Dynamic Thread Scheduling in Asymmetric Multicores to Maximize Performance-per-Watt

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

***Abstract*—Recent trends in technology scaling have enabled the incorporation of multiple processor cores on a single die. Depending on the characteristics of the cores, the multicore may be either symmetric (SMP) or asymmetric (AMP). Several studies have shown that in general, for a given resource and power budget, AMPs are likely to outperform their SMP counterparts. However, due to the heterogeneity in AMPs, scheduling threads is always a challenge. To address the issue of thread scheduling in AMP, we propose a novel dynamic thread scheduling scheme that continuously monitors the current characteristics of the executing threads and determines the best thread to core assignment. The real-time monitoring is done using hardware performance counters that capture several microarchitecture independent characteristics of the threads in order to determine the thread to core affinity. By controlling thread scheduling in hardware, the Operating System (OS) need not be aware of the underlying microarchitecture, significantly simplifying the OS scheduler for an AMP architecture.**

The proposed scheme is compared against a simple Round Robin scheduling and a recently published dynamic thread scheduling technique that allows swapping of threads (between asymmetric cores) at coarse grain time intervals, once every context switch ($\sim$20 ms for the Linux scheduler). The presented results indicate that our proposed scheme is able to achieve, on average, a performance/watt benefit of 10.5% over the previously published dynamic scheduling scheme and about 12.9% over the Round Robin scheme.

***Keywords*-Asymmetric Multicore Processor (AMP), Thread swapping, Instructions per cycle (IPC).**

I. Introduction

The relentless push in technology scaling driven by Moore's law has resulted in more transistors packed into a very small area. This led to improved transistor performance and a significant increase in frequency. However, this device miniaturization has resulted in a larger number of transistors per unit area leading unfortunately to a severe power density problem. Computer architects responded by integrating many cores on the same die [1] and lowering the operating frequency. To keep power density under check, these cores are also kept modest in their capabilities.

In general, multicore processors may be symmetric (SMP) or asymmetric (AMP). An SMP consists of many cores of the same type while in an AMP, the cores may be different from one another with respect to their functionality and/or performance [2]. Recently, a number of studies have shown that for a fixed budget (area or power or both), AMPs are likely to outperform SMPs [3], [4], [5]. The benefits of AMPs are intuitive as it is well known that different workloads have different resource requirements. In addition, within a workload, the flavor of the application may change from time to time resulting in distinguishable phases [6]. The computational resource requirements may often vary significantly from one phase of a given program to the other. In such a case, it may be beneficial to have different cores that cater to the needs of different applications and their phases such that resource utilization and hence performance/watt is maximized.

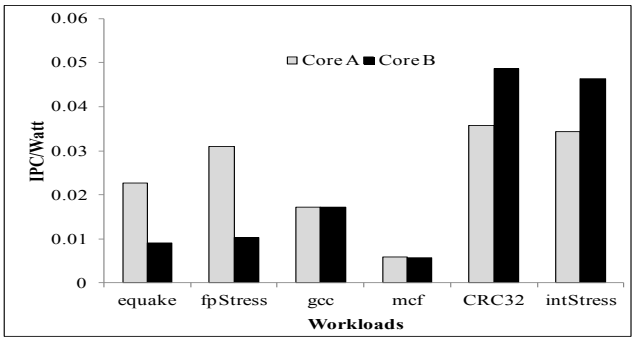

Figure 1. Performance-per-watt achieved for various workloads on two different core types A and B.

Thread scheduling is usually controlled by the OS and commonly used scheduling schemes include FIFO and Round Robin. These schemes work well for SMPs where the capability of all the cores is the same. However, thread scheduling in an AMP remains a challenge [7]. Consider, for example, Figure 1 where the performance/watt of few workloads executed on each of the two cores we consider is plotted. For now, let the two cores be called core A and core B. This figure shows that for some workloads, core A is the better option (e.g., *equake, fpStress*) while for some others core B is better (e.g., *CRC32, intStress*). There are also some workloads for which there is no significant difference in performance/watt achieved by either core (e.g., *gcc, mcf*). Clearly, a correct thread to core assignment can significantly improve the achieved performance/watt. This figure demonstrates that traditional OS thread scheduling techniques may not be optimal for an AMP. This is due to the heterogeneity in computation capability that the cores in

