Performance Per Watt Benefits of Dynamic Core Morphing in Asymmetric Multicores

Rance Rodrigues, Arunachalam Annamalai, Israel Koren, Sandip Kundu and Omer Khan†
Department of Electrical and Computer Engineering
University of Massachusetts at Amherst, University of Massachusetts at Lowell†
Email: {rodrigues, annamalai, koren, kundu}@ecs.umass.edu, Omer_Khan@uml.edu†

***Abstract*—The trend toward multicore processors is moving the emphasis in computation from sequential to parallel processing. However, not all applications can be parallelized and benefit from multiple cores. Such applications lead to under-utilization of parallel resources, hence sub-optimal performance/watt. They may however, benefit from powerful uniprocessors. On the other hand, not all applications can take advantage of more powerful uniprocessors. To address competing requirements of diverse applications, we propose a heterogeneous multicore architecture with a Dynamic Core Morphing (DCM) capability. Depending on the computational demands of the currently executing applications, the resources of a few tightly coupled cores are morphed at runtime. We present a simple hardware-based algorithm to monitor the time-varying computational needs of the application and when deemed beneficial, trigger reconfiguration of the cores at fine-grain time scales to maximize the performance/watt of the application. The proposed dynamic scheme is then compared against a baseline static heterogeneous multicore configuration and an equivalent homogeneous configuration. Our results show that dynamic morphing of cores can provide performance/watt gains of 43% and 16% on an average, when compared to the homogeneous and baseline heterogeneous configurations, respectively.**

***Keywords*- Dynamic Core Morphing (DCM), Asymmetric Multicore Processors (AMP), Instructions per cycle (IPC), Area-equivalent homogeneous multicore (HMG).**

I. 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 lowering processor frequency and integrating multiple processor cores on a die [38]. Still, a multicore die is limited by an overall power dissipation envelope that stems from packaging and cooling technologies. 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 [16], [32]. These workloads can benefit from more powerful cores to support higher instruction throughput. In order to achieve reasonable performance per watt, applications should have (i) low execution times 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. However, 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 per watt. Some workloads are memory bound, some are integer intensive, while some others are floating-point intensive. Thus, different workloads benefit from different resources. Even within a workload, the resource requirements may vary with time due to changes in program phases [3], [28]. Within a given resource budget, when computing demands are matched with processor capabilities, AMPs tend to perform better than SMPs [13], [18], [30]. The resource matching problem of AMPs has been well documented [5]. This problem is not limited to AMPs, as even SMPs may become asymmetric under processor reconfiguration schemes such as Dynamic Voltage and Frequency Scaling (DVFS) or partial shutdowns [25], [37]. In principle, developing multithreaded applications for cores performing at varying levels is notoriously difficult. Despite that, AMPs are gaining traction from smart phones [26] to integrated graphics processors [27] due to their power-performance benefits. We propose to improve the efficiency of AMPs (in terms of performance par watt) by adaptively matching the processor capability to the computing needs of the executing threads. We do so by either swapping threads between cores (of different capabilities) or by morphing core resources dynamically.

At a base level, we propose an AMP architecture where each core is resourced moderately in all areas, while featuring extra strength in a specific area such as integer or floating-point operations. The strength of the cores is non-overlapping. Thus, each core is suited for specific workloads. 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 with stronger resources of other core(s). There are several benefits to this approach. First, it allows applications to exploit the most suitable core for better performance. Second, individual cores remain modest in their sizing, therefore allowing the AMP to meet the cost and power targets. Third, when operated in the morphed mode, the realigned resources enable higher levels of performance for the applications that can benefit from them. Recent studies have shown that symmetric cores are unlikely to provide better performance than a heterogeneous multicore [13], [18]. Further studies [9], [16], [18], [33], [36] have shown that reconfigurable architectures may increase the benefits of AMPs even further. This provides a strong argument for our target multicore architecture. In this work we use hardware performance monitors to discover

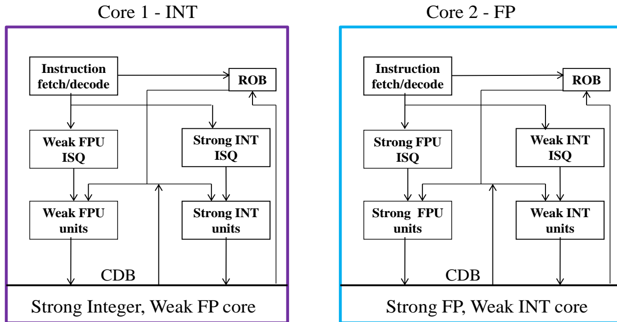


Figure 1. Baseline configuration for two heterogeneous cores.

thread to core affinity during runtime. Such discovery may trigger a thread swap or core morphing.

In the proposed scheme, the cores adapt to the time-varying computational requirements of an application. For the purpose of illustration, consider an AMP with two heterogeneous cores (see Figure 1). The first core is a 2-way superscalar capable of handling integer intensive operations, but has low performance for floating-point operations. The second core is capable of handling floating-point intensive operations, but has weak support for executing integer code. This is referred to as the baseline configuration in the paper. Core morphing in this example means that core 1 (in Figure 1) trades its weaker floating-point unit with core 2 to gain a stronger floating-point unit. This results in core 1 becoming strong on every front while core 2 becomes weaker on both fronts. When a change in the computation demands of the threads running on the two cores is detected, the system may decide to either perform *morphing*, return to the *baseline* configuration, or *swap* the two threads running on the cores. Detection of changes in the threads' behavior is done using hardware performance monitors that enable dynamic profiling of the threads by collecting their characteristics at runtime.

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 resulting performance-per-watt gain compared to the baseline heterogeneous configuration with static scheduling is significant.

