2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing

Scalable Thread Scheduling in Asymmetric Multicores for Power Efficiency

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

***Abstract*—The emergence of asymmetric multicore processors (AMPs) has elevated the problem of thread scheduling in such systems. The computing needs of a thread often vary during its execution (phases) and hence, reassigning threads to cores (thread swapping) upon detection of such a change, can significantly improve the AMP's power efficiency. Even though identifying a change in the resource requirements of a workload is straightforward, determining the thread reassignment is a challenge. Traditional online learning schemes rely on sampling to determine the best thread to core in AMPs. However, as the number of cores in the multicore increases, the sampling overhead may be too large. In this paper, we propose a novel technique to dynamically assess the current thread to core assignment and determine whether swapping the threads between the cores will be beneficial and achieve a higher performance/Watt. This decision is based on estimating the expected performance and power of the current program phase on other cores. This estimation is done using the values of selected performance counters in the host core. By estimating the expected performance and power on each core type, informed thread scheduling decisions can be made while avoiding the overhead associated with sampling. We illustrate our approach using an 8-core high-performance/low-power AMP and show the performance/Watt benefits of the proposed dynamic thread scheduling technique. We compare our proposed scheme against previously published schemes based on online learning and two schemes based on the use of an oracle, one static and the other dynamic. Our results show that significant performance/Watt gains can be achieved through informed thread scheduling decisions in AMPs.**

I. Introduction

Power density concerns in processor ICs led to the multicore era [1] where a single and very powerful processor has been replaced by several smaller cores with more modest computational capabilities. As long as the cores are identical, incoming program threads can be assigned to cores arbitrarily by the Operating System (OS). However, for a given power budget, Symmetric Multicore Processors (SMPs) have been shown to be outperformed by Asymmetric Multicore Processors (AMPs) that can cater to the needs of diverse workloads [2]–[4]. This however, is highly dependent on the way threads are assigned to the individual asymmetric cores and a non-optimal assignment may nullify the expected benefits of an AMP.

There have been a number of proposed thread scheduling schemes for AMPs [5]–[7]. Some of them rely on offline workload profiling [7], [8], while others rely on online learning via program sampling [9], [10]. Offline profiling is not a practical approach in general, since it relies on the availability of prior knowledge regarding all the applications that may execute on the AMP. An alternative to this approach is to learn program performance online via sampling. Here, whenever a new program phase is detected, the system is halted and the newly detected phase is sampled on each core type in the system. The information obtained is then used to reassess the thread to core assignment in the multicore. Thus, unlike offline profiling schemes, this approach does not require prior information about the workloads that will be run on the multicore. On the other hand, the number of samplings required increases with the number of core types in the AMP. Hence, this scheme will not scale well with the number of cores. Therefore, there is a need for a technique that will both scale with the number of cores in the AMP and will not rely on an oracular knowledge of the workloads.

In this paper, we propose a novel technique to schedule threads in an AMP such that performance/Watt is maximized. The key idea is the online estimation of both the performance and power of an application on all the other cores in the AMP, while it is being executed on the current core. This is made possible by using the performance counters of the current core. A relationship is established between the values of these counters in the core executing the application and the expected performance and power of this application if it would run on the other cores in the AMP. By estimating the performance and power on other core types, informed thread scheduling decisions can be made without any of the drawbacks of offline profiling and online learning. To illustrate our approach, we consider an 8-core AMP comprising of two high performance cores (HPerf core) with similar characteristics to an Intel Nehalem or AMD K10 processor, and six low power cores (LP core) similar to an Intel Atom or AMD Bobcat. This choice is in line with recent studies [5], [6], [9]. We present an extensive analysis to determine which hardware performance counters (HPCs) should be used to predict both performance and power. We then formulate expressions using the selected counters for estimating the performance and power on other cores in the AMP. These expressions are used to make real-time thread scheduling decisions in the AMP when dual threaded workloads are run. The proposed scheme is compared against the static baseline AMP (the same dual core type AMP with no thread swapping capability) with oracular knowledge of the best thread to core mapping and a previously proposed online learning scheme [9]. We also compare the proposed scheme to a greedy oracle scheduler. Our results indicate that the proposed scheme achieves significant performance/Watt improvements over all the baselines. In particular, on an

1550-6533/12 $26.00 © 2012 IEEE
DOI 10.1109/SBAC-PAD.2012.40

average, 2X gains are observed when comparing the proposed scheme to that based on online learning.