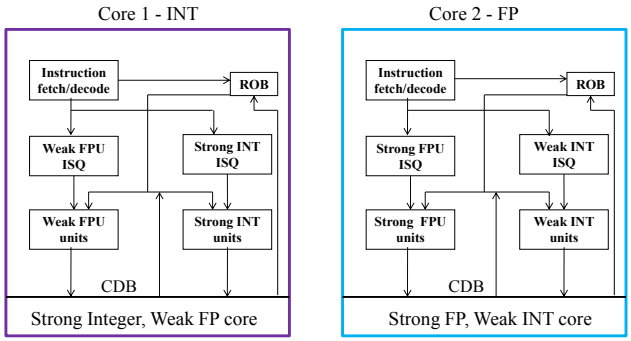

Figure 2. A heterogeneous dual-core system.

the AMP provide. To yield a better scheduling, either the OS needs to be aware of the underlying microarchitecture, which may significantly complicate the OS scheduler, or hardware assisted solutions need to be in place.

To address the dynamic hardware resources to thread matching problem of AMPs, we have recently proposed a low-cost non-invasive hardware mechanism to morph the asymmetric cores to provide better sequential performance whenever necessary [5]. This mechanism estimates the resource requirements of the application running on the core, thus enabling a better thread to core assignment. We have illustrated the benefits of our approach using an example of a dual core system (shown in Figure 2) where one core is designed to better handle integer (INT) instructions while the other core is capable of handling floating-point (FP) instructions efficiently. Whenever the threads running on the cores experience a change in resource demands, the online performance monitors detect it and as per certain rules, determine the optimal thread to core assignment. In this paper, we do not consider morphing and only focus on thread swapping in an effort to avoid the hardware overhead associated with morphing and explore the benefits of a swap only scheme. We elaborate on the proposed technique in section III. The rules that govern the optimal thread to core assignment may be based on either performance or performance/watt metric. We focus on performance/watt in this paper.

We compare our scheme against a recently proposed thread swapping technique [8]. There, the thread to core assignment is done by estimating the performance of a thread that is running on one core, on the other core in the asymmetric multicore. This estimation is done at coarse grain time intervals of once every context switch which is around 20 ms (40 million cycles on a 2GHz processor) for the Linux scheduler. They do this to avoid the penalties associated with thread swapping. However, our results indicate that a more fine grain evaluation of the thread to core assignment may provide higher performance/watt benefits.

The rest of the paper is organized as follows. In Section II, we survey the prior research and in Section III we outline our approach and point out the key differences between this approach and those previously proposed. In Section IV the experiments to determine the core parameters are described. We then describe in detail in Section V the reference dynamic scheme which is used for comparison. In Section VI we discuss our proposed dynamic thread scheduling scheme. Then, results and analysis are presented in Section VII. Section VIII presents our conclusions.

II. Related Work

With AMP architectures getting increasingly popular, there have been a number of proposed thread scheduling techniques for AMPs. Kumar et al. in [3] proposed an AMP consisting of cores of various sizes, all running the same ISA. Whenever a new program is run or a new phase [6] is detected, a sampling is initiated and the core which provides the best power efficiency is chosen. However, this work considered four cores and only a single thread was considered running which simplifies the AMP scheduling problem. They later extended this work for performance maximization of multithreaded applications [9]. A similar approach was described by Becchi et al. [10] in which the AMP consists of two types of cores, one small and one big. The thread to core assignment was determined by initiating a forceful swap between the big and small cores to find the corresponding performance ratio. Depending on the observed ratio, the threads were scheduled such that the overall system performance is maximized. However, such a scheduler is not scalable to an AMP with many different cores. In [11], Annavaram et al. present an AMP where the Energy Per Instruction (while running multithreaded applications) is kept within a budget by having a big core execute the serial portions and small cores execute the parallel portion of the code. In [12], Chen et al. use cores that differ with respect to issue width, branch predictor size and L1 caches. Using their proposed thread scheduling solution, they achieve Energy Delay Product, energy and throughput improvements.

Shelepov et al. in [13] do away with the need for 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 will never be able to take advantage of program phases. Khan et al. in [14] use regression analysis along with phase classification to find thread to core affinity in their considered AMP. In [15], Winter et al. study 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. Saez et al. [16] propose a comprehensive scheduler for AMPs consisting of small and big cores, that targets the performance of both single threaded as well as

multithreaded cases. They define a term called the utility factor which is the ratio of the time it takes to complete the task on the base configuration (small cores) to that on the alternate configuration using the last level cache miss rate information. In [2], Koufaty et al. determine thread to core assignment 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 the number of cache requests going to L2 and main memory) and internal stalls (front end not delivering instructions to the back end). Using the bias they schedule threads in the AMP such that the performance is maximized. In [8], Srinivasan et al. propose a thread to core assignment in an AMP by estimating the performance of the thread, running on one core type, on another using a formula.