The key contributions of this paper are:

1) A novel core morphing scheme that allows the AMP to morph at runtime. The reconfiguration allows the cores to match their hardware computation capabilities to the requirements of the executing workloads.
2) A novel hardware-based runtime algorithm to concurrently and non-invasively predict the performance per watt of applications while previously proposed dynamic schemes [30], [37] required sampling to determine the thread to core assignment. When the performance monitoring indicates that reconfiguration may be beneficial, our scheme either triggers a swap of the applications between cores, or performs a resource morphing, or reverts to the baseline configuration.

II. Related Work

Recently, reconfigurable multicores have received considerable attention. Core fusion was presented in [8] where the cores of a homogeneous CMP were reconfigured at runtime into stronger cores by "fusing" resources from the available cores. Another approach to fusion of homogeneous cores is presented in [10], 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 [2].

Aggregating cores in an SMP [8], [10], [36] offers more of the same resources and hence its performance benefits saturate as the Instruction Level Parallelism (ILP) saturates. Variations in computational demands of applications also exist. For example, in the SPEC benchmark [7], *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.

Heterogeneous architectures have been proposed to achieve higher performance per area and per watt [12]. In [3], Kumar et al. show their benefits in terms of reduced power using a single ISA heterogeneous CMP. Grochowski et al. [6] have explored various methods of reducing power consumption and report that heterogeneous cores are the most useful for this purpose.

In [13], Kumar et al. address the design of an AMP, targeting area and power efficiency. They use cores that match the resource requirements of certain types of workloads. Das et al. [21] have proposed an asymmetric dual-core processor with one core having strong integer instruction support while the other has strong floating-point support. The two cores can be fused into a single strong processor which retains the front-end of the INT core, and takes over the floating-point units of the FP core. The front-end of the FP core remains idle. Their scheme is thus static where the cores are either morphed or not for the entire program run. However, static morphing of the cores does not necessarily suit the different phases in an application and there is a need for an architecture that would dynamically adapt to the time-varying behavior of the applications.

There has been a number of schemes proposed for dynamic reconfiguration and thread migration in an AMP. In [30], Winter et al. explore thread scheduling and global power management techniques in AMPs. They compare different algorithms like brute force, greedy and local search for thread scheduling. All exmained schemes require sampling to determine the best thread-to-core assignment. Luo et al. [34] propose a thread allocation mechanism that dynamically determines how speculative threads are allocated in a multiple same-ISA heterogeneous multicore to achieve performance improvement with moderate energy increase. In [32], Gibson et al. propose a forward flow architecture where the execution logic can be scaled to meet the requirements of the incoming workloads. In [9], Najaf-abadi et al. propose core selectability where each "node" in the system consists of different types of cores that share common resources. Depending on the application, the respective core from the node is selected to serve that application. Chen et al. [33] propose a flexible multicore architecture for media processing which consists of one RISC processor, a reconfigurable controller, DSP blocks and Intellectual

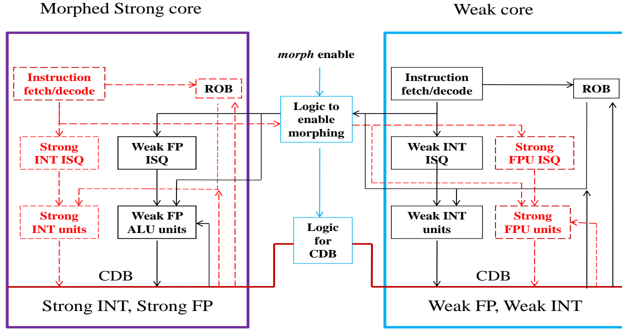


Figure 2. Morphed configuration for two heterogeneous cores. The red dotted lines/boxes indicate the connectivity for the strong morphed core configuration and the black solid lines/boxes indicate connectivity for the weak core.

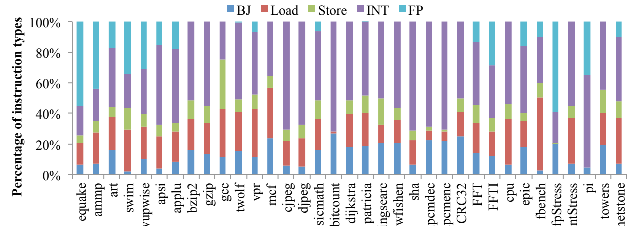


Figure 3. Percentage distribution of the instructions for the 35 benchmarks.

Property blocks.

III. Proposed Architecture

In this section, we describe in detail our proposed DCM scheme. To illustrate our approach, we consider two cores per tile: a FP core and an INT core where a multicore system may consist of as many such tiles as deemed appropriate. 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 queues (ISQ) and LSQ. The reasons for the specific values of these parameters of the individual cores are explained in Section IV. This scheme is similar to that proposed in [21]. However, significant enhancements were made to the scheme. Firstly, in [21], Das et al. did not explore the processor design space as in depth as we do. In addition to the parameters considered in [21], we performed design space exploration of the INT/FP ISQ and the INT/FP rename registers. Secondly, they explore performance benefits while we focus on performance/watt. Lastly, the architecture proposed in [21] is static while ours can dynamically reconfigure to meet changing application requirements.

In the baseline configuration (Figure 1) the cores operate independently providing good performance whenever a parallel workload with appropriate resource requirements is executed. When 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, as explained in Section IV, to suit the application running on it and reduce power. 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 operation, we also allow the two tightly coupled heterogeneous cores to swap their execution contexts.

The proposed DCM scheme is a hardware-only 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 hardware overhead for the proposed architecture which is similar to that in [21] can be estimated to be approximately 1%.

IV. Determining the Core Parameters

The design space for each core is extremely large including the exact sizes of individual structures (e.g., reorder buffers and issue queues). Our goal is to focus on a set of parameters that have the largest impact on the 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. If the cores are undersized, the results of core morphing would be biased and misleading.

