A

Improving Performance per Watt of Asymmetric Multicore Processors via Online Program Phase Classification and Adaptive Core Morphing

RANCE RODRIGUES, University of Massachusetts at Amherst
ARUNACHALAM ANNAMALAI, University of Massachusetts at Amherst
ISRAEL KOREN, University of Massachusetts at Amherst
SANDIP KUNDU, University of Massachusetts at Amherst

Asymmetric multicore processors (AMPs) have been shown to outperform symmetric ones in terms of performance and performance/watt. The improved performance and power efficiency are achieved when the program threads are matched to their most suitable cores. Since the computational needs of a program may change during its execution, the best thread to core assignment will likely change with time. We have therefore, developed an online program phase classification scheme that allows swapping of threads when the current needs of the threads justify a change in the assignment.

The architectural differences among the cores in an AMP can never match the diversity that exists among different programs and even between different phases of the same program. Consider, for example, a program (or a program phase) that has a high instruction level parallelism (ILP) and will exhibit high power efficiency if executed on a powerful core. We can not however, include in the designed AMP, such powerful cores since they will remain underutilized most of the time, and they are not power efficient when the programs do not exhibit high degree of ILP. Thus, we must expect to see program phases where the designed cores will be unable to support the ILP that the program can exhibit. We therefore, propose in this paper a dynamic morphing scheme. This scheme will allow a core to gain control of a functional unit that is ordinarily under the control of a neighboring core, during periods of intense computation with high ILP. This way, we dynamically adjust the hardware resources to the current needs of the application.

Our results show that combining online phase classification and dynamic core morphing can significantly improve the performance/watt of most multi-threaded workloads.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Adaptable architectures, Heterogeneous (hybrid) systems

General Terms: Design, Algorithms, Performance, Experimentation

Additional Key Words and Phrases: Asymmetric multicores, Dynamic Core Morphing (DCM), Hardware assisted core reconfiguration and thread scheduling

1. INTRODUCTION

The semiconductor industry has been driven by Moore's law for almost half a century. Miniaturization of device size has allowed more transistors to be packed into a smaller area while the improved transistor performance has resulted in a significant increase in frequency. Increased density of devices and rising frequency led, unfortunately, to a power density problem. The processor industry responded to this problem by lower-

This work has been submitted for consideration under the "Adaptive Power Management for Energy and Temperature Aware Computing Systems" special issue. It has been supported in part by a grant from SRC (Grant no. 1985.001) and NSF (Grant no. 0903191).
Author's addresses: R. Rodrigues, A. Annamalai, I. Koren and S. Kundu. Department of Electrical and Computer Engineering, University of Massachusetts at Amherst.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ing processor frequency and integrating multiple processor cores on a die [Held et al. 2006]. Still, a multicore die is limited by an overall power dissipation envelope that stems from packaging and cooling constraints. Consequently, most current multicores are composed of cores with relatively moderate capabilities as integration of high performance cores will result in higher cost and possibly breaching of heat dissipation limits.

For the majority of current applications, the capability of cores found in today's multicore systems is adequate. However, multicore processors are focused more on supporting Thread Level Parallelism (TLP) and hence sacrifice instruction throughput for certain workloads [Pericas et al. 2007; Gibson and Wood 2010]. These workloads can benefit from more powerful cores to support higher instruction throughput and better performance per power. In order to achieve high performance/watt, applications should have (i) low execution time which implies high performance, and (ii) low power. When sequential applications are encountered, higher performance may be achieved by either designing more powerful individual cores or by morphing the resources of a few simpler cores when the need arises. Incorporating complex cores in a multicore system goes against the basic premise of multicores, i.e., lowering the power density. Furthermore, resource and power are frequently wasted whenever such workloads are not encountered. Hence, on-demand resource morphing may provide a better alternative.

In general, multicore processors may be symmetric (SMP) or asymmetric (AMP). It is well known that different workloads require different processor resources for better performance/watt. Even within a workload, the resource requirements may vary with time due to changes in program phases [Kumar et al. 2003; Sherwood et al. 2003]. Within a given resource budget, when computing demands are matched with processor capabilities, AMPs tend to perform better than SMPs [Kumar et al. 2006; Hill and Marty 2008; Winter et al. 2010]. However, matching resources to applications' needs has been recognized as a difficult problem [Balakrishnan et al. 2005]. Despite that, AMPs are gaining traction from smart phones [van Berkel 2009] to integrated graphics processors [Intel ; AMD] due to their power-performance benefits. In this paper, we propose to adaptively match the processor capability to the computing needs of the executing threads and as a result improve the efficiency of AMPs (in terms of performance/watt). This is achieved by either swapping threads between cores (of different capabilities) or by morphing core resources dynamically.

We have previously explored the benefits of dynamic core morphing in an AMP architecture [Rodrigues et al. 2011]. In that paper, we studied a dual-core AMP (see Figure 1), where each core was resourced moderately in all areas, while featuring extra strength in either integer or floating-point operations. When a thread demanded strength in more than one area, the cores were 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 with stronger resources of other core(s) (see Fig-

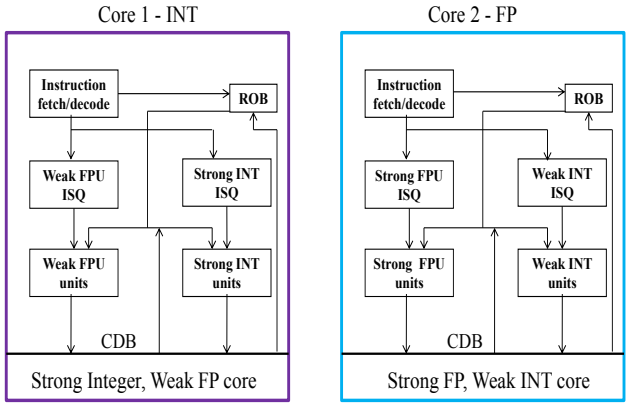


Fig. 1. Baseline configuration of the two heterogeneous cores [Rodrigues et al. 2011].

ure 2). Further details are provided in Section 3. Also, whenever deemed beneficial, threads were swapped between cores. The rules for triggering reconfiguration or swapping threads were determined by offline profiling [Rodrigues et al. 2011] and an online mechanism to determine current power consumption was assumed to be in place. From here on we refer to our previous core morphing scheme [Rodrigues et al. 2011] as Rule-based Dynamic Core Morphing (RDCM).

In this paper, we extend our prior work by adding a mechanism to trigger morphing or thread swapping online without offline learning. Online reconfiguration is made possible by detecting phases in the execution of programs and recording core performance and power consumption for each phase online. Whenever the same phase is encountered again, a simple table lookup enables determination of the best thread to core assignment with minimal overhead. As in our previous work, this functionality is provided by using hardware performance monitors. The online power estimation is also made possible through the use of performance event counters [Singh et al. 2009; Joseph and Martonosi 2001]. We call our proposed new online core morphing scheme Phase Classification based Dynamic Core Morphing (PCDCM).

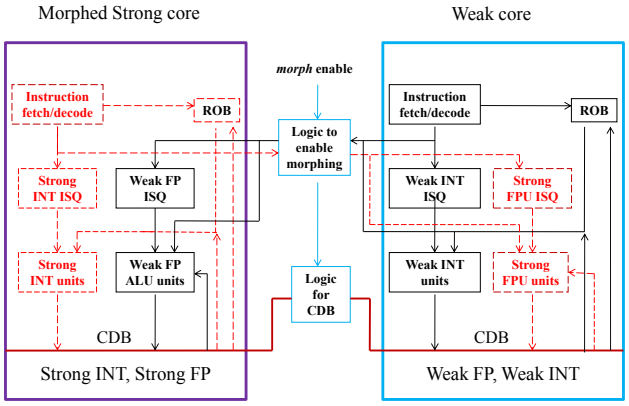


Fig. 2. Morphed configuration for the two heterogeneous cores. The red dotted lines/boxes indicate the connectivity for the strong morphed core configuration and the black solid lines/boxes indicate connectivity for the weak core [Rodrigues et al. 2011].

2. RELATED WORK

Reconfigurable multicores have recently received considerable attention. Core fusion was presented in [Ipek et al. 2007] where the cores of an SMP were reconfigured at runtime into stronger cores by "fusing" resources from the available cores. Another approach to fusion of homogeneous cores is presented in [Kim et al. 2007], where 32 dual-issue cores could be fused into a single 64-issue processor. Both schemes exhibit a high inter-core communication overhead. In addition, the reconfiguration overhead of critical units like the Reorder Buffer (ROB), issue queue and load/store queue has adversely affected the potential benefits. The difficulty in achieving good performance by fusing simple in-order cores into out-of-order (OOO) cores has been discussed in [Salverda and Zilles 2008]. It was also noted that aggregating cores in an SMP offers more of the same resources making it difficult to take full advantage of the high Instruction Level Parallelism (ILP) that some programs offer [Ipek et al. 2007; Kim et al. 2007].

There is significant variation in computational demand across applications. For example, in the SPEC benchmark [SPEC2000], *equake* is floating-point intensive while *gcc* is both integer and load-store intensive. Thus, to achieve acceptable performance for both workloads, the homogeneous cores would have to be designed such that they have a reasonably strong floating-point unit (FP), integer unit (INT) and load-store queues (LSQ). When only a strong FP or strong INT performance is needed, resources are idled. AMPs are better suited to handle such scenarios. Aggregating cores in an AMP provides opportunity to draw from the strengths of each core, creating potential

for achieving higher ILP than possible from aggregating SMPs. This motivates our current work.