III. Proposed Architecture and Differences from Our Previous Work [5]

As mentioned earlier, this work is an extension of our earlier work [5] where we considered two cores: a FP core and an INT core in a dual-core AMP (see Figure 2). 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 load-store queues (LSQ). The sizes of these cores were based on certain core sizing experiments and the details may be found in [5]. In that paper, for parallel operation, the architecture was able to either run in the baseline mode (retaining the current thread to core assignment) or swap threads between the two cores. To achieve higher sequential performance, resources between the two cores were morphed such that the architecture was transformed to a dual-core AMP where one core is strong on all fronts and the other is weak on all fronts. The strong core is the INT core which has taken over the strong FP execution datapath from the FP core and relinquished its own weak FP execution datapath to the FP core, which forms the weak core. We observed significant performance/watt benefits due to core morphing. However, to enable morphing of resources between the cores, special hardware is required to allow the INT core to transform into a strong core. To avoid the added complexity associated with morphing, we explore in this paper the benefits of only thread swapping (between the two cores). The swapping is controlled by performance monitors that record certain characteristics of the workloads being executed on the cores. Whenever these monitors detect an event (as defined by certain rules obtained offline by profiling a subset of workloads), the thread to core assignment can be changed if deemed beneficial.

IV. Determining the core parameters

As the design space for a core is extremely large and includes the sizes of individual structures (e.g., issue queues, reorder buffers and rename registers), we need to determine a set of parameters that would have the largest impact on the two heterogeneous cores. The size of the floating-point and integer core parameters should be chosen such that acceptable performance is achieved for a wide range of applications. We reused the extensive core sizing experiments done in our previous work [5]. The resulting configurations of the INT and FP cores are shown in Table I. The execution latencies for the cores were taken from one of our earlier studies [17]. We used SESC as our architectural simulator [18], and power is measured using Wattch [19] and CACTI [20] with modifications to account for *static power* dissipation.

Table I
Selected Core Configurations

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

Table II
Execution unit specifications for the cores. Latencies taken from [17] (cyc - cycles, 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
	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

For our experiments, we have selected 37 benchmarks: 15 benchmarks from the SPEC suite [21], 14 from the embedded benchmarks in the MiBench suite [22], one benchmark from the mediabench suite [23], and 7 additional synthetic benchmarks. These 37 benchmarks encompass most typical workloads, for example, scientific applications, media encoding/decoding and security applications.

V. The Hardware Monitoring and Prediction Engine Scheme

We compare our dynamic thread scheduling scheme to a recently proposed Hardware Monitoring and Prediction Engine (HPE) based scheduling [8]. By monitoring the performance characteristics (Cycles per Instruction (CPI) and stall cycles) of the application running on the current core, HPE estimates the performance of the application on other asymmetric cores. Based on this, applications that are expected to benefit more from a bigger core will be assigned to that core while those with lower expected benefit are assigned to smaller cores or cores that operate at lower frequencies. However, since the thread assignment decisions are made only every 20 ms in the HPE scheme, it is possible

to miss potential opportunities if applications exhibit phase changes within the 20 ms interval.

The cores considered in the HPE scheme have the same computational flavor, i.e., they are all general-purpose cores. The only difference between the cores is that one core is big (i.e., has faster execution units, bigger cache units, better branch prediction unit etc) or runs at a higher frequency, while the other is small or runs at a lower frequency. Since the cores used in our AMP have very different flavors, with one designed to better handle INT instructions while the other FP ones, the compute and stall scaling factors used in HPE scheme to estimate the performance cannot be applied here. We must therefore, extend the HPE scheme so it will cover cores of different flavors. Furthermore, the original objective of the HPE scheme has been performance using the metric CPI while our scheme's goal is performance/watt.