We used SESC as our architectural performance simulator [15], and CACTI [23] and Wattch [22] to estimate power.

A. Benchmarks

For our experiments, we have selected 35 benchmarks (see Table I): 13 benchmarks from the SPEC suite [7], 14 from the embedded benchmarks in the MiBench suite [14], one benchmark from the mediabench suite [17], and 7 additional synthetic benchmarks. These 35 benchmarks encompass most typical workloads, for example, scientific applications, media encoding/decoding and security applications. The instruction type distribution of the selected benchmarks is depicted in Figure 3 showing the diversity of resource requirements of the different workloads.

B. Core sizing

To determine the architectural parameters for the cores, we have started with a baseline configuration and then upsized the parameter under consideration and recalculated the instructions per cycle (IPC) metric for each core type. Based on the IPC, the most appropriate value for each parameter was selected. The baseline configuration along with the steps used for the parameter search are shown in Table II.

The parameters that were varied for design space exploration were the L1 and L2 caches, reorder buffer (ROB), load store queue

Table I
BENCHMARKS CONSIDERED

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

Table II
PARAMETER VARIATION STEPS FOR THE EXPERIMENTS

Parameter	Size	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

Table III
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

(LSQ), integer issue queue (INTISQ), floating-point issue queue (FPISQ), floating-point registers (FPREG), and integer registers (INTREG). For the sake of brevity only ROB sizing results are shown in Figure 4. In the figure, each curve represents the ratio of the performance for the core when going from a smaller to larger ROB. 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 to 256. Hence, the size 128 is chosen for the FP ROB. 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. For a fair comparison between our dual-core AMP and a 2-core Homogeneous (HMG) design, 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 were obtained by averaging those obtained for the INT and FP cores. As mentioned earlier, whenever the multicore enters the Morphed mode of operation, the FP core turns into a Weak core. Since this core is not expected

Table IV
EXECUTION UNIT SPECIFICATIONS FOR THE CORES. LATENCIES TAKEN FROM [21] (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

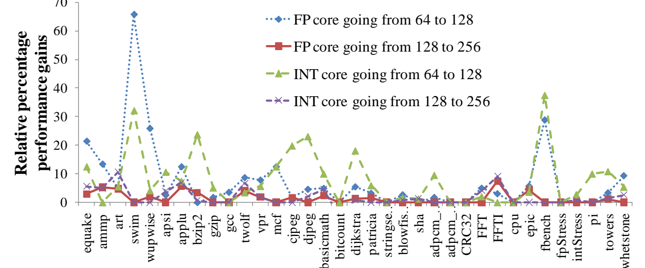


Figure 4. Ratio of the IPC for the core configurations when going from lower to higher sizes of ROB.

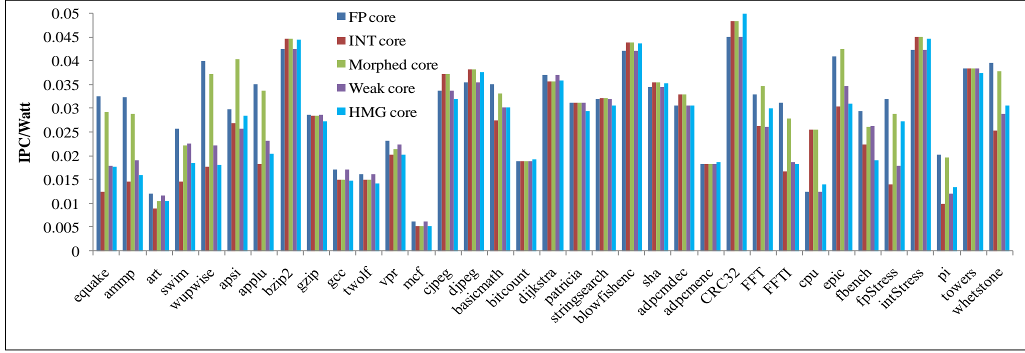
to provide a performance as high as the original FP core, we did similar sizing experiments to try and downsize this core for energy efficiency. The configuration for all core types is shown in Table III. We did not include the final configuration of the Morphed mode as it is nothing but a combination of the INT core with the FP units of the FP core. The performance/watt and performance of these cores are discussed in the next section. The specifications of the execution units is shown in Table IV. The execution unit specification values for the FP and INT cores were taken from [21] and the HMG execution unit specifications were derived from those once again by averaging.

V. PERFORMANCE/WATT AND PERFORMANCE OF THE CORE CONFIGURATIONS

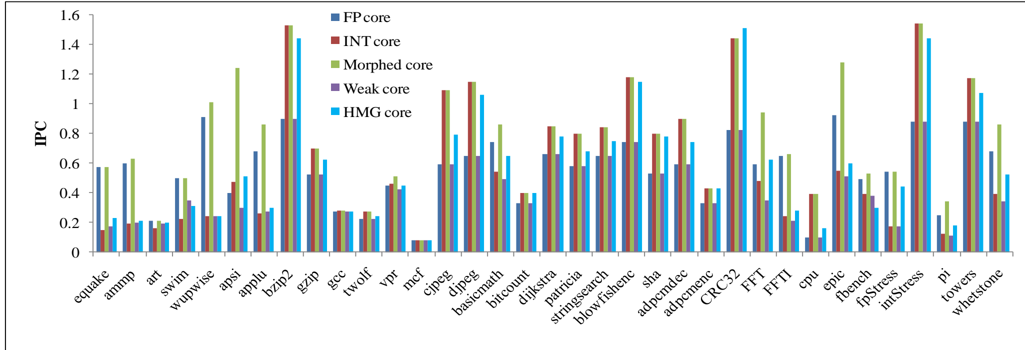
In this section we first analyze the performance/watt and performance of each core by running one application at a time on the various core types, i.e., FP, INT, Morphed, HMG and Weak cores.

