2013

A Dynamic Reconfiguration Framework to Maximize Performance/Power in Asymmetric Multicore Processors

Arunachalam Annamalai
University of Massachusetts Amherst

A DYNAMIC RECONFIGURATION FRAMEWORK TO MAXIMIZE PERFORMANCE/POWER IN ASYMMETRIC MULTICORE PROCESSORS

A Thesis Presented

by

ARUNACHALAM ANNAMALAI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2013

Electrical and Computer Engineering

A DYNAMIC RECONFIGURATION FRAMEWORK TO MAXIMIZE PERFORMANCE/POWER IN ASYMMETRIC MULTICORE PROCESSORS

A Thesis Presented

by

ARUNACHALAM ANNAMALAI

Approved as to style and content by:

Israel Koren, Co-chair

Sandip Kundu, Co-chair

Csaba Andras Moritz, Member

C.V. Hollot, Department Chair
Electrical and Computer Engineering

To my parents.

ACKNOWLEDGMENTS

I take this opportunity to thank and recognize all the people who have helped me with this thesis. Firstly, I would like to express my heartfelt gratitude to my advisors, Professor Israel Koren and Professor Sandip Kundu, for their constant guidance, motivation and encouragement. They have always been the enablers for going the extra mile and definitely without them this thesis would not have been a reality. Special thanks to them for supporting me financially through the grant 0903191 from the National Research Foundation. Secondly, I thank Professor Csaba Andras Moritz for being part of my thesis committee and sharing his valuable time, advice and suggestions throughout this work. I am grateful to Rance Rodrigues for all the help rendered and for guiding me through the initial phase of my masters. I thank all my friends in Amherst for providing me a healthy and stimulating study environment during my stay. I express my deepest gratitude to my mother and my brother for their constant encouragement and undoubted trust in my ability. Last but not the least, I thank God for this wonderful opportunity.

ABSTRACT

A DYNAMIC RECONFIGURATION FRAMEWORK TO MAXIMIZE PERFORMANCE/POWER IN ASYMMETRIC MULTICORE PROCESSORS

SEPTEMBER 2013

ARUNACHALAM ANNAMALAI

B.E, MADRAS INSTITUTE OF TECHNOLOGY, ANNA UNIVERSITY, INDIA

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Israel Koren and Professor Sandip Kundu

Recent trends in technology scaling have shifted the processing paradigm to multicores. Depending on the characteristics of the cores, the multicores can be either symmetric or asymmetric. Prior research has shown that Asymmetric Multicore Processors (AMPs) outperform their symmetric (SMP) counterparts within a given resource and power budget. But, due to the heterogeneity in core-types and time-varying workload behavior, thread-to-core assignment is always a challenge in AMPs. As the computational requirements vary significantly across different applications and with time, there is a need to dynamically allocate appropriate computational resources on demand to suit the applications' current needs, in order to maximize the performance and minimize the energy consumption. Performance/power of the applications could be further increased by dynamically adapting the voltage and frequency of the cores to better fit the changing characteristics of the workloads. Not only can a core be forced to a low power mode when its activity level is low, but the power saved by doing so could be opportunistically re-budgeted to the other cores to boost the overall system throughput.

To this end, we propose a novel solution that seamlessly combines heterogeneity with a Dynamic Reconfiguration Framework (DRF). The proposed dynamic reconfiguration framework is equipped with Dynamic Resource Allocation (DRA) and Voltage/Frequency Adaptation (DVFA) capabilities to adapt the core resources and operating conditions at runtime to the changing demands of the applications. As a proof of concept, we illustrate our proposed approach using a dual-core AMP and demonstrate significant performance/power benefits over various baselines.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION AND MOTIVATION

Advancements in technology allowed more transistors to be packed in a smaller area while the improved performance of transistors helped in achieving higher clock frequencies resulting in sharp increase in the power density. To combat this unsustainable increase in power density, the processor industry responded by lowering the frequency and integrating multiple cores on the same die [20, 25]. As a multicore die is still limited by an overall power dissipation envelope that stems from packaging and cooling technologies, most current multicores are composed of cores with relatively moderate capabilities.

In this chapter, we study the main problems that limit the current multicore systems in achieving high energy efficiency and present a proposal to address them.

1.1 Symmetric vs. Asymmetric Multicore Processors

Multicore processors, in general, may be symmetric (SMP) or asymmetric (AMP). An SMP consists of many cores of the same type while in an AMP, the cores may be different from one another with respect to their functionality and/or performance [31]. As a first step, there is a need to choose between the two types so as to achieve maximum performance/power for most applications.

Multicores execute diverse applications with a large variance in their instruction distribution. Figure 1.1 shows the instruction distribution of 38 benchmarks we consider, when run for 100 million instructions. As can be observed, some benchmarks are memory bound (e.g., *fbench, gcc*); some are floating-point intensive (e.g., *equake,*

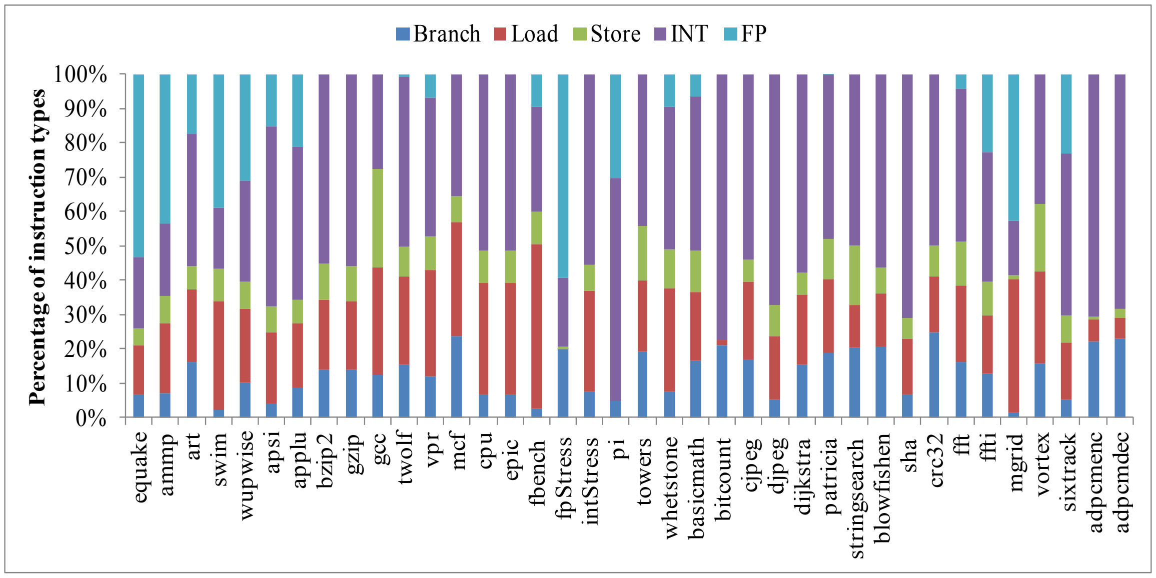


Figure 1.1. Distribution of the instruction types for 38 benchmarks.

fpStress, *ammp*) while others are integer intensive (e.g., *bitcount*, *sha*). It is evident from the figure that the resource requirements of applications vary significantly. Moreover, even within a given application, computational requirements may vary with time due to changes in program phases [32, 47]. Thus, different workloads benefit from different computational resources at different instants of time. Hence, homogeneous (symmetric) multicores with fixed computational resources are likely to miss potential opportunities to improve performance and reduce energy consumption. This leads us to our decision to focus on using a *heterogeneous computing fabric* with cores of diverse strengths that could efficiently cater to the needs of different applications and their phases. Our decision is in line with recent studies [18, 22, 33, 53] that show that AMPs can outperform their symmetric counterparts within a given power and area budgets when the computing demands of the applications are matched with the processor capabilities.

1.2 Motivation for Dynamic Reconfiguration Framework

The benefits of AMPs are, however, highly dependent on the way threads are assigned to the individual asymmetric cores and a non-optimal assignment may even have an adverse impact. Consider, for example, Figure 1.2 where the performance/Watt of a few workloads executed on two cores is plotted. For now, let the two cores be called core A and core B. This figure shows that for some workloads, core A

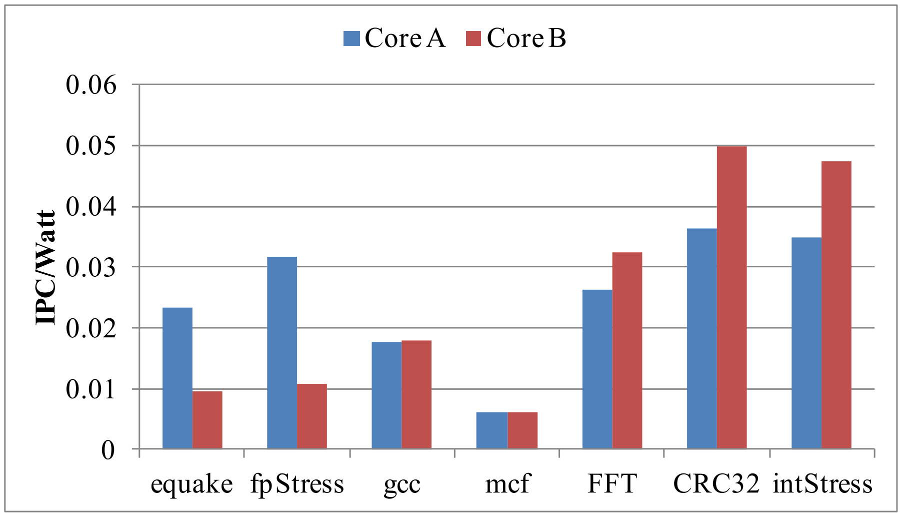


Figure 1.2. Performance/Watt achieved for different workloads on two different core types A and B.

is a better option (e.g., *equake, fpStress*) while for some others, core B is better (e.g., *CRC32, intStress*). There are also some workloads (e.g., *gcc, mcf*) for which there is no significant difference in performance/Watt achieved by either core. Clearly, a correct thread to core assignment is required to maximize the performance/power of the applications [5]. Furthermore, even a best static thread-to-core assignment may not suffice as the applications change phases during their execution. Hence, a dynamic thread swapping mechanism can further improve the performance/power.

But, thread swapping alone may not be sufficient for all applications/program phases. This is because multicore processors sacrifice instruction throughput for certain applications as they primarily focus on supporting Thread Level Parallelism (TLP) [17, 40]. To achieve reasonable performance/power, applications should have a lower execution time and consume less power. High performance for sequential applications could be achieved either by designing more powerful cores or morphing the resources of the existing cores on-demand to suit the applications' current needs. Incorporating complex cores in a multicore system may lead to under utilization of resources and even breaching of power dissipation limits. Therefore, there is a strong need for a scheme to morph the existing core resources on-demand. To this end, we present our **first proposition** to improve the performance/power efficiency of AMPs by adaptively matching the processor capabilities to the computing needs of the executing threads. Dynamic thread swapping along with on-demand resource morphing constitute the *Dynamic Resource Allocation (DRA)* capability of our scheme.

As with computational resources, different voltage/frequency levels of the processor may suit different phases of an application. By appropriately choosing the operating conditions, we would be able to improve performance and reduce power consumption further. In this regard, Dynamic Voltage and Frequency Scaling (DVFS) technique [24] has been widely employed to reduce power consumption. The voltage and frequency of a core can be lowered when it is idle or is in a low activity mode. For example, a memory bound application typically does not have sufficient instruction level parallelism (ILP) to keep the core busy while waiting for the long-latency memory accesses to complete [54]. Reducing the voltage and/or clock frequency of the core in such a case does not impact the overall performance greatly [24]. Intel's Turbo Boost technology enhances the performance of a high performing core through dynamic voltage and frequency boosting when the other cores are inactive [4, 43]. With an objective of increasing the overall system throughput and maximizing performance/power, we present our **second proposition** of incorporating a *Dynamic Voltage and Frequency Adaptation (DVFA)* capability as part of our reconfiguration framework. Dynamic Resource Allocation (DRA) and Dynamic Voltage and Frequency Adaptation (DVFA) capabilities together constitute our proposed Dynamic Reconfiguration Framework (DRF).

In this thesis, we first explore a rule-based approach for the proposed dynamic reconfiguration framework. We refer to this as RDRF where the DRA and DVFA decisions are made online based on rules developed by profiling offline a subset of workloads. We then present a prediction-based approach (PDRF) to address the observed limitations of RDRF. In PDRF, the decisions about the core reconfiguration and operating conditions are made online by predicting the expected performance/power of a thread at different voltage/frequency levels on all the available core-types in the AMP.

1.3 Overview of our proposed scheme

At a base level, we assume an AMP architecture that could dynamically allocate execution resources (DRA) and adapt the frequency and voltage of the cores (DVFA) at runtime to suit the time dependent behavior of the workload (see Figure 1.3). The objective of our scheme is to maximize performance while keeping power dissipation under check. Hence, we employ performance/Watt as the metric to evaluate our DRA mechanism and use throughput/Watt when the voltage/frequency of the cores are changed dynamically.

The baseline cores are resourced moderately in all areas, while featuring extra-strength in a specific area (e.g., integer or floating-point operations). The strength of the cores is non-overlapping and hence, each core is suited for specific application characteristics. When a thread demands strength in more than one area, the cores are morphed dynamically by realigning their execution resources such that one core gains strength in one or more additional area(s) by trading its moderate resources for the stronger resources of other core(s). Such morphing is not always the best solution, if a mismatch between the thread needs and the capabilities of the core executing

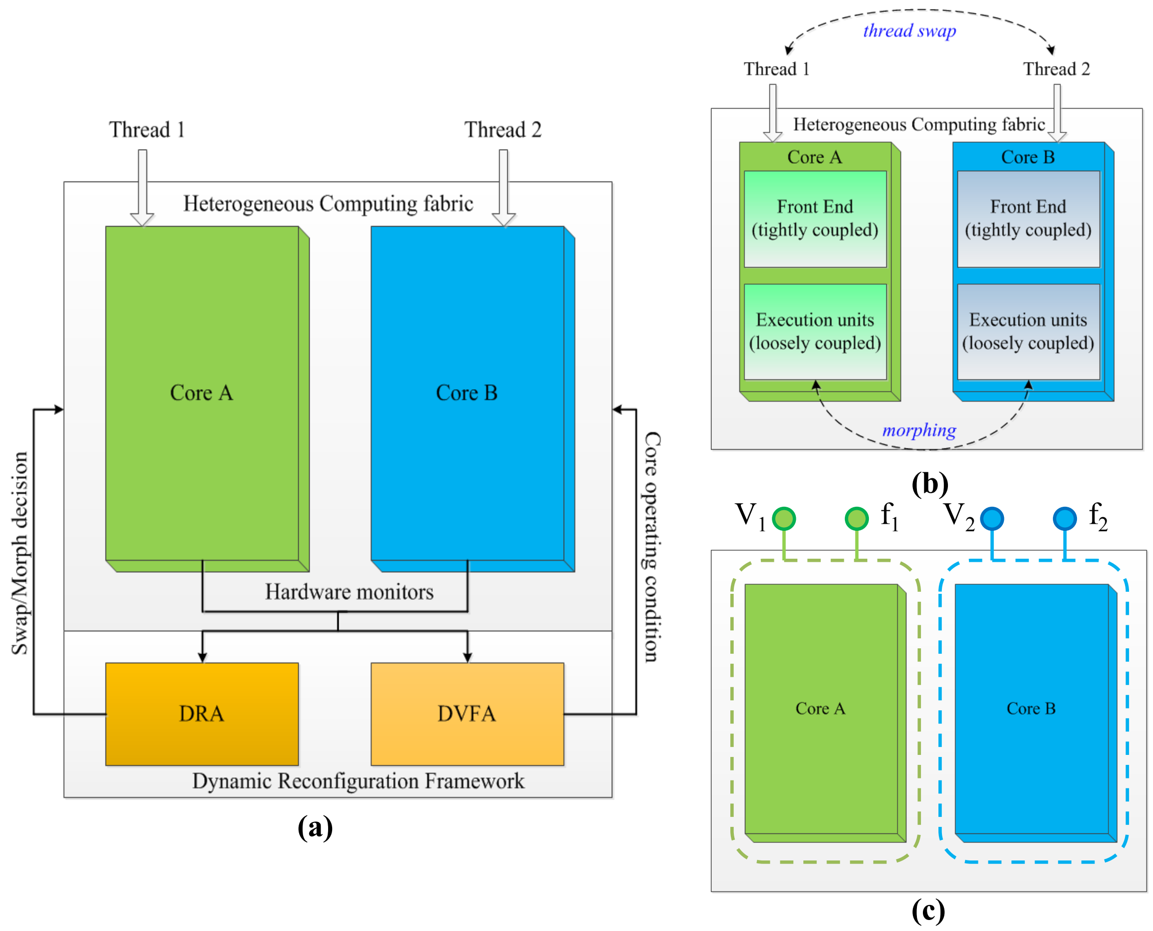

Thread 1
Thread 2
Heterogeneous Computing fabric
Core A
Core B
Hardware monitors
Swap/Morph decision
Core operating condition
DRA
DVFA
Dynamic Reconfiguration Framework
(a)
thread swap
Thread 1
Thread 2
Heterogeneous Computing fabric
Core A
Core B
Front End (tightly coupled)
Execution units (loosely coupled)
Front End (tightly coupled)
Execution units (loosely coupled)
morphing
(b)
V_1
f_1
V_2
f_2
Core A
Core B
(c)

Figure 1.3. (a) High-level view of the complete DRF. (b) Thread swap and core morphing as part of DRA. (c) DVFA capability of the scheme. Voltage(V)/frequency(f) of the cores changed dynamically.

it, is discovered, a thread swap may provide a better alternative. Thus, our AMP architecture supports moving from the *baseline* mode of operation to the *morphed* mode, returning to the baseline, and also supports *thread swap*. Hardware monitors (performance counters) are used to determine the thread-to-core affinity and trigger core reorganization at runtime to maximize performance/Watt. The main merits of the proposed DRA scheme shown in Figure 1.3(b) are:

1. It allows applications to exploit the most suitable core for better performance.

2. The individual cores remain modest in their sizing allowing the AMP to meet the overall cost and power targets.

3. The realigned resources in the morphed mode provide higher levels of performance for the applications that can benefit from them.

The above benefits are further increased with the added DVFA feature (see Figure 1.3(c)), where the frequency and voltage of the individual cores are changed (decreased or increased) dynamically in accordance with the workload behavior to maximize throughput/Watt while staying within the defined Thermal Design Power (TDP) limits.

1.4 Contributions of this thesis

1. A holistic energy-efficient scheme that dynamically allocates appropriate execution resources and/or changes the voltage and frequency of the cores at runtime to maximize performance/power.

2. A unified mechanism based on hardware counters that seamlessly triggers both core reconfiguration and voltage/frequency adaptation.

3. A mechanism to accurately predict the expected performance/power of the current program phase if it would run on other core-types in the AMP and at different voltage/frequency levels.

The rest of the thesis is organized as follows. We review prior work related to our approach in Chapter 2. We present our proposed scheme in Chapter 3. Chapter 4 describes our core sizing experiments. The different core configurations are evaluated in Chapter 5. We present and evaluate DRA as a stand-alone scheme in Chapter 6 and the complete rule-based dynamic reconfiguration framework is presented in Chapter 7. The prediction-based DRF (PDRF) which addresses the limitations of RDRF is discussed in Chapter 8 and an application of it for a commonly used dual-core AMP is studied in Chapter 9. Chapter 10 concludes the thesis and the possible extensions of this work are discussed in Chapter 11.

CHAPTER 2

RELATED WORK

As the proposed approach combines heterogeneity, dynamic resource allocation and voltage/frequency adaptation, we briefly review prior research on each of these fronts in this chapter.

2.1 Heterogeneous and Reconfigurable Architectures

Ipek *et al.* [23] have presented the concept of core fusion where the resources of several homogeneous cores are fused to form a single stronger core at runtime. Kim *et al.* [29] presented another approach to fusion of homogeneous cores where, for example, 32 dual-issue cores can be fused into a 64-issue processor. Both schemes exhibit a high inter-core global communication overhead and the potential benefits of fusion are negatively affected by the reconfiguration overhead of critical units like re-order buffer (ROB), issue (ISQ) and load/store queues (LSQ). Salverda *et al.* [45] discuss the difficulties in achieving good performance by fusing simple in-order cores into out-of-order (OOO) cores. Aggregating cores in a SMP [23, 29, 51] offers more of the same resources and hence its performance benefits saturate as the Instruction Level Parallelism (ILP) saturates.

Recent studies have shown that symmetric cores are unlikely to provide better performance than a heterogeneous multicore [22, 33]. Morad *et al.* [37] propose heterogeneous architectures that could be employed to achieve higher performance per area per Watt. The power benefits obtained by using a single ISA heterogeneous chip multiprocessor (CMP) is evaluated in [32]. Other references [10, 22, 38, 40] show that reconfigurable architectures may improve the benefit of AMPs even further. Das *et al.* [12] have proposed an asymmetric dual-core processor that could fuse a strong integer and a strong floating-point cores. Their scheme is static so the cores are either fused or not for the entire program run. Static morphing of the cores cannot suit all the different phases in an application and hence, there is a need for a scheme that could adapt dynamically to the time-varying program behavior.

2.2 Dynamic Thread Scheduling schemes

Earlier proposed thread scheduling schemes could be broadly classified into those that employ offline profiling, online learning via sampling and online estimation.

Offline profiling schemes: Khan *et al.* [26] propose regression analysis along with phase classification to identify thread to core affinity. Shelepov *et al.* [46] profile applications to determine architectural signatures based on cache misses. These signatures are obtained offline via profiling and are fixed for the lifetime of the program and hence their scheme do not take advantage of program phases. In [9], Chen *et al.* use cores in an AMP that differ with respect to issue width, branch predictor size and L1 caches. They use multi-dimensional curve fitting to determine the optimal thread to core assignment offline. All the above approaches are not practical as they require complete knowledge of the workloads that will be run on the multicore. In contrast, we deduce rules for reconfiguration by profiling only a few representative workloads in our rule-based dynamic reconfiguration framework (RDRF). The rules thus obtained are used globally for all applications, not limiting to the profiled set.