Extending the HPE to varying flavor cores with performance/watt as the objective is not straightforward and an analytical expression for estimating the performance/watt of a running application on the INT and FP cores is not likely to be derived. A procedure that can be used to do the extension consists of the following steps: (1) Select some application characteristic parameters that are relevant to the different flavors of the asymmetric cores. (2) Profile a certain number of representative benchmarks on the two types of cores, calculate the values of the parameters (selected in step (1)) and find out the corresponding performance/watt for each at fixed time interval of size 20 ms. (3) Construct a matrix that indicates, for the observed values of the selected parameters, the ratio of performance/watt achieved for the two types of cores. Comparing the ratio to 1 indicates by how much will one of the cores outperform the other core. (4) Using regression analysis derive an approximate expression that will be used to estimate the expected performance/watt ratio for any observed values of the application parameters.

Since the two heterogeneous cores in our example dual-core system differ mainly in their integer and floating-point execution capabilities, we use the INT and FP instruction percentages of the applications as the characteristic parameters. In step (2) we selected some representative benchmarks that are INT intensive (e.g., *bitcount, sha, intStress*), some that are FP intensive (e.g., *fpStress, equake, ammp*) and some that have a reasonable mix of both INT and FP instructions (e.g., *apsi, fft, pi*). We then simulated these representative benchmarks on both the cores and collected the values of the INT instruction percentage, FP instruction percentage and IPC/Watt every 20 ms for each of these benchmarks. In step (3), we constructed, using the results collected in step (2), the matrix that indicates the performance/watt ratio for the two cores. This matrix, in principle, can already be used to predict the performance/watt of an application (currently running on one core) on the other core given its INT and FP instruction percentages. However, not all entries in this matrix will contain some value (of the performance/watt ratio) while other entries may contain several values. Due to dependencies and stalls in the applications, even for the same INT and FP instruction percentages, there would be cases where the performance/watt ratio differs. To simplify the matrix and reduce its size we categorized the INT and FP instruction percentages into discrete bins and replaced the multiple values of the performance/watt ratio that corresponded to each bin by the statistical mode of all the values in the bin. Since the time interval between two decision points is high (20 ms), we found the mean and mode of the ratio to follow each other very closely. An example of such a matrix is shown in Figure 3. For example, if a thread currently being executed on the FP core has 80% INT instructions and 20% FP instructions, then its performance/watt on the INT core is estimated to be 1.3 times of its current performance/watt on the FP core.

INT\FP	0% - 20%	>20% - 40%	>40% - 60%	>60% - 80%	>80% - 100%
0% - 20%	0.61	0.51	0.36	0.3	0.18
>20% - 40%	0.78	0.58	0.41	0.33	-
>40% - 60%	1.14	0.6	0.53	-	-
>60% - 80%	1.3	0.63	-	-	-
>80% - 100%	1.6	-	-	-	-

Figure 3. An example performance/watt ratio matrix. Elements of the matrix represent the ratio of IPC/Watt on INT core to IPC/Watt on FP core.

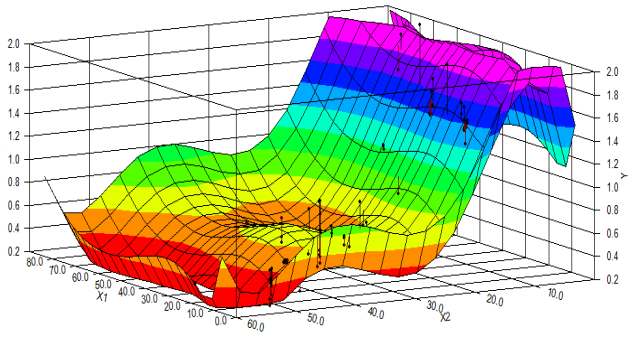

Figure 4. 3-Dimensional plot of the performance/watt ratio expression. (X1 - %INT, X2 - %FP, Y - Ratio of IPC/Watt on INT core to IPC/Watt on the FP core)

As an alternative to the ratio matrix, we can perform a 2-Dimensional curve fitting of all the results obtained in step (2). Using non-linear regression we derived an expression that best fits all the collected results. The resulting non-linear expression is best represented by the 3-Dimensional plot shown in Figure 4. At each decision point, the obtained expression can be used to estimate the performance/watt of the thread on the other core. For both matrix based or curve fitting based approaches, if the estimated speedup of the swapped configuration is more than 1.05 (5% speedup), the threads are allowed to swap. Else, the threads continue their execution in the current configuration.

VI. Dynamic Thread Scheduling scheme

Our proposed dynamic thread scheduling scheme consists of two components: an online monitor and a perfor-

mance 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 swap the threads.