II. Related Work

With AMPs becoming more common, a number of thread scheduling techniques have been recently proposed. We briefly overview the prior schemes which can be broadly classified into those that employ offline profiling, online learning via sampling and online estimation.

There has been a number of solutions based on offline profiling to determine the best thread to core scheduling in AMPs. Khan *et al.* [11] propose regression analysis along with phase classification to identify thread to core affinity. Shelepov *et al.* [7] profile applications to determine what they call architectural signature of the application. This signature (characterized by L2 cache misses) is unique for each core type and is used to determine the thread scheduling online. In [12], Chen *et al.* use cores in an AMP that differ with respect to issue width, branch predictor size and L1 caches. They use multi-dimensional curve fitting to determine the optimal thread to core assignment offline. All these approaches rely on offline profiling and are not practical, since they require knowledge of the workloads that will be run on the multicore.

Online learning based schemes offer a more practical solution to the AMP scheduling problem. Kumar *et al.* [10] proposed an AMP consisting of cores of various sizes, all running the same ISA. Whenever a new program is run or a new phase [13] is detected, sampling is initiated and the core which provides the best power efficiency is chosen. A similar approach was proposed by Becchi *et al.* [9] for performance maximization of an AMP consisting of two types of cores. Optimal thread scheduling was determined by forcing a thread swap between cores upon detection of phase change. Winter *et al.* [4] study power management techniques in AMPs via thread scheduling. They consider several algorithms, all based on program sampling. Even though these schemes are a practical alternative to the offline profiling based schemes, it is clear that with an increase in the number of cores in the system, the number of samples for each phase detected will be large and hence these schemes will experience a high overhead.

Online estimation based schemes are an improvement over the online learning schemes since they avoid sampling and the resulting overhead. Here, based on the current characteristics of a workload being executed, its performance on other core types of the system is estimated. However, the benefits of the scheme will be determined by the accuracy of the estimation. Saez *et al.* [6] propose a comprehensive scheduler for AMPs consisting of small and big cores using last level miss rates of an application to estimate its performance on each core type. It is, however, unclear whether using L2 misses alone is sufficient to make thread to core assignment decisions such that performance/Watt is optimized. The work closest to ours is that proposed by Srinivasan *et al.* [14], [15] and Koufaty *et al.* [5]. In [14], Srinivasan *et al.* estimate the performance of the thread currently running on one core type, on another core, using a closed form expression. These expressions were developed for specific cores and a general approach was not provided. Koufaty *et al.* [5] determine thread to core mapping in an AMP consisting of big and small cores, using program to core bias which is estimated online using the number of external stalls (proportional to cache requests going to L2 and main memory) and internal stalls (front end not delivering instructions to the back end). In both of these works, the objective is performance and not performance/Watt. Extending the above techniques to include power estimation is not straightforward. Rodrigues *et al.* [8] considered thread scheduling in an AMP by using predetermined rules but the cores that they consider are very specific and the extension to other types of cores is unclear.

Of the currently available scheduling schemes, the estimation-based ones offer the most practical and scalable solution, but they mostly focus on performance and not performance/Watt. In this paper, we propose an estimation based scheme for performance/Watt. The objective is to estimate not only performance but also the power of other cores in the AMP using counters in the host core.

III. Methodology

To evaluate our approach (detailed in the next two sections), we selected an 8-core AMP consisting of two core types at the two ends of the performance/power spectrum - a high-performance core (HPerf) and a low-power core (LP). This is one of the worst cases for a scheme for estimating the performance and power of the second core based on the activities observed in the first core. In the considered 8-core AMP, two cores are HPerf cores and 6 are LP cores. The list of core parameters and execution latencies used for both the core types are shown in Tables 1 and 2, respectively. Most of the core parameters and latencies were taken from [16]. It can be seen from Table 1 that the two cores are significantly different. We used SESC as our architectural performance simulator [17] and employed CACTI [18] and Wattch [19] to calculate power with modifications to account for static power. We are aware that Wattch has an error percentage of within 10% when compared to layout-level power estimation tools. Our focus is on estimating instantaneous power and we are mainly interested in detecting changes in the power profile (which may trigger dynamic thread re-scheduling). Hence, comparison of the estimated power (by using different counters) to the power calculated by Wattch is satisfactory. For our experiments, we have selected 38 benchmarks: 16 benchmarks from the SPEC suite [20], 14 from the embedded benchmarks in the MiBench suite [21], one benchmark from the Mediabench suite [22], and 7 additional synthetic benchmarks. These 38 benchmarks encompass most typical workloads, for example, scientific applications, media encoding and decoding and security applications. The instruction distribution of each of the considered workload is plotted in Figure 1.