Online learning schemes: These schemes offer a more practical solution to the AMP scheduling problem. Kumar *et al.* in [32] proposed an AMP consisting of cores of various sizes. Whenever a new program is run or a new phase is detected, a sampling is initiated and the core which provides the best power efficiency is chosen. Although, this work considered four cores, only a single thread was considered running which simplifies the AMP scheduling problem. The authors later extended their research to cover performance maximization of multithreaded applications [34]. A similar approach was proposed by Becchi *et al.* [6] for performance maximization of an AMP consisting of two types of cores. Optimal thread scheduling was determined by forcing a thread swap between cores upon detection of phase change. The number of samples required by the above schemes may be large for a many-core system.

Online estimation-based schemes: These schemes are an improvement over the online learning schemes since they avoid sampling and the resulting overhead. Here, based on the current characteristics of a workload being executed, its performance on other core types of the system is estimated. Saez *et al.* [44] propose a comprehensive scheduler for AMPs consisting of small and big cores using last level miss rates of an application to estimate its performance on each core type. In [31], Koufaty *et al.* determine thread to core mapping in an AMP consisting of big and small cores, using program to core bias which is estimated online using the number of external (proportional to cache requests going to L2 and main memory) and internal stalls (front end not delivering instructions to the back end). In [50], Srinivasan *et al.* estimate the performance of the thread currently running on one core type, on another core, using a closed form expression. These expressions were developed for specific cores and a general approach was not provided. Extending the above technique to include power estimation is not straightforward. Of the currently available scheduling schemes, the estimation-based ones offer the most practical and scalable solution. Still, most of the earlier schemes focus mainly on performance and, do not take into account the multiple voltage/frequency levels that may be available within the cores. Our proposed prediction-based dynamic reconfiguration framework (PDRF) addresses the above shortcomings and strives to maximize the overall throughput/Watt.

2.3 Dynamic Voltage and Frequency Adaptation schemes

Dynamically scaling the voltage and frequency of cores in CMPs has been established as an efficient technique for power reduction [24] and as a corrective measure for thermal emergencies [36, 7]. Ghasemazer *et al.* [16] address the problem of minimizing energy consumption in CMPs by selectively turning ON or OFF the cores and choosing the optimum voltage and frequency for each core using a three-level hierarchical framework. Intel's Turbo Boost technology allows a core to run at a higher frequency automatically if the multicore is operating below a given rated power and temperature limits [43]. The maximum frequency that could be reached is dependent on the number of active cores [4]. Similarly, AMD's Accelerated Processing Units (APUs) use the Turbo Core Technology to boost the frequency and performance of the cores staying within the defined power envelope [15]. Keramidas *et al.* [24] predict the performance and power consumption of super-scalar processors under different voltage and frequency combinations and implement a DVFS scheme based on stall cycles due to L2 misses. The benefits of per-chip adaptive frequency scaling in multicores by grouping applications with similar frequency-to-performance effects is explored in [54]. Energy-saving opportunities and nanosecond-scale voltage switching using on-chip voltage regulators are discussed in [30].

CHAPTER 3
PROPOSED SCHEME

In this chapter, we describe in detail our proposed scheme that adapts voltage/frequency and execution resources dynamically to different workloads and to program phases within those workloads at runtime. To facilitate such adaptation, the scheme implements Dynamic Resource Allocation (DRA) and Dynamic Voltage and Frequency adaptation (DVFA) which are discussed in detail next.

3.1 Dynamic Resource Allocation

To illustrate our approach, we consider two heterogeneous cores (see Figure 3.1) per tile. A multicore system may consists of as many such tiles as deemed appropriate making the scheme scalable. The first core is a 2-way super-scalar strong integer (INT) core, with high performance integer execution units but with low performance for floating-point operations, while the second, a strong floating-point (FP) core, features strong floating-point execution units but low performance integer execution units. The reason for this example architecture is the diversity in the instruction type distribution of the common benchmarks shown in Figure 1.1. By focusing on the distinct strength of the integer and floating-point execution units, we would be able to efficiently service a wide variety of non-overlapping applications in the baseline mode of operation. The front-end resources (e.g., number of virtual rename registers, sizes of ISQ and LSQ) vary between the two cores and are discussed in Section 4.2. This scheme is similar to the one proposed by Das *et al.* in [12]. However, significant enhancements were made to the scheme. Firstly, we have explored the processor

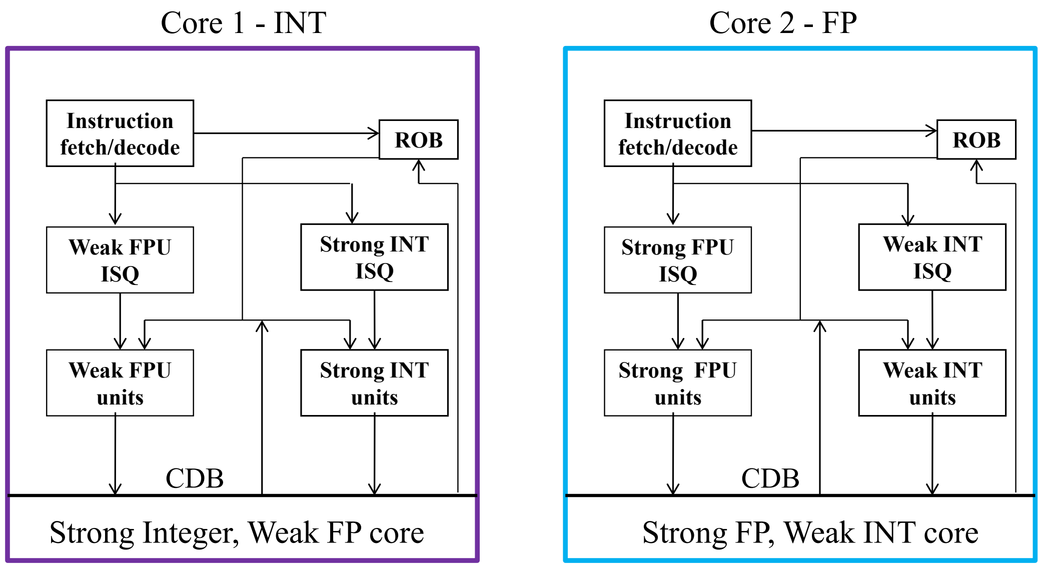


Figure 3.1. Baseline configuration for two heterogeneous cores.

design space in depth to determine the parameters of the baseline cores. Secondly, Das *et al.* [12] explore performance benefits while we focus on performance/Watt. Lastly, the architecture proposed in [12] is static while ours can dynamically reconfigure to meet changing application requirements.

In the baseline configuration (Figure 3.1), good performance is achieved by the cores while executing parallel workloads with appropriate resource requirements. However, when there is a need for a strong sequential performance by an application, dynamic resource *morphing* of the cores takes place. In the morphed mode, the INT core takes control of the strong floating-point unit of the FP core to form a "morphed strong" (strong) core while relinquishing control of its own weak floating-point unit to the FP core. The FP core is thus morphed into a "weak core." The strong core retains the front-end resources of the INT core. In contrast, the front-end resources of the FP core are appropriately sized down to suit the reduced needs of the application running on the weak core in order to save power. Hence, morphing results in two cores:

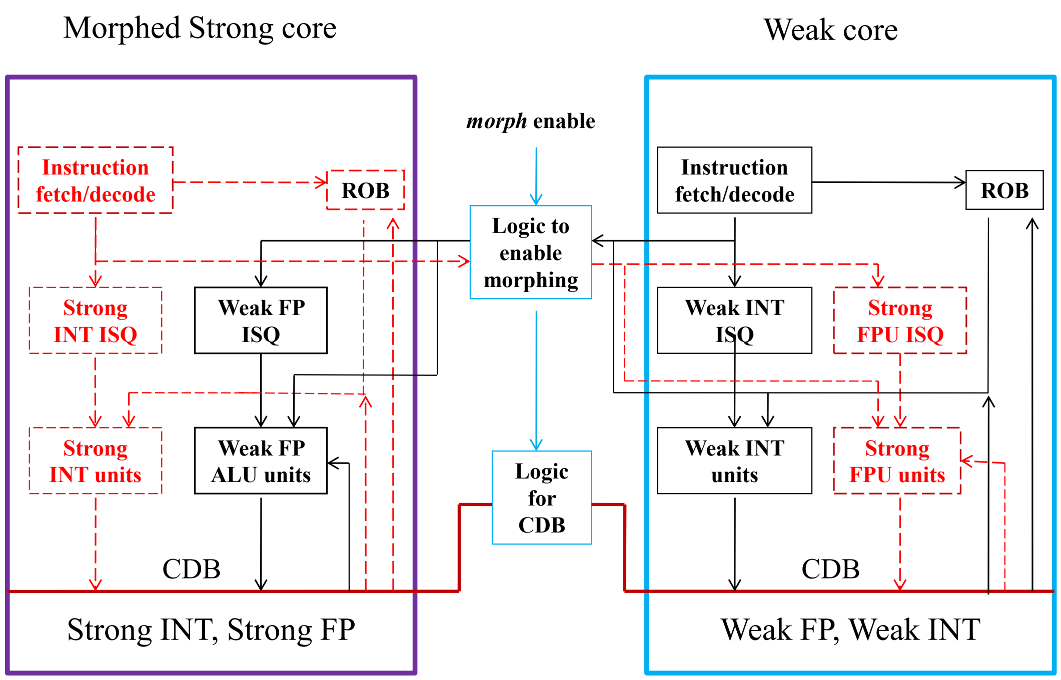


Figure 3.2. Morphed configuration for two heterogeneous cores. The red dotted lines/boxes indicate the connectivity for the strong core configuration while the black solid lines/boxes show the connectivity for the weak core.

1. A *strong core* capable of handling both integer and floating-point intensive operations efficiently.
2. A *weak core* with weak functional units consuming less power.

The proposed dynamic morphing of the cores is shown in Figure 3.2. When the morphed mode is no longer beneficial, the system reconfigures itself back to the baseline mode.

The behavior and characteristics of workloads tend to vary with time. Some applications may be floating-point intensive to start with and may have higher percentage of integer instructions after a certain point. Hence, swapping of the threads between the two baseline cores (strong integer (INT) and strong floating-point (FP)) under such scenarios would help in reducing the execution time significantly. Therefore, in addition to the baseline and morphed modes of operation, we also allow the two tightly coupled heterogeneous cores to swap their execution contexts.

3.2 Dynamic Voltage and Frequency Adaptation

We further leverage the DVFA feature to move each heterogeneous core individually to either the Low Power (LP) mode or the High Performance (HP) mode by monitoring the performance of the executing threads and the frequency of memory reference operations. When the Instructions per Cycle (IPC) of the thread is consistently low (likely due to memory intensive operations), the proposed scheme moves the corresponding core to the LP mode. On the other hand, if the performance of a thread is high, then the corresponding core is moved to the HP mode if the other core is either already in the LP mode or is ready to enter the LP mode. Hence, entering HP mode is conditioned on the other core being in the LP mode. This is done to ensure that the TDP limit of the multicore is not violated. Adding the DVFA feature to the dynamic allocation of resources (through morphing and thread swapping) further maximizes the performance/power benefits. Our results indicate that the proposed scheme performs much better in terms of increased throughput/Watt when compared to the static baseline heterogeneous cores and to the baseline heterogeneous cores with only one of these features.

CHAPTER 4

DETERMINING THE CORE PARAMETERS

We next describe in detail the experimental setup used in our experiments and the core sizing experiments that were carried out.

4.1 Simulator and Benchmarks

We used SESC as our architectural performance simulator [41], and measured power using Wattch [8] and CACTI [48] with modifications to account for *static power* dissipation. For our experiments, we have selected 38 benchmarks (see Table 4.1): 16 benchmarks from the SPEC suite [1], 14 from the MiBench suite [19], one benchmark from the mediabench suite [35], and 7 additional synthetic benchmarks. These 38 benchmarks encompass most typical workloads, for example, scientific applications, media encoding/decoding and security applications.

4.2 Core sizing

The design space for each core is extremely large including the exact sizes of individual structures (e.g., ROB and ISQ). Our goal is to focus on a set of parameters

Table 4.1. Benchmarks considered

	Benchmark
SPEC	apsi, ammp, equake, wupwise, twolf, swim, mcf, gcc, gzip, bzip2, vpr, art, applu, vortex, mgrid, sixtrack
MiBench	cjpeg, djpeg, basicmath, bitcount, dijkstra, patricia, stringsearch, blowfish, sha, adpcm, crc32, fft, ffti
Mediabench and others	epic, towers, intStress, fpStress, fbench, cpu, pi, whet-stone

Table 4.2. Parameter variation steps for the sizing experiments

Parameter	Initial configuration	Variation steps
DL1	32K	4-8-16-32
IL1	32K	4-8-16-32
L2	256K	32-64-128-256
LSQ	64 (each LD/SD)	16-32-48-64
ROB	256	32-48-64-128-256
INTREG	128	32-48-64-128
FPREG	80	32-48-64-80
INTISQ	128	16-32-64-128
FPISQ	64	8-16-32-64

that have the largest impact on the strong integer and the floating-point cores, and determine the size of these parameters for each core such that acceptable performance is achieved for a wide range of applications in the baseline configuration. If the cores are undersized, the results of core morphing would be biased and misleading.

To determine the architectural parameters for the cores, we have started with an initial configuration shown in Table 4.2 and then upsized the parameter under consideration and calculated the IPC metric for each core type. Based on the IPC, the most appropriate value for each parameter was selected. For the sake of brevity only ROB sizing results are shown in Figure 4.1. In the figure, each curve represents

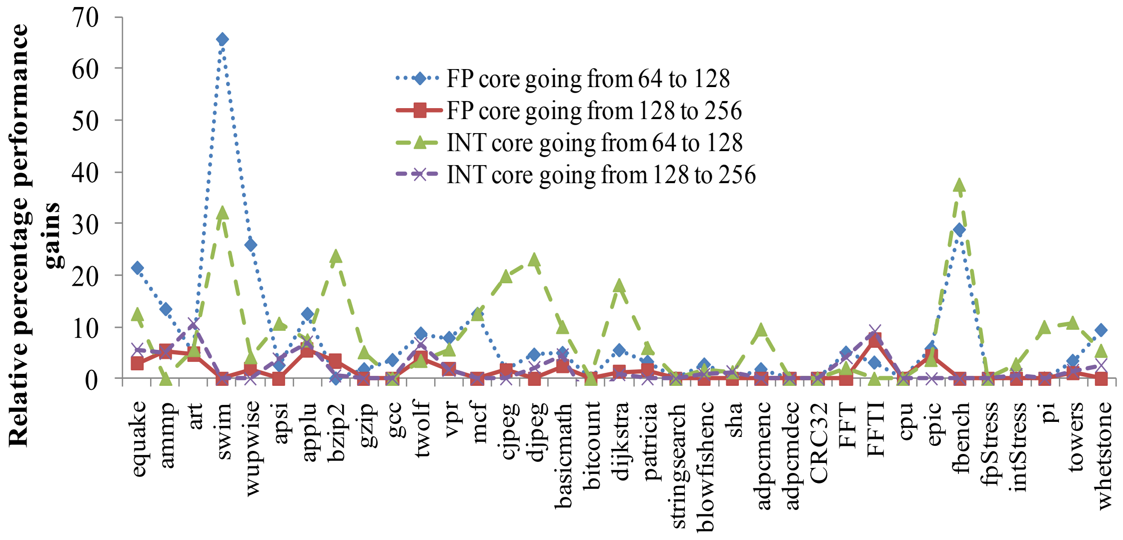


Figure 4.1. Ratio of the IPC for the core configurations when going from lower to higher sizes of ROB.

the ratio of the performance for the core when going from a smaller to larger ROB size. For the FP core, it can be seen that there are several benchmarks that benefit when going from ROB of size 64 to 128 (*equake, swim, applu, twolf, wupwise, fft, ffti and whetstone*) but such benefit is no longer seen when increasing the ROB size further to 256. Hence, ROB size of 128 is chosen for the FP core. Based on similar observations, the ROB for the INT core was also sized to 128. Similar sizing experiments were conducted for the rest of the parameters. To show the benefits of morphing, we also compare our DRA scheme against a dual-core homogeneous (HMG) design in Section 6.6. For a fair comparison between the two designs, the area of two HMG cores should match the sum of the areas of the FP and INT cores. Hence, the sizes of the structures for HMG core were obtained by averaging those obtained for the INT and FP cores.

Since the "weak core" is not expected to provide a performance as high as the original FP core, we further downsized it for higher energy efficiency. The configuration for all the core types is shown in Table 4.3. We did not include the final configuration of the "strong core" as it is nothing but a combination of the INT core with the FP units of the FP core. The specifications of the execution units of the INT and FP cores are shown in Table 4.4. The mentioned execution latencies are based on the experiments carried out by Vasan taking into account their impact on power,

Table 4.3. Core configurations after the sizing experiments

Parameter	FP	INT	HMG	Weak
DL1	4K	4K	4K	1K
IL1	4K	4K	4K	1K
L2	128K	128K	128K	64K
LSQ (each LD/SD)	32	32	32	32
ROB	128	128	128	64
INTREG	48	64	56	32
FPREG	64	32	48	32
INTISQ	32	32	32	16
FPISQ	32	16	24	8

Table 4.4. Execution unit specifications for the cores (P - Pipelined, NP - Not pipelined) [52].

Core	FP DIV	FP MUL	FP ALU
FP	1 unit, 18 cyc, P	1 unit, 10 cyc, P	2 units, 4 cyc, P
INT	1 unit, 60 cyc, NP	1 unit, 24 cyc, NP	1 unit, 10 cyc, NP
	INT DIV	INT MUL	INT ALU
FP	1 unit, 120 cyc, NP	1 unit, 30 cyc, NP	1 unit, 2 cyc, NP
INT	1 unit, 14 cyc, P	1 unit, 3 cyc, P	2 units, 1 cyc, P

performance and area [52]. A logical synthesis of the netlist using the mentioned latencies was also performed using Synopsys Design Compiler to illustrate that such a design could actually be implemented in practice.

4.3 Operating modes of the cores

Similar to the latest third generation Intel Core Processors [3], we envision the cores to operate in three modes: (i) nominal, (ii) low power (LP) and, (iii) high performance (HP) mode. The frequency levels of the cores are changed in steps of 133 MHz in accordance with [4]. Table 4.5 tabulates the different operating modes of the cores along with their voltage and frequency levels. It could be observed that the voltage/frequency of the core is decreased by two steps in LP mode resulting in significant power savings. The power thus saved could be redistributed to the other core in the AMP to boost the overall system performance.

Table 4.5. Core Operating Modes.

Mode	Voltage (V)	Frequency (GHz)
LP	0.9	1.734
Nominal	1.1	2
HP	1.2	2.133

CHAPTER 5

EVALUATING THE DIFFERENT CORE CONFIGURATIONS

In this chapter, we evaluate the effectiveness of our core sizing experiments by running each of the considered workloads on the various core types, i.e., FP, INT, strong and weak cores. Based on the objective set forth for the sizing experiments, we expect most of the applications to run reasonably well on one of the baseline cores (FP or INT) so that morphing is used sparingly. We present results of performance and performance/Watt evaluation and draw critical inferences from this analysis. We conclude the chapter with an in-depth study on the impact of program phases.

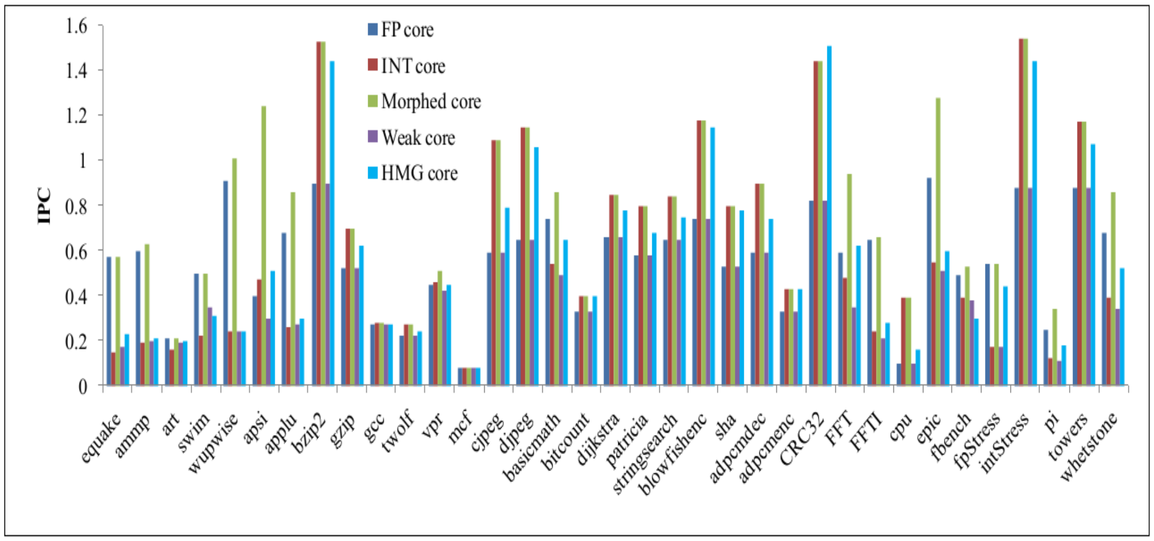


Figure 5.1. IPC of the considered benchmarks when run on each core configuration for 10 million instructions. Morphed core in the legend refers to the strong core.

5.1 Performance evaluation

We ran all the 38 considered benchmarks for 10 million instructions on each core configuration and the performance results are plotted in Figure 5.1. It can be seen from the figure that 7 benchmarks (*applu, wupwise, apsi, basicmath, epic, FFT, whetstone*) show benefits when run statically on the strong core. The obtained gains are significant and is even over 200% for *apsi*. However, as shown in the next section, this performance gain may not always result in a higher energy efficiency.