A. Online performance prediction

Prior knowledge about the computational needs of the applications is generally unavailable. Hence, an online mechanism is needed to characterize the time-varying computational resource requirements of the applications. Hardware support is required to detect changes in the application's behavior and then decide whether to swap the threads or not. The key program features that impact the performance/watt are continuously monitored and then used during dynamic thread scheduling. 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) of the threads running on the INT and FP cores which are then used to determine the thread to core mapping. We next describe the process that we have followed in order to make the scheduling decisions based on the instruction composition.

For our experiments, nine benchmarks from the suite of 37 (see Section IV) were chosen such that they included those that are: (i) INT intensive (*bitcount, sha, intStress*), (ii) FP intensive (*fpStress, equake, ammp*), and (iii) have a reasonable mix of INT and FP instructions (*apsi, fffti, pi*). These were the same 9 benchmarks described in Section V that were used to obtain the ratio matrix for the HPE scheme. Threads were run for 500 million instructions on both core types (INT and FP), 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 of the above mentioned benchmarks on both core types, two threads were chosen from the pool and after every window, the thread to core mapping that yields the best IPC/Watt was identified. The instruction distribution of both the threads in each window was also noted. The above experiment was repeated for 50 random combinations of two (out of the 9) threads. Averaging the values of the obtained FP instruction percentages (%FP) and INT instruction percentages (%INT) for all the 50 combinations, we came up with the swapping rules shown in Figure 5.

It can be seen that threads are swapped if the thread currently running on the INT (FP) core experiences a surge in FP (INT) instructions and the other thread on the FP (INT) core experiences a drop in its FP (INT) instructions. This way, our scheme ensures that swapping benefits both the threads. When both the threads have the same flavor of instructions (either INT intensive or FP intensive), it is

Dynamic Thread Scheduling:

1. Threads T_1 and T_2 assigned randomly to cores
2. Do Swap if:
   i. ($\%INT_{FP}$ >= 55) and ($\%INT_{INT}$ <= 35)
   OR
   ii. ($\%FP_{INT}$ >= 20) and ($\%FP_{FP}$ <=7)
3. If *no_swap* for 20 ms, do Swap if:
   i. ($\%INT_{FP}$ >= 55) and ($\%INT_{INT}$ >= 55)
   OR
   ii. ($\%FP_{INT}$ >= 20) and ($\%FP_{FP}$ >= 20)
4. End

- $\%INT_{FP}$: INT instruction percentage of thread on FP core
- $\%INT_{INT}$: INT instruction percentage of thread on INT core
- $\%FP_{INT}$: FP instruction percentage of thread on INT core
- $\%FP_{FP}$: FP instruction percentage of thread on FP core

Figure 5. Swapping conditions for the proposed scheme.

difficult to satisfy the condition mentioned in step (2) of Figure 5. Hence, to reduce the hampering of the thread running on the non-affine core, we force a swap every 20 ms under those scenarios (both threads having the same flavor). This ensures thread fairness for symmetric workloads as both threads are given equal opportunity to use the appropriate computational resources. The 20 ms interval was chosen so as to be consistent with the HPE scheme and the Linux scheduler.

B. Accounting for program phase changes

A tentative decision based on the conditions mentioned in Figure 5 is made at the end of every committed instruction window. However, to avoid too frequent swaps (and its associated overhead) 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 (to either swap or not) during the n most recent instruction windows. The number n of windows based on which the decision is made is referred to as history depth.

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 HPE scheme for the entire program execution. Different window sizes of 500, 1000 and 2000 committed instructions were considered and the history depth n was varied from 5 to 10. For each combination of window size and history depth, about 80 random combinations of two-thread workloads were run, assuming an overhead of 1000 cycles per reconfiguration (as described in the next section). The weighted IPC/Watt improvement for each individual experiment was then averaged to give a single value that represents the entire set as shown in Figure 6. It can be seen that the weighted IPC/Watt improvement is largest (10.5%) for window size of 1000 instructions and history depth of 5. Hence, our scheme will rely on the behavior of the threads during the recently committed 5000 (1000×5) instructions to make the swap decision. However, the obtained IPC/Watt benefit for the selected window size and history depth is