A. Performance/Watt evaluation

We ran our 35 benchmarks on each core configuration and the performance/watt results are plotted in Figure 5(a). We observe that 3 benchmarks (*apsi, epic, FFT*) in the morphed mode show notable gains. Out of these, *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 temporal distribution of the instruction mix in *apsi*. Having considered an architecture similar to ours, Das et al. [21] noted that whenever there is a phase in the program where there is a considerable mix of FP and INT instructions, the morphed core performs better than the others. They compared *applu* to *art* and showed why even though *applu* and *art* have the same instruction mix over a long run, it is the bursty behavior in the instruction mix that makes the difference. Since the morphed 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. 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



(a) IPC/Watt for all benchmarks



(b) IPC for all benchmarks

Figure 5. IPC/Watt and IPC for the 35 benchmarks considered when run on each core configuration for 10 million instructions.

reason why only 3 out of the 35 benchmarks show significant benefits when run on morphed core statically throughout their execution. In such cases, the power expended by the morphed core outweighs the obtained performance benefits resulting in poor performance/watt metric. This is evident from Figure 5 (b) where the morphed core performs either equally well or better than the other core configurations when only IPC is considered.

B. Performance evaluation

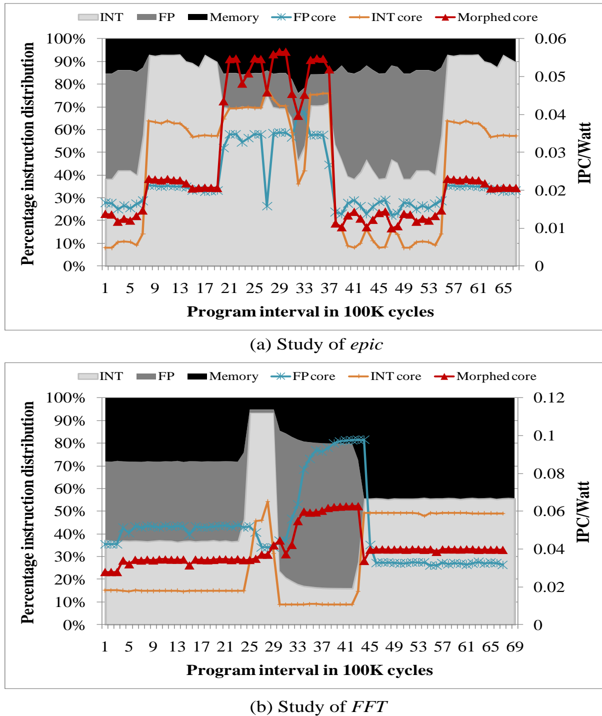
We also investigated the effects of our scheme on the performance of the benchmarks. The results obtained for the IPC are shown in Figure 5 (b). Here it can be seen that there is a bigger group of benchmarks (*applu, wupwise, apsi, basicmath, epic, FFT, whetstone*) that show benefits from morphing and even the gains are higher (>200% for *apsi*). However, this performance gain may not always result in a higher power efficiency.

C. Impact of program phases

We observed that over entire runs of 10 million instructions, some benchmarks benefit, some don′t while some others even lose out. Evaluation over entire runs does not take into account the changes in program behavior that is observed in most applications [3], [28]. In order to demonstrate the effect of program phases on performance/watt, we consider two benchmarks, *epic* and *FFT* that show benefit from morphing. We want to investigate the effect that instruction distribution of a benchmark may have on performance/watt.

1) Study of epic: The benchmark *epic* was run for a few billion instructions and the results are shown in Figure 6(a). The performance/watt for each core type (FP, INT and Morphed) is represented by the blue, orange and red curves, marked with an x, 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 - memory). It can be seen that for the first 19 data points, the morphed 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 morphed 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 performance/watt for that thread.

2) Study of FFT: FFT was also found to be one of the applications that benefit from morphing. The performance behavior when *FFT* is run on each core type is shown in Figure 6(b). It can be seen that even though over the entire run *FFT* shows a 10% benefit when running on the morphed core, at no point in the plot does the morphed core outperform the baseline cores. Initially up to data point 22, the FP core does the best. Then, for data points 22 to 26, the INT core exhibits a sudden increase in performance followed by another phase where the FP core does much better than any other core (80% better than the closest competitor). This continues until



(a) Study of *epic*

(b) Study of *FFT*

Figure 6. Zoomed view of variations in the performance of *epic* and *FFT* when run on each core configuration.

data point 41. After that, the instruction distribution shows a sudden drop in FP instructions and a surge in INT instructions. Hence, the INT core now provides the best performance. In summary, even though *FFT* can benefit from running on the morphed core for the entire run, swapping the threads between the two cores may provide the same or even better results. As mentioned earlier, in the morphed mode when *FFT* is executed on the strong morphed core, the second thread (executed along with *FFT*) is assigned to the weaker core potentially decreasing its performance. Swapping the threads may provide a better solution in such a case as the second thread's performance may not be greatly compromised. 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. The details of our dynamic decision making scheme are outlined in the next section.

VI. Dynamic Core Morphing Mechanism

Our DCM mechanism consists of two components: an online monitor and a performance predictor. The online monitor continuously and non-invasively profiles certain aspects of the execution characteristics of the committed instructions. The performance predictor collects the profiled application characteristics and, using the most recently collected information, determines whether to continue execution in the current configuration, or transition to another configuration.