5.2 Performance/Watt evaluation

Figure 5.2 shows the performance/Watt evaluation of the cores. It could be observed that the number of benchmarks that benefit from the strong core has now reduced from 7 to 3 (*apsi, FFT, epic*). Even the achieved performance/Watt benefits are significantly lower. Of the 3 cases, *apsi* shows 35% improvement over its closest competitor, the FP core. This benefit is more modest for the benchmarks *epic* and *FFT* (10%). The reason why *apsi* shows substantial benefits is related to the tem-

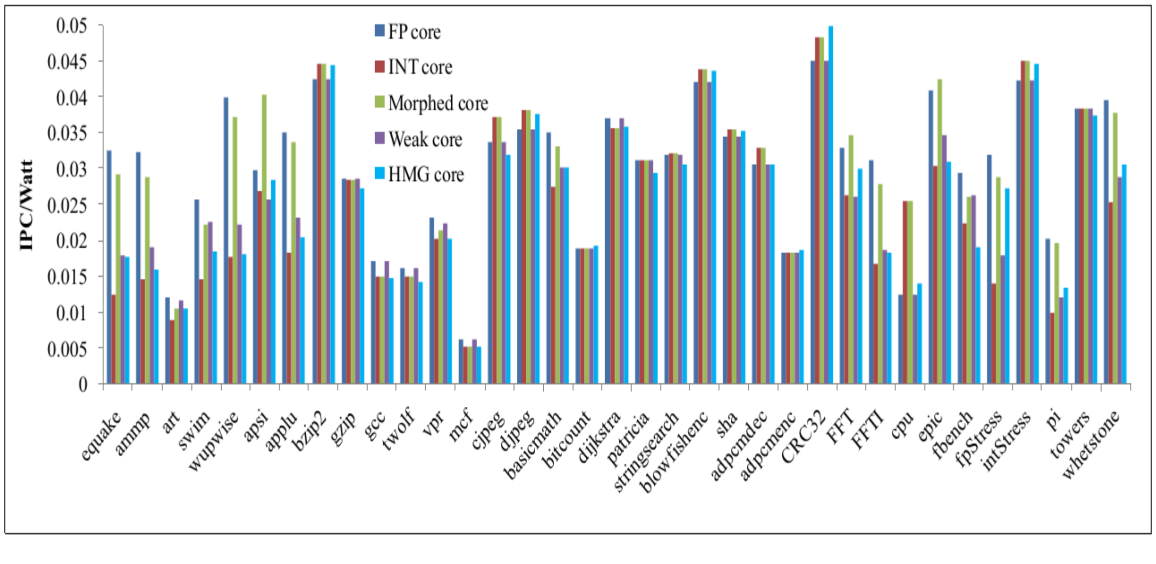


Figure 5.2. IPC/Watt of the considered benchmarks when run on each core configuration for 10 million instructions. Morphed core in the legend refers to the strong core.

poral distribution of the instruction mix in *apsi*. Having considered an architecture similar to ours, Das et al. [12] noted that whenever there is a phase in the program with a considerable mix of FP and INT instructions, the morphed strong core performs better than the others. Since the strong core can handle a mix of both FP and INT instructions, the performance is improved and at the same time resources are better utilized, and as a result a higher performance/Watt is achieved.

5.2.1 Inferences from performance and performance/Watt evaluations

We observed that over entire runs of 10 million instructions, some benchmarks benefit, some don't while some others even lose out. However, the above analysis reflects only the static behavior. But, many programs exhibit phases and each core configuration might be beneficial for different phases [32, 47]. Hence, running the benchmark statically on the same core may miss opportunities to maximize performance/Watt. This is the reason why only 3 out of the 38 benchmarks show performance/Watt benefits when run statically on the strong core throughout their execution. In rest of the cases, the power expended by running them on the strong core outweighs the obtained performance benefits resulting in poor performance/Watt metric. This is evident from Figure 5.1 where the strong core performs either equally well or better than the other core configurations when only IPC is considered.

In summary, the main inferences from the core evaluation experiments are:

- There is a need to use the morphed mode sparingly and it should be opted for only if the expected performance benefits outweigh the additional power overheads.

- Program phases may have a significant impact on choosing the right core configurations. This motivates us to study the impact of program phases in the next section.

5.3 Impact of program phases

In order to demonstrate the effect of program phases on performance/Watt, we consider the benchmark *epic* that shows benefit from morphing. We want to investigate the effect the instruction distribution of an application may have on performance/Watt. The benchmark *epic* was run for two billion instructions and the results are shown in Figure 5.3. The performance/Watt for each core type (FP, INT and strong) is represented by the blue, orange and red curves, marked with an $*$, a dot and a triangle ($\triangle$), respectively. The distribution of instruction types at each

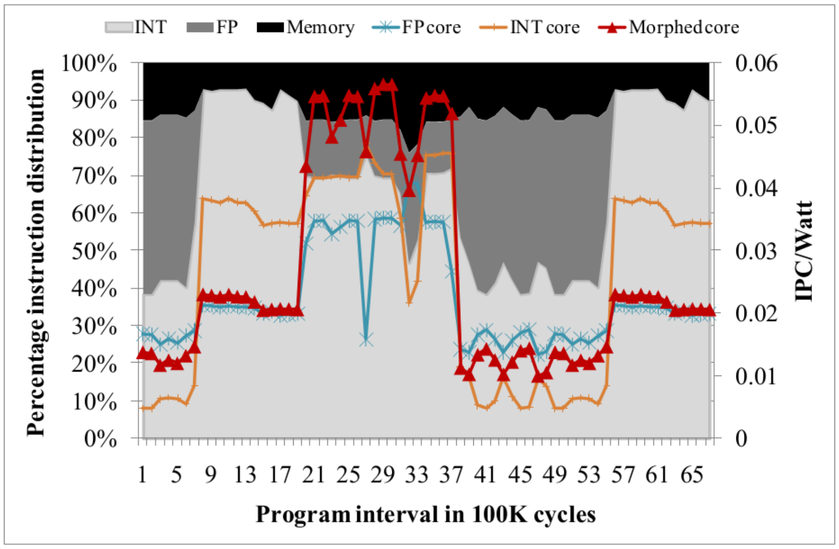


Figure 5.3. Zoomed view of variations in the performance/Watt of *epic* when run on each core configuration. Morphed core in the legend refers to the strong core.

time instant is represented by the area in the increasingly darker shades (light grey - INT, dark grey - FP, black - memory). It can be seen that for the first 19 data points, the strong core does not outperform either the FP or the INT core. Hence, staying in the baseline mode is advisable. However, for the data points 20 to 37, the strong core does much better than the other cores (35% on average when compared to the nearest competitor, the FP core). Hence, there is a possibility of considerable performance/Watt gains to be made here by morphing. After that, going back to the baseline mode once again proves beneficial. This shows that by monitoring the program behavior at a more fine-grain level, there are more opportunities for gains to be made by either morphing or coming out of it. At the same time, even though gains are made for *epic*, careful consideration must be given to the performance/Watt of the second thread running on the AMP which upon morphing gets assigned to the weak core, potentially resulting in a drop in its performance/Watt.

Thus, it can be seen that depending on the time-dependent behavior of an application, morphing or swapping may be the right choice. The decision whether to swap or morph should be based on the current instruction mix of the executing workloads.

CHAPTER 6

DYNAMIC RESOURCE ALLOCATION MECHANISM

Changing the voltage and frequency levels of the cores individually calls for voltage regulator modules (VRMs) on a per core basis which may be expensive for some architectures. Hence, we first evaluate dynamic resource allocation (DRA) mechanism as a stand-alone scheme to explore its benefits.

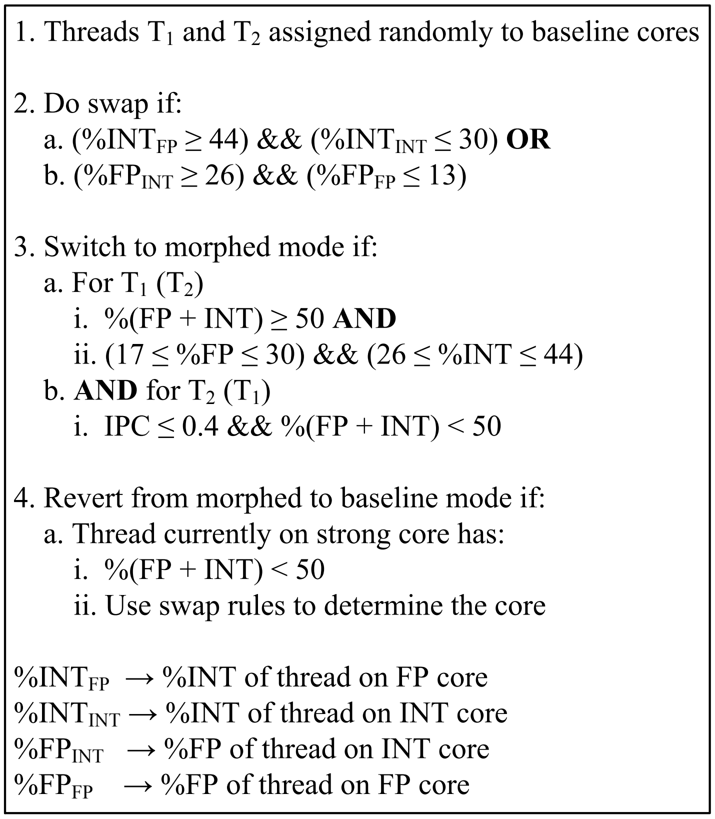
6.1 Hardware counters to trigger reconfigurations

Prior knowledge about the computational needs of the applications is generally unavailable. Hence, an online mechanism is needed to detect changes in the application's behavior that may impact performance/Watt and then decide whether to reconfigure the cores. Since power cannot be extracted at runtime, we use other program attributes as proxy for power when estimating performance/Watt.

From the study of *epic* in Section 5.3, we observed that there is a strong correlation between the performance/Watt achieved on different core-types and the current instruction distribution of the executing workload. Hence, we employ hardware counters to monitor the instruction composition (percentage of floating-point (%FP) and integer instructions (%INT)) of the workloads. Further, when switching to the morphed mode, the other thread gets executed on the weak core. We need to ensure that the performance of this thread is not greatly compromised. Therefore, in addition to the instruction composition counters, our DRA mechanism keeps track of the IPC of the threads. The employed counters are similar to those used by Khan et al. in [27]. We next describe the process that we have followed to deduce rules for morphing/swapping based on the instruction composition and IPC.

6.2 Offline Profiling

Offline profiling experiments were run to arrive at the suitable switching conditions for core reconfigurations. For our profiling experiments, twelve benchmarks from the suite of 38 (see Section 4.2) were chosen such that they included those that (i) benefit from morphing/swapping (*apsi, epic, fft*), and (ii) those that did not (e.g., *equake, art, applu*).



```
1. Threads $T_1$ and $T_2$ assigned randomly to baseline cores

2. Do swap if:
   a. ($\%INT_{FP} \geq 44$) && ($\%INT_{INT} \leq 30$) OR
   b. ($\%FP_{INT} \geq 26$) && ($\%FP_{FP} \leq 13$)

3. Switch to morphed mode if:
   a. For $T_1$ ($T_2$)
      i. $\%(FP + INT) \geq 50$ AND
      ii. ($17 \leq \%FP \leq 30$) && ($26 \leq \%INT \leq 44$)
   b. AND for $T_2$ ($T_1$)
      i. $IPC \leq 0.4$ && $\%(FP + INT) < 50$

4. Revert from morphed to baseline mode if:
   a. Thread currently on strong core has:
      i. $\%(FP + INT) < 50$
      ii. Use swap rules to determine the core

$\%INT_{FP}$  $\rightarrow$ %INT of thread on FP core
$\%INT_{INT}$ $\rightarrow$ %INT of thread on INT core
$\%FP_{INT}$  $\rightarrow$ %FP of thread on INT core
$\%FP_{FP}$   $\rightarrow$ %FP of thread on FP core
```

Figure 6.1. Rules for Dynamic Resource Allocation.

The threads were executed on each core type, and IPC, IPC/Watt and the instruction distributions were noted for fixed number of committed instructions, referred to as window. Once this data was available for each benchmark on all core types, two threads were chosen from the pool and after every window, the core configuration that yields the best IPC/Watt was identified. The corresponding instruction distribution of both the threads in those windows were also noted. For example, at the end of a window, while running a combination of *apsi* and *fft*, if it is noticed that the performance of running *apsi* on the strong core and *fft* on the weak core is higher than the baseline mode, this point (corresponding %INT and %FP of the threads) is marked as a potential switch point from baseline to morphed mode. Similarly, preferred switching points to come out of the morphed mode and to swap threads were identified. Averaging the values of %FP, %INT and IPC that we have observed for the 100 combinations of two (out of the 12) threads, we set the rules for reconfiguration shown in Figure 6.1.

It can be seen that for the *morphed* mode, we keep track of not only the floating-point and integer instructions, but also their sum. At the same time, minimum and maximum bounds are also set for the %FP and %INT individually, such that when these bounds are violated, the threads should continue to run on the baseline configuration. A morphed to baseline mode switch takes place when the total percentage of FP and INT instructions go below 50. At this point, all the benefits of morphing have diminished and it is better to operate in the baseline mode.

6.3 Weighted and geometric speedup definition

We have used weighted and geometric speedups extensively in this thesis as a measure of the achieved benefits. For example, weighted IPC/Watt improvement is used in the next section. Hence, we first define the metrics before using them.

$S_0 = (IPC/Watt_{thread0})_{proposed}/(IPC/Watt_{thread0})_{baseline}$

$S_1 = (IPC/Watt_{thread1})_{proposed}/(IPC/Watt_{thread1})_{baseline}$

$Speedup_{weighted} = (S_0 + S_1)/2$

$Speedup_{geometric} = \sqrt[2]{S_0 \times S_1}$

6.4 Accounting for program phase changes

As shown in Figure 6.2, a tentative decision based on the rules mentioned in Figure 6.1 is made at the end of every committed instructions window. However, to avoid too frequent reconfigurations we prefer to wait until the new execution phase

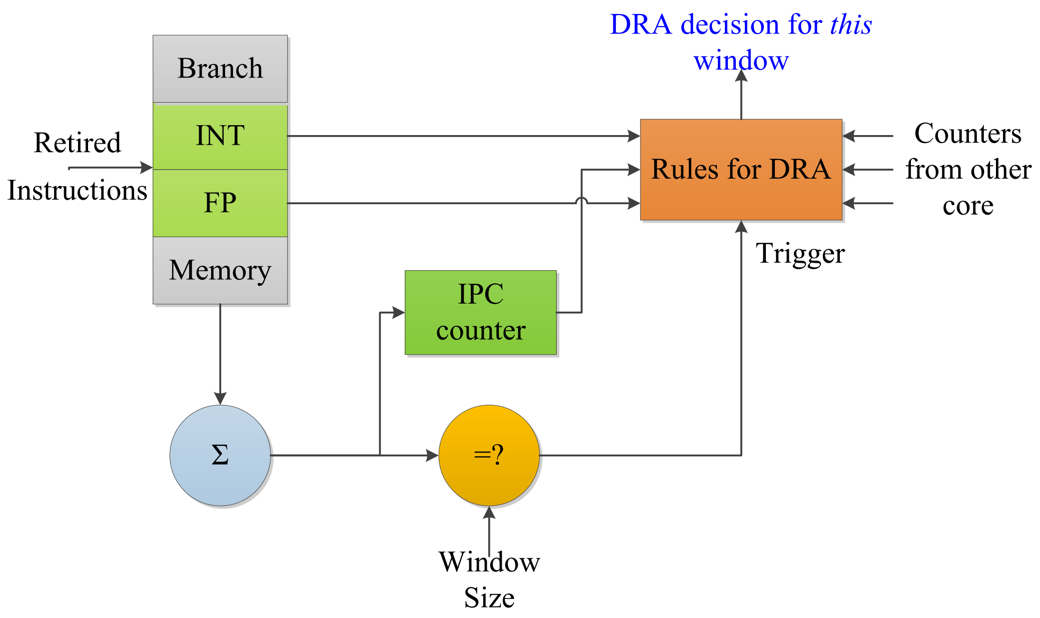


Figure 6.2. Tentative DRA decision for the current window using hardware counters.

of the thread has stabilized and only then switch from one mode to another. To this end, we base our reconfiguration decision on the most frequent tentative decision made during the n most recent instruction windows. For example, if in the last n windows, morphing was the most frequent decision, it may be predicted that the threads have entered a phase where morphing will yield the best results. We call the number of windows n after which a final reconfiguration decision is made as history depth.

Both the history depth and the size of the individual window have to be determined experimentally. We have conducted a sensitivity study to quantify their impact on the quality of the reconfiguration decisions. Various window sizes of 250, 500 and 1000 instructions were considered and the history depth n was varied from 3 to 20. For each combination of window size and history depth, about 100 multiprogrammed workloads were run with a random combination of benchmarks from our set of 38. All experiments were run until at least one of the threads completed 40 million instructions. A reconfiguration overhead of 1000 cycles has been considered in these experiments (discussed in detail in the next section). The weighted perfor-

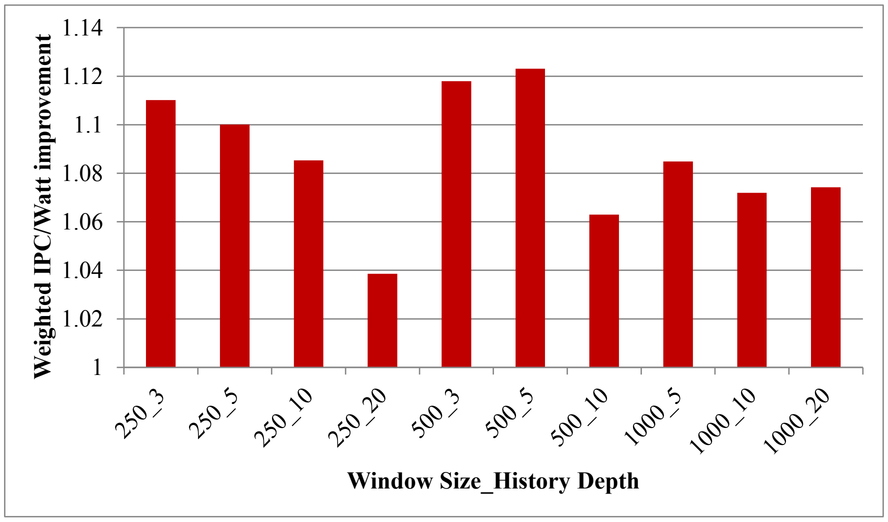


Figure 6.3. Sensitivity analysis for determining window size and history depth for DRA mechanism.

mance/Watt improvement of DRA over the static baseline configuration (shown in Figure 3.1) obtained from each individual experiment was then averaged to give a single value that represents the entire set that is shown in Figure 6.3. It can be seen that the best speedup is obtained for a window size of 500 instructions and a history depth of 5. Hence, we chose a window size of 500 instructions and a history depth of 5 for our experiments. We describe the overheads associated with the DRA mechanism in the next section.

6.5 Overheads associated with DRA mechanism

There are three main overheads that need to be considered for the proposed DRA mechanism: (i) hardware overhead, (ii) reconfiguration overhead, and (iii) communication overhead in the morphed mode.

6.5.1 Hardware Overhead

The first overhead is related to the additional hardware required to support core morphing. As shown in Figure 6.4, the FP operands are held in the reservation station until they are issued for execution. Depending on the mode of operation (baseline

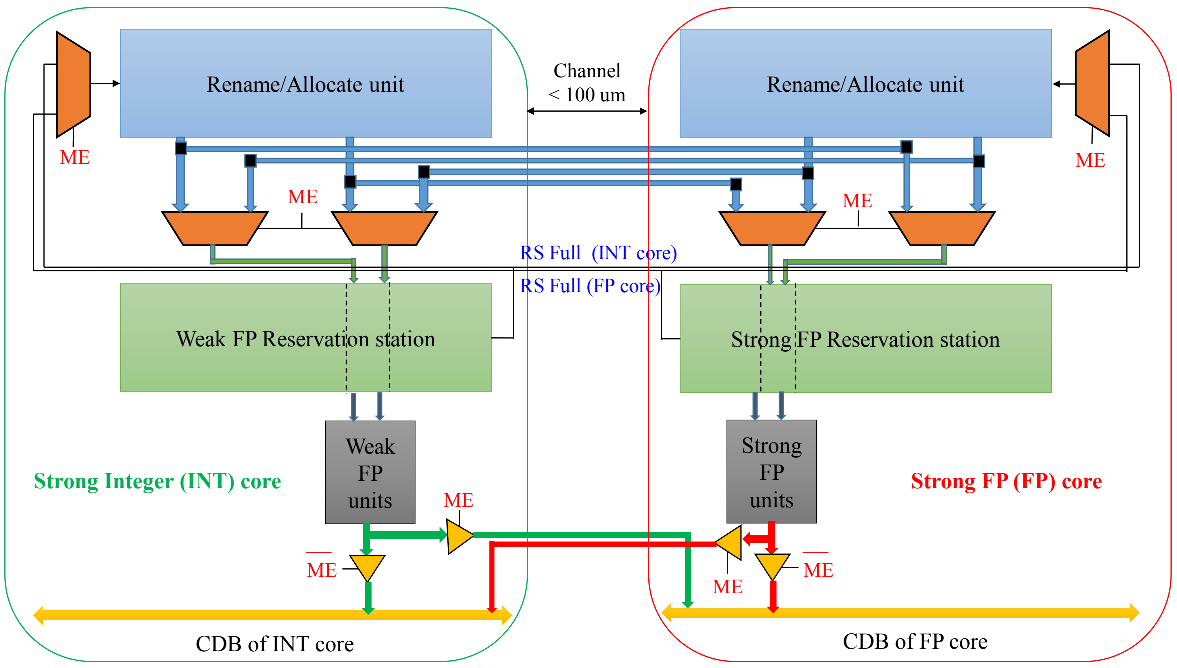


Figure 6.4. Hardware required to support core morphing.

or morphed), the operands for execution could come either from the same core or the other core. Hence, there is a need to first multiplex the operands from both the cores. The select signal for the multiplexers is the *morph enable* (ME) signal which indicates the current mode of operation. When the execution completes, the result of the FP operation is passed on to the common data bus (CDB) of the same core or the other core depending on the value of *morph enable*. One possible implementation using tri-state buffers is shown in Figure 6.4. Considering 32-bit FP operations, 64 2:1 multiplexers and 64 tri-state buffers per core, and 192 core-to-core communications are required for this purpose. The distance between the two cores is typically less than 100 μm and hence two inverters would be sufficient to send a signal from one core to the other.