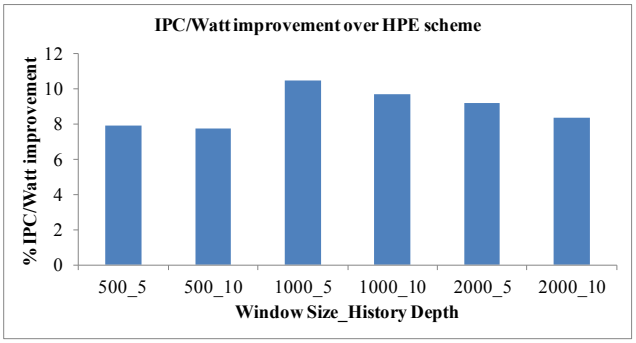

Figure 6. Performance/Watt sensitivity analysis for determining window size and history depth.

greater than the overall average (8.9%) only by 1.6% which clearly shows that small changes in selected window size and history depth will only have a marginal impact on the results.

C. Reconfiguration Overhead

Thread swapping incurs overheads. The overheads stem from flushing the pipelines, exchanging architectural states between the cores and warming the caches. These overheads can vary from one architecture (with no support for swapping) to another (one with ISA support for swapping). Moreover, the cost of swapping could vary significantly depending on whether a shared cache is used for exchanging architectural states or not. Srinivasan et al. have set the core migration overhead to be 450 microseconds [8] which is about 0.9 million cycles for a 2 GHz processor. To show the impact of the overhead on the overall achieved gains, experiments (80 random combinations of two-thread workloads) were run with a swapping overhead of between 1000 cycles to 1 million cycles. We have observed that the average weighted IPC/Watt improvement (arithmetic mean of the weighted IPC/Watt improvement obtained for the 80 random combinations) using our scheme over the HPE scheme, drops by only about 0.9% when a reconfiguration overhead of 1 million cycles was used over 1000 cycles. The results discussed in Section VII are with a swapping overhead of 1000 cycles.

D. Hardware-based Scheduling versus OS Scheduler

The proposed dynamic thread scheduling scheme is a hardware-based solution which is autonomous and isolated from the OS level scheduler which makes it scalable and OS-independent. 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 with the goal of optimizing performance/watt. Our scheme can notify the OS whenever a thread swap happens. As the scheme relies on decision-making at fine-grained time slices, after the commit of every 5000 instructions, there are about 100000 decision-making points for an execution of 500 million instructions. Had we relied on OS for this service, the resulting overhead would be prohibitive. Moreover, based on our experimental results (presented in Section VII), in much less than 1% of the 100000 decision-making points, swapping of threads actually happened.

VII. Evaluation

In this section we present the IPC/Watt improvement achieved using our proposed dynamic thread scheduling scheme for a wide number of multiprogrammed workloads over two other dynamic scheduling schemes – the HPE-based scheduling described in Section V and Round Robin scheduling. Experiments were run with decision intervals of 10 and 20 context-switch periods for Round Robin scheduling similar to [8]. We observed that the Round Robin scheduling with decision intervals of 10×20 ms performs better than with a interval of 20×20 ms. Hence, the results of Round Robin scheduling shown below are with a decision intervals of 200 ms where the applications are swapped between the INT and FP cores every 200 ms. From the pool of all 37 benchmarks, 80 random combinations of two benchmarks were chosen and run on the dual-core until one of the threads completed 500 million instructions.

For the sake of clarity, only 30 combinations (out of the 80) are shown in Figures 7 and 8 which depict the weighted and geometric improvements in IPC/Watt (in percentages) when using the proposed dynamic thread scheduling scheme against HPE and Round Robin scheduling, respectively. The shown 30 combinations include the 10 worse results (out of the 80), the 10 best results and 10 that showed average (8% - 19%) benefits with respect to the weighted IPC/Watt metric. The overall performance/watt of the system could be degraded if one thread benefits at a greater expense of the other. To account for the system fairness, we have employed geometric speedup in our evaluation.

As shown in Figure 7, consistent improvement in IPC/Watt is obtained using the proposed scheme over the HPE scheme where the swapping decisions are made at coarse grain time intervals of 20 ms. The 20 ms fixed time interval used by the HPE scheme may not be optimal for benchmarks that either change phases too frequently or when the observed new phases last only for a short period. These shortcomings are overcome by our proposed scheme where the scheduling decisions are made at fine-grained time intervals (once every 5000 committed instructions). This makes our scheme more efficient as it is able to track and adapt to the program phases closely. The percentage IPC/Watt improvement obtained over the Round Robin scheme is even more significant as it is a static scheme [8]. The employed Round Robin scheme unconditionally swaps the applications between the INT and FP cores every 200 ms. Hence, it is possible that at times, the Round Robin scheme makes unwanted swaps that reduce the performance/watt of the applications.