Kumar et al. in [Kumar et al. 2003] proposed an AMP consisting of cores of various sizes, all running the same ISA. Whenever a new program is run or a new phase [Sherwood et al. 2003] is detected, a sampling is initiated and the core which provides the best power efficiency is chosen. This work considered four cores but only a single thread was considered running which greatly simplifies the AMP scheduling problem. The authors later extended this work for performance maximization of multithreaded applications [Kumar et al. 2004]. A similar approach was proposed by Becchi et al. [Becchi and Crowley 2006] in which the AMP consisted of two types of cores, one small and one big. The thread to core assignment was determined by forcing a swap between the big and small cores to find the performance ratio of the big to small cores. Depending on the ratio, the threads were scheduled such that the overall system performance is maximized. Kumar et al. [Kumar et al. 2006] address the design of an AMP, targeting area and power efficiency. They use cores that match the resource requirements of certain types of workloads. Annavaram et al. [Annavaram et al. 2005] use an AMP such that the Energy Per Instruction (EPI), while running multithreaded applications, is kept within a budget by having a big core execute the serial portions and the small cores execute the parallel portion of the code. [Najaf-abadi et al. 2009] propose core selectability where each "node" in the system consists of different types of cores that share common resources. There has been a number of schemes proposed for dynamic reconfiguration and thread migration in an AMP. Gibson et al. [Gibson and Wood 2010] propose a forward flow architecture where the execution logic can be scaled to meet the requirements of the incoming workloads. Li et al. [Li et al. 2007] schedule threads in an AMP such that the thread-queue length of the cores is kept proportional to the computing power of the cores. Chen et al. [Chen and John 2009] propose a thread scheduling solution for AMPs by matching the resource requirements of the threads to the available cores. They used cores that differ with respect to issue width, branch predictor size and L1 caches. Using the proposed algorithm, they achieved Energy Delay Product (EDP), energy and throughput improvements. Winter et al. [Winter et al. 2010] discuss power management techniques in AMPs via thread scheduling. They consider various algorithms and all of them require sampling on the core types to determine the best thread to core mapping.

Shelepov et al. [Shelepov et al. 2009] avoid sampling to determine thread to core mapping in an AMP by introducing what they call architectural signature. This signature is characterized by cache misses for the various core configurations and using this, they schedule threads such that the overall runtime is reduced for the multithreaded application. These signatures are determined offline via profiling and are fixed for the lifetime of the program. Hence, this solution is unable to take advantage of program phases. Khan et al. [Khan and Kundu 2010] propose regression analysis along with phase classification to find thread to core affinity. Luo et al. [Luo et al. 2010] propose the use of speculative threads in a same-ISA AMP to achieve performance improvement with moderate energy increase. Saez et al. [Saez et al. 2010] propose a comprehensive schedule for AMPs consisting of small and big cores, that considers both single threaded as well as multithreaded performance. They define a term called the utility factor (UF) which is the ratio of the time it takes to complete the task on the base (small cores) to that on the alternate configuration. This term is measured online using the last level cache miss rate to determine the thread to core assignment online. Koufaty et al. [Koufaty et al.] determine thread to core mapping in an AMP consisting of big and small cores using program to core bias. This bias is estimated online using the number of external stalls (proportional to cache requests going to L2 and main memory) and internal stalls (front end not delivering instructions to the back end).

Table I. Core configurations after the sizing experiments

Param	FP	INT	HMG	Weak
DL1	4K	4K	4K	1K
IL1	4K	4K	4K	1K
L2	128K	128K	128K	64K
LSQ	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 II. Execution unit specifications for the cores (P - Pipelined, NP - Not pipelined)

Core	FP DIV	FP MUL	FP ALU
FP	1 unit, 12 cyc, P	1 unit, 4 cyc, P	2 units, 4 cyc, P
INT	1 unit, 120 cyc, NP	1 unit, 30 cyc, NP	1 unit, 10 cyc, NP
HMG	1 unit, 66 cyc, NP	1 unit, 17 cyc, P	2 units, 7 cyc, P
	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, 12 cyc, P	1 unit, 3 cyc, P	2 units, 1 cyc, P
HMG	1 unit, 66 cyc, NP	1 unit, 16 cyc, P	2 units, 1 cyc, P

Using the bias they accordingly schedule threads in the AMP such that performance is maximized. Srinivasan et al. [Srinivasan et al. 2011] propose thread to core scheduling in an AMP by estimating the performance of the thread currently running on one core type, on another, using a closed form expression. The proposed scheme comes to within 3% of the oracular model used by the authors.

3. PROPOSED ARCHITECTURE

In this section, we describe in detail the Dynamic Core Morphing (DCM) scheme. To illustrate our approach, we consider a multicore processor constructed from tiles of asymmetric processors, where each tile consist of two cores: a FP core and an INT core (see Figure 1). The FP core features strong floating-point execution units but low performance integer execution units, while the INT core features exactly the opposite. Other differences between the cores include the number of virtual rename registers, issue queue (ISQ) and Load/Store Queue (LSQ) entries. We ran core sizing experiments (please refer to our earlier work [Rodrigues et al. 2011] for details) to determine these parameters and the final sizes of the various core parameters are shown in Table I. The details of the execution units for the two cores are shown in Table II. The HMG core in Table I represents an area equivalent homogeneous core used for comparison purposes where the area of the two HMG cores is approximately the sum of the areas of the INT and FP cores. The parameters for this core were obtained by averaging the size of the structures in the FP and INT cores.

In the baseline configuration (Figure 1) the cores operate independently providing good performance for affine workloads. Whenever a higher sequential performance is needed, a dynamic morphing of the cores takes place. In this configuration, the INT core takes control of the strong floating-point unit of the FP core to form a strong "morphed core" while relinquishing control of its own weak floating-point unit to the FP core. The FP core thus becomes a "weak core." Morphing results in two cores: (i) a strong single-threaded core capable of handling both integer and floating-point intensive applications efficiently, and (ii) a weak core which consumes less power and does not provide high performance. Instead of retaining the front end of the FP core as is, its resources are appropriately sized down (see the last column in Table I), to suit the application running on it and further reduce power. The result of the proposed dynamic morphing of the cores is shown in Figure 2. If the morphed mode is no longer beneficial, the system reconfigures itself back to the baseline mode.

The behavior of many applications tends to vary with time. Some may be floating-point intensive to start with and after a certain point may have higher percentage of integer instructions and vice-versa. Hence, the ability to swap threads between the two baseline cores could reduce the execution time significantly. Reduced execution time would improve the performance/watt with less idling and thus more efficient utilization of resources. Therefore, in addition to the baseline and morphed modes of opera-

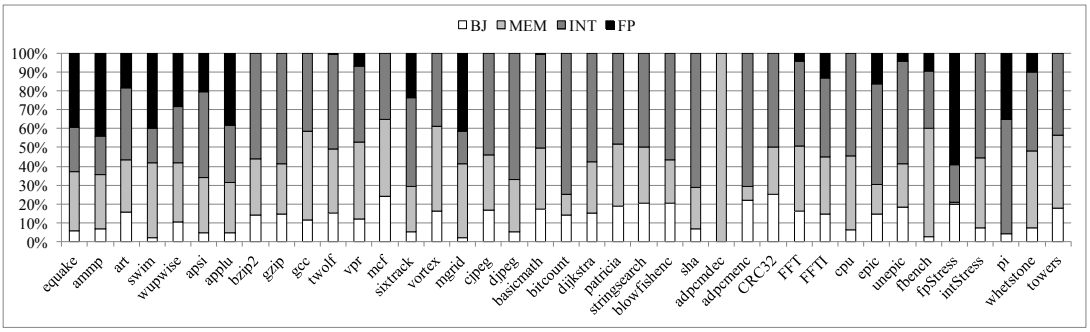


Fig. 3. Instruction distribution of the workloads when run for 5 billion instructions.

tion, we also allow the two tightly coupled heterogeneous cores to swap their execution contexts.

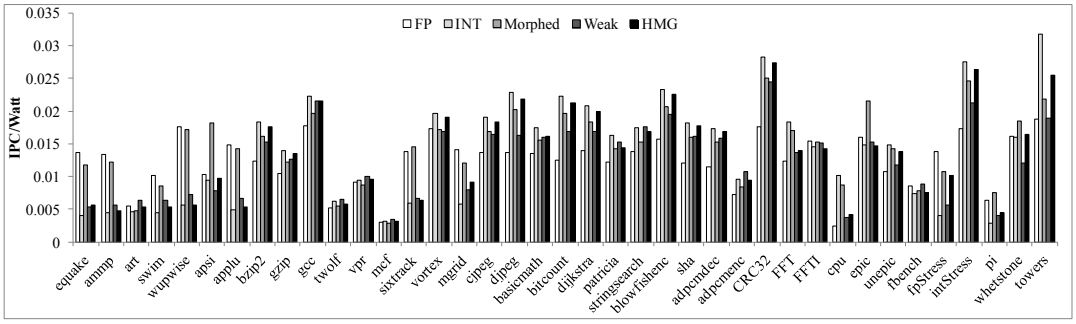
The proposed DCM scheme is a hardware-assisted solution that is autonomous and isolated from the Operating System (OS) level scheduler. We assume that only the initial scheduling is done by the OS in the baseline configuration. From then on, the thread to core assignment is managed autonomously by our scheme to optimize performance/watt at fine-grain time slices. The overhead of the required hardware support to enable swapping and core morphing at runtime is minimal, as they reuse the existing resources for task swaps and sleep states. The hardware overhead for the proposed architecture has been previously estimated by us in an earlier work [Das et al. 2010] to be approximately 1%.

4. PERFORMANCE/WATT AND PERFORMANCE EVALUATION

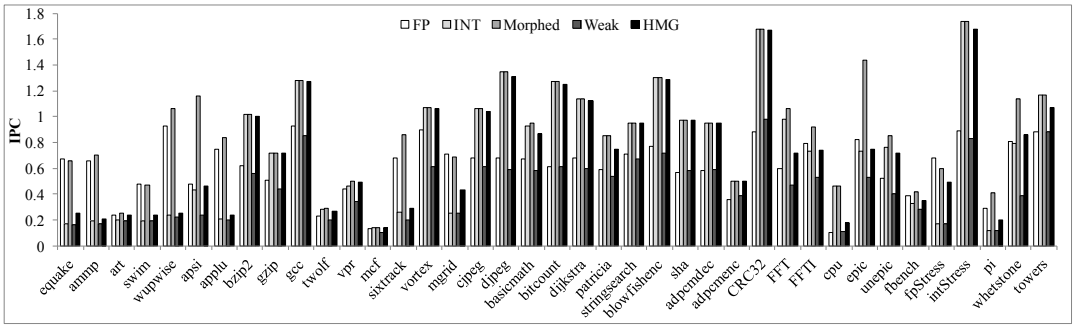
To evaluate our scheme, the SESC architectural simulator [Renau 2005] was used, and power was measured using Wattch [Brooks et al. 2000] and CACTI [Shivakumar et al. 2001] with modifications to account for *static power* dissipation. For the experiments, 38 benchmarks were selected: 16 benchmarks from the SPEC suite [SPEC2000], 14 embedded benchmarks from the MiBench suite [Guthaus et al. 2001], one benchmark from the mediabench suite [Lee et al. 1997], and 7 additional synthetic benchmarks. These 38 benchmarks encompass most typical workloads, for example, scientific applications, media encoding/decoding and security applications. In this section, the performance/watt and performance of each core is analyzed by running one application at a time on the various core types, i.e., FP, INT, Morphed, HMG and Weak cores.