IV. Performance/Watt analysis of the two core types

The two core types that comprise our AMP have very different characteristics with one designed for high performance,

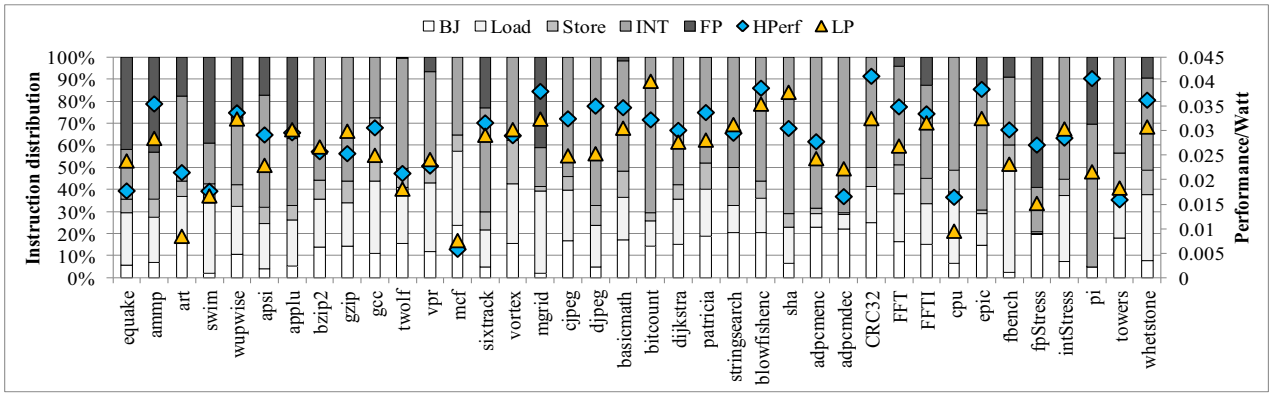

Fig. 1. Instruction distribution and IPC/Watt for the 38 benchmarks considered when run on each core type for 1 billion instructions.

TABLE I
CHOSEN CORE PARAMETERS

Param	LP	HPerf	Param	LP	HPerf
Issue	2	6	INTREG	64	96
FPREG	64	80	INTISQ	NA	36
FPISQ	NA	24	LS units	1	3
LSQ	NA	32	ROB	NA	128
L1(I/D)	32K	32K	L2	512K	2M
Freq (GHz)	2.4	2.4	Type	In-order	OOO

TABLE II
EXECUTION UNIT SPECIFICATIONS FOR THE CORES. (P - PIPELINED, NP - NOT PIPELINED, PP - PARTIALLY PIPELINED)

Core	FP DIV	FP MUL	FP ALU
LP	1 unit, 60 cyc, NP	1 unit, 4 cyc, PP	1 unit, 5 cyc, P
HPerf	1 unit, 21 cyc, P	1 unit, 5 cyc, P	2 units, 3 cyc, P
	INT DIV	INT MUL	INT ALU
LP	1 unit, 207 cyc, NP	1 unit, 10 cyc, P	2 unit, 1 cyc, P
HPerf	1 unit, 23 cyc, P	1 unit, 8 cyc, P	8 units, 1 cyc, P

while the other for low power. To quantify the difference in the capabilities of the cores, we ran all the 38 benchmarks on both the core types (LP and HPerf cores) for 1 billion instructions, after skipping the initial 5 billion that include the program initialization. The performance/Watt results are shown in Figure 1. It can be seen that for some workloads, the HPerf core performs better than the LP core (*ammp, CRC32, pi*) while it is vice-versa for certain other workloads (*equake, bitcount, sha*). The performance per watt is a function of the resource utilization. Efficient resource utilization leads to better figures. In general, for benchmarks which are branch or memory intensive, HPerf core resource utilization is not optimal and hence the performance per watt is lower than that of the LP core. Clearly, for eight threaded workloads, a correct thread to core scheduling will yield significant benefits, while an incorrect one, will have a much lower performance/Watt.

Figure 1 depicts the average behavior over 1 billion instructions and as such only indicates the achieved IPC/Watt due to a fixed thread to core assignment. Many programs exhibit phases with varying computational demands and each core in the AMP may be beneficial for different phases during the program execution. A dynamic thread to core assignment will be able to adapt to the time-dependent program behavior.

