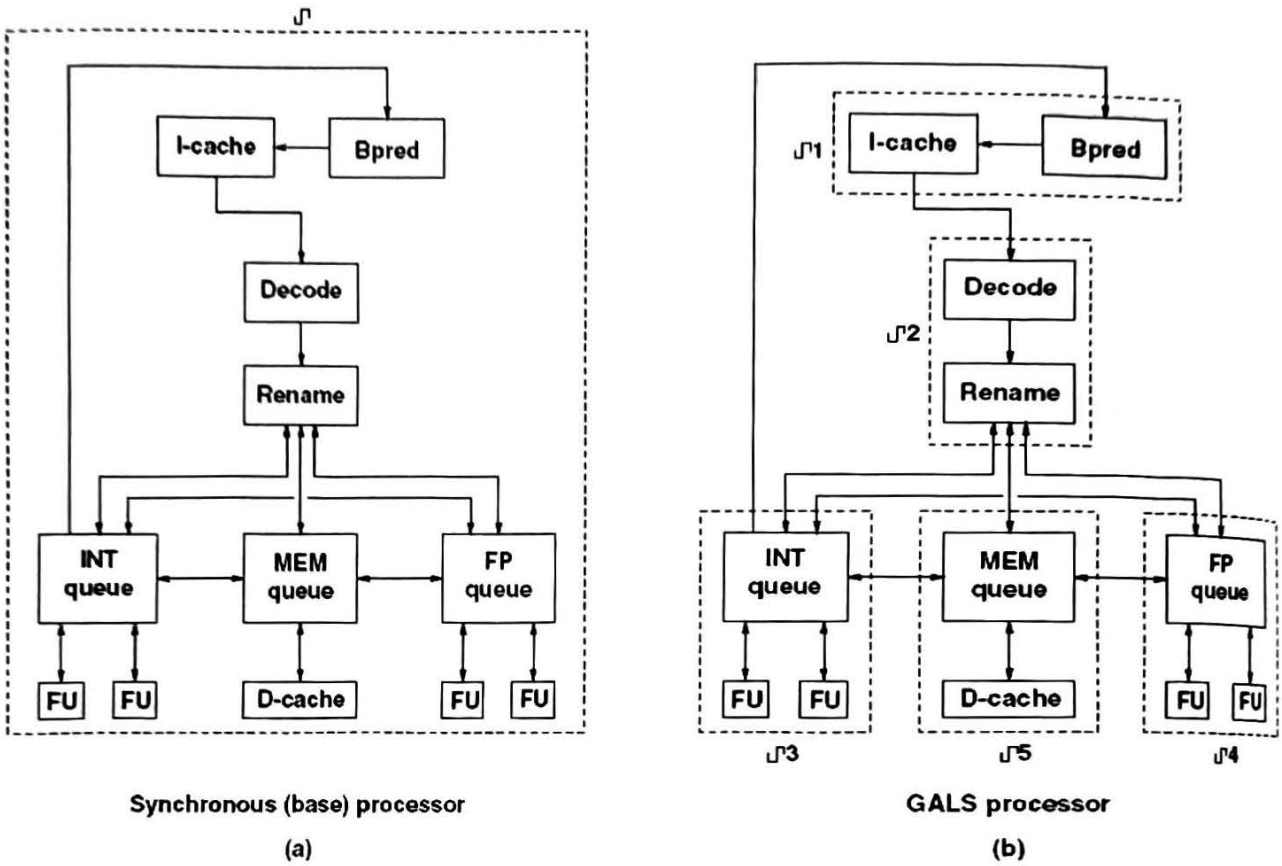

FIGURE 3.5: Synchronous versus GALS processor. Reproduced from [117]. Copyright 2002 IEEE.

clock domains, the drop in performance ranges between 5% and 15%, while power consumption is reduced by 10% on the average. Thus, fine-grained voltage scaling allows GALS to match or exceed the power efficiency of fully synchronous approaches.

In a similar timeframe, research from Albonesi's group also explored DVFS opportunities in MCD processors [199, 200]. Similar to the Iyer and Marculescu MCD division of the CPU (Figure 3.5), Semeraro et al. divide the processor into five domains: Front end, Integer, Floating point, Load/Store, and External (Main Memory). The division is shown in Figure 3.6 along with the relevant clock parameters. The domains interface via queues.

The work by Semerao et al. primarily focused on the control policies by which (V, f) settings could be optimized for such microarchitectures. Their early work used offline scheduling to accomplish good (V, f) settings, but subsequent work explored control-theoretic approaches for managing MCD processors.