The 38 benchmarks were run on each core configuration for 5 billion instructions (instruction distribution shown in Figure 3.) and the performance/watt results are plotted in Figure 4(a). We observe that 5 benchmarks (*apsi, sixtrack, epic, pi, whetstone*) in the morphed mode show notable gains. Out of these, *apsi* shows 82% improvement over its closest competitor, the FP core. This benefit is more modest for the benchmarks *epic* (35%), *whetstone* (17%), *pi* (12%) and *sixtrack* (5%). The reason why *apsi* shows substantial benefits is related to the temporal distribution of the instruction mix in *apsi*. We have observed that this happens due to the bursty nature of the instruction types encountered when executing *apsi*. For additional details, please refer to our earlier work [Das et al. 2010].

What is depicted in Figure 4(a) represents the average behavior over 5 billion instructions. However, many programs exhibit phases and each core configuration might be beneficial for different phases in the program execution. Hence, running the benchmark statically on the same core configuration may miss opportunities to maximize performance/watt. This is the reason why only 5 out of the 38 benchmarks show ben-



(a) IPC/Watt for all benchmarks



(b) IPC for all benchmarks

Fig. 4. IPC/Watt and IPC for the 38 benchmarks considered when run on each core configuration for 5 billion instructions.

efits when run on the morphed core throughout their execution. In the rest of the 33 cases, the power expended by the morphed core outweighs the obtained performance benefits resulting in poor performance/watt. This is evident from Figure 4(b) that shows the IPC for all benchmarks on the four types of cores. As can be seen from this figure, the morphed core performs either equally well or better than the other core configurations when only IPC is considered. Moreover, there is a bigger group of benchmarks (*ammp, wupwise, apsi, applu, sixtrack, FFT, FFTI, epic, unepic, fbench, pi, whetstone*) that show significant benefits from morphing and the gains are even higher ($>$150% for *apsi*). As we have seen, such performance gain does not always result in a higher power efficiency.

4.1. Impact of program phases

It was observed that over entire runs of 5 billion instructions, some benchmarks benefit, some don't, while some others even lose out upon morphing, when performance/watt is considered. Evaluation over entire runs does not take into account the changes in program behavior that is observed in most applications [Kumar et al. 2003; Sherwood et al. 2003]. In order to demonstrate the effect of program phases on performance/watt, a detailed study of the benchmark *epic* that shows benefit from morphing is presented next. The objective is to investigate the effect that the varying instruction distribution of a benchmark may have on performance/watt.

The benchmark *epic* was run for a few million instructions and the results are shown in Figure 5. The performance/watt for each core type (FP, INT and Morphed) is represented by the blue, orange and red curves, marked with a $\times$, a dot and a triangle, respectively. The distribution of instruction types at each time instant is represented by the area in the increasingly darker shades (light grey - INT, dark grey - FP, black

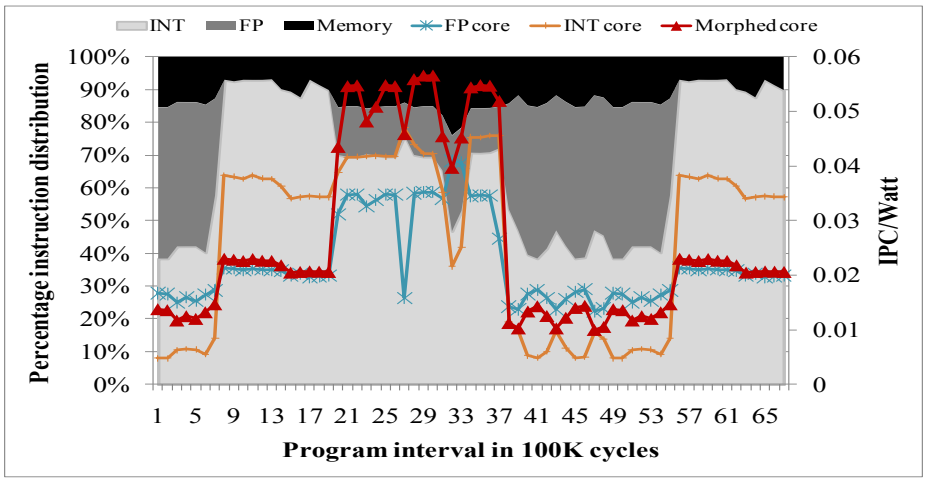


Fig. 5. Zoomed view of variations in the performance/watt of *epic* when run on each core configuration [Rodrigues et al. 2011].

- memory). For the first 19 data points, the morphed core does not outperform either the FP or the INT core and as a result, staying in the baseline mode of execution is advisable. For the data points 20 to 37, the morphed core performs 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 by morphing. During subsequent stages of execution, the baseline mode of execution again proves beneficial. This shows that by monitoring the program behavior at a more fine-grain level, there are more opportunities for improving the power efficiency by either morphing or exiting the morphed mode. 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 performance/watt for that thread.

A similar in-depth study was carried out for the benchmark *FFT* in our earlier work [Rodrigues et al. 2011]. It was noticed that even though *FFT* can benefit from running on the morphed core for the entire run, swapping the threads between the two cores, instead of morphing, may provide even better results. The study of the above two benchmarks infer that the decision to swap or morph should be based on the current behavior (e.g., instruction mix) of the executing workloads. In the next section, we describe in detail our dynamic decision making scheme.

5. DYNAMIC ONLINE RECONFIGURATION

We have seen that the performance/watt of an application running on one of the cores in the AMP is directly related to its instruction distribution. In this section, we describe the mechanism used to detect changes in an application behavior and accordingly trigger reconfiguration. We first give a brief overview of the Rule-based Dynamic Core Morphing (RDCM) scheme we proposed in [Rodrigues et al. 2011] and then introduce the new proposed Phase Classification based Dynamic Core Morphing (PCDCM) scheme.

5.1. Our previous Rule-based Dynamic Core Morphing scheme (RDCM)

In [Rodrigues et al. 2011], to detect the instantaneous affinity of an application to a core, we used an approach based on offline profiling. There, we selected a subset (12) of the workloads (training set) and ran profiling experiments on them. In these experiments, the instruction distribution as well as the performance/watt for each workload on each core type were observed, for a fixed window of committed instructions (500 instructions). The thread to core affinity as a function of instruction distribution was then established, using this information, in the form of rules (see Figure 6). These rules were then used to trigger morphing of cores or swapping of threads whenever application behavior changes are detected online. To evaluate the efficacy of these rules, experiments were run on the entire set of workloads (including those that were not part of the training set). The workloads were run for 40 million instructions and it was observed that the offline rule based method performed on average 16% better with respect to performance/watt over the baseline AMP with static thread to core mapping. Since this scheme used profiling, whenever applications that were not part of the training set were encountered, it is likely that not always were the optimal decisions made. Furthermore, we considered only a single AMP baseline for comparison. The thread to core assignment in that baseline stays the same throughout the lifetime of the program. There have been a few proposals [Becchi and Crowley 2006; Srinivasan et al. 2011] that allow swapping threads between cores. As a result, we decided to explore a new dynamic decision making mechanism, one that would be independent of training set workloads and would be compared to a more sophisticated baseline.

Algorithm for dynamic reconfiguration:

1. Threads T_1 and T_2 assigned randomly to cores
2. Do Swap if:
   i. ($\%INT_{FP}$ >= 44) and ($\%INT_{INT}$ <= 30)
      OR
   ii. ($\%FP_{INT}$ >= 26) and ($\%FP_{FP}$ <=13)
3. Go from baseline to morphed mode if:
   i. For T_1 (T_2)
      a. %(FP + INT) >= 50 and
      b. (17<=%FP<=30) and (26<=%INT<=44)
   ii. And T_2 (T_1)
      a. IPC <= 0.4 and
      b. %(FP + INT) < 60
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
5. End

- $\%INT_{FP}$ – 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

Fig. 6. The rules used for dynamic reconfiguration in the RDCM scheme

5.2. The proposed Phase Classification based Dynamic Core Morphing (PCDCM) scheme

The desired dynamic decision making mechanism should be able to determine online, the affinity of a program or program phase to a core without any prior knowledge of the workload. In addition, the scheme's overhead should not outweigh its benefits. We have seen above that the affinity of