V. DYNAMIC THREAD SCHEDULING

Determining the affinity of a program phase to a core in the AMP is crucial for establishing a dynamic thread scheduling scheme. Since prior knowledge about the computational needs of the different workload phases is generally unavailable, there is a need to determine them online. Moreover, the dynamic thread scheduling scheme should consider reassignment of a thread only when that thread has moved to a new and stable phase otherwise the scheme's overhead will become prohibitive. Even before determining the affinity of a phase to a core, there is a need to detect and successfully classify stable phases of execution in a program. Only stable phases should be considered since short-lived (unstable) phases do not justify thread reassignment. We present the scheme that we adopt for phase classification next. This is followed by the online mechanism used to determine the program phase to core affinity.

A. Phase classification mechanism

A number of schemes have been proposed for phase classification, e.g., [23], [24]. We adopt here (after some modifications) the Instruction Type Vector (ITV) scheme presented by Khan *et al.* [11] due to its simplicity, but other phase classification schemes can also be used. ITVs are created using hardware counters that count the number of committed instructions of certain types (9 types were used in [11]) during a specified interval. This interval corresponds to a fixed number n of committed instructions with the value of n to be determined. Whenever an instruction is retired, the appropriate instruction counter is incremented. After n instructions have committed, the resulting 9-element vector is captured and compared to the ITV of the previously identified phase. If the difference between the two (measured as sum of differences between the instruction types of the currently executing and previously encountered phase) is greater than a threshold, Δ (another parameter that needs to be determined), then this is potentially a new phase. If the difference is less than the threshold, then the same phase has repeated. A newly detected phase is classified as stable when at least m consecutive intervals (of n committed instructions each) had a difference of less than Δ. The number m is another parameter of the scheme that needs to be determined. Khan uses a phase table to store

all previously detected stable phases. Since the scheme that we propose does not need to store prior phase information, we do not include a phase table in our implementation. Moreover, since we consider only LP and HPerf cores, we reduced the ITV from 9 to 4 components, corresponding to integer, floating-point, memory and branch instructions. Additional details of the algorithm to detect and classify phases can be found in [24]. Khan determined the parameters of the phase classification, namely n, m, and Δ by experimentation. Since the benchmarks that we consider are different from those in [24], and the components of the ITV are different, we have redone these experiments to determine the three phase classification parameters. After extensive experiments, we have set the phase classification parameters to (i) interval length n = 100K instructions, (ii) threshold Δ = 7.5% and, (iii) m = 4, based on the phase classification quality metrics defined in [24].

B. Determining program affinity to a core online

Once a phase classification mechanism is in place, we need to identify the affinity of the current phase to the different cores in the AMP. The objective here is to non-invasively predict program performance on other core types without the drawbacks of online learning based on sampling. Hardware performance counters (HPCs) reveal information about the characteristics of the thread currently being executed. We therefore, decided to develop a scheme to predict power and performance of an executing application on the host core, as well as other cores in the AMP using HPCs. Our scheme is described in detail in the next section.

C. Using performance counters to determine thread to core affinity

Hardware performance monitoring counters (HPCs) reveal considerable amount of information about the performance and power consumption of a thread [25], [26]. Most prior research dealing with such estimations use HPCs to predict these characteristics on the same core and not on another core in the AMP. To make thread to core assignment decisions, there is a need to estimate the performance and power of the thread on the host core as well as on the potential core where it may be executed. Performance on the host core can be directly collected from the *IPC* counter, but there is a need to estimate the power on the host core, as well as the expected performance and power of the thread if it would be executed on the other cores. Thus, we need to identify a set of counters that will enable prediction of power on the host core as well as performance and power on the other cores. Our objective is to shortlist potential counters with the most impact.

The performance counters studied by us can be grouped as follows:

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

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

• **Miss/Hit counters**: Cache hits and misses play a significant role in performance or power consumption of a core. In this regard, the following event counters: *L1 hit, L1 miss, L2 hit, L2 miss, page hit* and, *TLB miss* are considered.

• **Retired instructions counters**: Performance/power consumption can vary significantly depending on the type of the retired instructions (INT, FP, Memory, Branch). Hence we considered retired instructions counters.

• **Stalls**: The activity of the processor will be low when it experiences dependencies (data or resource conflicts) frequently. We consider stalls due to reservation stations, re-order buffer (ROB), load/store queues (LSQ), register renaming and RAT (Register Alias Table). We refer to this counter as *Dispatch Stalls*.