A. Performance prediction at fine grain time slices

Prior knowledge about the computational needs of the applications is generally unavailable. Hence, an online mechanism is

```
Algorithm for dynamic reconfiguration:
1.  Threads T1 and T2 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 T1 (T2 )
           a.   %(FP + INT) >= 50 and
           b.   (17<=%FP<=30) and (26<=%INT<=44)
     ii. And T2 (T1 )
           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
```

Figure 7. Transition conditions for DCM scheme

needed to characterize the time-varying computational and resource requirements of the applications. Hardware support is needed to detect changes in the application's behavior and then decide whether to reconfigure the cores. The key program features that impact the performance/watt are continuously monitored and then used during dynamic core morphing. Since power is not a property that can be extracted during runtime, we use other program attributes as proxy for power when optimizing performance/watt. We use hardware counters that monitor the instruction composition (floating-point and integer) and the IPC of the thread that may be assigned to the weak core upon morphing. The required counters are similar to those used by Khan et al. [29] to keep track of instruction type distributions and IPC. We next describe the process that we have followed in order to make the morph/swap decisions based on the instruction composition and IPC.

For our experiments, twelve benchmarks from the suite of 35 (see Section IV) 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*). Threads were run for 40 million instructions on each core type, and IPC/Watt as well as 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 instruction distribution of both the threads in each window was 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 morphed core and *fft* on the weak core is higher than the baseline mode, this point 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. In this way, we found potential trigger points for morphing, swapping and reverting to baseline mode. Averaging the values of the percentage of FP instructions (%FP), percentage of INT instructions (%INT) and IPC that we have observed for the 132 combinations of two (out of the 12) threads, we set the rules for reconfiguration that are included in the algorithm in Figure 7.

It can be seen that for *morphed* mode, we keep track of not only the floating-point and integer instructions, but also their sum. This is because performance/watt benefits were observed

for the *morphed* mode only when the executing thread exhibited a reasonable mix of floating-point and integer instructions. We found that the combined percentage of FP and INT instructions should be higher than 50 (%FP + %INT $\geq$ 50). At the same time, minimum and maximum bounds are set for the %FP and %INT individually, such that when these bounds are violated, the threads should continue to run on the baseline configuration. In addition, it is also important to keep track of the IPC of the other thread to make sure that its performance is not greatly compromised by core morphing. We found this value to be 0.4. Moreover, we experimentally deduced that it would benefit to assign the second thread to the weak core only when its %FP + %INT $<$ 60.

Similarly, a morphed to baseline mode switch takes place when the total percentage of FP and INT instructions goes below 50 (%FP + %INT $<$ 50). At this point, all the benefits of morphing have diminished and it is better to operate in the baseline mode. Swapping is beneficial if the thread currently running on the INT core experiences a surge in FP instructions and the other thread on the FP core experiences an increase in INT instructions. Based on our experiments, we found the optimal values for the percentages of floating-point and integer instructions for the two cores to achieve higher performance/watt through swapping. As can be seen from the swapping condition in Figure 7, our dynamic morphing scheme benefits both the threads.

B. Accounting for program phase changes

A tentative decision based on the conditions mentioned in Figure 7 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 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 for the most recent 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. The history depth (indicated by n) and the size of the individual window have to be determined experimentally. We have conducted a sensitivity study to quantify the impact of window size and history depth on the quality of the reconfiguration decisions. The best choice would be the one that yields the largest weighted performance/watt speedup over the entire program execution.

1) Determining the best window size and history depth: Various window sizes of 250, 500, 1000 and 2000 instructions were considered. Based on the activity within a chosen period consisting of n windows, a reconfiguration decision is made. The history depth n was varied from 5, 10, 20, 50, 100 and 200 in our experiments. For example, if window size 500 is chosen with history depth 10, the scheme will rely on the behavior of the threads during the 500$\times$10=5000 recently committed instructions to make the reconfiguration decision. For each combination of window size and history depth, about 140 multiprogrammed workloads were run with a random combination of benchmarks from our set of 35. All experiments were run until at least one of the cores executed 40 million instructions. The weighted speedup in terms of performance/watt obtained from each individual experiment was then averaged to give a single value that represents the entire set. For example, when running an experiment with window size 250

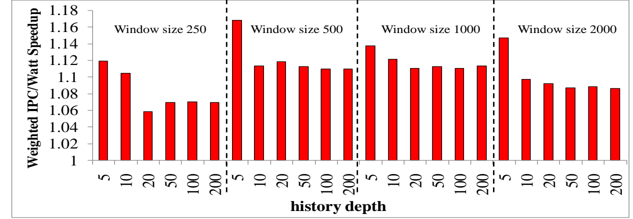


Figure 8. Performance sensitivity analysis for determining window size and history depth.

and history depth 20, 140 results were obtained and these were averaged to obtain the single representative value in Figure 8. From the weighted speedup results in Figure 8, it can be seen that the best speedup (taking into account a certain overhead for reconfiguration) is obtained for a window size of 500 instructions and a history depth of 5. A reconfiguration overhead of 400 cycles has been considered in these experiments as discussed in the next section.