In a conventional processor, when the reservation station is full, an “RS Full” signal is asserted that stalls further issuing of instructions. As the allocation can happen into the reservation station of either core in the proposed DRA mechanism, there is a need to multiplex the “RS Full” signals of both the cores. Table 6.1 lists the

Table 6.1. Complete hardware overhead to support core morphing

Gate type	Count
2:1 multiplexers	130
Tri-state buffers	128
Core-to-core communications	194
Inverters	388

complete hardware overhead to support core morphing. By this analysis, we observe that the hardware overhead is much less than 1% considering the total core area and gate count.

6.5.2 Reconfiguration Overhead

Core reconfiguration requires both the cores to stall execution. For swapping threads between the cores we need to flush the pipelines, exchange architecture states and warm the caches. Hence, the performance impact due to reconfiguration should be accounted for when considering the benefits of DRA.

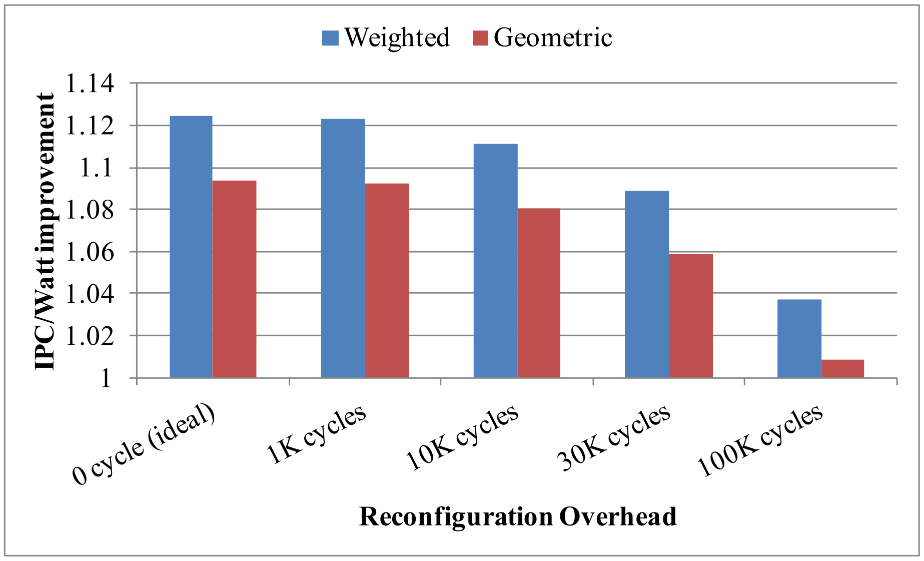


Figure 6.5. Impact of reconfiguration overhead on achieved performance/Watt benefits of using DRA over static baseline configuration.

To quantify the impact of reconfiguration overhead, experiments were run varying the penalty from 0 cycle (ideal case) to 100K cycles. The performance/Watt benefits achieved over the static baseline was used as the qualifying metric in these experiments. There were only 65 reconfigurations per run on an average while executing

40 million instructions. As shown in Figure 6.5, we observed only about 1.3% degradation in the achieved benefits when the reconfiguration penalty was increased from 0-cycle to 10K cycles. Even when the overhead was as high as 30K cycles, the proposed DRA mechnaism still achieved about 9% (6%) weighted (geometric) IPC/Watt improvement over the static baseline. Considering the current memory access latencies and processor-memory bus width [2], it would take less than 30K cycles to even sequentially refill the L1 caches (both instruction and data) of both the cores. This analysis shows that the proposed DRA mechanism has the potential to achieve significant performance/Watt benefits even when reconfiguration incurs a high penalty. We extended the above experiment to investigate the point at which the overheads of reconfiguration outweigh the achieved benefits. We found that only with a penalty of 100K cycles (50 μs for a 2 GHz processor) per reconfiguration, the achieved gains of the proposed scheme are almost nullified. With dedicated support for state swapping (e.g., Intel's Sandy Bridge [43]), far lower overheads can be expected and we used a reconfiguration overhead of 1000 cycles in our DRA experiments.

6.5.3 Communication Overhead

As mentioned in Section 6.5.1, the floating-point operands and results are transferred between the cores in the morphed mode. We analyzed the impact of this additional communication latency that arises due to the use of execution units that belong to one core by the other core. We ran experiments varying this communication latency overhead as 0, 1, 3, 5 and 10 cycles. As shown in Figure 6.6, the DRA mechanism achieves about 12.6% (9.6%) weighted (geometric) improvement in performance/Watt over the static baseline in the ideal case (0-cycle overhead) when the above communication happens without any cost. With a more realistic overhead of 1-cycle, the gains drop only by about 0.3%. Even in the extreme case when it takes 10 cycles to send the operands across, the DRA mechanism still achieves about 9.3% (a

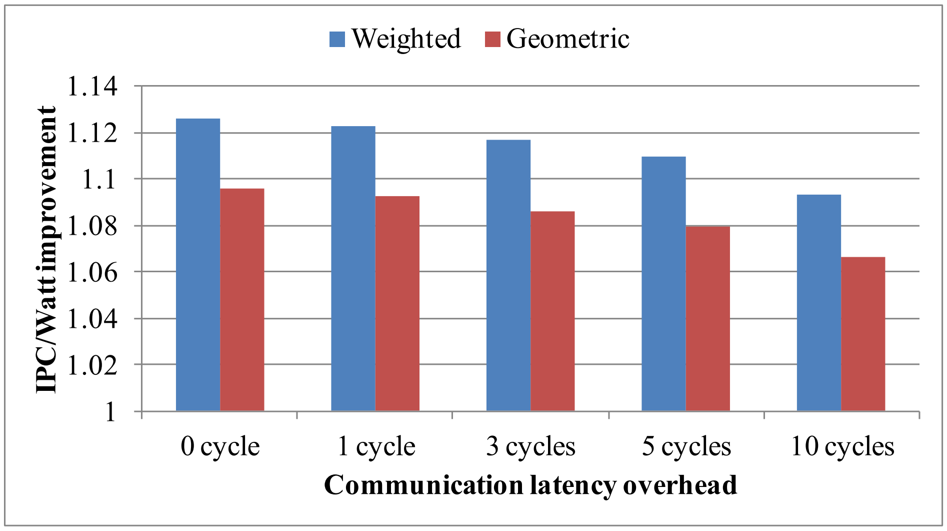


Figure 6.6. Impact of communication overhead on achieved performance/Watt benefits of using DRA over static baseline configuration.

drop of about 3.3% with respect to the ideal case) weighted IPC/Watt improvement over the static baseline. This analysis shows that even the communication latency overhead has very minimal impact on the achieved benefits of DRA. We have assumed a communication latency overhead of 1 cycle in all our experiments.

6.6 Evaluation

Having discussed the required preliminaries, we now present the evaluation of our DRA scheme. The performance/Watt achieved using our DRA scheme is compared against that of the homogeneous multicore and the baseline heterogeneous multicore in this section. For the heterogeneous baseline, we assume that the best thread to core assignment is known in advance while for our DRA scheme, a random initial thread to core assignment is made. Without loss of generality, all the baselines considered in this thesis were given this advantage (best initial thread-to-core assignment) while the proposed scheme starts with a random assignment. The hope is that the proposed scheme will detect the best assignment shortly after the programs begin to run. We first present an in-depth study for a single benchmark combination and then present the results for a large number of other benchmark combinations.

6.6.1 Detailed time-slice analysis of workload performance

An in-depth analysis for the benchmark combination {*applu, art*} is shown, at time slice intervals of 10,000 cycles, in Figure 6.7 with respect to weighted IPC/Watt improvement. For the combination {*applu, art*}, it can be seen that there are five

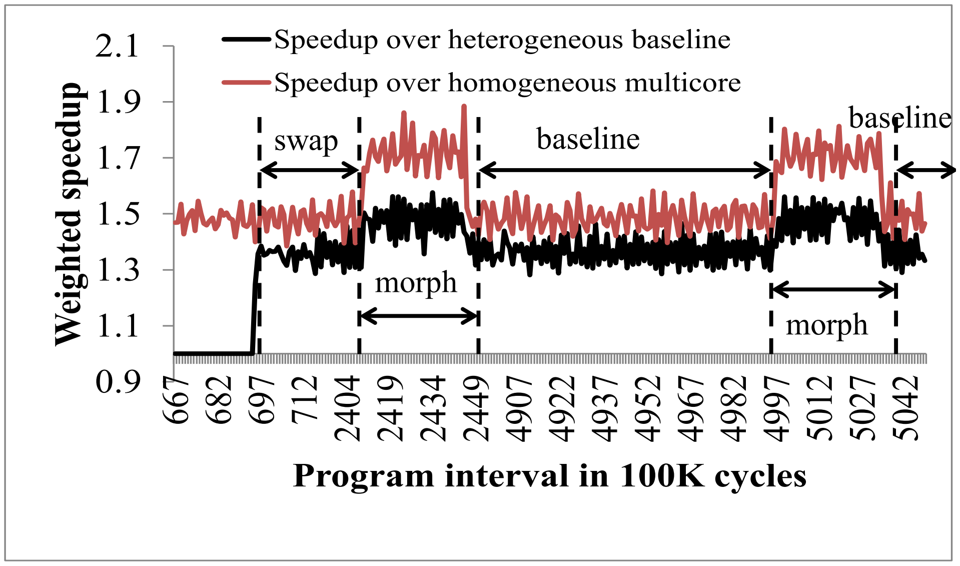


Figure 6.7. Weighted IPC/Watt speedup of DRA scheme vs. the homogeneous and heterogeneous baselines for {*applu, art*} combination.

reconfigurations: one *swap*, two *morph* and two back to *baseline* mode. Initially, up to data point 682, the DRA scheme performs as well as the static heterogeneous scheme as they both have the same initial thread-to-core assignment. However, the DRA scheme outperforms the homogeneous scheme in this region. This is due to the fact that both threads show different behavior (*applu* is more FP intensive and *art* is INT intensive in this phase) and since AMP is better suited to handle such workloads, there is a considerable benefit over the homogeneous baseline. Later, after data point 682, a swap of the threads take place and as a result, there is a jump in IPC/Watt when compared to the heterogeneous baseline, but not much of a difference when compared to the homogeneous multicore. This is because the homogeneous multicore is capable of handling all types of workloads and this particular change in the phase does not make much of a difference. The benefit over the heterogeneous multicore

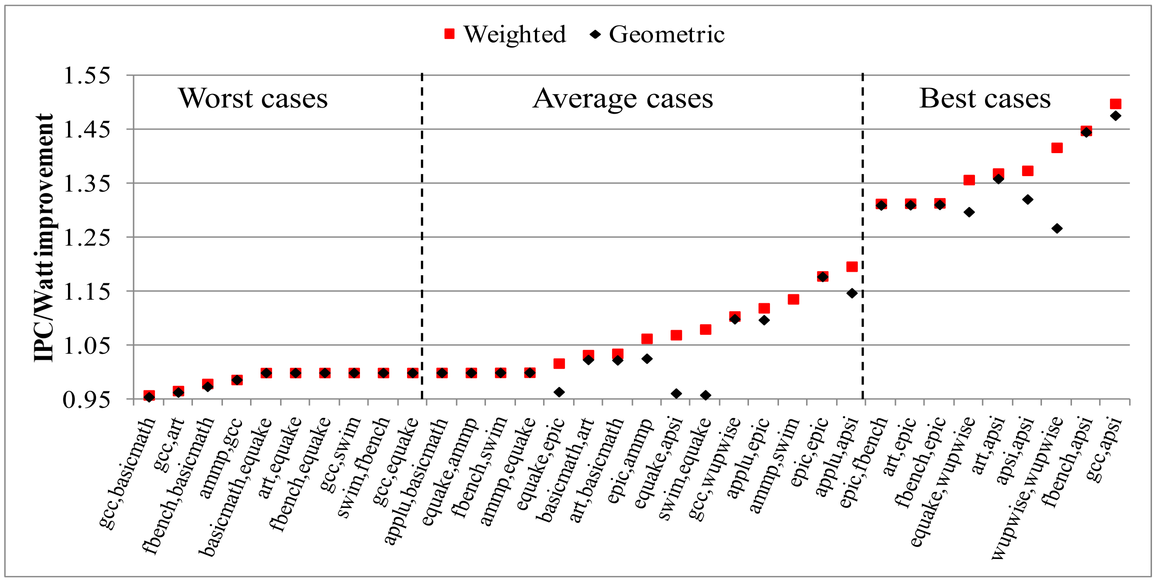


Figure 6.8. Performance/Watt improvement of DRA scheme over heterogeneous baseline for different multiprogrammed workloads.

can be attributed to the fact that the DRA scheme takes full advantage of the phase change. Then, *morphing* takes place at data point 2404 at which a sudden jump in speedup is observed for both the curves. But this jump is more pronounced in DRA vs. homogeneous curve. This is due to the fixed resources present in the homogeneous dual-core. As can be seen from the curves, even the heterogeneous baseline is better suited to the applications running on the multicore (due to their contrasting behavior) than the homogeneous one.

6.6.2 Overall Performance

Results are now presented for 35 combinations of benchmarks showing the weighted and geometric speedup with respect to the heterogeneous baseline and homogeneous multicore in Figures 6.8 and 6.9, respectively.

The 35 combinations were chosen out of a pool of 100 randomly generated benchmarks combinations where all 38 benchmarks participated and not just the 12 that were used to construct the DRA rules in Figure 6.1. The selected 35 combinations include the 10 worst results, the 10 best results and 15 that showed average benefits

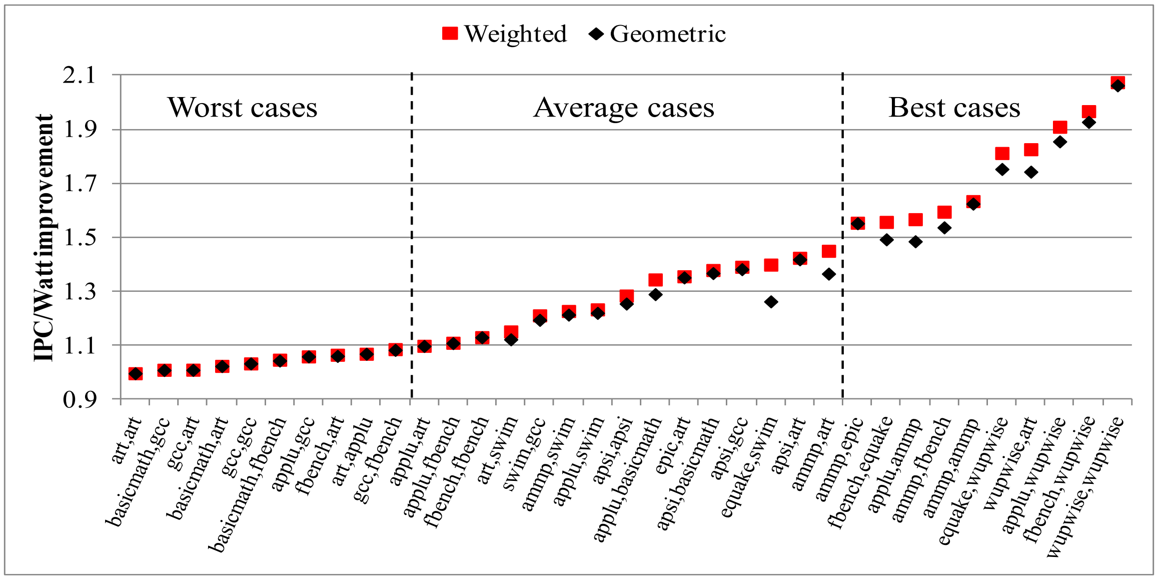


Figure 6.9. Performance/Watt improvement of DRA scheme over homogeneous baseline for different multiprogrammed workloads.

with respect to the weighted speedup. When comparing against the heterogeneous baseline (Figure 6.8), it can be seen that there are a few combinations (e.g., {*gcc, basicmath*}, {*fbench, basicmath*}) where the proposed scheme does slightly worse than the heterogeneous baseline (about 4.5% and 2.5% for the two cases). There are two possible reasons for this: (i) the thread to core assignment is random for our scheme and no reconfiguration takes place during the run, (ii) the scheme mispredicts. Case (i) happens when the two threads do not satisfy the swap/morph conditions at the same time and hence no change in the operating mode takes place. Case (ii) can happen occasionally for any prediction scheme. However, the proposed scheme achieved an average weighted (geometric) IPC/Watt improvement of about 12.3% (9.3%) over the static baseline configuration considering all the 100 combinations.

The results obtained when comparing the DRA scheme to the homogeneous multicore are shown in Figure 6.9. It can be seen that only for the symmetric workload combination of {*art, art*}, the proposed DRA mechanism performed worse than the homogeneous baseline. In general, the homogeneous baseline may perform better for workloads that do not exhibit distinct program phases and have the same flavor, i.e., either both are FP or INT intensive. However, when there are many program phase changes when executing symmetric workloads, the proposed DRA scheme does much better (consider {*ammp, ammp*} which shows about 63% benefit). On an average, for the 100 combinations, the proposed DRA scheme achieved a performance/Watt improvement of about 41% over the homogeneous baseline.

CHAPTER 7

RULE-BASED DYNAMIC RECONFIGURATION FRAMEWORK

We incorporate the Dynamic Voltage and Frequency Adaptation (DVFA) capability and present the complete rule-based dynamic reconfiguration framework (RDRF) in this chapter.

7.1 Extensions to include DVFA capability

The hardware counters and the reconfiguration rules discussed in Chapter 6 should now be extended to determine the appropriate operating conditions of the cores.

7.1.1 Hardware counters required

Our DVFA mechanism exploits the 'memory boundness' of the program. When the core is busy with memory intensive operations and the IPC is low, it is moved to the low power (LP) mode where both the voltage and frequency are lowered to values mentioned in Table 4.5. On the other hand, if the IPC of the thread is high and the core is busy servicing compute intensive operations, the voltage and frequency are boosted if the other core is in LP mode. The DVFA scheme reverts back to the default operating conditions when these modes are no longer beneficial, by monitoring the IPC of the threads.

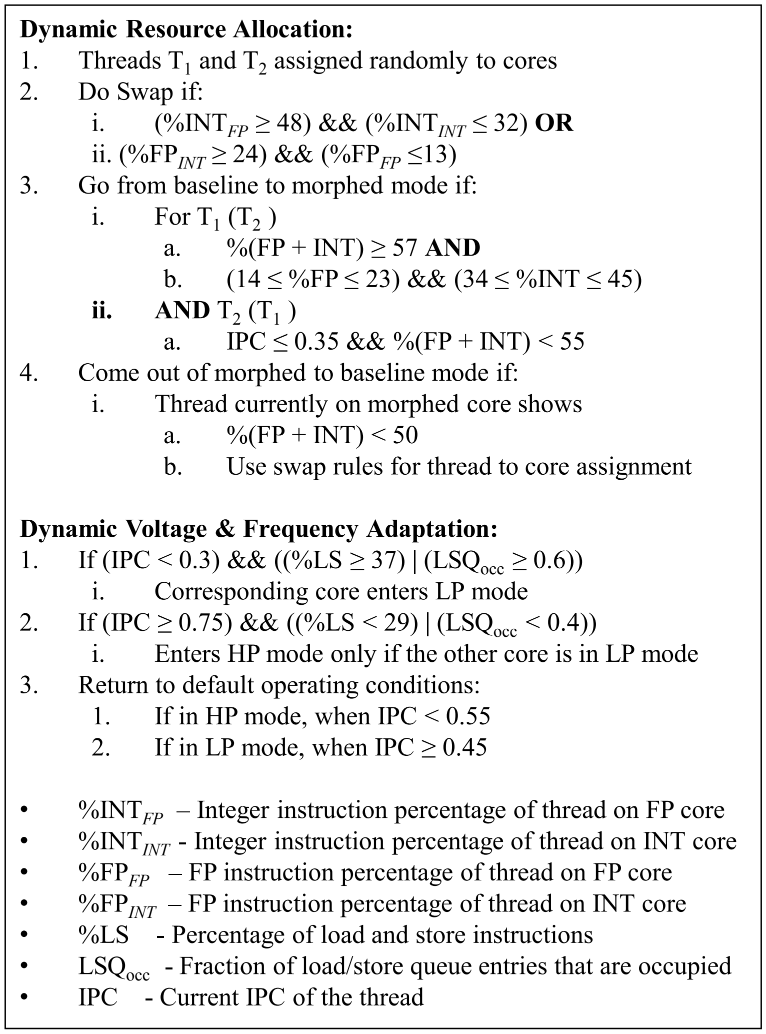
The 'memory boundness' of the program is tracked through the monitoring of the frequency of the load/store instructions (%LS) and is further strengthened by monitoring the Load/Store Queue (LSQ) occupancies of the cores. The complete set of counters employed by the rule-based DRF is shown in Table 7.1.

Table 7.1. Hardware counters used by RDRF

Counters	Parameters monitored in a window
Instruction Composition	%INT, %FP, %LS instructions
IPC	Current IPC of the threads
LSQ Occupancy	Fraction of occupied LSQ entries

7.1.2 Modifications to profiling experiments

The offline profiling experiments discussed in Section 6.2 were extended to determine rules for the DVFA mechanism. As the voltage/frequency levels of the cores are changed at runtime which impacts the cycle time, we used throughput (instructions per second (IPS))/Watt as the metric to determine the optimal switching points. The new rules thus developed for both the mechanisms are shown in Figure 7.1.



Dynamic Resource Allocation:

1. Threads T_1 and T_2 assigned randomly to cores
2. Do Swap if:
   i. $(\%INT_{FP} \geq 48)$ && $(\%INT_{INT} \leq 32)$ **OR**
   ii. $(\%FP_{INT} \geq 24)$ && $(\%FP_{FP} \leq 13)$
3. Go from baseline to morphed mode if:
   i. For T_1 (T_2)
      a. $\%(FP + INT) \geq 57$ **AND**
      b. $(14 \leq \%FP \leq 23)$ && $(34 \leq \%INT \leq 45)$
   ii. AND T_2 (T_1)
      a. $IPC \leq 0.35$ && $\%(FP + INT) < 55$
4. Come out of morphed to baseline mode if:
   i. Thread currently on morphed core shows
      a. $\%(FP + INT) < 50$
      b. Use swap rules for thread to core assignment

Dynamic Voltage & Frequency Adaptation:

1. If $(IPC < 0.3)$ && $((\%LS \geq 37) \mid (LSQ_{occ} \geq 0.6))$
   i. Corresponding core enters LP mode
2. If $(IPC \geq 0.75)$ && $((\%LS < 29) \mid (LSQ_{occ} < 0.4))$
   i. Enters HP mode only if the other core is in LP mode
3. Return to default operating conditions:
   1. If in HP mode, when $IPC < 0.55$
   2. If in LP mode, when $IPC \geq 0.45$

- $\%INT_{FP}$ – Integer instruction percentage of thread on FP core
- $\%INT_{INT}$ - Integer instruction percentage of thread on INT core
- $\%FP_{FP}$ – FP instruction percentage of thread on FP core
- $\%FP_{INT}$ – FP instruction percentage of thread on INT core
- %LS - Percentage of load and store instructions
- LSQ_{occ} - Fraction of load/store queue entries that are occupied
- IPC - Current IPC of the thread

Figure 7.1. Rules for DRA and DVFA.