1) Performance / Power Modeling: To shortlist the most influential performance counters, we used correlation between the counters and the metric that is to be estimated. Estimating power on the same core is not difficult and has been done in prior publications using 3 to 4 counters [25], [26]. In contrast, estimation of the metrics on the other core is not straightforward. Our objective is to use the least number of counters to predict all the required metrics. The reason behind this is not just to save hardware, but also to reduce the number of counters that have to be monitored simultaneously. In current processors, the same counters are used for monitoring multiple events and it is not possible to simultaneously obtain the count for two different events from the same hardware counter [25]. We searched for counters that showed high correlation to power and performance of the other core. Since we are interested in swapping threads (between the LP and HPerf cores) at runtime, we need to estimate the performance/Watt of a thread currently running on LP core, on HPerf core and vice-versa. To this end, we need to analyze, offline, the correlation between the performance counters of the LP (HPerf) core to the power and performance of the thread if it would execute on the HPerf (LP) core. To accomplish this, we identified eight representative benchmarks from the set of 38, such that they included: INT intensive (*intStress,bitcount*), FP intensive (*fpStress,equake*), load/store intensive (*gcc*), have high IPC (*apsi*) and low IPC (*mcf,ammp*). The 8 benchmarks were run on both the cores (LP and HPerf) for 1 billion instructions and the value of the above mentioned performance counters for both the cores were sampled periodically after the commit of every 100K instructions (equal to interval length n described earlier). All the counter values obtained were normalized with respect to the number of cycles elapsed during that period. We then computed the correlation between the normalized counter values of one core and the observed power and performance on the other core, and the results are plotted in Figures 2 and 3.

As can be seen from the figures, the observed correlation to both IPC and power is not very high as the counters used to estimate the performance and power are in the other core. From the initial set of 15 counters, we shortlisted *L2 miss, TLB miss, # Fetched instructions, IPC, Power, retired INT, L1 hit*

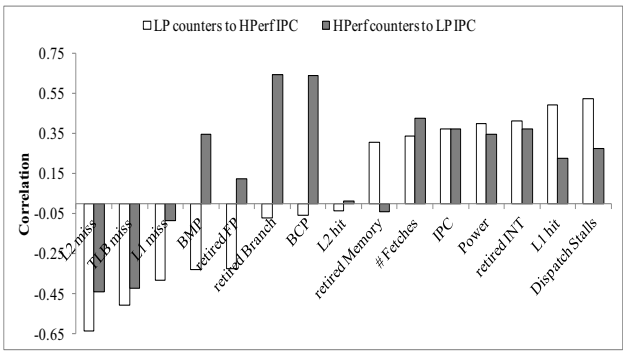

Fig. 2. Correlation of various performance counters in one core to the observed IPC on the other core.

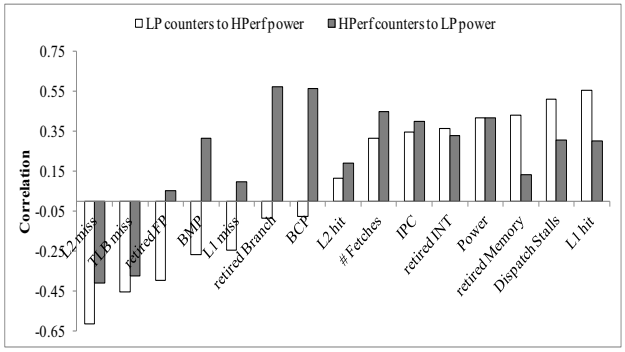

Fig. 3. Correlation of various performance counters in one core to the power consumed by the other core.

and *Dispatch Stalls* as they showed reasonable correlation to both IPC and power on the other core. To reduce the number of performance counters that are involved in the estimated IPC and power expressions for the other core, we investigated the correlation of each of the above selected parameters to the rest. The one which correlates well with many other parameters could be used as a proxy for the rest. We found the *# Fetched instructions* to have a high correlation to power, while *IPC* of the current core correlated well with *retired INT* and *L1 hit* counters. Therefore, based on this observation, we chose *L2 miss, TLB miss, # Fetched instructions, IPC* and *Dispatch Stalls* as the main performance counters to be used in our estimation scheme. Having the same set of counters for both the metrics (performance and power on the other core) and for both the core types (LP and HPerf) greatly simplifies the estimation mechanism.