C. Reconfiguration Overhead

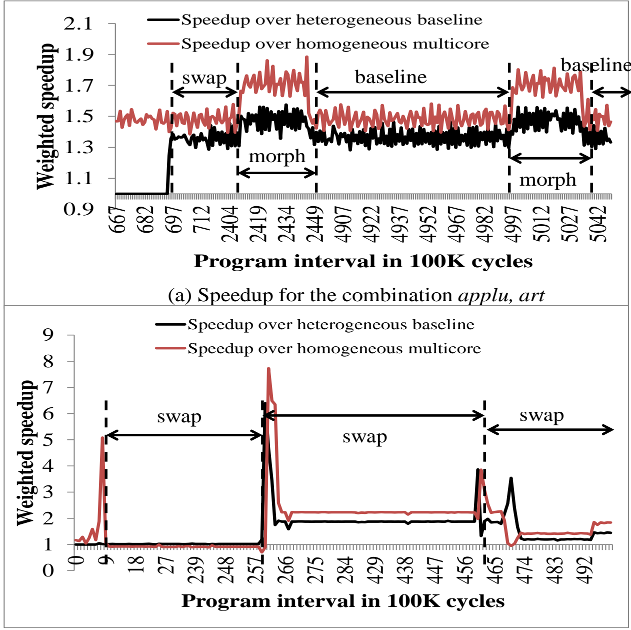
Core reconfiguration requires both the cores to stall execution. For swapping threads between the cores we need to flush the pipelines, exchange architecture states and warm the caches. Hence, the performance impact due to reconfiguration should be accounted for when calculating the weighted speed-ups. Some architectures may be very slow at transitioning states between cores (order of thousands of cycles) while others may have ISA support for state swapping between cores and thus exhibit much smaller penalties (order of hundreds of cycles). To quantify the reconfiguration overhead, experiments were run with pseudo-random combinations of benchmarks. The window size and the history depth were set to 500 and 5. It was found that a change in operating mode of the AMP happened forty times on an average in the experiments while executing 40 million instructions. Even when the reconfiguration overhead is as high as 10,000 cycles, the overall penalty is 400,000 cycles out of the 40 million cycle, i.e., about 1%. With dedicated support for state swapping, far lower overheads can be expected and we used an overhead of 400 cycles in our experiments.

VII. Evaluation

Performance/watt of our proposed 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 the dynamic scheme, a random initial thread to core assignment was made. The hope is that the dynamic scheme will detect the best assignment shortly after the programs begin to run. The two threads were run on the dual-core until completion and weighted/geometric speedup were calculated. We first present in depth studies for a few benchmark combinations and then present the results for a large number of other benchmark combinations.

A. Detailed time-slice analysis of workload performance

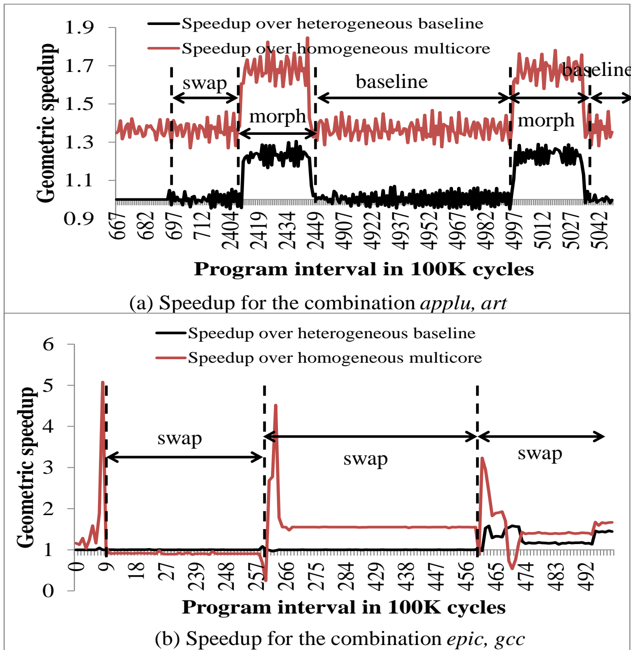
An in-depth analysis for the two benchmark combinations, ({*applu, art*} and {*epic, gcc*}) is shown, at time slice intervals of 10,000 cycles, in Figures 9 and 10 with respect to weighted and geometric speedups, respectively. Note that the time scale on the x-axis is not continuous. This was done because we wanted to show only the interesting sections of the plot. For the combination



(a) Speedup for the combination *applu, art*

(b) Speedup for the combination *epic, gcc*

Figure 9. Weighted speedup of the dynamic scheme vs. the homogeneous and heterogeneous baselines with respect to IPC/Watt.



(a) Speedup for the combination *applu, art*

(b) Speedup for the combination *epic, gcc*

Figure 10. Geometric speedup of the dynamic scheme vs. the homogeneous and heterogeneous baselines with respect to IPC/Watt.

applu, art in Figure 9(a) (weighted) and Figure 10(a) (geometric), it can be seen that there are five reconfigurations: one *swap*, two *morph* and two back to *baseline* mode. Initially, up to data point 682, the dynamic scheme performs as well as the static heterogeneous scheme as they both have the same initial thread-to-core assignment. However, the dynamic 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 the 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 can be attributed to the fact that the dynamic 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 the dynamic 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. Following that, the dynamic scheme returns to the *baseline* mode as the instruction composition changes and the performance/watt drops a little. It may be noticed that the obtained geometric speedup is relatively smaller.

A similar trend in speedup is seen for the benchmark combination $\{epic,\ gcc\}$ in Figure 9(b) (weighted) and Figure 10(b) (geometric) where there are three *swaps* and at each reconfiguration there are jumps in performance/watt speedup. *epic* is an application that has overall a high percentage of both FP and INT instructions but, as was observed in Figure 6(a), it has phases where there are very few or no FP instructions. *gcc* on the other hand, does not show any FP activity throughout. Hence, there are cases where both the threads would benefit from being assigned to the INT core. This conflict is resolved by the dynamic scheme by measuring the performance gain that each thread will have (by means of instruction composition) and assigning accordingly the thread that is predicted to provide better performance/watt to the INT core. It may also be noted that during the run (data points 9 to 257 for both weighted and geometric plots), the dynamic scheme does slightly worse than the homogeneous one and only just better than the heterogeneous scheme (about 1%). This happens as the decisions to swap or morph are made only when both cores satisfy the conditions defined in Figure 7. Sometimes only one core satisfies the conditions resulting in no mode change. However, at data point 257, both cores have complied and hence a swap is performed and gains are observed. Notice that relatively smaller gains are achieved here when the geometric speedup is considered. After that, another swap happens at data point 465 and gains are observed with respect to both types of speedup. On average, the dynamic scheme outperforms the heterogeneous baseline and the homogeneous multicore in the long run.