7.2 Complete framework and role of Microvisor

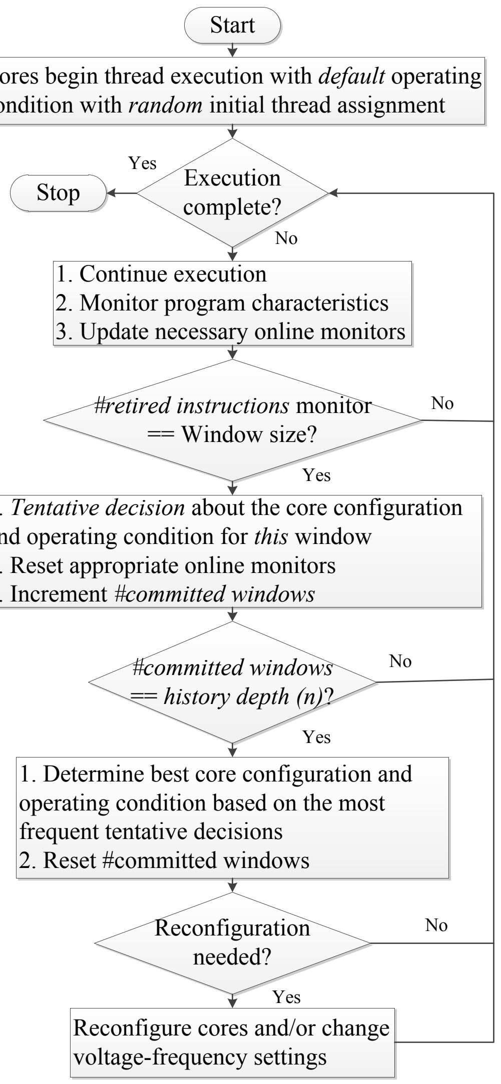

Start
Cores begin thread execution with *default* operating condition with *random* initial thread assignment
Execution complete?
Yes
Stop
No
1. Continue execution
2. Monitor program characteristics
3. Update necessary online monitors
#retired instructions monitor == Window size?
No
Yes
1. *Tentative decision* about the core configuration and operating condition for *this* window
2. Reset appropriate online monitors
3. Increment *#committed windows*
#committed windows == *history depth (n)*?
No
Yes
1. Determine best core configuration and operating condition based on the most frequent tentative decisions
2. Reset #committed windows
Reconfiguration needed?
No
Yes
Reconfigure cores and/or change voltage-frequency settings

Figure 7.2. Flowchart of the rule-based dynamic reconfiguration framework.

We have described the individual components of the proposed RDRF, namely: (i) the appropriate counters to monitor the program characteristics, and (ii) rules to trigger core reconfigurations and voltage/frequency adaptation. However, we still need a way to seamlessly govern the above autonomous mechanisms so that the intervention of the operating system (OS) can be limited. We envision a software layer called *microvisor* to perform this task. It is similar to IBM's millicode [21] and functions the way as proposed by Khan *et al.* in [28]. This software layer is invisible to the OS and is resident between the OS and hardware. The role of the microvisor could be better understood by studying the flowchart of RDRF shown in Figure 7.2.

To start with, a random initial assignment of the threads is made to the baseline INT and FP cores, which operate at default voltage and frequency. The counters mentioned in Table 7.1 non-invasively monitor the characteristics (instruction composition and IPC) of the threads and LSQ occupancy of the cores. At the end of every committed window, the microvisor is invoked to sample the counter values of both the cores at that time instant. The microvisor is aware of the established rules for DRA and DVFA shown in Figure 7.1.

After sampling the counter values, it applies the rules and makes a tentative decision about the best core configuration and operating condition for that window. A final decision is made by the microvisor after observing the last n windows. Upon a thread swap, the critical OS data structures (e.g., timer registers, interrupt vector addresses) pertaining to the two processes are also exchanged by the microvisor, thus completely isolating the OS. Further, whenever the voltage/frequency of the cores need to be changed, the microvisor takes the responsibility of writing the corresponding voltage ID (VID) values to the platform registers. Thus, the microvisor strives to relieve the OS from the low-level processor details while our proposed DRF works underneath to maximize performance/power.

The next section details the overheads associated with the proposed rule-based dynamic reconfiguration framework (RDRF).

7.3 RDRF Overheads

RDRF incurs the following overheads: (i) reconfiguration overhead for core morphing/thread swapping, (ii) communication latency overhead, (iii) DVFA overhead, and (iv) microvisor invocation overhead. The reconfiguration and communication latency overheads were discussed in Section 6.5. To be conservative, we assumed an overhead to refill one-fourth of the L1 caches in the new core-type upon a thread swap. Without loss of generality, this is the reconfiguration overhead used henceforth in this thesis. For the considered cache sizes of the cores (see Table 4.3), this is estimated to be about 1.75 μs ($\sim$3500 cycles).

The scheme incurs a higher overhead for DVFA. Firstly, changing the voltage levels of the cores (V_{cpu}) individually requires VRMs on a per-core basis. However, industry has moved in this direction and many of the current processors already support this capability. Secondly, the processor should be halted while the phase-locked loop (PLL) relocks to the new frequency. The PLL relock time in the latest Intel processors is 5 μs.

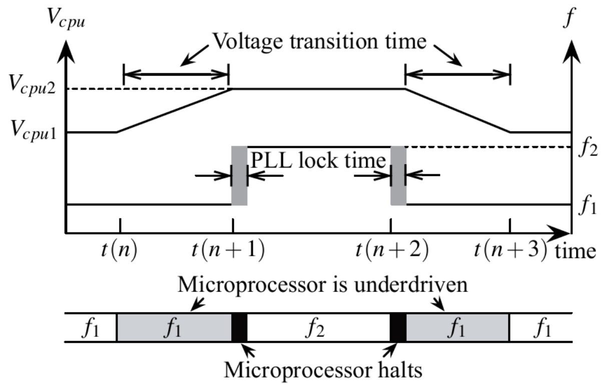


Figure 7.3. Overheads associated with DVFA. Figure taken from [39].

In addition to the PLL relock time, while scaling up the voltage/frequency, the processor operates at the lower frequency until the voltage has risen to the new value (see Figure 7.3) [39]. This performance loss during the voltage transition time should also be considered. Based on the DVFA overhead expressions deduced by Park *et al.* [39], the average performance loss for the considered voltage levels is about 3.5 μs.

Table 7.2. RDRF overheads

Type	Overhead
Thread swap	1.75 μs
Voltage/frequency downscaling	5 μs
Voltage/frequency upscaling	8.5 μs
Microvisor invocation	~0.29 μs

The microvisor invocation overhead is the most frequent of all as it happens for every committed instruction window. But, the associated overhead is relatively small as it only involves collecting the counter values of the two cores and evaluating the inequalities mentioned in Figure 7.1. This can be assumed to be at most a few hundred clock cycles and we observed this to have negligible impact on our results. In our experiments, we have conservatively assumed an overhead of 500 cycles for each microvisor invocation. Depending on the frequency of operation of the cores, the invocation overhead ranges between 0.23 μs and 0.29 μs. Table 7.3 presents the summary of the overheads considered in our RDRF experiments.

7.3.1 Accounting for program phases

A high-level picture of our rule based decision process is shown in Figure 7.4. Based on the conditions mentioned in Figure 7.1, tentative decisions regarding the core configuration and the appropriate voltage and frequency levels are made by the

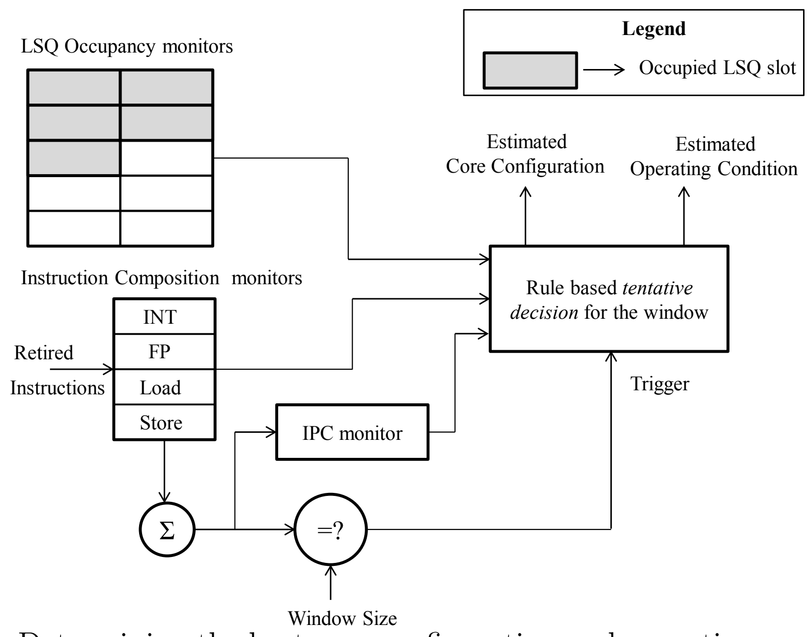

Legend
Occupied LSQ slot
LSQ Occupancy monitors
Estimated Core Configuration
Estimated Operating Condition
Rule based *tentative decision* for the window
Instruction Composition monitors
INT
FP
Load
Store
Retired Instructions
Trigger
IPC monitor
Σ
=?
Window Size

Figure 7.4. Determining the best core configuration and operating condition for the current window using hardware counters.

microvisor at the end of every committed instruction window. The chosen window size should be sufficiently large so that the microvisor invocation overhead (500 cycles) would be negligible. A sensitivity study was performed varying the window size from 25K to 75K, the results of which are shown in Figure 7.5. Here too, the metric used

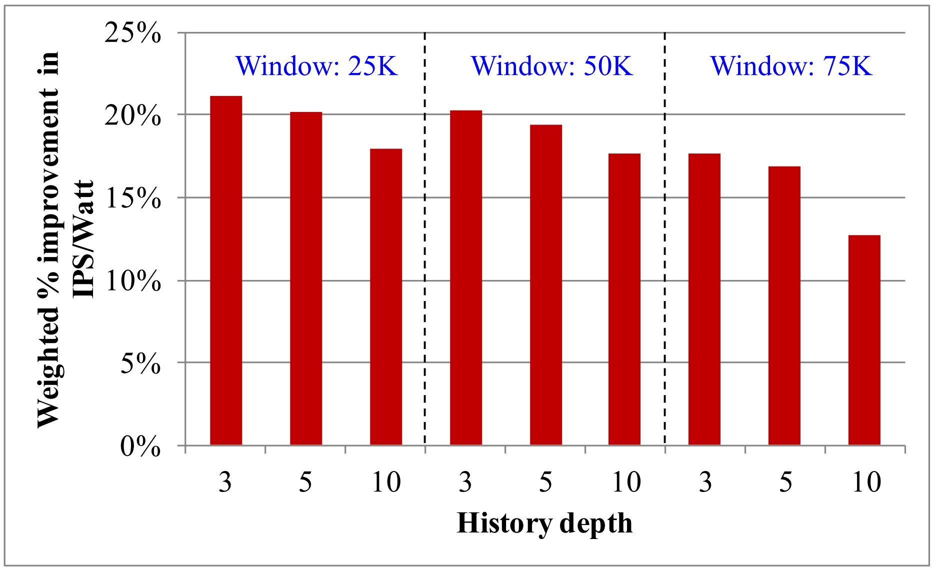


Figure 7.5. Sensitivity analysis for determining window size and history depth for RDRF.

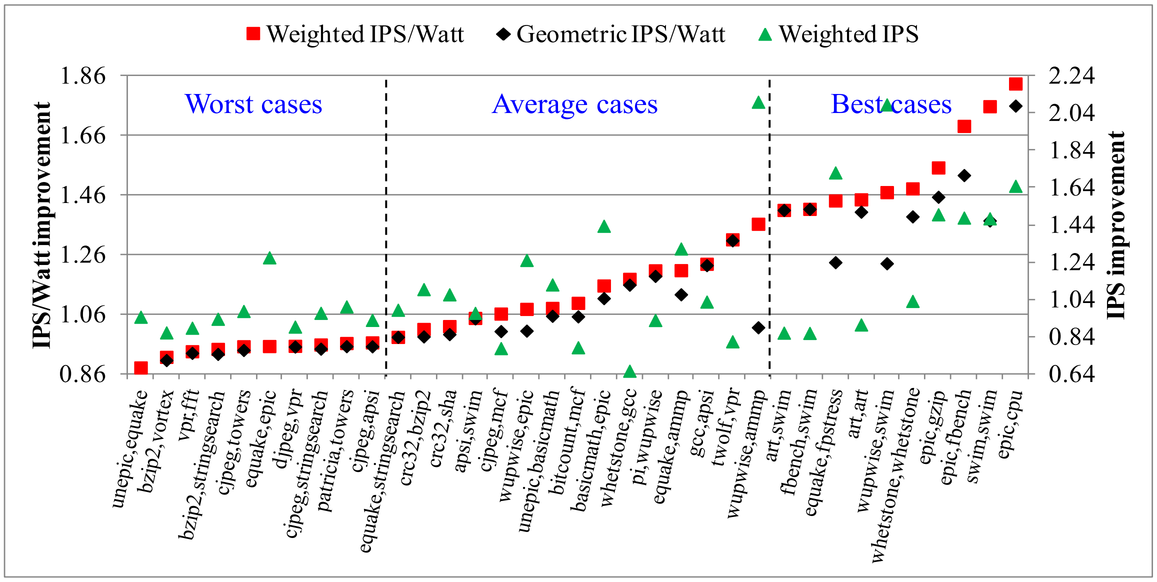


Figure 7.6. IPS/Watt and IPS improvement using RDRF over the static baseline for different multiprogrammed workloads.

was IPS/Watt instead of IPC/Watt. Significant weighted IPS/Watt benefits (21.2%) were achieved over the static baseline for a window size of 25K instructions and a history depth of 3, which is used in all our RDRF experiments.

7.4 Evaluation

To illustrate the efficiency of the proposed RDRF, we compare the throughput/Watt and throughput metric of our scheme against three baseline heterogeneous configurations – static, with DVFA capability only and with the DRA feature only. As before, we show only the 10 worse results (out of the 100), the 10 best results and 15 that showed average benefits in Figures 7.6, 7.7 and 7.8.

As shown in Figure 7.6, significant improvement in IPS and IPS/Watt was obtained using the proposed scheme when compared to the static baseline heterogeneous configuration which lacks the capability to adapt to the time-varying behavior of the workload. The only scenario where the static configuration was able to perform better or match RDRF was when no reconfiguration happened for the two-benchmarks

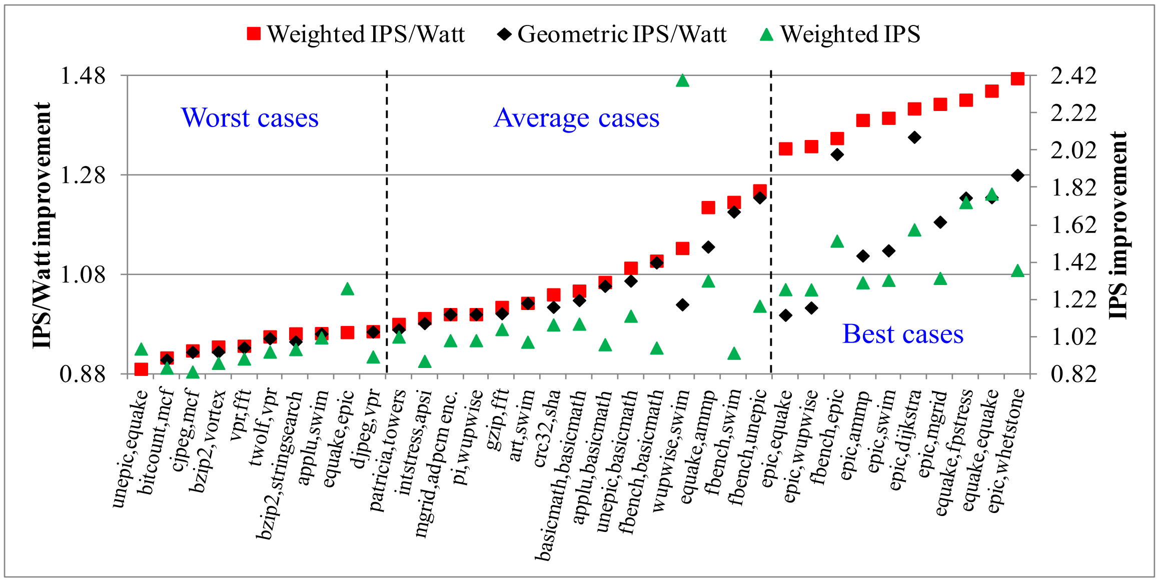


Figure 7.7. IPS/Watt and IPS improvement using RDRF over the DVFA-only baseline for different multiprogrammed workloads.

pairs (for example, {*unepic,equake*}, {*bzip2,vortex*}). Using the proposed scheme there was on average (for all the 100 combinations) a 21.2% (13.5%) weighted (geometric) improvement in throughput/Watt and about 21.7% weighted improvement in throughput over the static baseline.

As expected, relatively lower benefits were observed when comparing RDRF against the baseline configuration with either the DVFA only or the DRA only capability. These reference baseline configurations have some capability (either to change the voltage/frequency levels or morph the execution resources) to adapt to the time-varying behavior of the workload. It could be noted that for few combinations like {*bitcount,mcf*}, {*cjpeg,mcf*} in Figure 7.7 and {*equake,epic*}, {*djpeg,vpr*} in Figure 7.8, the proposed scheme performs worse than the two baseline configurations. There are three possible reasons for this: (i) The best thread-to-core initial assignment is assumed for the baseline configurations while it is random for the proposed scheme. This gives an added advantage to the baseline configurations when either no or very late reconfigurations happen using RDRF. (ii) Due to the dual feature of DVFA and DRA in the proposed RDRF, in a few rare cases the earlier transitions (either fre-

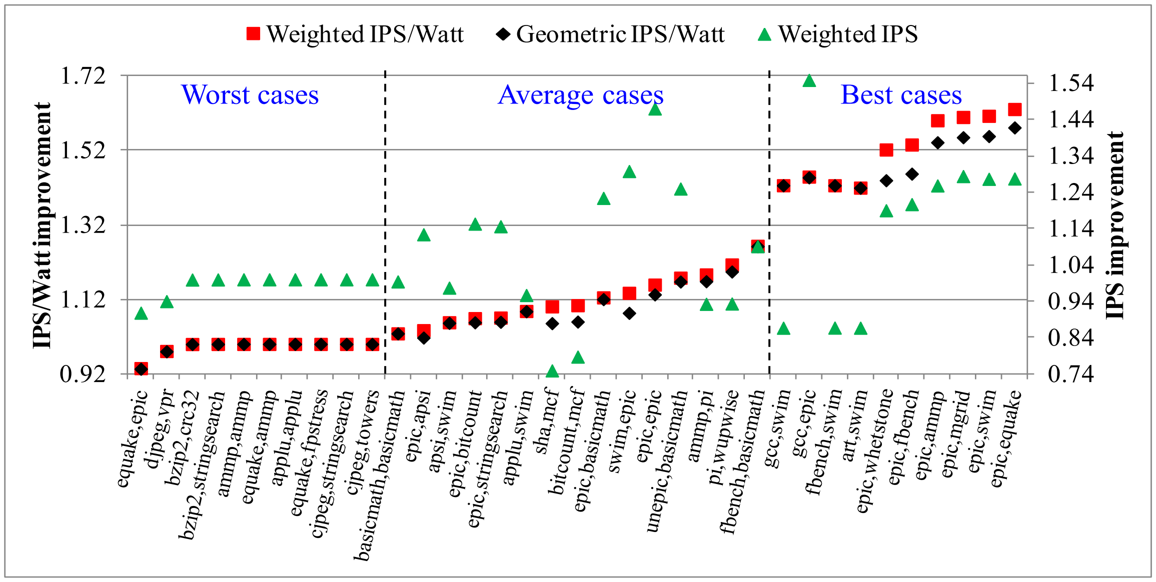


Figure 7.8. IPS/Watt and IPS improvement using RDRF over the DRA-only baseline for different multiprogrammed workloads.

quency/voltage adaptation or resource morphing) made by our scheme prevents some useful reconfigurations to happen in the future. For example, a decision to morph the cores could have been turned down by RDRF as the second core had been in the HP mode (and is about to come out of it) because of which its performance is slightly better than the defined conditions for a thread to be assigned to the weak core (see Figure 7.1). (iii) The scheme mispredicts. This could happen occasionally for any prediction scheme whose rules are determined by analyzing a subset of applications.

We also noticed many combinations that stressed the need for a scheme to have both DRA and DVFA. Consider for example, the pair {*art,swim*} for which voltage and frequency were scaled twice during the program execution for both the proposed scheme and the baseline configuration with DVFA. Hence, no significant IPS/Watt improvement was seen for this benchmark pair when compared against the baseline with only DVFA. However, the threads executed without any reconfiguration in the static baseline and the baseline with DRA-only capability. Hence, there was about 42% weighted increase in IPS/Watt for the mentioned workload pair compared to the static baseline configuration and the one with DRA alone. There were few other workload pairs like {*equake,ammp*}, {*equake,fpStress*} where the baseline configuration with DRA was able to follow the proposed scheme closely while significant benefits were achieved over the baseline with DVFA-only capability. These examples illustrate the capability of RDRF to satisfy the diverse needs of different applications and result in a significant performance/power improvement.