We then used the traces obtained from the 8 selected benchmarks to express the observed performance and power on the other core as a function of the chosen performance counters in the current core. A multi-dimensional curve fitting and regression analysis was performed to obtain expressions for the estimated performance and power for both the core types and these are shown in Table III. A similar procedure was followed to estimate power on the host core using its own counters. We observed that the same set of counters, selected for estimating metrics on the other core, shows a reasonably high correlation to the observed power on the host core too (figure not included due to space constraints). The expression

TABLE III
POWER AND PERFORMANCE ESTIMATION OF THE OTHER CORE USING THE PERFORMANCE COUNTERS OF THE CURRENT CORE. *L2m - L2 miss, TLBm - TLB miss, S - Dispatch Stalls, F - # Fetched instructions*

Estimating Parameter	Expression
LP IPC	exp(-41.8 × L2m - 30.2 × TLBm - 3.4 × S + 6.5 × IPC - 2.9 × F + 1.44)
HPerf IPC	exp(-389.8 × L2m - 19.6 × TLBm + 3.9 × S + 20.3 × IPC - 22 × F - 3.6)
LP Power	exp(-1.5 × L2m - 2.2 × TLBm - 0.6 × S + 1.2 × IPC - 0.5 × F + 2.9)
HPerf Power	exp(-126.5 × L2m - 4.7 × TLBm + 3.9 × S + 4.2 × IPC - 6.2 × F - 0.4)

TABLE IV
ONLINE POWER ESTIMATION FOR THE HOST CORE USING ITS OWN PERFORMANCE COUNTERS. *L2m - L2 miss, TLBm - TLB miss, S - Dispatch Stalls, F - # Fetched instructions*

Estimating Parameter	Expression
LP Power	exp(1.3 × L2m + 1.5 × TLBm + 0.5 × S + 0.5 × IPC + 0.03 × F + 1.7)
HPerf Power	exp(-0.48 × L2m + 4.6 × TLBm - 0.35 × S + 1.3 × IPC - 0.5 × F + 3.3)

obtained for the online power estimation for the considered dual-core type AMP is shown in Table IV.

The accuracy of the expressions obtained was then measured for all 38 workloads. Counter values from the HPerf core were used to estimate its own power as well as the performance and power of the LP core and vice versa. We observed that on an average, the derived expressions estimated power on the host core with a 6.5% error, and IPC and power on the other core with an error of 32% and 9%, respectively. The resulting IPC/Watt average estimation error for the host core was 8%, and was 34.2% for the other core. Even though the errors in estimating metrics for the other core are quite high, they proved to be adequate for our purpose of making online thread scheduling decisions. A high estimation error is not important if the right thread to core assignment is made most of the time. We found in our experiments that the proposed estimation based scheme made the right thread scheduling decision 92% of the time, which is acceptable. As will be seen in Section VI, the 8% erroneous decisions do not have a significant effect on the benefits of the proposed scheme.

D. The complete thread scheduling framework

Having a phase classification mechanism and a scheme to approximately estimate the power and performance of the thread on other cores, we still need a way to govern these two autonomous mechanisms and decide on thread reassignments. The task of managing the phase classification mechanism and the collection of data from the selected performance counters is assumed to be handled by a software layer called the Microvisor. A similar layer has been used by Khan [24] and was previously developed by IBM [27]. Additional details on this software layer may be found in those papers. We now describe the working of the entire system, as managed by Microvisor.

The flowchart of the procedure followed in the proposed scheme is shown in Figure 4. Eight workloads are run on the dual-core type AMP consisting of six LP and two HPerf

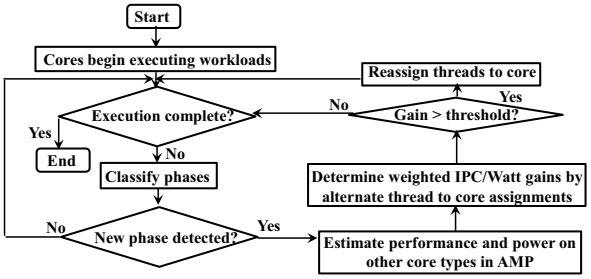

Fig. 4. The thread scheduling flowchart.

cores. Whenever a phase change is detected for any one of the threads by our phase detection mechanism, the power on the host core as well as the power and performance of the thread if executed on the other core are estimated by Microvisor, based on the chosen performance counters (*L2 miss, TLB miss, IPC* and *# Fetched instructions*) of the host core. The performance and power of the other core type running other threads are also collected. The performance/Watt is then calculated for the current and the alternate thread to core assignment. Based on this, the current thread to core assignment may be changed.