It could be noted that for few combinations like $\{swim,fpStress\}$, $\{bitcount,ADPCM\ enc.\}$ in Figure 7 and $\{intStress,ADPCM\ dec.\}$, $\{CRC32,bzip2\}$ in Figure 8, the

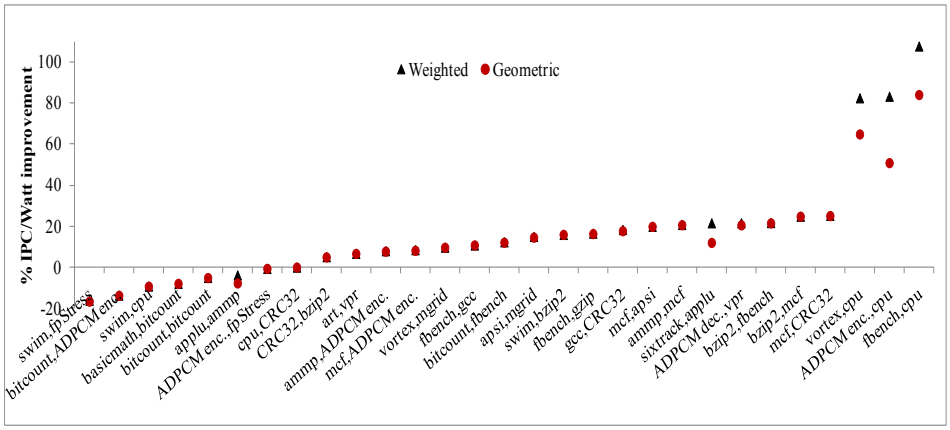

Figure 7. IPC/Watt improvement using the proposed dynamic thread scheduling scheme over HPE scheme for different multiprogrammed workloads.

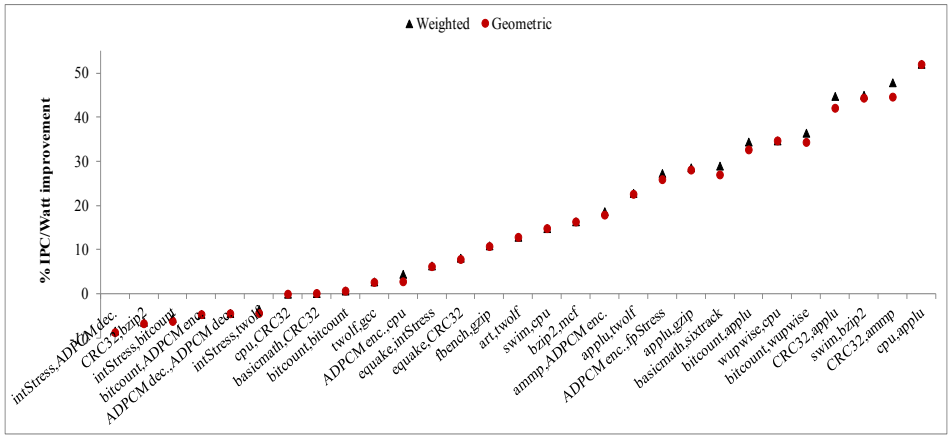

Figure 8. IPC/Watt improvement using the proposed dynamic thread scheduling scheme over Round Robin scheme for different multiprogrammed workloads.

proposed scheme performs slightly worse than the other two schemes. There are few possible reasons for this: (i) At times, fewer swaps happen using the proposed scheme as both the threads do not comply to the swapping rules at the same time. This may not be the case with the HPE scheme where the final swapping decision is driven by the estimated weighted speedup and with the Round Robin scheme where swaps happen unconditionally. (ii) As the swapping decision for the proposed scheme is made purely based on the INT and FP instruction percentages of the threads, it is possible for the scheme to sometimes mispredict. The underlying assumption is that when the INT instruction percentage is high, we expect the thread to perform better on the INT core. However, swapping the thread from FP to INT core may not help when the application experiences dependencies or memory stalls. We plan to improve upon these scenarios by including the performance (IPC) and last-level cache miss rate information into our swapping conditions.