Considerable benefits are achieved by RDRF against both the non-static baseline configurations when workload pairs (like {*epic,ammp*}, {*fbench,swim*}) that require both DVFA and DRA to maximize performance/power are encountered. Moreover, the number of combinations that benefit from RDRF and result in a significant increase in IPS/Watt is much higher than the number of those that do not (only 11% of the 100 combinations showed >3% degradation compared to the baseline configuration with DVFA while the number of combinations that were slightly degraded was 5% compared to the baseline configuration with DVFA). On average, for the considered 100 combinations, there was about 12% and 16% weighted improvement in throughput/Watt using RDRF over the baseline heterogeneous configurations with DVFA-only and DRA-only capability, respectively.

7.5 Limitations of RDRF

Although the proposed RDRF achieves good performance/power benefits, the scheme suffers from the following limitations that need to be addressed:

- *Continuous monitoring:* The scheme requires continuous monitoring of the program characteristics though the number of reconfigurations on an average was small.

- *Scalability issue:* More importantly, developing DRF rules for a many-core system that has more than two cores per tile is complicated.

CHAPTER 8

PREDICTION-BASED DYNAMIC RECONFIGURATION FRAMEWORK

We introduce in this chapter a new reconfiguration framework that is based on prediction. The presented prediction-based dynamic reconfiguration framework (PDRF) tries to address the shortcomings of RDRF.

8.1 Overview of Prediction-based Dynamic Reconfiguration Framework

We have already noticed that the computational resource requirements of the threads usually change only when the programs change phases. Therefore, it may be sufficient to look for opportunities to reconfigure and/or change the voltage/frequency levels of the cores only when a phase change is detected for any of the threads. Thus, the *continuous monitoring* requirement of RDRF can be avoided. By opportunistically making such reconfiguration decisions only when a new phase is encountered, the associated overheads can be lowered further. In addition, an informed thread-to-core assignment can be made if we could predict the expected performance/power of the current program phase at different voltage/frequency levels on all the available core-types in the AMP. With such a prediction mechanism, the proposed scheme could be extended to a many-core system.

The above two features form the central idea of our PDRF: (i) opportunistic decision making and, (ii) predicting the expected throughput/Watt of the current phase on other core-types at different voltage/frequency levels. By covering the entire search space, an informed decision about the core configuration and the operating conditions is made. The prediction is made possible by employing hardware performance counters (HPCs) of the host core. A relationship is established between the values of these counters in the core executing the application and the expected throughput/Watt of this application if it would run on the other cores in the AMP and at different voltage/frequency levels. A high-level view of the proposed PDRF is shown in Figure 8.1.

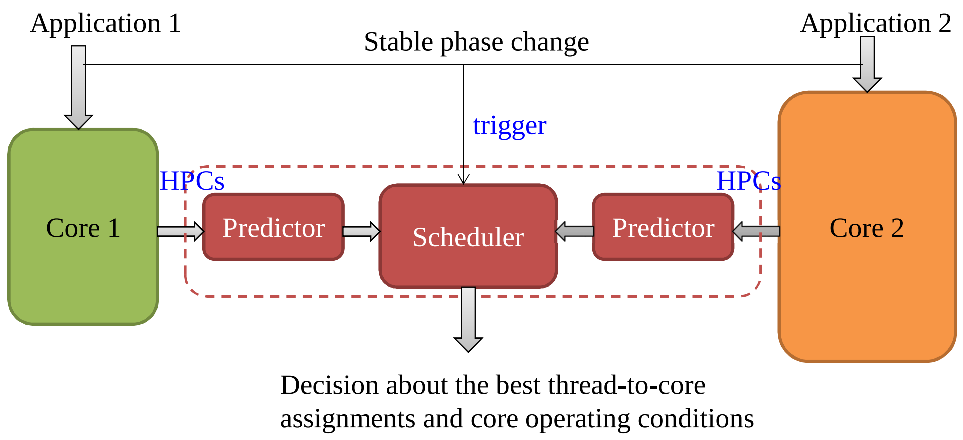


Figure 8.1. High-level view of PDRF.

8.2 Considered core configurations and operating conditions

As a proof of concept, we illustrate the benefits of PDRF using the considered baseline cores (INT and FP). In this prototype study, we do not consider morphing and focus only on thread swapping to explore its benefits. The latest Intel and AMD processors employ DVFS to a great extent for their “Enhanced Intel Speedstep Technology” and “AMD PowerNow! Technology,” respectively. Further, the frequencies of the cores are changed in a wider range in such processors. In alignment with that, we consider two power states for the baseline cores covering the extremes of the frequency spectrum. The considered voltage/frequency levels of the cores are tabulated in Table 8.1. Thus, the presented PDRF can be viewed as a **dynamic thread scheduling scheme** that makes informed thread-to-core assignments taking into ac-

Table 8.1. Voltage/Frequency levels considered for the baseline cores.

DVFS level	Operating voltage	Frequency
Level 1 (Normal mode)	1.1 V	2 GHz
Level 2 (Boost mode)	1.3 V	3 GHz

count the multiple voltage/frequency levels that may be available within the cores in the AMP. We next describe our phase detection and throughput/Watt prediction mechanisms that form the basis for PDRF.

8.3 Phase detection mechanism

To keep the overheads at bay, a good thread scheduling scheme should consider reassigning threads and/or changing the voltage/frequency levels only when a thread has moved to a new and stable phase. Therefore, there is a need to detect stable phase changes in a program even before determining the best thread-to-core affinity or the appropriate power state (voltage/frequency levels). The program phase detection mechanism should ignore short-lived unstable phases that do not warrant thread reassignment or change in core operating conditions.

A number of phase classification mechanisms have been proposed in the literature [13, 28]. After certain modifications, we adopt the phase classification scheme based on Instruction Type Vectors (ITVs) proposed by Khan *et al.* [26] owing to its simplicity. In their scheme, ITVs are formulated using hardware counters that count the number of committed instructions of certain types (9 in [26]) during a specified interval. A fixed number n of committed instructions constitute the above interval, with the value of n to be determined. The appropriate instruction counter is incremented whenever an instruction is retired. After the commit of n instructions, the resulting 9-element vector is captured and compared to the ITV of the previously identified phase. If the sum of differences between the instruction types of the previously encountered and currently executing phase is greater than a threshold, Δ (another parameter that needs to be determined), then this is potentially a new

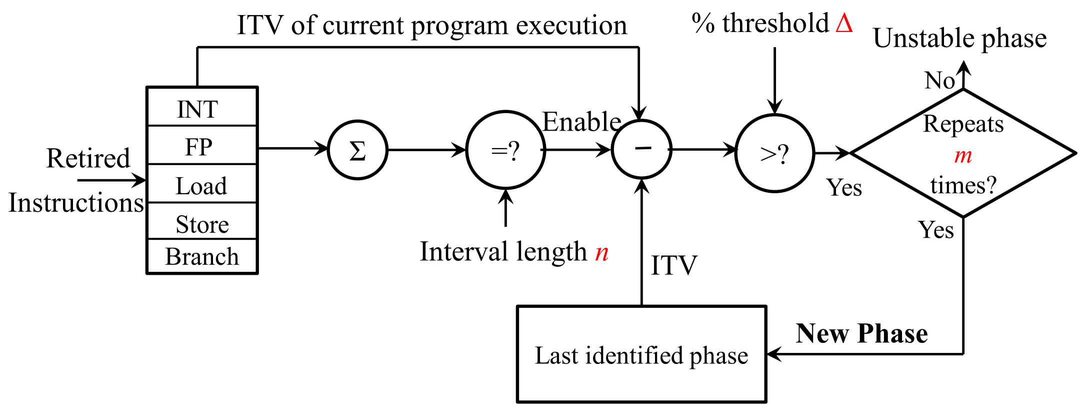


Figure 8.2. Our phase detection mechanism.

phase. The scheme qualifies a newly detected phase as stable only when at least m (the third and the last phase classification parameter that should be determined) consecutive intervals have their ITV differences smaller than Δ. Additional details about their scheme could be found in [26]. Due to the nature of the considered baseline cores (INT and FP), further classifying integer and floating-point instructions as ALU, multiply or divide does not offer any significant benefit. Therefore, we reduce the ITV from 9 to 5 elements corresponding to floating-point, integer, load, store and branch instructions. Our modified phase detection mechanism is shown in Figure 8.2. Khan *et al.* determined the phase classification parameters (n, m, and Δ) by experimentation. Since the baseline core configurations and the benchmarks that we consider are very different from those in [26], we have redone the experiments and found the parameters to be: (i) interval length n = 150K instructions, (ii) threshold $\Delta = 7.5\%$ and, (iii) $m = 4$ [42]. For every 150K instructions committed by either thread, the microvisor is invoked. The microvisor captures the current ITV and compares it to that of the previously identified phase to detect any phase changes. We have assumed an overhead of 500 cycles for every microvisor invocation as it involves executing a few instructions.

8.4 Determining program affinity online by predicting the expected throughput/Watt

Next, we need to determine online the affinity of the current program phase to the core-types and voltage/frequency levels in the AMP. The objective that our scheme tries to maximize is throughput/Watt which is the product of IPC/Watt and frequency. Since the frequency of each power state is known beforehand, the proposed scheme tries to non-invasively predict the expected IPC/Watt of the current phase at different operating conditions on both the core-types. Hardware performance counters (HPCs) have been observed to reveal significant information about the characteristics of the thread currently being executed [11, 49]. We therefore, decided to develop a scheme to predict IPC/Watt of an executing application on the host core, as well as on other cores in the AMP at all the available voltage/frequency levels using HPCs. To do so, we need to first identify a set of counters that could be used for estimation and then choose a small subset that would have the largest impact on IPC/Watt.

8.4.1 Hardware Performance Counters (HPCs) explored

We examined 14 different HPCs which can be grouped as follows. It is to be noted that none of the counter values would change by changing the voltage/frequency levels of the corresponding core.

• **Instructions per Cycle (IPC)**: Power consumption of the processor is dependent on its activity and the IPC counter provides a good measure of it.

• **Fetch counters**: The IPC metric considers only the retired instructions, but in a processor, many instructions are executed speculatively and then flushed from the pipeline. To account for these, we considered *# Fetched instructions (F)* and, *Branch mispredictions (BMP)*.

• **Miss/Hit counters**: Cache hits and misses play a significant role in performance or power consumption of a core. In this regard, the following event counters:

L1 hit (L1h), L1 miss (L1m), L2 hit (L2h), L2 miss (L2m) and, *TLB miss (TLBm)* are considered.

• **Retired instructions counters**: Performance or power consumption can vary significantly depending on the type of the retired instructions (*integer (INT), floating-point (FP), Load (Ld), Store (St), Branch (Br)*). Hence, we considered retired instructions counters.

• **Stalls**: The activity of the processor will be low when it experiences dependencies (data or resource conflicts) frequently. We consider stalls due to reservation stations, re-order buffer (ROB), load/store queues (LSQ), register renaming and RAT (Register Alias Table). We refer to this counter as *Stalls (S)*. A single unified counter is assumed for this purpose which is incremented whenever the corresponding structure is full and an attempt is made to allocate a new entry.

8.4.2 Performance/Power Modeling

As power cannot be extracted at runtime, there is a need to estimate IPC/Watt even on the same core at the current operating condition. This results in total of 4 predictions (2 for each core and 2 for each operating mode) within the same core. Further, to make thread swapping decisions, we need to predict the expected IPC/Watt of a thread running on INT (FP) core, on FP (INT) core at both operating modes. This accounts for another 4 predictions thereby increasing the total number of predictions required to 8.

Our intent is to use the least number of counters (from the available 14) to predict IPC/Watt at a reasonably high precision. The objective of this is not to save hardware, but, to minimize the number of counters that need to be monitored simultaneously. Once the right set of counters is chosen, we could employ multi-dimensional curve fitting and regression analysis to obtain expressions for IPC/Watt using the selected counters. To perform this analysis, we identified 12 represen-

```
1. Initialize:
   a. selected_counters = NULL
   b. untried_counters = {All 14 counters}

2. for i = 1 to total #counters
        a. Select a counter $C_i$ from the untried_counters set that best fits
           IPC/Watt along with the list of counters in selected_counters set
        b. Exclude $C_i$ from untried_counters set and add it to
           selected_counters set
        c. Store $R^2$ coefficient and selected_counters set corresponding to this
           iteration
   end

3. Plot $R^2$ coefficient obtained for each iteration

4. Explore selected_counters around the saturating region of the plot. Choose
   the one that offers a good trade-off between accuracy and the #counters
```

Figure 8.3. Pseudocode of our counter selection algorithm.

tative benchmarks from the set of 38, such that they include: integer intensive (*intStress,bzip2,gzip*), floating-point intensive (*fpStress,equake,ammp*), load/store intensive (*gcc,whetstone,swim*) and, branch intensive (*mcf,twolf,art*) benchmarks. These 12 benchmarks were run on both the cores at the two operating modes for 1 billion instructions, after skipping the initial 5 billion. The value of the 14 performance counters along with the observed IPC/Watt were sampled periodically after the commit of every 150K instructions (equal to the interval length n used in phase detection mechanism). All the obtained counter values were normalized with respect to the interval length n so that the same IPC/Watt expressions could be used for a different interval length while making runtime thread scheduling decisions.

8.4.3 Our counter selection approach

To accomplish the task of making the right choice of HPCs, we devised an efficient heurestic that searches the counter space iteratively. During each iteration, our counter selection algorithm picks a new counter that best fits IPC/Watt along with the set of counters already chosen in the previous iterations. We tried only linear

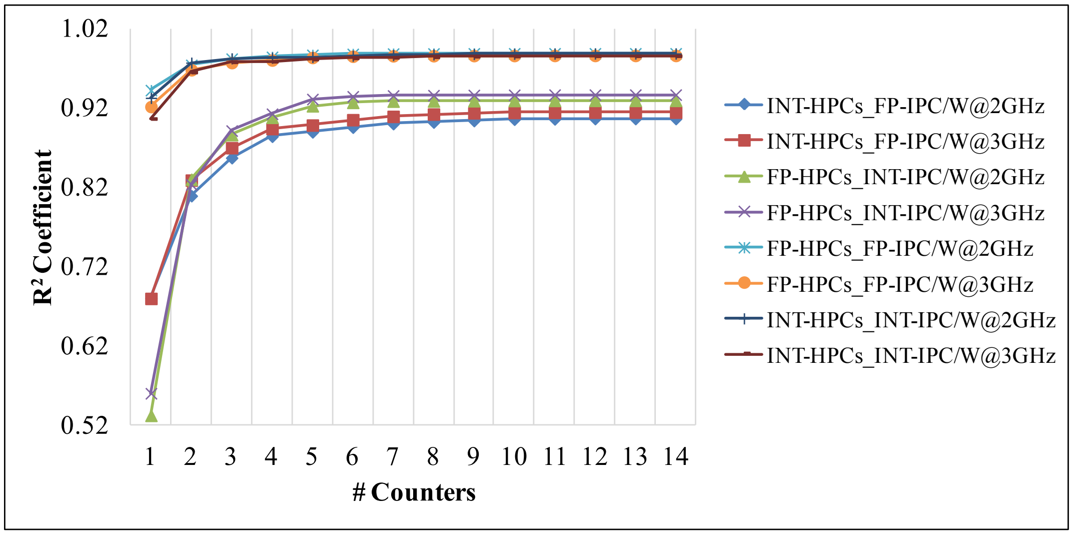


Figure 8.4. Variation in the R^2 coefficient with increasing number of HPCs while predicting IPC/Watt on the same core and the other core. The HPCs of the first core-type in the legend name is used to estimate the IPC/Watt on the second core-type in the legend name at the mentioned operating frequency. For example, legend INT-HPCs_FP-IPC/W@2GHz corresponds to IPC/Watt prediction on the FP core in the normal mode using the HPCs of the INT core.

models for curve-fitting and the best fit is qualified by the R^2 coefficient. During the initial few iterations, the value of the R^2 coefficient increases steeply as more counters are added, but it tends to saturate later. The best set of counters is around the region where the R^2 coefficient tends to saturate. The pseudocode of our counter selection algorithm is shown in Figure 8.3.

Figure 8.4 shows the value of the R^2 coefficient obtained during each iteration of the algorithm while predicting IPC/Watt both on the same core and the other core. It is evident that a reasonably high value of the R^2 coefficient is achieved for the same core predictions (the top 4 curves in Figure 8.4) and it saturates after 2 counters. Consequently, we used only two counters for IPC/Watt estimation on the same core. However, the value of the R^2 coefficient achieved while predicting the IPC/Watt on the other core (the bottom 4 curves) by using the HPCs of the host core is significantly lower than that on the same core (the top 4 curves). Further, the curves tend to saturate only after the fourth iteration indicating that 4 or more counters

Table 8.2. Union of HPCs chosen for the other core prediction during the first 4 iterations.

Iteration #	Union of HPCs
1	*L1m, IPC*
2	*L1m, IPC, BMP, FP*
3	*L1m, IPC, BMP, FP*
4	*L1m, IPC, BMP, FP, St*

Table 8.3. IPC/Watt expressions obtained for the normal mode.

HPCs of/Prediction on	Expression
INT/FP	$-0.12 \times L1m - 0.34 \times BMP + 0.01 \times IPC + 0.02 \times FP + 0.04$
FP/INT	$0.03 \times IPC - 0.07 \times FP + -0.71 \times BMP - 0.04 \times St + 0.04$
INT/INT	$0.02 \times IPC - 1.3 \times 10^{-2} \times S + 0.03$
FP/FP	$0.04 \times IPC - 4.6 \times 10^{-2} \times L1H + 0.02$

of the host core may be needed to adequately predict the IPC/Watt on the other core. The union of HPCs chosen during the first 4 iterations of the algorithm while predicting the IPC/Watt on the other core (corresponding to the bottom 4 curves in Figure 8.4) is shown in Table 8.2. As can be seen from the table, considering all the 4 cases, there were only 5 different HPCs chosen at the end of the fourth iteration. This indicates that there are common HPCs (e.g., BMP, FP, IPC) that are used for different predictions. We found the prediction accuracy achieved using 4 counters to be adequate and the same was employed for the other core estimation. For the sake of brevity, we show only 4 out of the 8 expressions that correspond to IPC/Watt prediction in the normal mode in Table 8.3.

8.4.4 Evaluating the accuracy of IPC/Watt prediction

We evaluated the accuracy of our prediction using all the 38 workloads, not limiting to the trained 12. The average absolute percentage error in IPC/Watt estimation for all the 8 cases is shown in Figure 8.5. Due to better quality of fit (higher value of R^2 coefficient), a much higher accuracy (average error of less than 10%) was achieved for estimating IPC/Watt on the same core when compared to the other core. In con-

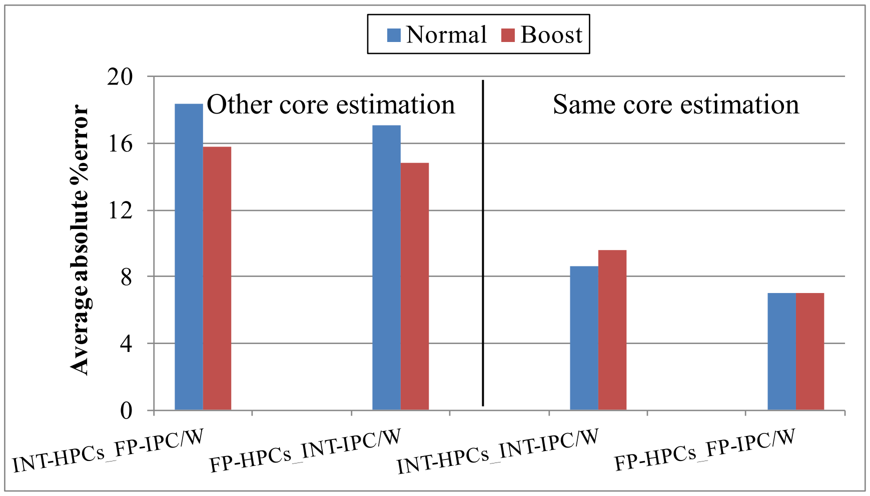


Figure 8.5. Average absolute percentage error in IPC/Watt estimation. Description of names in x-axis: HPCs of the first core-type is used to estimate the IPC/Watt (IPC/W) on the second core-type.

trast, the maximum average error was about 18.4% when predicting the IPC/Watt on FP core in normal mode using the HPCs of INT core. Overall, our proposed scheme achieves reasonably high accuracy in predicting the IPC/Watt both on the same core and on the other core at different core operating conditions.

In addition, we also analyzed the distribution of the error in IPC/Watt prediction. For the sake of brevity, only the error distribution for the worst case (IPC/Watt estimation on the other core) is shown in Figure 8.6. It is evident from Figure 8.6 that most of the sample points are contained within +/- 1σ, reflecting the high accuracy of our prediction scheme. About 92% (95%) of the samples corresponding to IPC/Watt estimation on INT (FP) core using the HPCs of FP (INT) core fall within this range. This is inline with our expectation of having low prediction error and hence, we could expect our prediction scheme to make good thread scheduling decision most of the time.

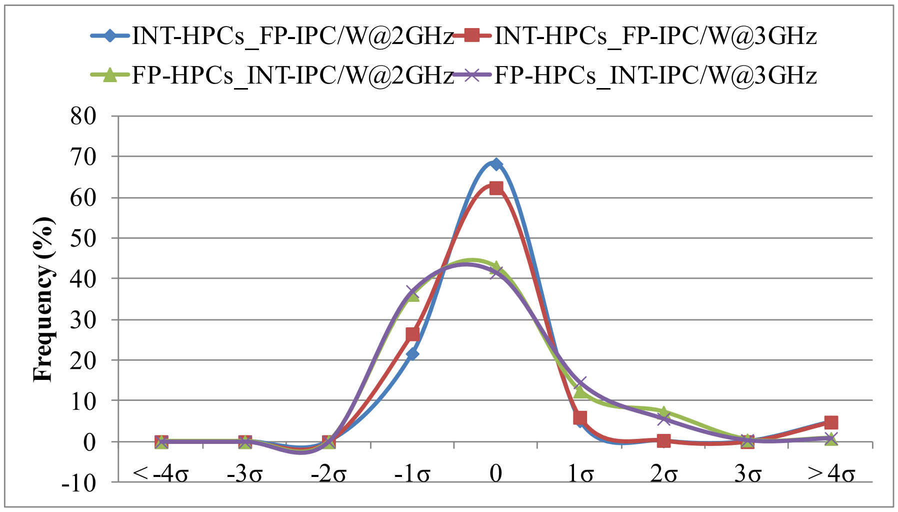


Figure 8.6. Distribution of error in estimating IPC/Watt on the other core using HPCs of the host core. Horizontal axis indicates the number of standard deviations by which the observations are off from the mean (average %error) while the vertical axis indicates the frequency of such occurrences.