The number of potential thread to core assignments to assess increases with the number of simultaneous phase changes for the various workloads. For a single phase change, when the thread on the LP core changes phase, there are two potential threads that it may swap with, i.e. the two threads on the HPerf cores. Similarly, for a phase change in a thread being executed on the HPerf core, there are six threads that it may swap with. Hence, for single phase changes, there are up to six combinations that have to be assessed. We found in our experiments that 92% of the time only a single phase change is detected and the maximum number of simultaneous phase changes detected was 3 (0.2% of the time). Hence, the number of combinations to assess was far lower than the worst case of 8 simultaneous phase changes. Using the estimated performance/Watt of the various threads in an alternate configuration, the weighted performance/Watt improvement (geometric or harmonic speedups may also be used) projected for the new thread to core assignment over the current one is calculated. If the weighted speedup is over 3% (called decision threshold; detailed study was conducted to set this value), the threads are swapped between the two cores. If not, the current thread to core assignment is maintained. Swapping threads between cores incurs an overhead due to context switch and cold cache misses. Rodrigues *et al.* [8] have estimated this overhead to be 400 cycles. We assume, conservatively, a swapping overhead of 1K cycles. We observed the system to be not very sensitive to this overhead. Another source of overhead is the invocation of the Microvisor. This was observed to be invoked, on an average, 700 times per run, but this overhead is relatively small as it involves collection of counter statistics and evaluation of the expression. This can be assumed to be at most a few hundred cycles and we found this to have negligible effect on the results. By using phase classification, the proposed scheme needs to make decisions only when stable phases are detected, which is not very often. Hence, the overheads associated with decision making are kept at bay. The proposed scheme is evaluated next and compared against various baselines.

VI. Evaluation

In this section, we report the results of our evaluation experiments. Multi-programmed workloads were run on the AMP until one of the threads executed 1 billion instructions. The phase classification parameters were set to: Interval n = 100K, Δ = 7.5% and stable phase interval m = 4.

We now describe the baselines that will be used for comparison. The performance/Watt improvement achieved by the proposed scheme over each of the baselines is then presented.

A. Baseline configurations considered

We compare our proposed scheme to the following baseline configurations:

• **Static**: Here the thread to core assignment is *static*, i.e., it never changes. This fixed assignment is based on oracular knowledge of the best assignment over the entire run of the workloads and as such is not practical.

• **Online learning-based (*O_Learning*) swapping scheme with sampling overheads**: Threads are dynamically swapped between the cores in this scheme. Detection of phases (based on the ITV scheme) is used as a trigger to initiate a possible swap and the learning is done by sampling the newly detected phase on the other core type of the AMP. This baseline constitutes a modified version of the scheme proposed by Becchi et al [9]. Sampling incurs an overhead and it is assumed to be 1M cycles [9]. Thread swapping overheads are also considered here.

• **Greedy oracle (*G_Oracle*)**: This baseline is capable of swapping threads between the cores. The trigger is once again phase detection, but the thread to core decisions are made based on oracular knowledge at that instant in time, regarding the best current reassignment of threads to cores. No learning overheads are considered for this baseline but thread swapping overheads are taken into account.

B. Performance per watt analysis over the baselines

We considered three speedup metrics to compare our proposed scheme to the baselines. We first define the following terms:

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

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

The various speedups considered are:

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

From the set of 38 workloads, we randomly selected 100 combinations of eight threaded workloads and had them executed using the proposed as well as each of the baseline schemes. We have plotted a subset (30 of the 100) of those results for various baselines in Figures 5, 6 and 7 for the *Static*, *O_Learning* and *G_Oracle* baselines. The shown 30 combinations include the 10 worst results (out of the 100), the 10 best results and 10 that showed average benefits with

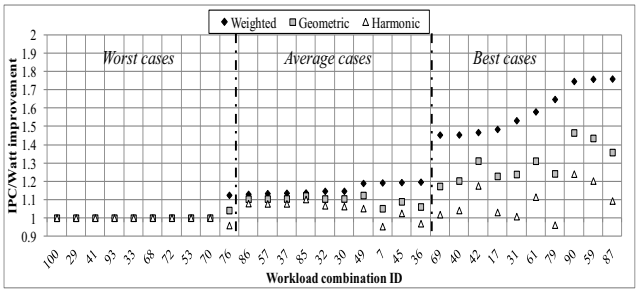

Fig. 5. IPC/Watt improvement of the proposed scheme against the Static baseline.