Still, the number of combinations that benefit from our scheduling scheme is much higher than those that do not. Only 7 out of 80 combinations (8.75%) showed minor degradation compared to the HPE scheme while the number reduced further to 7.5% (6 out of 80 combinations) when compared to the Round Robin scheme. The benefit of our proposed dynamic thread scheduling scheme is further

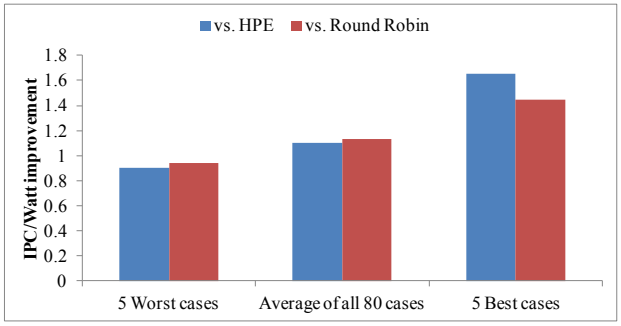

Figure 9. Worst, Average and Best case IPC/Watt improvements obtained using the proposed Dynamic Thread Scheduling scheme.

illustrated in Figure 9 which shows the overall average, worst case and best case IPC/Watt improvements over the HPE and Round Robin schemes.

As can be seen in Figure 9, the mean of the five worst cases contribute to a 10% degradation relative to the HPE scheme and about 6% degradation when compared to Round Robin scheme. However, on an average, for all the 80 combinations put together, we achieve a weighted (geometric) IPC/Watt improvement of about 10.5% (9.1 %) over the HPE scheme and 12.9% (12.4%) over Round Robin scheduling. Moreover, the average weighted IPC/Watt improvement obtained for the five best cases is as high as 65% over the

HPE scheme and 45% over Round Robin scheduling. These IPC/Watt improvements clearly demonstrate the potential benefits our scheme offers.

VIII. Conclusions

We have presented a novel dynamic thread scheduling scheme for AMPs that consist of cores of different flavors. Using simple and low-cost hardware performance counters, we efficiently swap threads between the two cores at fine-grained time intervals to maximize the overall performance/watt. Our results show that the proposed dynamic thread scheduling scheme outperforms the coarse grain HPE scheme / Round Robin scheduling schemes on an average by 10.5% / 12.9% with respect to weighted speedup and by 9.1% / 12.4% with respect to geometric speedup, respectively. The methodology described here for an INT and FP cores can be followed for other types of asymmetric cores that can be designed.

References

[1] J. Held, J. Bautista, and S. Koehl, "From a few cores to many: A tera-scale computing research review," White Paper, 2006.

[2] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, 2010.

[3] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003*.

[4] E. Grochowski, R. Ronen, J. Shen, and H. Wang, "Best of both latency and throughput," in *Proceedings of IEEE International Conference on Computer Design*, 2004.

[5] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, and O. Khan, "Performance Per Watt Benefits of Dynamic Core Morphing in Asymmetric Multicores," in *Proceedings of the 20th International conference on Parallel Architectures and Compilation Techniques*, 2011.

[6] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[7] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proceedings of 32nd International Symposium on Computer Architecture*, 2005.

[8] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer, "Efficient interaction between OS and architecture in heterogeneous platforms," *SIGOPS Oper. Syst. Rev.*, vol. 45, 2011.

[9] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," in *Proceedings of 31st Annual International Symposium on Computer Architecture*, 2004.

[10] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd conference on Computing frontiers*, 2006.

[11] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's law through EPI Throttling," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[12] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Annual Design Automation Conference*, 2009.

[13] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, 2009.

[14] O. Khan and S. Kundu, "A self-adaptive scheduler for asymmetric multi-cores," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, 2010.

[15] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "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*, 2010.

[16] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer systems*, 2010.

[17] A. Das, R. Rodrigues, I. Koren, and S. Kundu, "A study on performance benefits of core morphing in an asymmetric multicore processor," in *IEEE International Conference on Computer Design*, 2010.

[18] J. Renau, "SESC: SuperESCalar simulator," Tech. Rep., http://sesc.sourceforge.net, 2005.

[19] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture*, 2000.

[20] P. Shivakumar, N. P. Jouppi, and P. Shivakumar, "Cacti 3.0: An integrated cache timing, power, and area model," Compaq Western Research Laboratory, Tech. Rep., 2001.

[21] "The Standard Performance Evaluation Corporation (spec cpi2000 suite). http://www.specbench.org/osg/cpu2000."

[22] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, 2001.

[23] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.