B. Overall Performance

Having discussed the in-depth behavior for two benchmark combinations, results are now presented for 40 combinations of benchmarks with respect to the weighted and geometric speedup in Figures 11 and 12, respectively.

The 40 combinations were carefully chosen out of a pool of 150 randomly generated benchmarks combinations where all 35 benchmarks participated and not just the 12 that were used to construct the algorithm in Figure 7. The selected 40 combinations include the 10 worst results, the 10 best results and 20 that

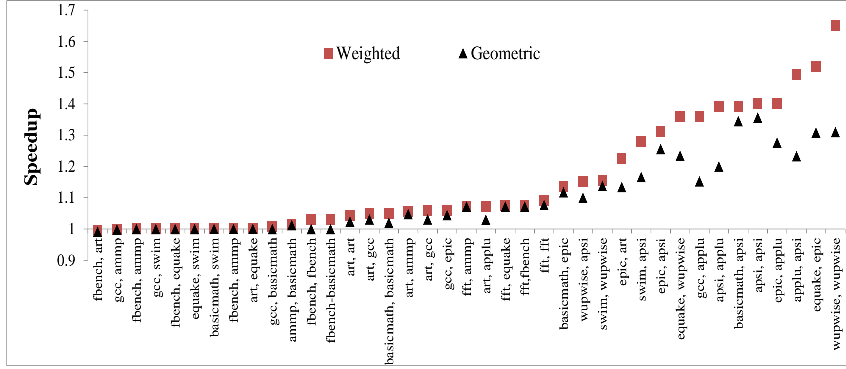


Figure 11. Weighted performance/watt speedup of DCM scheme over heterogeneous baseline for different multiprogrammed workloads.

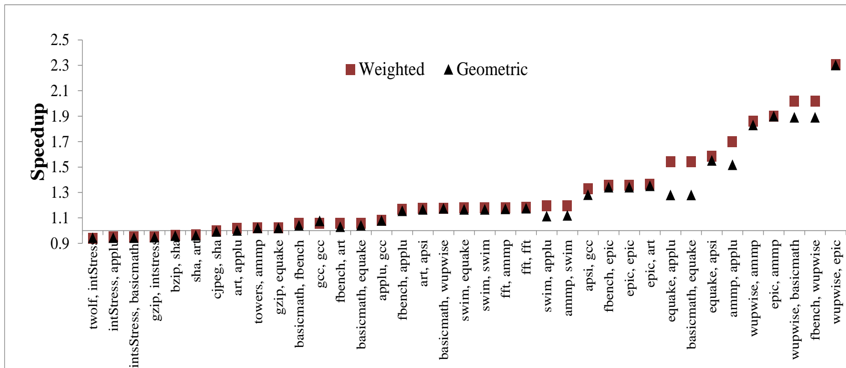


Figure 12. Weighted performance/watt speedup of DCM scheme over Homogeneous Multicore for different multiprogrammed workloads.

produced average (5%-30%) benefits with respect to the weighted speedup. When comparing against the heterogeneous baseline (Figure 11), it can be seen that there are a few combinations (e.g., {*fbench, art*}, {*gcc, ammp*}) where the dynamic scheme does slightly worse than the heterogeneous baseline (about 1% for both weighted and geometric speedup). There are two possible reasons for this: (i) the thread to core assignment is random for the dynamic scheme and no reconfiguration takes place during the runs, (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 number of combinations that benefit from the dynamic scheme is much higher (94% for weighted and 92% for geometric) than those that do not. The results obtained when comparing the dynamic to the homogeneous multicore are shown in Figure 12. It can be seen that the worst case performance/watt degradation is higher here (about 6% for both weighted and geometric metrics). The reasons for the performance/watt loss are: (i) threads running on the multicore have the same nature, i.e., either both are FP or INT intensive, or (ii) mispredictions by the dynamic scheme. Case (i) is due to the fact that for symmetric workloads that show very small program phase changes, SMPs are expected to do better than the AMPs. At the same time, when there are many program phase changes when using symmetric workloads, the dynamic scheme does much better (consider {*fft, fft*} which shows about 27% benefit). Case (ii) is again expected to happen occasionally. Still, overall the dynamic scheme does better than the homogeneous multicore for 96% (93%) of the workloads when using the weighted (geometric) speedup metric.

VIII. Conclusions

We have presented a novel DCM architecture for heterogeneous multicores. We described the implementation of trigger mechanisms for dynamic reconfiguration to maximize performance/watt. The hardware overhead of the proposed technique is negligible at less than 1% of the total area. The proposed heterogeneous core architecture consists of tiles, where each tile features one core with strong support for floating-point operations, and another with strong support for integer code. Reconfiguration mechanisms include thread swapping, and dynamic core morphing at runtime by realigning resources of the given baseline cores to form a strong and a weak core. Using the proposed dynamic reconfiguration scheme, substantial performance and performance/watt gains are achieved over purely static heterogeneous configuration or homogeneous cores (with capability corresponding to the average of the heterogeneous cores). Our results show that the DCM architecture outperforms the static heterogeneous/homogeneous architectures on an average by 16%/43% with respect to weighted speedup and 7%/32% with respect to geometric speedup, respectively.

References

[1] S. Ghiasi and D. Grunwald, “Aide de camp: Asymmetric dual core design for power and energy reduction,” University of Colorado Technical Report CU-CS-964-03, 2003.

[2] P. Salverda and C. Zilles, “Fundamental performance constraints in horizontal fusion of in-order cores,” International Symposium on High Performance Computer Architecture, 2008.

[3] R. Kumar, K. I. Farkas and N. P. Jouppi et al., “Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction,” International Symposium on Microarchitecture, 2003.

[4] R. Kumar, D. M. Tullsen and P. Ranganathan et al., “Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance,” International Symposium on Computer Architecture, 2004.