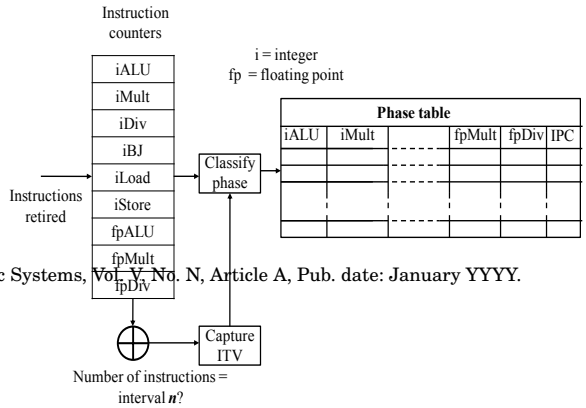
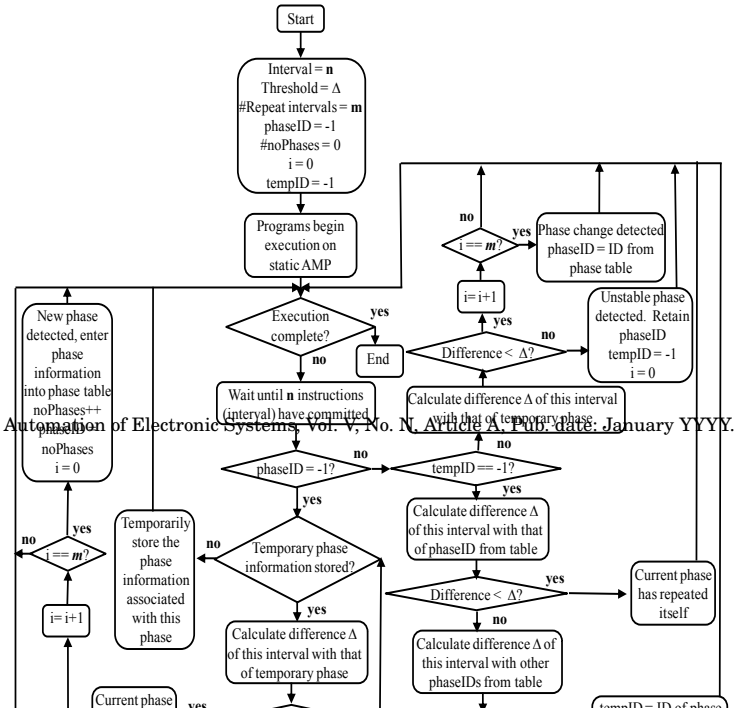


Fig. 7. Online recording of application behavior via hardware counters and phase table as done by Khan et al. in [Khan and Kundu 2011].

a program can be determined based on its instruction distribution. This distribution can be memorized in a history table and provide the basis for a phase classification mechanism as described next.

5.2.1. ITV based phase classification. Instruction type vectors (ITV) were introduced by Khan et al. in [Khan and Kundu 2011] for the purpose of program phase classification. We adopt this scheme and modify it to better suit our purposes. The ITVs are created using a circuit similar to that shown in Figure 7 where hardware counters are used to count the number of committed instructions of certain types (9 types as shown in the figure) during a specified interval. This interval corresponds to a fixed number n of committed instructions with the value of n to be determined. Whenever an instruction is retired, the appropriate instruction counter is incremented. After n instructions have committed the resulting 9-element vector is captured and compared to previously stored ITVs in the Phase Table. The already stored ITVs correspond to previously encountered stable phases where a phase is classified as stable when at least m consecutive intervals (of n committed instructions each) had almost identical ITVs. The number m is another parameter of the scheme that needs to be determined. The newly captured ITV is compared to each stored ITV by calculating the sum of the absolute differences between their corresponding nine elements. If this sum is smaller than a pre-specified threshold Δ (a parameter that needs to be determined), then the newly encountered phase is assigned the same phase ID as the one it was compared against. This signifies that we expect the current behavior of the program to be very similar to its behavior during the previously encountered phase with likely the same performance and performance per watt. If however, the sum exceeds the threshold value Δ, it becomes a potentially new phase but it needs to repeat m times before being assigned a new stable phase ID. Every program may exhibit during its execution several short-lived intermittent phases that do not justify actions like thread swapping or core morphing. It is important therefore, to distinguish between stable and unstable phases. The resulting algorithm to detect and classify phases is shown in Figure 8. Experiments were carried out to determine the phase classification parameters, i.e., n, m and Δ. Details on these experiments can be found in Appendix **??**. We found that in general, the phase classification mechanism provides best benefits when the interval range n is between 50K - 200K instructions, the %threshold is between 5 - 15% and the stable phase interval is between 2 - 8. For the rest of our experiments, these parameters have been set as: n = 150K, %threshold = 7.5 and the stable phase interval m=4.

5.2.2. Extending the phase table to include performance and power entries. We made a few changes to Khan's [Khan and Kundu 2011] phase classification scheme as shown in Figure 9. There are two major differences. (i) The number of instruction types in the ITV vector has been re-



duced from 9 to 4, and (ii) there are additional entries in the table to indicate the estimated IPC and power for each core type in the AMP. Since the cores in the AMP mainly differ with respect to their capability of processing integer and floating-point instructions, we aggregate all integer instructions into a single entry and all floating-point instructions into another single entry. We also aggregate load and store instructions into a single entry called Mem. As is shown in Appendix B, such a reduction does not compromise the benefits of the online mechanism. Further, to be able to use effectively the information about already classified stable phases, there is a need to collect per core type in the AMP, the performance and power for a given phase. We do that by augmenting the phase table with 2 additional entries per core type, one each for IPC and power. Since there are 4 core types in the considered AMP (FP, INT, Morphed and Weak), there are 8 entries corresponding to the estimated IPC and power for the given phase on each core type. Whenever a new stable phase is identified, our scheme will store in the phase table the approximate values of the IPC and the power consumed by each core during that phase.

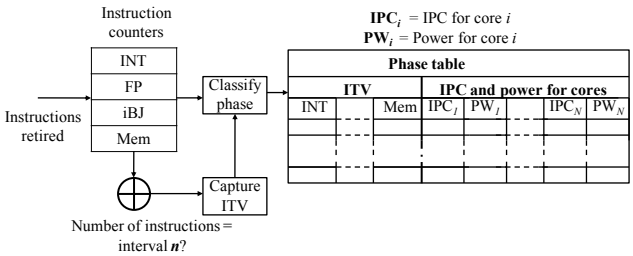


Fig. 9. Extending the phase table with IPC and Power entries for each core in the AMP. Note that the number of instruction types in the ITV vector has been reduced from 9 to 4.

Online measurement of IPC is straightforward, but the same cannot be said about power measurement. To estimate power, we use performance event counters, available in almost every processor, as a proxy for power. Computer architects have used performance monitoring counters as a proxy for estimating the power consumption, for example, IPC and cache misses [Joseph and Martonosi 2001; Contreras and Martonosi 2005; Singh et al. 2009]. The accuracy of such estimates is not high, but still sufficient for comparing the power consumed by different cores executing the same program. We adopted a similar approach to estimate power online using performance counters.

If the approximate IPC and power consumption is available for each phase of an application on each processor, a simple table lookup suffices to determine the best thread to core mapping for future occurrences of the classified phase. In this paper, we use a simple dynamic online learning approach and in the next sub-section we explain it and discuss its associated overhead.

5.2.3. Online performance and power estimation. The phase classification mechanism tracks the percentage of instructions in the most recently retired interval of n instructions. Whenever a new phase is detected, or when a previously classified phase is encountered again, it indicates that there is a change in the composition of instructions being executed for the application.We have seen previously (Figure 5) that such a change may indicate that the resource requirements of the program being executed have changed and it may be better to run the current phase of the program on another core in the AMP. In order to then determine the best thread to core assignment, performance estimation of that phase on each core type in the AMP is needed. As mentioned earlier, we achieve this by dynamic online learning where the newly detected phase is run on each core in the AMP. A similar scheme has been used by Kumar et al. [Kumar et al. 2003] and Becchi et al. [Becchi and Crowley 2006]. However, Kumar et al. sample the program on each core type in the AMP, each time a new phase is detected even if it has been previously encountered. Becchi et al. force a thread swap between cores whenever a new phase is detected to estimate performance of the phase on each core type. Sampling is clearly needed when new phases are detected, but not when a previously encountered phase is detected again, if the information related to the phase is available. Hence, during the proposed online learning process, the program is executed once on each core type and the observed IPC and power information are stored in the phase table. Since the AMP has 4 possible core types, this process must be repeated 4 times.

The overheads of the online scheme stem from the online learning mechanism and context switch on thread swap. We quantify the details of this overhead and its effect on the benefits of our scheme in Section 6.

5.2.4. Putting it all together. So far, individual components of the system (the online phase detection, performance and power estimation techniques) have been described. We now describe how each of these autonomous mechanisms work together (as shown in Figure 10). A software called the microvisor [Khan and Kundu 2011] is used to initialize and manage the phase classification mechanism as well as the performance and power estimation mechanisms. This software is invisible to the OS and is resident in between the OS and hardware. It collects information from the phase table and makes the best thread to core assignment. This functions the same way as that proposed by Khan et al. [Khan and Kundu 2011] or IBM's millicode [Heller and Farrell 2004].

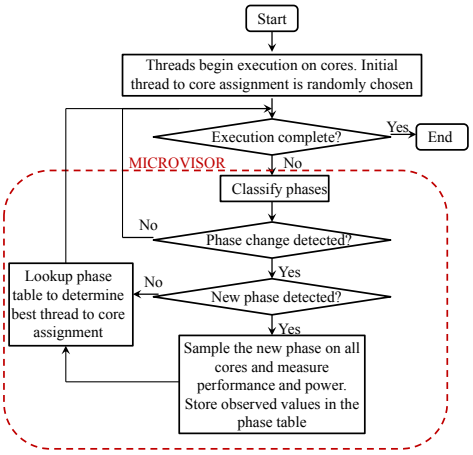


Fig. 10. Elements of the proposed PCDCM working together. The part of the algorithm controlled by the software layer (Microvisor) is indicated by the dotted red rectangle.

The microvisor. The microvisor monitors the operation of the AMP. It has control and access to the phase table of each core type. It also generates the triggers to morph/unmorph the cores. It is invoked whenever there is a new phase detected or a previously detected phase is encountered (phase change). If a new phase is detected, the microvisor controls the process of the sampling mechanism to estimate the IPC and it also collects the counter values that are used to estimate power. The phase table is then updated with the IPC and power information for each core type. If a phase change is detected, phase tables are looked up to fetch the IPC and power values for

that phase which are then used to make thread scheduling decisions. Since the microvisor does some computation whenever phases are detected or repeated, it incurs an overhead. We discuss this overhead in detail in Section 6.3.