8.5 Complete framework

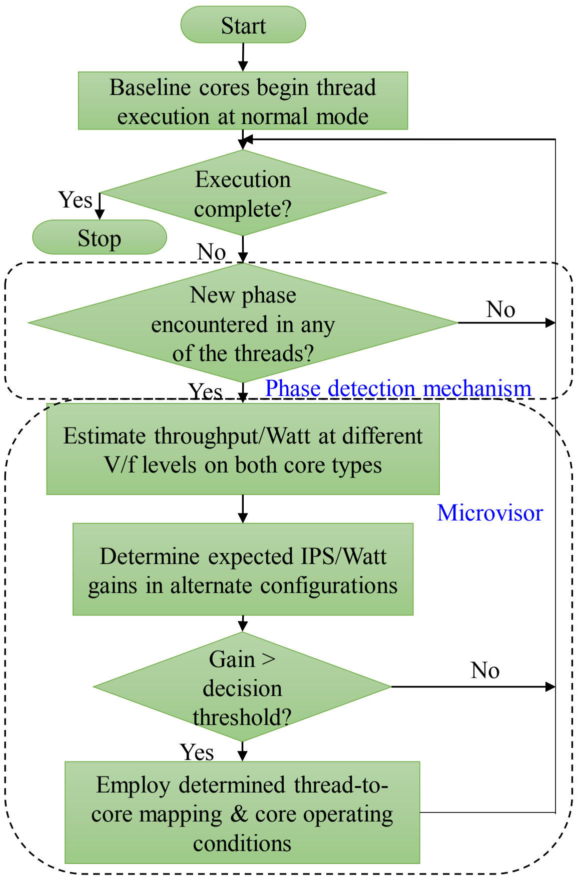

Start
Baseline cores begin thread execution at normal mode
Execution complete?
Yes
Stop
No
New phase encountered in any of the threads?
No
Yes
Phase detection mechanism
Estimate throughput/Watt at different V/f levels on both core types
Microvisor
Determine expected IPS/Watt gains in alternate configurations
Gain > decision threshold?
No
Yes
Employ determined thread-to-core mapping & core operating conditions

Figure 8.7. Flowchart of PDRF.

Figure 8.7 shows the flowchart of PDRF. We assume the microvisor discussed in Section 7.2 to coordinate the predictions and thread scheduling decisions whenever a new phase is detected for either of the threads. Whenever a stable phase change is detected for one of the threads, the microvisor is invoked to predict the expected throughput/Watt of the current execution phases of both the threads at different operating modes on the two core-types. This prediction is done using the current values of the chosen HPCs of the corresponding host cores.

Based on the above predictions, the microvisor calculates the projected geometric throughput/Watt gain (weighted metric could also be used) in moving to each of the possible new states (combinations of different thread-to-core assignment and voltage/frequency levels) over the current one. If the maximum geometric speedup is greater than 5% (called decision threshold to account for the swapping and DVFS overheads; detailed sensitivity study was conducted to determine this value), the corresponding new thread-to-core assignment and core operating conditions are opted for. Else, the current thread-to-core mapping and core operating conditions are maintained.

The PDRF incurs the same overheads as that of RDRF (see Table 7.3). But, the performance loss is as high as 25 μs for the voltage/frequency levels mentioned in Table 8.1. Hence, whenever the scheme switches from normal to boost mode, an upscaling overhead of 30 μs (5 μs for relocking the PLL and 25 μs for performance loss) is incurred. A discussion on the scalability of PDRF is presented next.

8.6 Discussion on scalability of PDRF

The proposed PDRF is based on the throughput/Watt prediction of the current program phase at different voltage/frequency levels on the available core-types in the AMP. We expect the methodology to remain the same if only the number of cores (with same core-types and operating modes) in the system increases. However, increasing the core-types and/or the operating modes increases the search space significantly. For the considered dual-core AMP with two operating modes, the number of potential next states for the current configuration is 7. Just by increasing the operating modes from 2 to 3, the number of potential next states increases from 7 to 17. For such cases, making an integrated decision about the thread-to-core mapping and core operating conditions may become too time consuming. Therefore, we may need to employ a sequential approach of determining the best thread-to-core

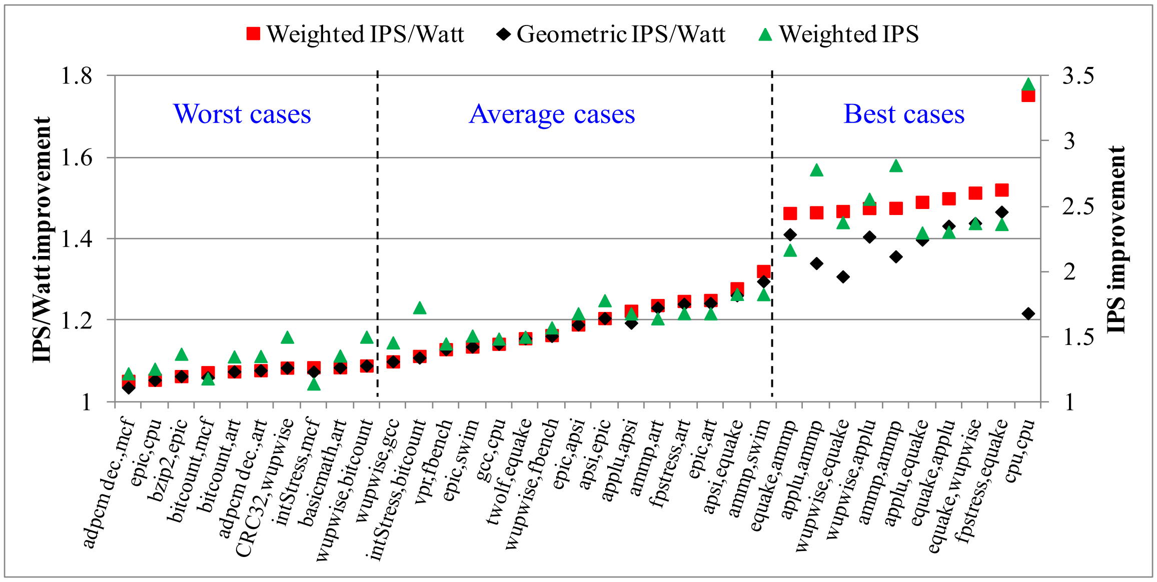


Figure 8.8. Throughput/Watt and throughput improvement using PDRF over the static baseline for INT/FP dual-core AMP.

mapping first and then choosing the appropriate voltage/frequency levels of the cores individually. By this, we would be able to drastically reduce the number of potential next states for the current configuration. Irrespective of either integrated or sequential decision-making, the proposed PDRF could be deployed for a many-core system. Having discussed the necessary requisites, we evaluate PDRF in the next section.

8.7 Evaluation

We compare the PDRF against the static, *swap-only*, and *DVFS-only* baselines by running 100 random combinations of 2-threaded workloads. As before, only the baselines were given the advantage of best initial thread-to-core assignment while a random assignment was assumed for PDRF. It should be noted that the trigger for both *swap-only* and *DVFS-only* baselines is phase detection. Further, both the baselines employ the same mechanism, i.e., throughput/Watt prediction to make their respective reconfiguration decisions. A detailed analysis of the comparison against each of the baselines is presented next.

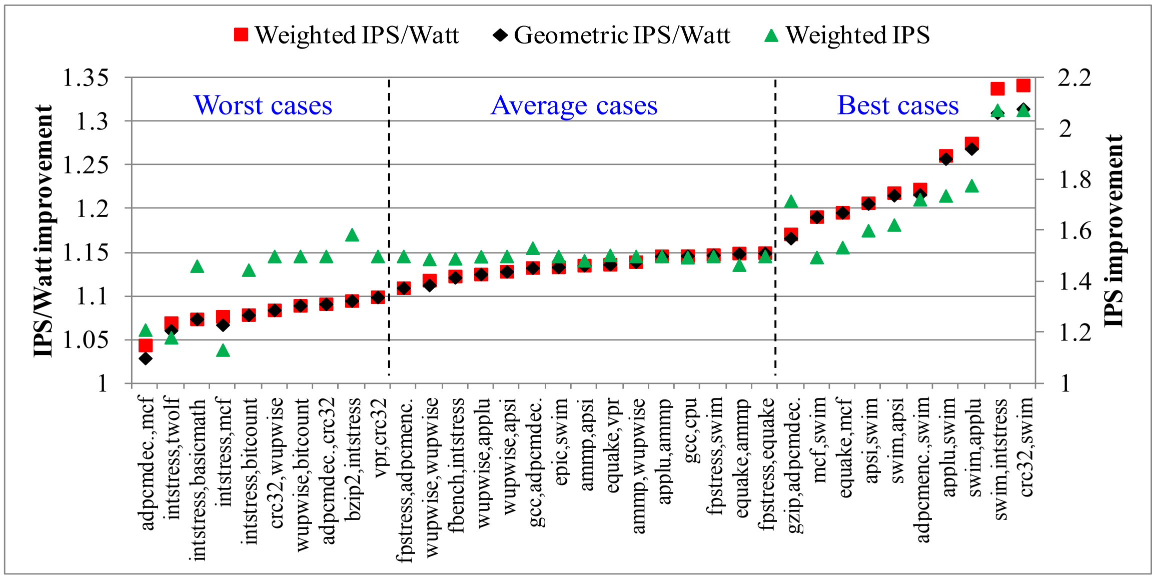


Figure 8.9. Throughput/Watt and throughput improvement using PDRF over the *swap-only* baseline for INT/FP dual-core AMP.

vs. ***Static*****:** This is the baseline heterogeneous AMP with a *static* thread-to-core assignment, i.e., it never changes during the program execution. The baseline lacks the capability to adapt to the time-varying behavior of the workload. Though a thread may have affinity for a certain core or an operating mode over the entire run, there may be periods where this thread would be more affine to another core or a power state in the AMP. By taking advantage of program phases and adapting to the thread needs, the proposed scheme achieves significant throughput and throughput/Watt benefits over this baseline (see Figure 8.8). Even for the worst case, PDRF was better than the *static* baseline by 5.5% when considering weighted throughput/Watt improvement. On an average, considering all the 100 combinations, PDRF achieved a 24.3% (21%) weighted (geometric) improvement in throughput/Watt over this baseline. Furthermore, by opportunistically opting for the boost mode for high-compute intensive phases, PDRF resulted in much higher throughput improvement of about 79%, on an average, over the *static* baseline. These results demonstrate the need for a dynamic scheme that can adapt the available core resources and operating modes to the program phase behavior.

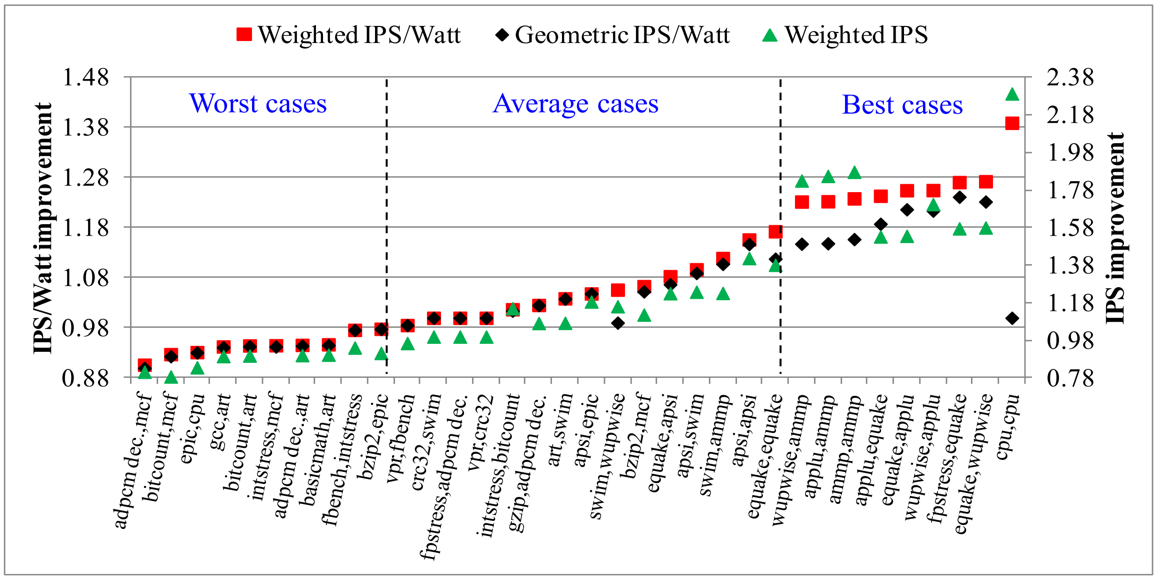


Figure 8.10. Throughput/Watt and throughput improvement using PDRF over the *DVFA-only* baseline for INT/FP dual-core AMP.

vs. *Swap-only*: This is a dynamic baseline that can swap threads between cores at runtime. As could be seen from Figure 8.9, a substantial increase in throughput/Watt is achieved using PDRF even over the *swap-only* baseline. Again, we did not encounter any combination where the *swap-only* scheme performed better than PDRF. As expected, PDRF performed much better than the *swap-only* scheme for cases when DVFA would come in handy and when opportunities to swap threads are limited. Both these cases occur for workloads that are primarily integer (INT) or floating-point (FP) intensive and do not exhibit many phases. For such cases, once the affine core is chosen, the execution can be significantly speeded up by pushing the corresponding core to the boost mode. In line with our expectation, we observe many uni-flavored workloads (e.g., *intStress, adpcm* are INT intensive while *fpStress, equake* are FP intensive) among the best performing cases. On an average, for the 100 combinations, PDRF achieved a weighted (geometric) throughput/Watt improvement of about 15.7% (14.9%) and a weighted throughput improvement of about 53.9% over the *swap-only* baseline.

vs. *DVFA-only*: This baseline has the capability to dynamically boost the voltage/frequency levels of the cores. In contrast to the previous two baselines, there were few benchmark combinations (e.g., {*adpcm dec.,mcf*}, {*bitcount,mcf*}) in Figure 8.10 for which PDRF performed worse than *DVFA-only* scheme. In the worst case, the IPS/Watt degradation is about 10%. This is because for few rare cases, PDRF ended up making wrong thread scheduling decision due to error in throughput/Watt prediction at the time of decision making. As a result of this, PDRF performed few non-beneficial thread swaps and opted for boosting the voltage and frequency of the cores at a much later stage of the program execution. Since PDRF is an opportunistic scheme that looks for thread scheduling opportunity only upon a phase change, a wrong thread scheduling decision made, is retained for the entire phase, magnifying its impact. The PDRF achieves significant benefits over the *DVFA-only* baseline when workloads with distinct INT/FP phases (e.g., *wupwise, ammp*) or symmetric workload combinations (e.g., {*equake, equake*}, {*cpu, cpu*}) are encountered. An average weighted (geometric) throughput/Watt improvement of about 7.5% (5.4%) and weighted throughput improvement of about 20% was achieved by the proposed PDRF over this baseline.

CHAPTER 9

EVALUATING PDRF FOR A LOW-POWER/HIGH-PERFORMANCE DUAL-CORE

Most of the current asymmetric multicore research has focused on designs with small and big cores [18, 31, 44]. The reconfiguration frameworks, RDRF and PDRF, presented in the earlier chapters were evaluated using the custom baseline cores (INT and FP). To explore the potential of the proposed approach further, we employ PDRF for a more commonly studied dual-core AMP consisting of low-power (LP) and high-performance (HP) cores in this chapter.

9.1 LP and HP core parameters

The considered LP and HP cores are at the two ends of the power/performance spectrum. This is one of the worst cases for a scheme for predicting the throughput/Watt on the HP core based on the activities observed in the LP core and vice versa. The parameters used for both the cores is shown in Table 9.1. Most of these parameters and the execution latencies were taken from [14]. It can be seen from Table 9.1 that the two cores are significantly different. The HP core is a 4-way issue, out-of-order (OOO) core with large core resources (e.g., integer (INT)/ floating-point

Table 9.1. Chosen core parameters for LP and HP cores

Param	LP	HP	Param	LP	HP
Issue	2	4	LS units	1	2
INTREG	64	96	LSQ	NA	32
FPREG	64	80	ROB	NA	128
INTISQ	NA	36	L1(I/D)	32K	32K
FPISQ	NA	24	L2	512K	2M
Type	In-order	OOO			

(FP) registers, issue queues, L2 cache) while the LP core is a 2-way issue, in-order core with minimal resources to cater to low power applications. Similar to the INT and FP cores described in Chapter 8, the LP and HP cores can also operate either in normal or boost mode and, the corresponding voltage and frequency levels in the two modes are shown in Table 9.2.

Table 9.2. Voltage/Frequency levels of LP and HP cores.

Core-type	Normal	Boost
LP	0.81 V / 1 GHz	0.9 V / 1.6 GHz
HP	1.1 V / 2 GHz	1.3 V / 3 GHz

9.2 Counter selection for IPC/Watt estimation

Due to the significant difference in the microarchitecture of the baseline cores, the counters used for IPC/Watt estimation on INT/FP cores may not work for LP/HP cores. Hence, the expressions for IPC/Watt estimation pertaining to all the 8 cases (4 for the same core and 4 more for the other core) were re-trained for the LP/HP dual-core AMP. A subset of the 8 expressions that correspond to IPC/Watt prediction in the normal mode is shown in Table 9.3.

Table 9.3. IPC/Watt expressions trained for the normal mode.

HPCs of/Prediction on	Expression
LP/HP	$-1.2 \times \text{BMP} - 0.1 \times \text{L1m} + 0.04 \times \text{Br} + 3.6 \times 10^{-4} \times \text{S} + 0.05$
HP/LP	$-0.28 \times \text{L1m} - 0.04 \times \text{Ld} + -0.5 \times \text{BMP} + 0.1 \times \text{TLBm} + 0.08$
LP/LP	$0.2 \times \text{IPC} - 8 \times 10^{-4} \times \text{S} + 0.04$
HP/HP	$0.02 \times \text{IPC} - 0.01 \times \text{L1m} + 0.04$

9.3 Evaluating the accuracy of IPC/Watt prediction

As it is evident from Figure 9.1, we achieved a reasonably high prediction accuracy in estimating the IPC/Watt at both the operating modes and on the two core-types. The maximum average error was about 16.4% when predicting the IPC/Watt on the LP core in normal mode using the HPCs of the HP core. Figure 9.2 shows the error distribution of the worst case, predicting the IPC/Watt on the other core in normal

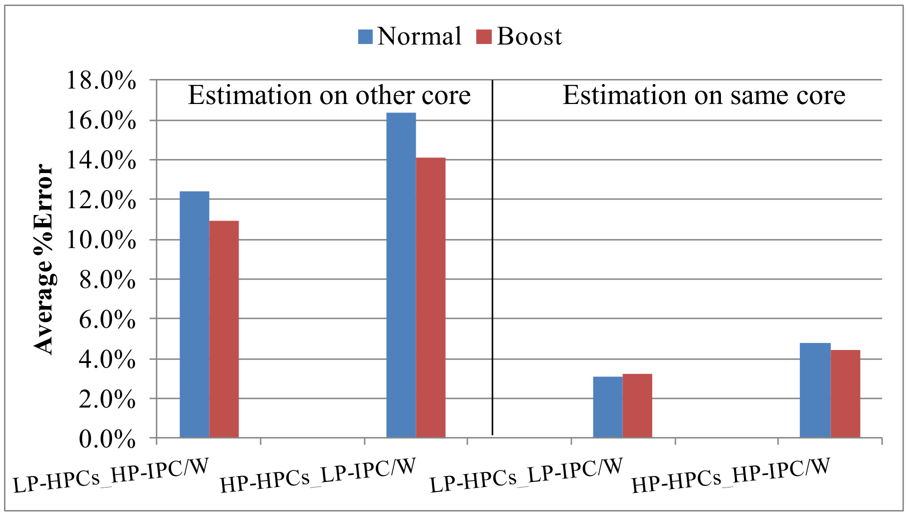


Figure 9.1. Average percentage error in IPC/Watt (IPC/W) estimation.

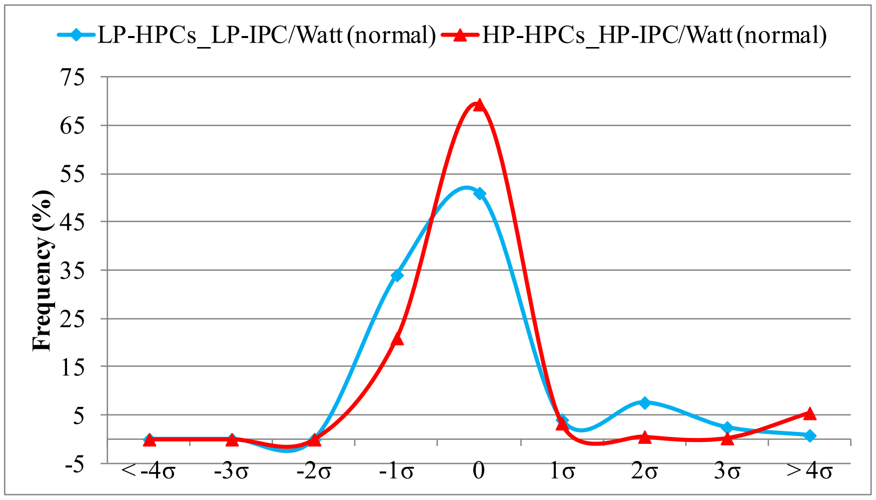


Figure 9.2. Distribution of error in estimating IPC/Watt on the other core using HPCs of the host core.

mode using the HPCs of the host core. The high accuracy of the prediction is reflected even in this figure as majority (about 90%) of the sample points are contained within +/- 1σ. This analysis clearly illustrates the capability of the described prediction mechanism to work for different architectures.