Offline approach: Semeraro et al. use an offline approach in [200] to select the times and frequency values for DVFS in an application. The application is executed (at maximum speed) in a simulator that creates an event trace. The events correspond to primitive operations in the processor (for example, for a load instruction: fetch, dispatch, address calculation, memory access, and commit events are traced). The events are connected with resource constraints and

Parameter	Value(s)
Domain Voltage	0.65 V — 1.20 V
Domain Frequency	250 MHz — 1.0 GHz
Frequency Change Rate	49.1 ns/MHz
Domain Clock Jitter	110ps, normally distributed about zero
Synchronization window	30% of 1.0 GHz clock (200ps)

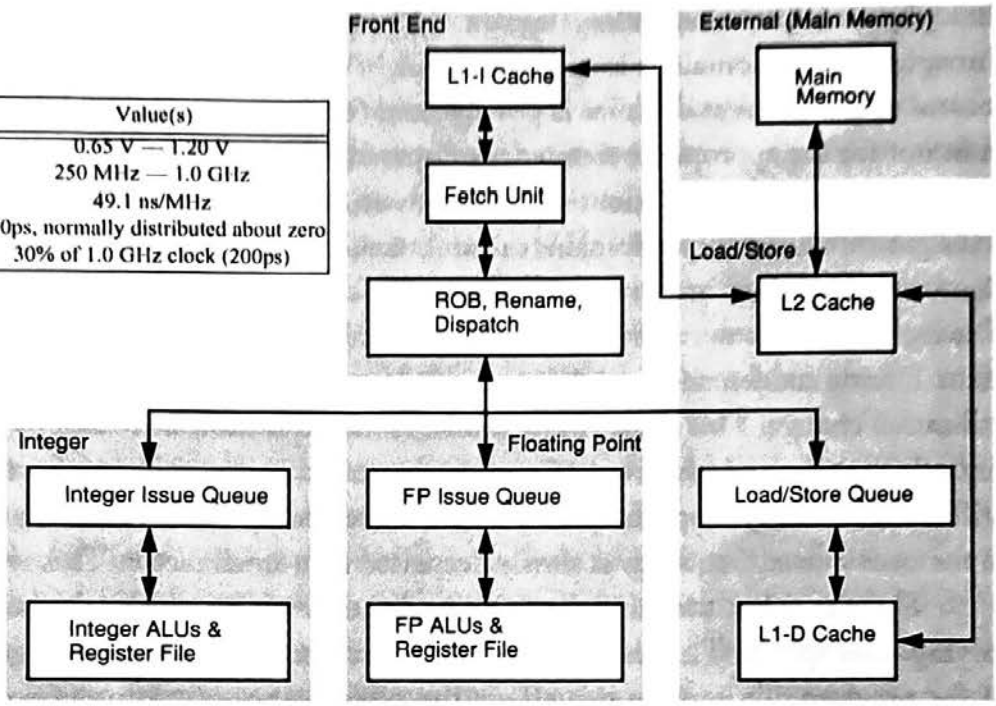

FIGURE 3.6: MCD processor and clock parameters. Adapted from [199].

data dependencies into a temporal-ordered directed acyclic graph (DAG). A DAG is created for an interval of 50K instructions and is then processed in two phases.

In the first phase, each event in the DAG that is not on the critical path is stretched, as if each instruction could run at its own frequency. A multi-pass "shaker" algorithm tries to distribute slack evenly in the DAG wherever it exists. This step concludes when all slack in the DAG is removed and each instruction is assigned to run at one of the allowed frequencies (e.g., one of the 32 frequencies for the Transmeta Crusoe or one of the 320 in the Intel XScale).

Since executing each instruction at a different frequency is not practical, the second phase processes the results of the first phase and aims to find a single *minimum* frequency *per interval for each domain*. This is done under the constraint that each domain finishes its work with no more than a fixed—externally set—factor of time dilation. Finally, intervals with the same or similar frequencies are merged together to create larger combined intervals—and this is continued recursively—with the intent of reducing the number of reconfigurations. In contrast to the first phase where DVFS reconfiguration was considered instantaneous and for-free, in the second phase realistic DVFS overhead (for the two processors studied) is taken into account.

Online approach: By analyzing the resource utilization of various CPU structures in [199], Semeraro et al. discovered that there is a significant correlation between the number of valid entries in the input queues for each domain and the desired frequency of the domain as derived by their offline method (see above). In other words, the *occupancy* of these queues reveals the

throughput of the corresponding domain.[3] When a queue fills up—its utilization increases—the throughput of its domain is low. The analysis of Semeraro et al. revealed that (i) decentralized control of the different domains is possible and (ii) the utilization of the input queues is a good indicator for the appropriate frequency of operation.