Determining the best thread to core assignment. Whenever a phase change is detected or a new phase is classified, a different thread to core assignment may be needed to optimize performance per watt of the applications being executed. This thread to core assignment is determined by the microvisor that receives inputs from the various elements of the PCDCM scheme. The various thread to core assignments for the proposed AMP are: (i) $thread_0$ running on the FP core and $thread_1$ on the INT core ([FP, INT]) or vice versa ([INT, FP]) (ii) $thread_0$ running on the morphed core and $thread_1$ on the weak core ([MR, WK]) or vice versa ([WK, MR]). Whenever the thread to core assignment in the AMP needs to be reassessed (whenever there is a phase change or a new phase is detected in any of the two applications being executed on the AMP), the microvisor looks up the phase table for each application being run and estimates the gain in performance per watt that a change in thread to core assignment may have. To do this, either the weighted, geometric or harmonic speedup metrics (defined in Section 6.2) may be used to calculate the gain in performance per watt of the new potential thread to core assignment over the current one. We use the weighted metric in our experiments, but other metrics may be used. Whenever threads are swapped between cores, the phase tables are also moved accordingly.

6. EVALUATION

In this section, the proposed online PCDCM scheme is evaluated. In these experiments, multi-programmed workloads were run on the AMP. Execution stops when 5 billion instructions of either thread are retired. The phase classification parameters were set to: Interval n = 150K, %threshold Δ = 7.5 and stable phase interval m = 4 based on our search described in Appendix **??**.

We now describe the baselines that will be used for comparison. The performance/watt improvement achieved by PCDCM scheme over each baseline is then evaluated. Since the proposed scheme relies on dynamic online learning in order to determine the affinity of a newly detected phase to a core in the AMP, we present a study on the effect of this overhead on the benefits.

6.1. Baseline modes considered

We compare the proposed PCDCM scheme to the following baseline configurations:

(1) Static: This is the same baseline that we used in [Rodrigues et al. 2011]. Here the AMP does not feature morphing or swapping of threads, but the thread to core assignment is based on oracular knowledge of the best assignment.
(2) Swap: Here threads are allowed to swap between the cores. The decision to swap is made in the same way as the proposed dynamic online scheme using the ITV phase detection. The only difference is that the cores are not allowed to morph.
(3) HMG: This baseline consists of two homogeneous cores with parameters as described in Section 3. This dual-core processor is symmetric and occupies the same area as the FP and INT core AMP.
(4) RDCM: This is the offline profiling-based DCM scheme that we proposed in [Rodrigues et al. 2011]. As described earlier, core reconfiguration in this scheme relies on rules derived offline by profiling a subset of the workloads considered. These rules are then applied to trigger either morphing or thread swapping whenever deemed beneficial.

6.2. Performance per watt analysis over the baselines

We considered three speedup metrics for comparing the proposed scheme to the baselines. We define the following terms:

$$S_0 = (IPC/Watt_{thread0})_{dynamic}/(IPC/Watt_{thread0})_{baseline}$$
$$S_1 = (IPC/Watt_{thread1})_{dynamic}/(IPC/Watt_{thread1})_{baseline}$$

The various speedups considered are:

(1) Weighted:
$Speedup_{weighted} = (S_0 + S_1)/2$
(2) Geometric:
$Speedup_{geometric} = \sqrt[2]{S_0 \times S_1}$
(3) Harmonic:
$Speedup_{harmonic} = 2/(1/S_0 + 1/S_1)$

From the set of 38 workloads, we randomly selected 100 combinations of two threaded workloads and executed those on the proposed PCDCM scheme as well as on each of the baselines. We have plotted a subset (40 of the 100) of those results in Figures 11 and 12. We have sorted the 100 results from smallest to largest values of the weighted IPC/Watt improvements. Of the 40 shown in the plot, 10 were the worst cases, 10 best cases and 20 cases in between. It is clear that in general (see Figures 11 and 12), a significant IPC/Watt improvement is observed when compared to any baseline. Also, amongst the worst cases for the baselines (static and HMG), it can be see that the IPC/Watt degradation is not very high (0.86 in the worst case against HMG). Even when compared against the dynamic baselines (Swap and RDCM), it can be seen that significant IPC/Watt improvement is achieved on average.

6.2.1. Analysis of results. We now provide detailed analysis and reasons on why the PCDCM scheme performs better on average than both the static as well as the dynamic baselines.

Static. In this baseline, thread to core assignment is kept the same from start to end of the run. However, the assignment is assumed to be done by an oracle and as such, can not be done in practice. It can be seen that significant IPC/Watt improvement is achieved by PCDCM over this baseline. This scheme never takes advantage of phase changes or changes in resource demands. Even if over the entire run, a thread has an affinity for a certain core (as determined by the oracle), there may be periods where this thread would be more affine to another core. The PCDCM scheme is equipped with the phase classification mechanism and hence is able to react during such periods by rescheduling the threads. For example, over an entire run of 5 billion instructions, the workload *equake* shows an affinity to the FP core (see Figure 3). However, during the experimental run, 11 phases were detected for *equake* and affinity for the INT, Morphed or even the Weak core was observed during those phases. The PCDCM scheme detects these phases, re-evaluates the thread to core mapping and hence optimizes IPC/Watt. Hence, the PCDCM scheme achieves significant improvement in IPC/Watt over the static baseline. Still, there are a few workload combinations (3 out of of 100) where the PCDCM scheme performs slightly worse than this baseline (see Figure 11(a)). For these workloads, even though phases are detected and classified, at no point did PCDCM trigger a reconfiguration, but phases were detected and the sampling overhead increased the runtime. As a result, the IPC/Watt improvement is less than 1. However, on an average, for all the 100 combinations (see Figure 13

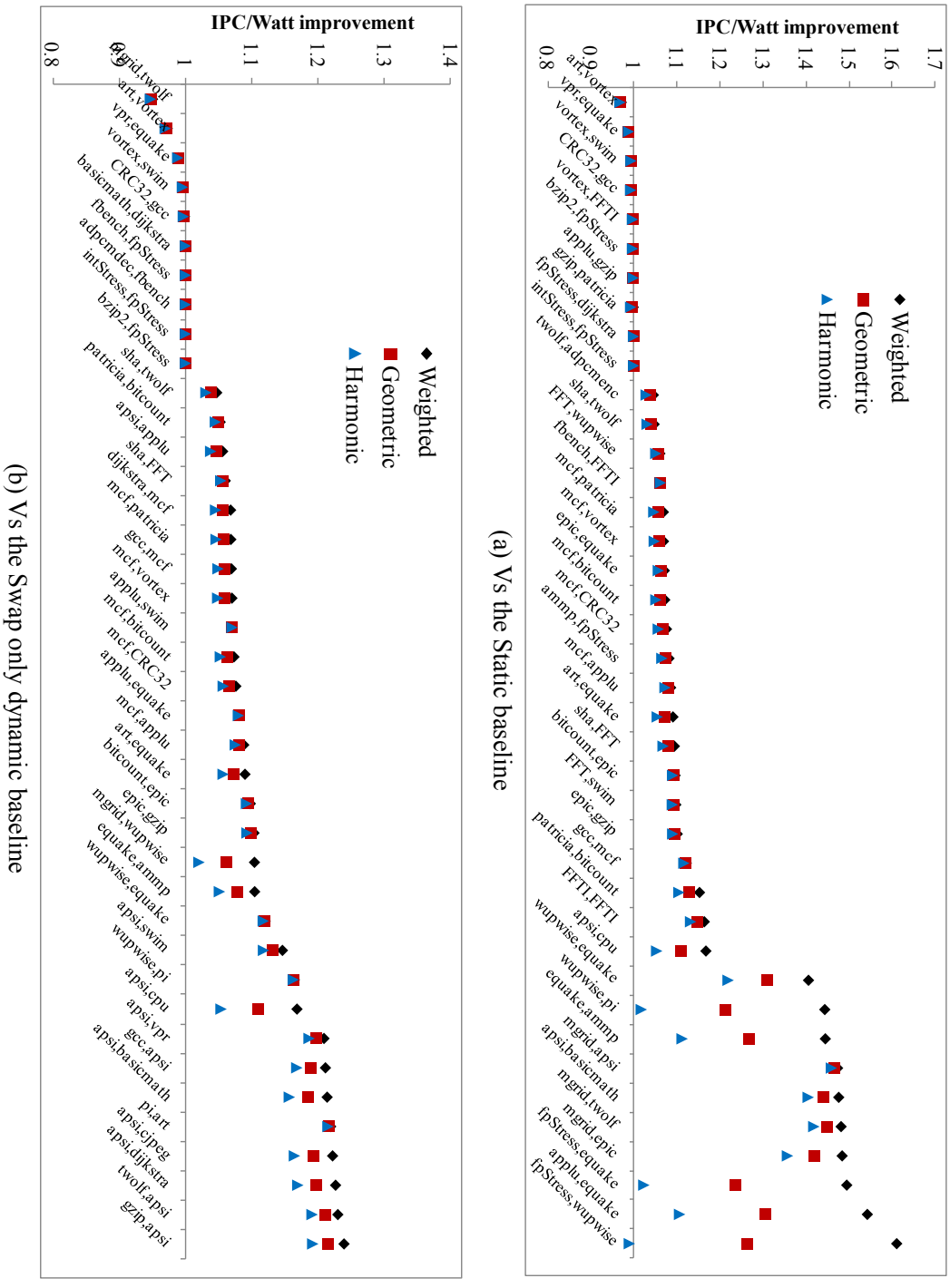


Fig. 11. IPC/Watt improvement over the various baselines for a subset of the workload combinations.