[5] S. Balakrishnan, R. Rajwar and M. Upton et al., “The Impact of Performance Asymmetry in Emerging Multicore Architectures,” SIGARCH Computer Architecture News, 2005.

[6] E. Grochowski, R. Ronen and J. Shen et al., “Best of Both Latency and Throughput,” International Conference on Computer Design, 2004.

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

[8] E. Ipek, M. Kirman and N. Kirman et al., “Core fusion: accommodating software diversity in chip multiprocessors,” International Symposium on Computer Architecture, 2007.

[9] H. H. Najaf-abadi, N. K. Choudhary and E. Rotenberg, “Core-Selectability in Chip Multiprocessors,” International Conference on Parallel Architectures and Compilation Techniques, 2009.

[10] C. Kim, S. Sethumadhavan and M. S. Govindan et al., “Composable Lightweight Processors,” International Symposium on Microarchitecture, 2007.

[11] R. Kumar, N. P. Jouppi and D. M. Tullsen, “Conjoined-Core Chip Multiprocessing,” International Symposium on Microarchitecture, 2004.

[12] T. Morad, U. C. Weiser and A. Kolodny, “ACCMP - Asymmetric Cluster Chip Multi-Processing,” CCIT Technical Report 488, 2004.

[13] R. Kumar, D. M. Tullsen and N. P. Jouppi, “Core architecture optimization for heterogeneous chip multiprocessors,” International Conference on Parallel Architectures and Compilation Techniques, 2006.

[14] M. R. Guthaus, J. S. Ringenberg and D. Ernst et al., “MiBench: A free, commercially representative embedded benchmark suite,” IEEE International Workshop on Workload Characterization, 2001.

[15] J. Renau et al. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[16] M. Pericas, A. Cristal and F. J. Cazorla et al., “A Flexible Heterogeneous Multi-Core Architecture,” International Conference on Parallel Architecture and Compilation Techniques, 2007.

[17] C. Lee, M. Potkonjak and W.H. Mangione-Smith, “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems,” International Symposium on Microarchitecture, 1997.

[18] M.D. Hill and M. R. Marty, “Amdahl’s Law in the Multicore Era,” Computer, vol.41, no.7, pp.33-38, July 2008.

[19] D. Shelepov, J. C. S. Alcaide, S. Jeffery et al., “HASS: a scheduler for heterogeneous multicore systems,” SIGOPS Operating System Review, 2009.

[20] M. Pericas, A. Cristal and R. Gonzalez et al., “A decoupled kilo-instruction processor,” International Symposium on High Performance Computer Architecture, 2006.

[21] A. Das, R. Rodrigues, I. Koren and S. Kundu, “A Study on the Performance Benefits of Core Morphing in an Asymmetric Multicore Processor,” International Conference on Computer Design, 2010.

[22] D. Brooks, V. Tiwari and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” International Symposium on Computer Architecture, 2000.

[23] P. Shivakumar and N. P. Jouppi, “Cacti 3.0: An integrated cache timing, power and area model,” Technical report, Compaq Western Research Laboratory, 2001.

[24] M. Suleman et al., “ACMP: Balancing Hardware Efficiency and Programmer Efficiency,” Technical Report TR-HPS-2007-001, 2007.

[25] R. Joseph, “Exploring Salvage Techniques for Multi-core Architectures,” HPCRI-2005 Workshop, February 2006.

[26] C. H. V. Berkel, “Multi-core for Mobile Phones,” International Conference on Design, Automation and Test in Europe, 2009.

[27] O. Khan and S. Kundu, “A self-adaptive scheduler for asymmetric multi-cores,” Great Lakes Symposium on VLSI , 2010.

[28] T. Sherwood, S.Sair and B. Calder, “Phase tracking and prediction,” Proceedings of 30th Annual International Symposium on Computer Architecture, pp. 336- 347, 2003.

[29] O. Khan and S. Kundu, “Thread Relocation: A Runtime Architecture for Tolerating Hard Errors in Chip Multiprocessors,” IEEE Transactions on Computers, pp. 651-665, May, 2010.

[30] J. A. Winter, D. H. Albonesi and C. A. Shoemaker, “Scalable thread scheduling and global power management for heterogeneous many-core architectures,” International Conference on Parallel Architectures and Compilation Techniques, 2010.

[31] M. A. Watkins and D. H. Albonesi, “Dynamically managed multithreaded reconfigurable architectures for chip multiprocessors,” International Conference on Parallel Architectures and Compilation Techniques, 2010.

[32] D. Gibson and D. A. Wood, “Forwardflow: a scalable core for power-constrained CMPs,” International Symposium on Computer Architecture, 2010.

[33] T. F. Chen, C. M. Hsu and S. R. Wu, “Flexible heterogeneous multicore architectures for versatile media processing via customized long instruction words,” IEEE Transactions on Circuits and Systems for Video Technology, pp. 659-672, May 2005.

[34] Y. Luo, V. Packirisamy and W. C. Hsu et al., “Energy efficient speculative threads: dynamic thread allocation in Same-ISA heterogeneous multicore systems,” International Conference on Parallel Architectures and Compilation Techniques, 2010.

[35] B. C. Lee and D. Brooks, “Efficiency Trends and Limits from Comprehensive Microarchitectural Adaptivity,” International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII), 2008.

[36] D. Tarjan, M. Boyer and K. Skadron, “Federation: Repurposing scalar cores for out-of-order instruction issue,” Design Automation Conference, pp.772-775, 2008.

[37] R. Teodorescu and J. Torrellas, “Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors,” International Symposium on Computer Architecture, pp.363-374, 2008.

[38] Intel Corporation. From a Few Cores to Many: A Tera-scale Computing Research Overview, Whitepaper, 2006.