9.4 Evaluation

Having discussed the accuracy of IPC/Watt prediction for the LP and HP cores, we evaluate the potential benefits of PDRF for the considered baseline cores (LP and

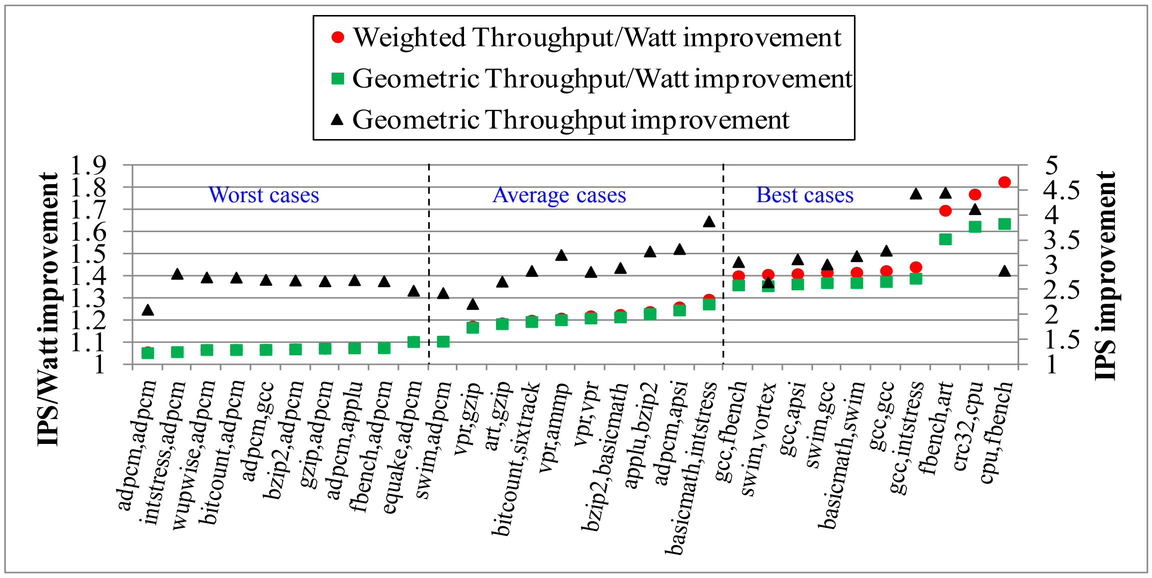


Figure 9.3. Throughput/Watt and throughput improvement using PDRF over the static baseline for LP/HP dual-core AMP.

HP). The same baselines discussed in Section 8.7 were used and the throughput/Watt and throughput achieved using PDRF and the baselines were compared for a large number (about 120) of multiprogrammed workloads. An in-depth analysis of the comparison results is presented next.

vs. ***Static*****:** By taking advantage of the program phases, the PDRF achieved significant throughput and throughput/Watt benefits over the *static* baseline (see Figure 9.3). Of the 120 combinations, we did not find any case where this baseline performed better than PDRF. On an average, considering all the 120 combinations, PDRF achieved a 27.2% (25%) weighted (geometric) improvement in throughput/Watt over this baseline. Furthermore, by opportunistically opting for the boost mode and efficiently making use of the HP core for high-compute intensive/high-ILP program phases, PDRF resulted in much higher throughput improvement of about 190%, on an average, over the *static* baseline.

vs. ***Swap-only*****:** PDRF achieved a throughput/Watt improvement of about 5.3% over the *swap-only* baseline even for the worst case (see Figure 9.4). Using the proposed scheme, there was, on average, a 13.7% (13%) weighted (geometric) improve-

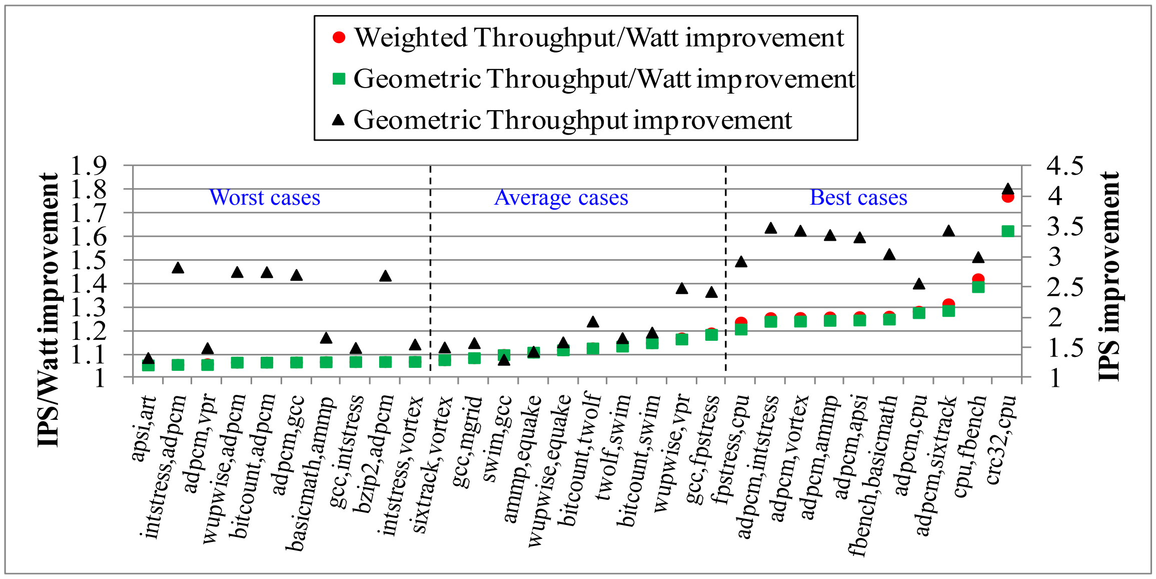


Figure 9.4. Throughput/Watt and throughput improvement using PDRF over the *swap-only* baseline for LP/HP dual-core AMP.

ment in IPS/Watt and about 91% improvement in IPS over the *swap-only* baseline. The drop in the achieved throughput/Watt gain relative to the *static* baseline reflects the adaptable nature of the *swap-only* baseline, where at least the appropriate core-type is chosen to suit the current execution phase of the threads.

We analyzed the benchmark combinations at the right end of Figure 9.4 for which we achieve maximum IPS/Watt improvement over the *swap-only* baseline. It is interesting to note that most of them are either compute-memory intensive benchmark combinations (e.g., {*fbench,basicmath*}) or both are compute intensive benchmark combinations (e.g., {*adpcm,cpu*}). In the case of {*fbench,basicmath*} combination, the benchmark *fbench* is memory intensive with about 58% load/store instructions while the benchmark *basicmath* is compute intensive. For such compute-memory intensive benchmark combinations, besides deciding the best thread-to-core assignments, our scheme makes use of DVFA to good extent. During high-IPC/high-ILP phases of compute intensive benchmark, our scheme pushes the HP core to boost mode resulting in much faster execution and hence, better IPS/Watt. This is supported by much higher IPS speedup for these combinations over the *swap-only* baseline (speedup of

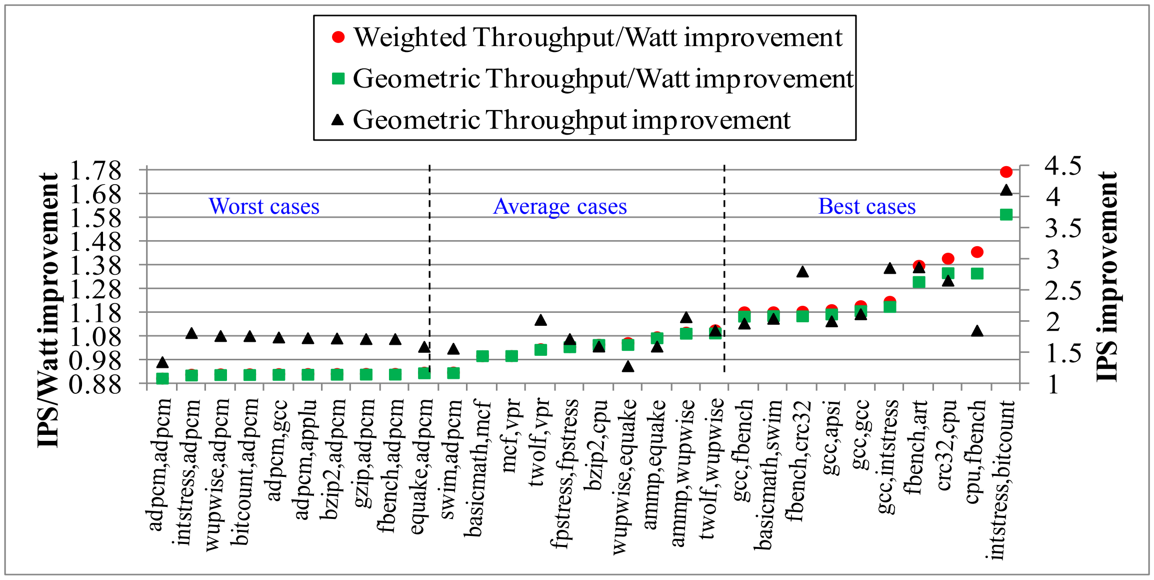


Figure 9.5. Throughput/Watt and throughput improvement using PDRF over the *DVFA-only* baseline for LP/HP dual-core AMP.

about 3 for {*fbench,basicmath*}). For cases when both the threads go through high compute intensive phases at about the same time (e.g., {*adpcm,cpu*}), our scheme pushes the HP core to boost mode, clearing the conflict for better resources (HP core) quickly. During this time, the performance of the thread executing on non-affine core is improved by opting for the boost mode within the LP core. Once the conflict clears up, the latter thread is migrated to HP core. These cases clearly substantiate the need for dynamically changing the voltage/frequency of the cores besides swapping threads.

vs. *DVFA-only*: The voltage and frequency of the cores are chosen so as to maximize throughput/Watt in *DVFA-only* baseline. In contrast to the previous two baselines, there were few benchmark combinations (e.g., {*adpcm,adpcm*}, {*bitcount,adpcm*} in Figure 9.5) out of the 120 for which our scheme performed worse than *DVFA-only* scheme. We have already observed a probable reason for this in Section 8.7. Nevertheless, these worst case scenarios were infrequent (only 10 out of 120 combinations resulted in degradation >3%) and even in the worst case, the observed IPS/Watt degradation was only about 9.5%. On an average, considering all the 120 combinations, the proposed PDRF achieved a throughput/Watt improvement of about 8% over the DVFA-only baseline.

We also analyzed the cases for which our scheme performs much better than the *DVFA-only* scheme. We observed that most of such cases were for symmetric workload combinations (both the threads having affinity for the same core-type, e.g., {*gcc,gcc*}, {*intStress,bitcount*} - both are integer intensive). By swapping threads, our scheme efficiently shares the affine resource (preferred core-type) while one of the threads is forced to execute on the non-affine core throughout its execution in *DVFA-only* scheme. Hence, there is a definite need for a scheme to support thread swapping besides DVFA.

Furthermore, we analyzed the best 10 cases for which our scheme achieves maximum IPS/Watt speedup over *swap-only* (see Figure 9.4) and *DVFA-only* (see Figure 9.5) baselines. We noticed that there were only 2 benchmarks combinations ({*crc32,cpu*} and {*cpu,fbench*}) that were in common between the two. This is very encouraging for our proposed scheme as it clearly shows that the benefits of dynamic thread swapping and DVFA are mostly non-overlapping. As a result, different kinds of benchmark combinations could benefit from either of them, indicating the potential benefits of schemes (like the one proposed) that seamlessly combine the two approaches.

CHAPTER 10
CONCLUSIONS

We have presented a novel dynamic reconfiguration framework (DRF) for AMPs which strives to maximize performance/power of the applications. The proposed DRF is equipped with dynamic resource allocation (DRA), and voltage/frequency adaptation (DVFA) capabilities. Two approaches were explored for the proposed DRF: one (RDRF) works based on rules established offline and the other (PDRF) by predicting online the expected performance/power of the thread at different voltage/frequency levels on all the available core-types in the AMP. We have devised an unified trigger mechanism using hardware performance counters (HPCs) for both RDRF and PDRF.

To illustrate our approach, we considered a dual-core: one core with support for strong integer code execution and another core that could handle floating-point operations efficiently. Aligning with the time-dependent behavior of the applications and their computational demands, our proposed DRF dynamically swaps the executing threads or morphs the cores at runtime by realigning resources of the given baseline cores to form a strong and a weak core. In addition, appropriate voltage/frequency levels are chosen dynamically to maximize performance/power of the applications. We have demonstrated the potential of PDRF for varied baseline core architectures. Our results show that proposed DRF achieves significant throughput/Watt benefits over different baselines.

CHAPTER 11
FUTURE WORK

We discuss in this chapter the possible extensions to this thesis.

- *Adaptive fetch throttling:* Hardware counters were extensively used in this thesis to trigger reconfigurations and predict performance/power. One possible future work is to leverage them for an adaptive fetch throttling mechanism. When the difference between the *Fetched instructions* and the *Retired instructions* counters is large or when the value of *Branch misprediction* counter is high, then it is a clear indication that the processor is executing many speculative instructions. Execution of these instructions unnecessarily burns more power without contributing to the actual computation. Hence, under such scenarios it may be beneficial from a power perspective to dynamically reduce the fetch width.

- *Phase-based performance/power prediction:* Performance/power prediction was done using a single trained expression for all application phases in this thesis. Different performance/power expressions could be trained for different program phases which could then be used online. Employing such a phase-based performance/power models may improve the accuracy of the prediction even further.

- *Opportunistic execution outsourcing:* With very minimal modification to the hardware support for core morphing (see Figure 6.4), the proposed DRA mechanism could be extended to support execution outsourcing. When the processor is stalled due to the lack of execution resources, the subsequent instructions could make use of the (execution) resources of the other core if they are available. Such opportunistic execution outsourcing could be deployed for performance improvement or even for fault tolerance.

BIBLIOGRAPHY

[1] The Standard Performance Evaluation Corporation (Spec CPI2000 suite). http://www/specbench.org/osg/cpu2000.

[2] The Nehalem Preview: Intel Does It Again. http://www.anandtech.com/show/-2542/5.

[3] Third generation Intel Core Processors. http://www.intel.com/content/dam/-www/public/us/en/documents/datasheets/3rd-gen-core-family-mobile-vol-1-datasheet.pdf.

[4] Intel Corporation. Intel Turbo Boost Technology in Intel Core Microarchitecture (Nehalem) Based Processors, White Paper, (2008).

[5] Balakrishnan, S., Rajwar, R., Upton, M., and Lai, Konrad. The impact of performance asymmetry in emerging multicore architectures. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on* (2005), pp. 506–517.

[6] Becchi, M., and Crowley, P. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers* (2006), CF '06, ACM, pp. 29–40.

[7] Brooks, D., and Martonosi, M. Dynamic thermal management for high-performance microprocessors. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on* (2001), pp. 171–182.

[8] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (2000), pp. 83–94.

[9] Chen, J., and John, L.K. Efficient program scheduling for heterogeneous multi-core processors. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE* (2009), pp. 927–930.

[10] Chen, T., Hsu, C., and Wu, S. Flexible heterogeneous multicore architectures for versatile media processing via customized long instruction words. *Circuits and Systems for Video Technology, IEEE Transactions on 15*, 5 (2005), 659–672.

[11] Contreras, G., and Martonosi, M. Power prediction for Intel XScale reg; processors using performance monitoring unit events. In *Low Power Electronics and Design, 2005. ISLPED '05. Proceedings of the 2005 International Symposium on* (2005), pp. 221–226.

[12] Das, A., Rodrigues, R., Koren, I., and Kundu, S. A study on performance benefits of core morphing in an asymmetric multicore processor. In *Computer Design (ICCD), 2010 IEEE International Conference on* (2010), pp. 17–22.

[13] Dhodapkar, A.S., and Smith, J.E. Comparing program phase detection techniques. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (2003), pp. 217–227.

[14] Fog, A. The microarchitecture of Intel, AMD and VIA CPU. Tech. rep., Copenhagen University College of Engineering.

[15] Foley, D., Steinman, M., Branover, A., Smaus, G., Asaro, A., Punyamurtula, S., and Bajic, L. AMD's "LLANO" FUSION APU, Hot Chips (2011), Paper: HC23.19.930.

[16] Ghasemazar, M., Pakbaznia, E., and Pedram, M. Minimizing energy consumption of a chip multiprocessor through simultaneous core consolidation and dvfs. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on* (2010), pp. 49–52.

[17] Gibson, D., and Wood, D. A. Forwardflow: a scalable core for power-constrained CMPs. In *37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, pp. 14–25.

[18] Greenhalgh, P. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White Paper* (2011).

[19] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on* (2001), pp. 3–14.

[20] Held, J., Bautista, J., and Koehl, S. From a Few Cores to Many: A Tera-scale Computing Research Review, (2006).

[21] Heller, L. C., and Farrell, M.S. Millicode in an IBM zSeries processor. *IBM Journal of Research and Development 48*, 3.4 (2004), 425–434.

[22] Hill, M.D., and Marty, M.R. Amdahl's Law in the Multicore Era. *Computer 41*, 7 (2008), 33–38.

[23] Ipek, E., Kirman, M., Kirman, N., and Martinez, J. F. Core fusion: accommodating software diversity in chip multiprocessors. In *34th Annual International Symposium on Computer Architecture* (2007), ISCA '07, pp. 186–197.

[24] Keramidas, G., Spiliopoulos, V., and Kaxiras, S. Interval-based models for runtime DVFS orchestration in superscalar processors. In *7th ACM International Conference on Computing Frontiers* (2010), CF '10, pp. 287–296.

[25] Khan, O., and Kundu, S. A model to exploit power-performance efficiency in superscalar processors via structure resizing. In *20th Symposium on Great Lakes Symposium on VLSI* (2010), GLSVLSI '10, pp. 215–220.

[26] Khan, O., and Kundu, S. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI* (2010), ACM, pp. 397–400.

[27] Khan, O., and Kundu, S. Thread Relocation: A Runtime Architecture for Tolerating Hard Errors in Chip Multiprocessors. *Computers, IEEE Transactions on 59*, 5 (2010), 651–665.

[28] Khan, O., and Kundu, S. Microvisor: A Runtime Architecture for Thermal Management in Chip Multiprocessors. *T. HiPEAC 4* (2011), 84–110.

[29] Kim, C., Sethumadhavan, S., Gulati, D., Burger, D., Govindan, M.S., Ranganathan, N., and Keckler, S.W. Composable Lightweight Processors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007* (2007).

[30] Kim, W., Gupta, M.S., Wei, Gu-Yeon, and Brooks, D. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008* (2008).

[31] Koufaty, D., Reddy, D., and Hahn, S. Bias scheduling in heterogeneous multicore architectures. In *Proceedings of the 5th European conference on Computer systems* (2010), EuroSys '10, ACM, pp. 125–138.

[32] Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., and Tullsen, D.M. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on* (2003), pp. 81–92.

[33] Kumar, R., Tullsen, D. M., and Jouppi, N. P. Core architecture optimization for heterogeneous chip multiprocessors. In *15th International Conference on Parallel Architectures and Compilation Techniques* (2006), PACT 2006, pp. 23–32.

[34] Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., and Farkas, K.I. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on* (2004), pp. 64–75.

[35] Lee, C., Potkonjak, M., and Mangione-Smith, W.H. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on* (1997), pp. 330–335.

[36] Li, Y., Skadron, K., Brooks, D., and Hu, Zhigang. Performance, energy, and thermal considerations for SMT and CMP architectures. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on* (2005), pp. 71–82.

[37] Morad, T., Weiser, U., and Kolodny, A. ACCMP - Assymetric Cluster Chip Multi-Processing. In CCIT Technical Report 488, (2004).

[38] Najaf-abadi, H.H., Choudhary, N.K., and Rotenberg, E. Core-Selectability in Chip Multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on* (2009), pp. 113–122.

[39] Park, J., Shin, D., Chang, N., and Pedram, M. Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors. In *Low-Power Electronics and Design (ISLPED), 2010 ACM/IEEE International Symposium on* (2010), pp. 419–424.

[40] Pericas, M., Cristal, A., Cazorla, F.J., Gonzalez, R., Jimenez, D.A., and Valero, M. A Flexible Heterogeneous Multi-Core Architecture. In *16th International Conference on Parallel Architectures and Compilation Techniques, 2007. PACT 2007* (2007).

[41] Renau, Jose. SESC Simulator, http://sesc.sourceforge.net., (2005).

[42] Rodrigues, R., Annamalai, A., Koren, I., and Kundu, S. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Trans. Des. Autom. Electron. Syst. 18*, 1 (Jan. 2013), 5:1–5:23.

[43] Rotem, E., Naveh, A., Rajwan, D., Ananthakrishnan, A., and Weissmann, E. Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge, Hot Chips (2011), Paper: HC23.19.920.

[44] Saez, J. C., Prieto, M., Fedorova, A., and Blagodurov, S. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems* (2010), EuroSys '10, ACM, pp. 139–152.

[45] Salverda, P., and Zilles, C. Fundamental performance constraints in horizontal fusion of in-order cores. In *IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008* (2008).

[46] Shelepov, D., Saez, J. C., Jeffery, S., Fedorova, A., Perez, N., Huang, Zhi Feng, Blagodurov, S., and Kumar, V. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev. 43*, 2 (Apr. 2009), 66–75.

[47] Sherwood, T., Sair, S., and Calder, B. Phase tracking and prediction. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on* (2003), pp. 336–347.

[48] Shivakumar, P., and Jouppi, N. P. Cacti 3.0: An Integrated Cache Timing, Power, and Area Model. Tech. rep., (2001).

[49] Singh, K., Bhadauria, M., and McKee, S. A. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News 37*, 2 (July 2009), 46–55.

[50] Srinivasan, S., Zhao, L., Illikkal, R., and Iyer, R. Efficient interaction between OS and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev. 45*, 1 (Feb. 2011), 62–72.

[51] Tarjan, D., Boyer, M., and Skadron, K. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE* (2008), pp. 772–775.

[52] Vasan, A. Performance and power evaluation by Resource Sizing in an Asymmetric Multicore system. Tech. rep., University of Massachusetts at Amherst.

[53] Winter, J. A., Albonesi, D. H., and Shoemaker, C. A. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *19th International Conference on Parallel Architectures and Compilation Techniques* (2010), PACT 2010, pp. 29–40.

[54] Zhang, X., Shen, K., Dwarkadas, S., and Zhong, R. An evaluation of per-chip nonuniform frequency scaling on multicores. In *2010 USENIX Conference on USENIX Annual Technical Conference* (2010), USENIXATC'10, pp. 19–19.