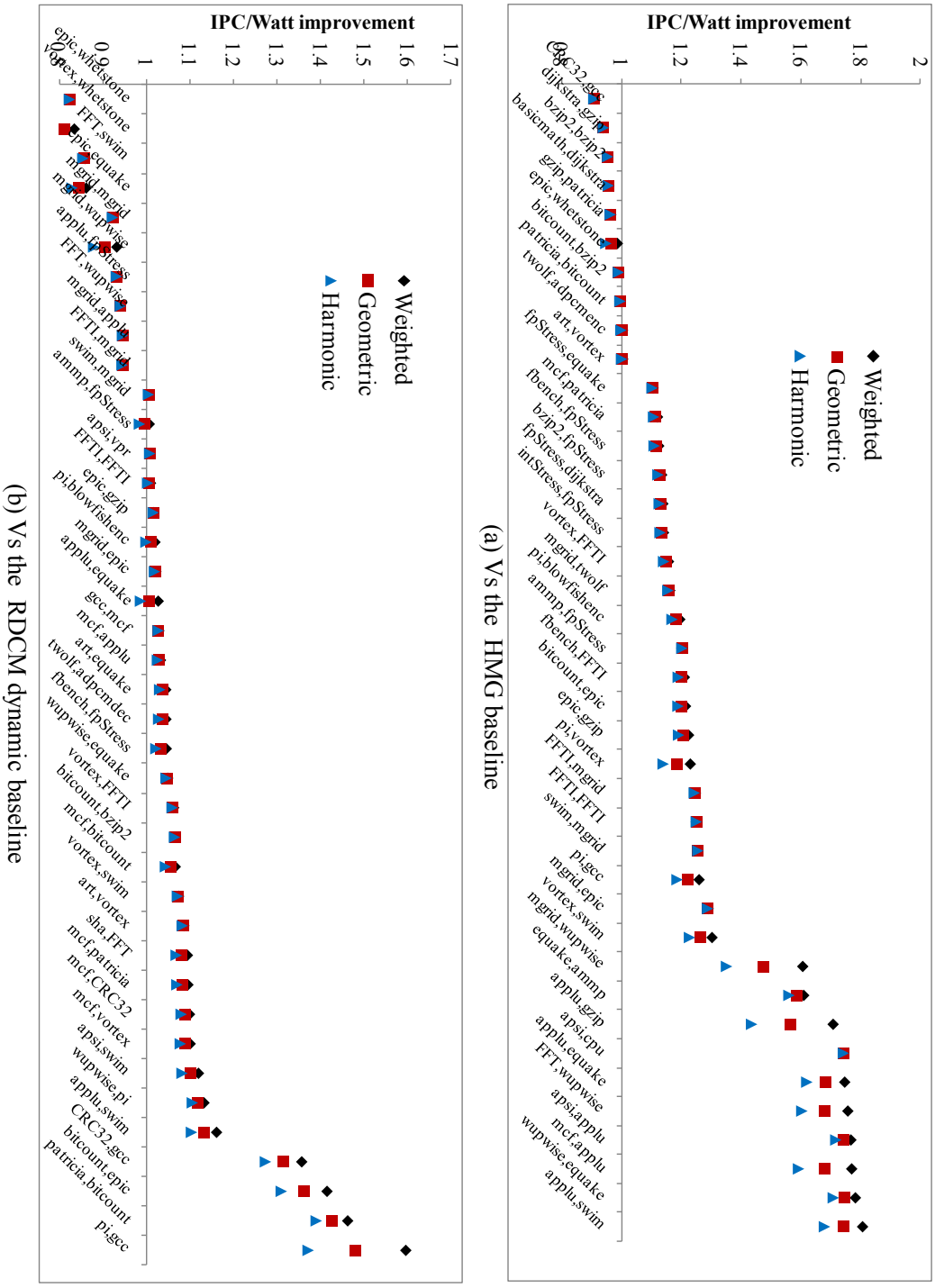
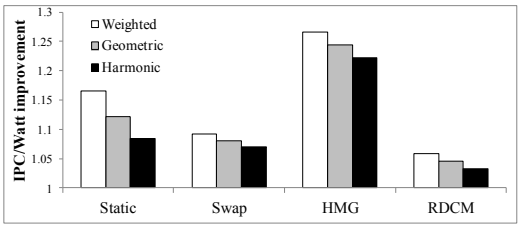


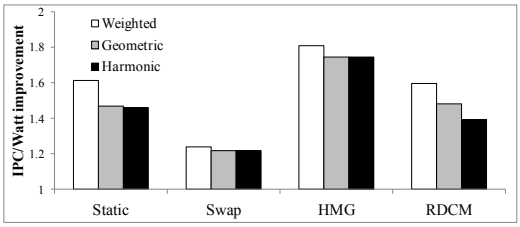
Fig. 12. IPC/Watt improvement over the various baselines for a subset of the workload combinations.

where average, maximum and minimum improvements over all baselines are plotted), a significant improvement of 16% is observed with respect to weighted IPC/Watt, which more than justifies the rare cases where no reconfigurations take place.

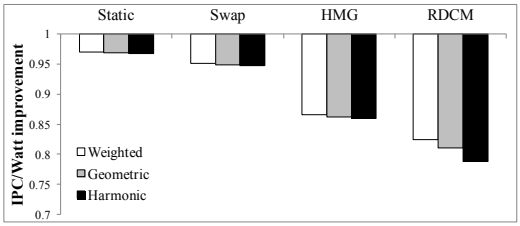
Swap. This is one of the two baselines that are dynamic. Here, whenever deemed beneficial, the threads are swapped between cores. The decision to trigger swapping is determined by the same mechanism that is used by the PCDCM scheme, but, this baseline is unable to morph core resources. Although this scheme is dynamic, it can be seen that the IPC/Watt improvements are significant on average (see Figure 13(a)). Also, there are only four cases where IPC/Watt improvement is < 1 (the leftmost workload combinations in Figure 11(b)). By allowing cores to morph, the execution of the thread on the Morphed core is accelerated, while that on the Weak core is slowed down. As a result, the phase combinations that are encountered between the two workloads, when the cores have morphing capability and when they do not, are very different. This results in sometimes, different reconfiguration decisions made by the PCDCM and swap-only schemes. For example, when running the workload combination *mgrid,twolf* (leftmost combination in Figure 11(b)) where a speedup of 0.97 was observed, the PCDCM scheme performed morphing 10 times, while the swap scheme made no reconfiguration. Since the proposed scheme is greedy in its decision making, thread re-scheduling decisions are made even for the short lived phases. Hence sometimes, the overheads outweigh the benefits which is what led to the PCDCM scheme performing slightly worse. This however, is not a frequent occurrence and it happens only in 4 out of the 100 combinations of workloads. There are also fourteen workload combinations where the IPC/Watt improvement is 1, which suggests that both schemes make the same decisions. However, in the rest of the 82 of 100 workload combinations, the PCDCM scheme significantly outperforms the swapping only dynamic baseline. This happens as there are some workloads that have phases that have a good mix of FP and INT instructions. In such cases the swap only scheme must arrive at a compromise, while the PCDCM scheme detects these phases, and accelerates them by mapping those phases on the Morphed core. Even though the other thread must now be mapped onto the Weak core, the benefits achieved by morphing results in an overall IPC/Watt improvement in the system. The workload combination *wupwise,pi* is an example of such an occurence. Sometimes, the thread that will be mapped onto the Weak core may be affine to it. For example, consider the worklod combination *gcc,apsi* in Figure 11(b). The workload *apsi* shows significant IPC/Watt improvement (>200%) for a couple of its phases when mapped onto the Morphed core and the workload



(a) Average IPC/Watt improvement



(b) Maximum IPC/Watt improvement



(c) Minimum IPC/Watt improvement

Fig. 13. Average, maximum and minimum IPC/Watt improvement of the proposed scheme over the various baselines.

gcc is memory intensive (see Figure 3).
Thus, *apsi* is naturally affine to the Morphed core and *gcc*, to the Weak core most of the time. Hence, when running the workload *apsi,gcc*, the PCDCM scheme achieves almost a 20% improvement over the swap only baseline which is unable to take advantage of such scenarios. The average IPC/Watt improvement over 100 combinations of workloads was found to be 9% (see Figure 13(a)).

It may be noted that the phase classification based "swap" scheme achieves a weighted IPC/Watt improvement of about 8% over the static baseline. Hence, such a dynamic swap scheme may be beneficial for architectures that do not include the hardware support for morphing.

HMG. This baseline is an area-equivalent symmetric multicore. In general, IPC/Watt is significantly improved by the PCDCM scheme, when compared to this baseline. On the other hand, the number of cases where PCDCM performs worse is on the higher side (9 out of the 100 combinations). Moreover, the worst case weighted IPC/Watt improvement of 0.86 is one of the worst when compared to all other baselines. This happens because the HMG baseline is well suited to running certain homogeneous workload combinations. For example, the left most workload combinations in Figure 12(a), i.e. *CRC32, gcc*, *dijkstra, gzip* and *bzip2, bzip2* are all INT intensive. In such cases, having a homogeneous multicore may be a better option as both the threads are affine to the same core type in the AMP, the thread assigned to the other core type will suffer with respect to performance. This is evident from Figure 12(a). If however, one of the workloads being executed has FP instructions, PCDCM may perform better even if those workloads are similar. As an example, consider the symmetric workload combination *FFTI, FFTI* in Figure 12(a) which shows a weighted IPC/Watt improvement of 25% when run on the PCDCM scheme. This happens as *FFTI* shows phases which are FP/INT intensive or have both. Phases that have a reasonable proportion of INT and FP instructions are naturally affine to the Morphed core. PCDCM detects those and makes intelligent thread mapping decisions to improve IPC/Watt. On an average for the 100 combinations, PCDCM scheme achieves 26% IPC/Watt improvement over the HMG baseline (see Figure 13(a)).

RDCM. This is our earlier published work [Rodrigues et al. 2011]. Here decisions to morph or trigger thread swapping are governed by rules obtained offline through profiling a subset of 12 workloads (*apsi, applu, art, swim, equake, epic, fft, twolf, ammp, gcc, mcf, bzip2*). Even though a considerable IPC/Watt improvement was achieved over the static baseline by this scheme, there is still room for further improvement. The PCDCM achieves IPC/Watt improvements even over this scheme. Since PCDCM determines reconfiguration decisions online, it is not dependent on the training set. As a result, it makes better decisions as compared to the RDCM scheme when workloads that are not part of the training set, are encountered. For example, consider the workload combination of *patricia,bitcount* which is toward the right extreme in Figure 12(b). Here, a weighted IPC/Watt improvement of 40% was observed. None of the two were part of the training set. Still, there are workload combinations where the IPC/Watt improvement of PCDCM over RDCM is < 1. This happens as the RDCM scheme makes thread scheduling decisions at smaller granularities on instruction window (500 instructions as compared to 150K by the PCDCM scheme). Hence, the RDCM scheme is better equipped to detect and take advantage of short lived program phases, some of which may never even be seen by the PCDCM scheme due to its large instruction window. Such short lived phases that showed benefits from Morphing were observed for the workloads *epic, FFT* and *whetstone*. From Figure 12(b), it can be seen that the majority of the workload combinations where IPC/Watt improvement was less than 1, had one of these workloads. Still, on an average, PCDCM achieves 6% IPC/Watt

improvement over RDCM as seen in Figure 13(a). One of the major benefits of the proposed PCDCM scheme is that it will always make intelligent scheduling decisions irrespective of the incoming workloads, unlike the RDCM scheme, the benefits of which depends on the training set used. Further, the RDCM schemes rules are only valid for the architecture considered in this paper. For different architectures, a different set of rules may have to be determined. This is not the case for the proposed scheme which will work just fine for any core types. The PCDCM scheme is therefore, more scalable than the RDCM scheme.