Based on the observations of their analysis, Semeraro et al. devised an online DVFS control algorithm for multiple domains called Attack/Decay. This is a decentralized, interval-based algorithm. Decisions are made independently for each domain at regular sampling intervals. The algorithm tries to react to changes in the utilization of the issue (input) queue of each domain. During sudden changes, the algorithm sets the frequency aggressively to try to match the utilization change. This is the *Attack* mode. If the utilization increased by a significant amount since the last interval, the frequency is also increased by a significant factor. Conversely, when utilization suddenly drops, frequency is also decreased. In the absence of any significant change in the issue queue, frequency is slowly decreased by a small factor. This is the *Decay* mode.

The algorithm tries to balance the utilization of the various domains independently by varying their speeds. This, however, does not account for a natural change in the performance of the program. To capture this effect, the performance of the processor in terms of IPC is tracked from interval to interval. If there is an inherent change in the IPC that is not related to frequency adjustments in the domains, then no frequency changes are allowed for the next interval. The IPC is the only global information needed in this algorithm.

On average, across a wide range of MediaBench, Olden, and Spec2000 benchmarks, their algorithm achieved a 19% reduction (from a non-DVFS baseline) in energy-per-instruction and a 16.7% improvement in energy-delay product. The approach incurred a modest 3.2% increase in cycles per instruction (CPI). Interestingly, their online control-theoretic approach was able to achieve a full 85.5% of the EDP improvement offered by the prior offline scheduling approach. Wu et al. extended the online approach using formal control theory and a dynamic stochastic model based on input-queue occupancy for the MCDs [228].

3.4.2 Dynamic Work-Steering for MCD Processors

As an alternative to dynamic voltage/frequency scaling of multiple clock domains, one can *statically* provide multiple components for the same function clocked at different frequencies. For example, one can provide a fast and power-hungry pipeline and a slow but power-efficient pipeline. With two such pipelines, the problem is no longer about selecting domain frequencies to eliminate stack, but instead is about steering instructions to the appropriate slow or fast

[3]The occupancy of the instruction queue is also used for resizing it to reduce its switching activity factor as we discuss in Section 4.6.3.

pipeline to accomplish this. This is a *work-steering* strategy that is also revisited in Chapter 4 for *effective capacitance* optimizations (Section 4.13).[4]

Fields, Bodik, and Hill use this work-steering approach as an example of how instruction slack can be exploited [76]. In their study, they show that there exists a significant slack in instructions. In many instances, instructions can be delayed several cycles without any impact on the program's critical path and hence its performance [76]. Furthermore, they classify the instruction slack into *local*, *global*, and *apportioned*. *Local* slack exists when an instruction can be delayed without any impact on any other instruction. *Global* slack exists when delaying an instruction does not delay the last instruction of the program (i.e., there is no impact on the total execution time). *Apportioned* slack refers to the amount of slack for a *group of instructions* that can be delayed together without impact on execution. Apportioned slack depends on how it is calculated from the instructions' individual slack [76].

To measure slack, Fields et al. use an offline analysis (similar to the Semeraro et al. offline approach used in [200] and described in Section 3.4.1) that creates a dependence graph of the execution taking into account both data dependencies and microarchitectural resource constraints. Their offline approach allows the calculation of all three types of slack. Their results show that there is enormous potential for exploiting slack by slowing down instructions [76].

More interestingly, Fields et al. show that one can dynamically predict slack in hardware. This is of significance since it allows for the possibility of *fine-grain*—on a per-instruction basis—control policies. Online control policies discussed previously for DVFS in MCD processors cannot treat each instruction individually. There is simply no possibility of dynamically changing the frequency of execution individually for each instruction; instead, the frequency of each domain is adjusted according to the aggregate behavior of all the instructions processed in this domain over the course of a sampling interval (Section 3.4.1).

With work steering, the execution frequencies are fixed for each execution pipeline—as is the case for the fast and slow pipelines in Figure 3.7–and the instructions are steered toward the appropriate pipeline. All that is needed to implement work steering is to have a good idea of the slack of each instruction. And this is where prediction comes into play. According to Fields et al., for 68% of the static instructions, 90% of their dynamic instances have enough slack to double their latency. This slack "locality" allows slack prediction to be based on sparsely sampling dynamic instructions and determining their slack.

Slack prediction would not be feasible if slack could not be measured efficiently at run-time. To determine whether an instruction has slack, Fields et al. employ an elegant delay-

[4]There too, multiple components are provided, offering a range of power/performance characteristics, and work (computation) is dynamically steered according to run-time conditions and goals.