6.3. Overheads vs. benefits

The PCDCM scheme uses online dynamic learning and phase tables, controlled by the microvisor to provide performance per watt benefits. In this subsection, we quantify the software and hardware overheads of the scheme and also present the effect of the overheads on its benefits.

6.3.1. Software overheads. The proposed PCDCM scheme relies on a dynamic online learning mechanism to make thread to core decisions online. This mechanism has an overhead, as indicated earlier. The overheads arise due to microvisor function, sampling to determine IPC and power information and the times when cores need to swap thread contexts.

As described earlier, the microvisor is invoked whenever a new phase or phase change is detected. Table lookup is then performed and the information is used to determine the weighted speedup metric which is then used to determine the best thread to core mapping based on the newly detected phase. We estimate the overhead of this procedure to be a few hundred cycles every time it happens. We set this number conservatively, as 500 cycles for our experiments. It was observed in our experiments (consisting of 100 combinations) that there were around 5 phases detected on an average and, the maximum number of phases detected was 17. Also, phase changes were detected around 800 times on average and the maximum number of phase changes detected was 2020. Hence, the overhead due to microvisor invocation was found to be (3 + 800) times 500 cycles which equals 401K cycles on average and, (17 + 2020) times 500 cycles which equals around 1M cycles overhead for the worst case.

The second source of overhead is that of the IPC and power sampling. When a new phase is detected, it is sampled on each core type for the defined interval length n (150K instructions) and stable phase interval m of 4. Hence, a total of 600K instructions is executed on each core type during the sampling phase. On an average, we estimated that it would take around 2.5M cycles to execute this. During this dynamic online learning process, the system continues with one of the core types and hence only 75% of the 2.5M cycles is the actual overhead of sampling. Thus, a significant portion of the overhead is due to online learning. In our experiments, as described earlier, average/maximum phases detected were 5/17 and the corresponding average/maximum overhead due to sampling were 12.5/42.5 million cycles. Considering that each experiment runs for a few billion cycles, this overhead in cycles comes to be around 0.2% on an average and, 0.8% in the worst case. It is worth noting that since we use the phase table, we avoid the sampling process whenever the same phase is detected again. If that were not the case, sampling would have to be done 2020 times in the worst case, for every phase change and this would have significantly increased the overheads. The third source of overhead stems from the context switch whenever the microvisor determines that the cores must swap their contexts or morphing of resources must take place to maximize

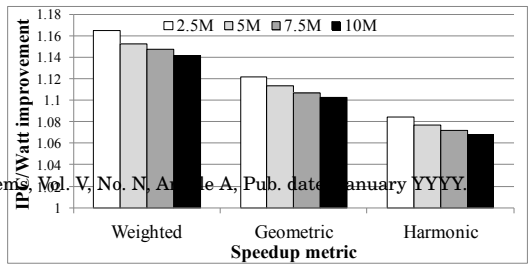


Fig. 14. Weighted, geometric and harmonic IPC/Watt improvement over the static baseline for increasing overhead for dynamic online learning.

performance per watt. It was observed in our experiments that the thread swaps and hardware reconfigurations happened around 90 times on average and around 1000 times in the worst case. Again this overhead can vary from one architecture to the other. Architectures with support for thread swapping may incur up to a thousand cycles overhead while it may be significantly larger for those without such support. We have assumed this overhead to be 1K cycles and hence the overhead due to thread swapping/morphing may be estimated to be 90K and 1M cycles on an average and in the worst case, respectively. Both of these are negligible considering that we execute the benchmarks for billions of cycles. We experimented with various context switch overheads of 10K, 50K, 100K and 1M cycles and found those to have negligible effect on the benefits of the proposed scheme.

Of the three sources of overhead, the overhead due to sampling dominates the others. To quantify the effect of these overheads on IPC/Watt improvement of our approach, we increased the sampling overhead from 2.5 to 10 million cycles. The result is plotted in Figure 14. It can be seen that even with a pessimistic overhead of 10 million cycles per sampling process, the scheme still achieves benefits of 14% over the static baseline (a drop of 2%), with respect to all three speedup metrics. Hence, we can conclude that the proposed scheme has a low sensitivity to the sampling overhead.

6.3.2. Hardware overhead. As mentioned earlier, in our experiments we noticed the average number of phases detected to be about 5. For the worst case scenario, this number went up to 17. Therefore, about 20 phase table entries may be sufficient for most cases. Each entry in the phase table captures the ITV, the performance and power information of the phase on each core types. Hence, an entry in the table consists of 12 fields, totaling to about 240 fields (12 fields/entry $\times$ 20 entries) for the entire phase table. Even if each field requires 32 bits, the size of the phase table would be less than 1 KB. Clearly, this is a small overhead considering the total gate count of the processor.

7. CONCLUSIONS

In this paper we presented an online program phase classification based thread to core assignment scheme to improve performance per watt of an asymmetric multiprocessor system. The studied AMP architecture features two cores: one with strength in floating-point computation and the other in integer intensive workload. By morphing the two cores, we obtained a core that is strong in both integer and floating-point computations, but this has resulted in the second core becoming much weaker. We deployed adaptive core morphing alongside thread swapping, at runtime, to reassign threads to cores using the above program phase classification. Whenever a new phase is detected, the phase is sampled on each core configuration to obtain an estimate of the performance/watt using performance monitors. This information is then stored in a table for future reference. When an already classified phase is detected again, a simple table lookup is performed to determine the best thread to core assignment. To evaluate the scheme, several static and dynamic reconfiguration alternatives were considered. Using the proposed dynamic reconfiguration scheme, substantial performance/watt gains are achieved. Our results show that the proposed PCDCM scheme, on an average, outperforms the static heterogeneous baseline by about 16%, the homogeneous baseline by 26% and the best dynamic baseline by 6%, with respect to weighted IPC/Watt. Since the proposed scheme is based on online learning with no prior knowledge regarding the

individual capabilities of the individual cores, it is not limited to the considered INT, FP dual-core but is applicable to any heterogeneous AMP.

REFERENCES

AMD. www.amd.com.

ANNAVARAM, M., GROCHOWSKI, E., AND SHEN, J. 2005. Mitigating amdahl's law through epi throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*. ISCA '05.

BALAKRISHNAN, S., RAJWAR, R., UPTON, M., AND LAI, K. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*. ISCA '05. IEEE Computer Society, Washington, DC, USA, 506–517.

BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*. CF '06.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*.

CHEN, J. AND JOHN, L. K. 2009. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference*. DAC '09.

CONTRERAS, G. AND MARTONOSI, M. 2005. Power prediction for intel xscale® processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on Low power electronics and design*. ISLPED '05. ACM, New York, NY, USA, 221–226.

DAS, A., RODRIGUES, R., KOREN, I., AND KUNDU, S. 2010. A study on performance benefits of core morphing in an asymmetric multicore processor. In *Computer Design (ICCD), 2010 IEEE International Conference on*.

GIBSON, D. AND WOOD, D. A. 2010. Forwardflow: a scalable core for power-constrained cmps. In *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. ACM, New York, NY, USA, 14–25.

GUTHAUS, M., RINGENBERG, J., ERNST, D., AUSTIN, T., MUDGE, T., AND BROWN, R. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*.

HELD, J., BAUTISTA, J., AND KOEHL, S. 2006. White paper from a few cores to many: A tera-scale computing research review.

HELLER, L. C. AND FARRELL, M. S. 2004. Millicode in an ibm zseries processor. *IBM Journal of Research and Development 48,* 3.4, 425 –434.

HILL, M. AND MARTY, M. 2008. Amdahl's law in the multicore era. *Computer 41,* 7, 33 –38.

INTEL. www.intel.com.

IPEK, E., KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. F. 2007. Core fusion: accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News 35*, 186–197.

JOSEPH, R. AND MARTONOSI, M. 2001. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 international symposium on Low power electronics and design*. ISLPED '01. ACM, New York, NY, USA, 135–140.

KHAN, O. AND KUNDU, S. 2010. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the 20th symposium on Great lakes symposium on VLSI*. GLSVLSI '10.

KHAN, O. AND KUNDU, S. 2011. Microvisor: A runtime architecture for thermal management in chip multiprocessors. *T. HiPEAC 4*, 84–110.

KIM, C., SETHUMADHAVAN, S., GOVINDAN, M. S., RANGANATHAN, N., GULATI, D., BURGER, D., AND KECKLER, S. W. 2007. Composable lightweight processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 40. IEEE Computer Society, Washington, DC, USA, 381–394.

KOUFATY, D., REDDY, D., AND HAHN, S. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*. EuroSys '10.

KUMAR, R., FARKAS, K., JOUPPI, N., RANGANATHAN, P., AND TULLSEN, D. 2003. 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*.

KUMAR, R., TULLSEN, D., RANGANATHAN, P., JOUPPI, N., AND FARKAS, K. 2004. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*.

KUMAR, R., TULLSEN, D. M., AND JOUPPI, N. P. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. PACT '06.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. MICRO 30.

LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. SC '07.

LUO, Y., PACKIRISAMY, V., HSU, W.-C., AND ZHAI, A. 2010. Energy efficient speculative threads: dynamic thread allocation in same-isa heterogeneous multicore systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. PACT '10.

NAJAF-ABADI, H., CHOUDHARY, N., AND ROTENBERG, E. 2009. Core-selectability in chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*. 113 –122.

PERICAS, M., CRISTAL, A., CAZORLA, F. J., GONZALEZ, R., JIMENEZ, D. A., AND VALERO, M. 2007. A flexible heterogeneous multi-core architecture. In *Proceedings of the 16th International Conference fmdahlon Parallel Architecture and Compilation Techniques*. PACT '07. IEEE Computer Society, Washington, DC, USA, 13–24.

RENAU, J. 2005. Sesc: Superescalar simulator.

RODRIGUES, R., ANNAMALAI, A., KOREN, I., KUNDU, S., AND KHAN, O. 2011. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. 121 –130.

SAEZ, J. C., PRIETO, M., FEDOROVA, A., AND BLAGODUROV, S. 2010. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European conference on Computer systems*. EuroSys '10.

SALVERDA, P. AND ZILLES, C. 2008. Fundamental performance constraints in horizontal fusion of in-order cores. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 252 –263.

SHELEPOV, D., SAEZ ALCAIDE, J. C., JEFFERY, S., FEDOROVA, A., PEREZ, N., HUANG, Z. F., BLAGODUROV, S., AND KUMAR, V. 2009. Hass: a scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev. 43*.

SHERWOOD, T., SAIR, S., AND CALDER, B. 2003. Phase tracking and prediction. In *Proceedings of the 30th annual international symposium on Computer architecture*. ISCA '03.

SHIVAKUMAR, P., JOUPPI, N. P., AND SHIVAKUMAR, P. 2001. Cacti 3.0: An integrated cache timing, power, and area model. Tech. rep.

SINGH, K., BHADAURIA, M., AND MCKEE, S. A. 2009. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News 37*, 46–55.

SPEC2000. The standard performance evaluation corporation (spec cpi2000 suite).

SRINIVASAN, S., ZHAO, L., ILLIKKAL, R., AND IYER, R. 2011. Efficient interaction between os and architecture in heterogeneous platforms. *SIGOPS Oper. Syst. Rev. 45*, 62–72.

VAN BERKEL, C. 2009. Multi-core for mobile phones. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.* 1260 –1265.

WINTER, J. A., ALBONESI, D. H., AND SHOEMAKER, C. A. 2010. Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. PACT '10.

Online Appendix to: Improving Performance per Watt of Asymmetric Multicore Processors via Online Program Phase Classification and Adaptive Core Morphing

RANCE RODRIGUES, University of Massachusetts at Amherst
ARUNACHALAM ANNAMALAI, University of Massachusetts at Amherst
ISRAEL KOREN, University of Massachusetts at Amherst
SANDIP KUNDU, University of Massachusetts at Amherst

A. DETERMINATION OF PHASE CLASSIFICATION PARAMETERS

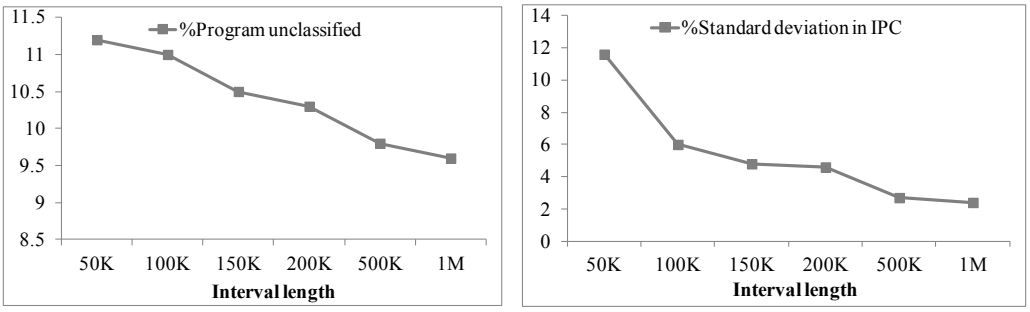
Khan et al. in [Khan and Kundu 2011] have observed that the value of n (the number of committed instructions in the fixed interval) is an important factor in phase classification, as choosing too small an interval may result in too many phase changes. On the other hand, a very large interval may not classify any phases at all.

Experiments were conducted by Khan et al. to determine the parameters of the phase classification mechanism namely: (i) interval length (n) (ii) phase detection threshold (Δ) and (iii) stable phase interval (m). We reran these experiments with an eye to (a) simplify the phase detection hardware and (b) improve accuracy of prediction against a much larger and diverse set of benchmarks. Based on these experiments, we reduced the ITV vector length from 9 to just 4, which cuts down the size of phase detection hardware by nearly half. As will be seen in Appendix B, such a reduction has little or no effect on the benefits of the phase chassification mechanism.

In order to determine the parameters for the phase classification mechanism, a number of interval lengths n were experimented with between 1K to 1M instructions. The threshold Δ was varied between 2.5 and 25% and the stable phase interval m was varied between 1 to 16. In order to measure the quality of the phase classification mechanism, we define the following two quality metrics (i) percentage of the program that can be classified into stable phases and (ii) standard deviation of the IPC between intervals classified under the same phase ID. Reconfiguration decisions can only be made if the thread under consideration is in a stable phase of execution. However, if the standard deviation of IPC between phases classified under the same phase is too high, there may be a large disparity between the estimated IPC/Watt improvement and reality. Hence, in general it is desirable that most of the program is classified as stable, and at the same time the standard deviation in IPC between phases classified under the same ID is as low as possible.

In general, we found that smaller intervals result in a high proportion of unstable phases and higher standard deviation in IPC between intervals classified under the same phase. Increasing the interval size results in a reduction of unstable phases and standard deviation in IPC between phases. This happens due to the averaging effect that takes place with an increasing interval size. However, too large an interval size may result in the entire program being classified into single phase. The phase classification mechanism will then not be able to detect changes in program behavior and hence adapt the architecture such that IPC/Watt is maximized. Hence the interval size must be small enough to detect small changes in program behavior, but at the same time, the two quality metrics described must be optimized. Thus, as the first step in optimization space exploration, we only considered those combinations of phase classification parameters that yielded % unstable phases and % standard deviation in IPC

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000



(a) %Program unclassified Vs Interval length (b) %Standard deviation in IPC Vs Interval length

Fig. 15. Sensitivity of the phase classification quality metrics to increasing interval length (n). Note that the results for combinations of phase classification parameters with the same interval length have been averaged.

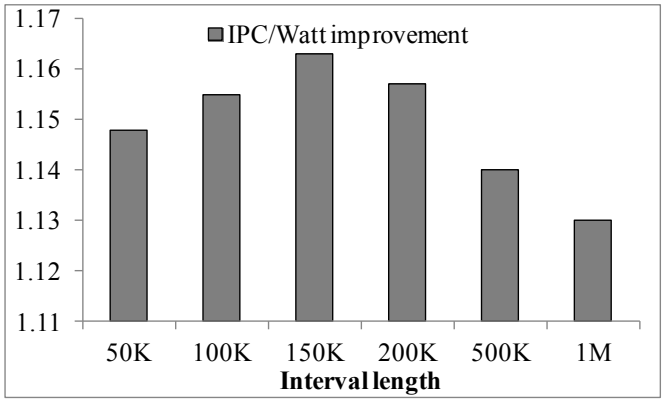


Fig. 16. IPC/Watt improvement for various interval sizes. Note that the results for combinations of phase classification parameters with the same interval length have been averaged.

to be below 12% (12% chosen as this resulted in greater than 65% reduction in the optimization space). The various combinations of phase classification parameters hence shortlisted fall within the pool where the interval n varies from 50K to 1M instructions, the threshold Δ is between 7.5 and 15% and the stable phase interval m varies from 2 to 8. The general trends observed in % program classified as unstable and % standard deviation in IPC for increasing interval size is shown in Figure 15. Note that we have averaged results obtained for combinations of phase classification parameters with the same interval size, in order to show the results in a single plot.

The ultimate purpose of phase classification is objective maximization, which in our case is IPC/Watt. Hence, for each shortlisted combination of phase classification parameters, we ran 100 random combinations of two threaded workloads from the the set of 38 and calculated the weighted IPC/Watt improvement over the static baseline with oracular thread to core assignment. Once again, to show the results in a single plot, we averaged results observed for the same interval size. The results are plotted in Figure 16. It can be seen that IPC/Watt improvement is the best for the interval size of 150K. From the considered combinations of phase classification parameters that had interval length as 150K, we found the largest speedup when using %threshold (Δ) as 7.5% and stable phase interval (m) as 4, which is what we used for our experiments in this paper.

B. ITV VECTOR LENGTH VS. PERFORMANCE/WATT BENEFITS

As mentioned earlier, the proposed scheme may not need the details of all the nine types of instructions. To illustrate the effect on the quality of phase classification going

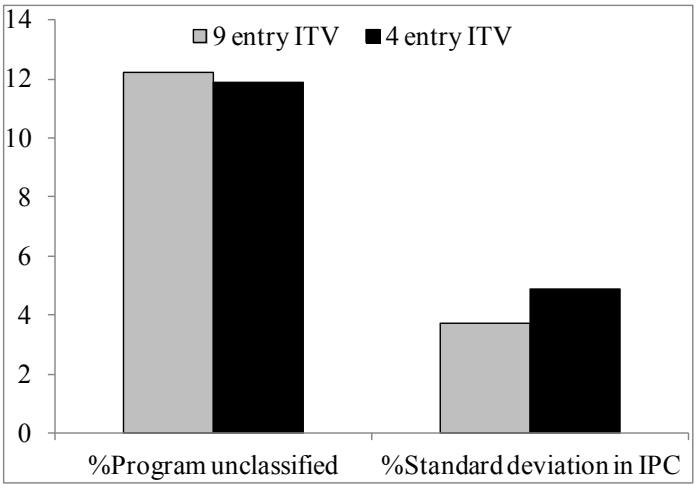


Fig. 17. %Program unclassified and % standard deviation in IPC when using a 9 entry and 4 entry ITV. It can be seen that quality degrades a little with respect to standard deviation in IPC, going from 9 to 4 entries which is expected.

from a 9 to a 4 entry ITV, we measured both the quality defining factors for both and the results are plotted in Figure 17. In this experiment, interval length n was kept at 150K instructions, %threshold Δ was kept at 7.5% and the stable phase interval m was kept at 4.

It can be seen that there is only a small quality degradation with respect to standard deviation of the IPC which is expected. This reduction in ITV length made a difference of less than 1% in the achieved IPC/watt benefits. Hence, we use a 4 entry ITV to save hardware.