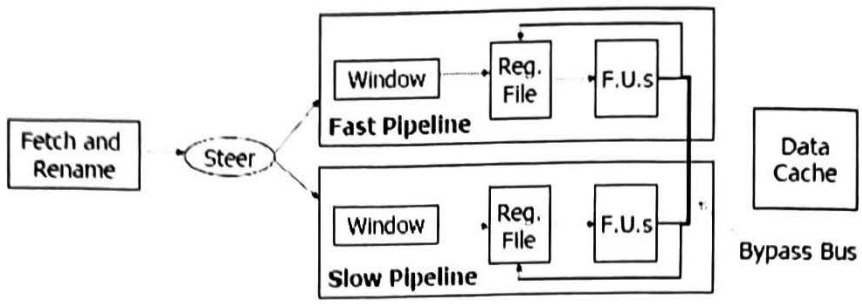

FIGURE 3.7: Work steering for a fast and a slow pipeline. Reproduced from [76]. Copyright 2001 IEEE.

and-observe approach. An instruction is delayed for a number of cycles and then observed to see if it becomes critical.

Criticality, in turn, is tested with a token that is passed from instruction to instruction, starting from the delayed instruction, and passed to all dependent instructions. The token is *dropped* if it is not passed to an instruction at the very last moment before it becomes ready to execute. In other words, the token is dropped if it has slack. If the token is still in existence well after an instruction is delayed, then the instruction, thus far, is in the critical path.

Slack determined by sampling is stored, per-instruction, in a PC-indexed predictor. This prediction is used in subsequent dynamic instances of the instruction for steering. If an instruction's predicted slack can accommodate the increased latency of a slow pipeline, then the instruction can be steered to this pipeline for execution without an impact on the performance. Fields et al. confirm this in their results, showing that a control policy based on slack prediction is second-best, in terms of performance, only to the ideal case of having two fast pipelines instead of a fast and a slow pipeline [76]. However, execution in the slow pipeline yields significant benefits in power consumption.

3.4.3 DVFS for Multi-Core Processors

As chip multiprocessors (CMPs) become the predominant general-purpose, high-performance microprocessor platform, it becomes important to consider how DVFS management can be applied to them most effectively. One major design decision concerns whether to apply DVFS at the chip level or at the per-core level. As with other MCD designs, per-core DVFS is considered more expensive; it requires more than one power/clock domain per chip, and synchronizer circuits are required to avoid metastability between domains. On the other hand, multiple clock domains may be employed anyway for circuit design or reliability reasons, in addition to voltage and frequency control.

Research has explored the benefits of per-core versus per-chip DVFS for CMPs. For example, on a four-core CMP in which DVFS was being employed to avoid thermal emergencies (rather than simply to save power), a per-core approach had 2.5× better throughput than a per-chip approach [67]. This is because the per-chip approach must scale down the entire chip's (V, f) when even a single core is nearing overheating. With per-core control, only the core with a hot spot must scale (V, f) downwards; other cores can maintain high speed unless they themselves encounter thermal problems.

While a multi-core processor can be used to run independent programs for throughput, its promise for single-program performance lies in thread-level parallelism. Managing power in a multicore when running parallel (multi-threaded) programs is currently a highly active area of research. Many research groups are tackling the problem, considering both symmetric architectures which replicate the same core and asymmetric architectures that feature a variety of cores with different power/performance characteristics [146].[5] Independent DVFS for each core [15], a mixture chip-wide DVFS and core allocation [153], or work-steering strategies at the program level in heterogeneous architectures [146, 170] are considered.

3.5 HARDWARE-LEVEL DVFS

The main premise in much of the DVFS work is that a system, a task, or a program can be slowed down with disproportionally small impact on its performance (or the perception of performance for interactive tasks), while at the same time obtaining significant savings in power consumption by voltage scaling. This can only be achieved by intelligently reducing frequency to remove *slack*: idle time in the system, slack in tasks with deadlines, or instruction slack due to memory accesses in memory-bound program phases. A similar idea can be applied at the hardware level. Ernst, et al. proposed a DVFS variation intended to remove slack in the timing of the hardware itself. Their approach is called Razor [73].

The driving motivation is to scale the supply voltage as low as possible for a given frequency while still maintaining correct operation. What prevents scaling the voltage below a critical level for a given frequency is the built-in margins in a process technology. For a given frequency, a voltage level is allowed which is safely above the lowest voltage level needed for the *worst-case process and environment variability* in the design. In other words, the relation between voltage and frequency is such that it guarantees correct operation with significant margin from the worst-case scenario.

This, however, diminishes the value of DVFS since the useful voltage range for DVFS shrinks with each new process technology. Going below the critical voltage level (*subcritical voltage*) for a given frequency invites trouble: timing faults. Faults, however, are unlikely to

[5]Similarly to the architecture described in Section 3.4.2 but allowing multiple cores to be active at the same time.