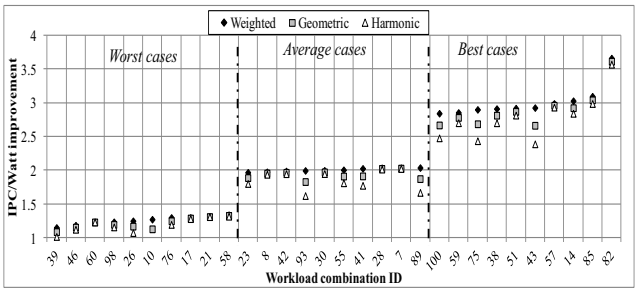

Fig. 6. IPC/Watt improvement of the proposed scheme against the O_Learning baseline.

respect to the weighted IPC/Watt metric. It is clear that in general, considerable IPC/Watt improvement is achieved over the *Static* baseline and in particular, the *O_Learning* baseline, where speedup of up to 3.5X is observed. Amongst the worst cases, it can be seen that an IPC/Watt degradation is observed when comparing against the static baseline (0.99). However, when comparing to the *O_Learning*, even the worst case speedup is 1.14 which shows that the overhead of sampling negates the benefits of the learning-based approach. When compared to the *G_Oracle* baseline, barring a few rare cases, there are no notable gains, as expected. We have also plotted

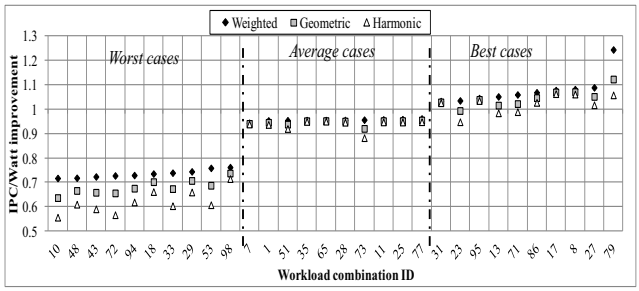

Fig. 7. IPC/Watt improvement of the proposed scheme against the G_Oracle baseline.

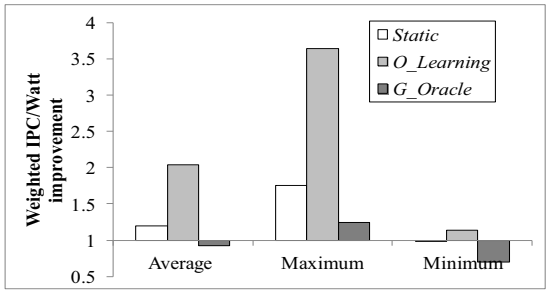

Fig. 8. Speedup of the proposed scheme against the Static, O_Learning and the G_Oracle schemes.

the average, minimum and maximum weighted IPC/Watt gains that the proposed scheme achieves over the baselines in Figure 8. It can be seen that on an average, the proposed scheme performs around 20% better than the *Static* baseline with respect to weighted improvement, but what is more noteworthy is that the gain is 200% when compared to the *O_Learning* scheme. The reason for this is the overhead due to sampling (discussed in detail in sub-section VI-B1). It can also be seen that the proposed scheme comes to within 92% of what the *G_Oracle* scheme achieves with respect to average weighted gains, which is very encouraging. We provide detailed analysis on these results next.

1) Analysis of results:

a) Static: In this baseline, the thread to core assignment is kept the same throughout the execution. This thread to core assignment is based on an oracle and as such, cannot be done in practice. Still, it can be seen that significant IPC/Watt improvement is achieved by the proposed scheme over this baseline (Figure 5). This baseline never takes advantage of phase changes or changes in resource demands. Even if over the entire run, a thread has an affinity for a certain core, there may be periods where this thread would be more affine to another core in the AMP. Hence, the proposed scheme achieves significant improvement in IPC/Watt over the *Static* baseline. Still, there are a few workload combinations where the *Static* baseline performs better. This is mainly due to the mispredictions made by the proposed scheme and the fact that some workloads do not experience many phase changes. However, looking at the average, it is clear that there are only a few mispredictions. The overall benefits (20% on average for weighted gains) more than justify the losses due to mispredictions.

b) O_Learning: This baseline is dynamic and whenever deemed beneficial, the threads are swapped between cores. The decision to trigger swapping is determined by the same mechanism that is used by the proposed scheme, i.e., phases detected by the phase classification mechanism. Every time a phase change is detected, this scheme initiates an online sampling mechanism. Hence, this scheme is expected to predict thread to core reassignment more accurately than the proposed scheme. However, as mentioned earlier, it suffers from a learning overhead. We found that on an average, there are approximately 700 such events, significantly increasing the overhead of this baseline. This is the reason why the benefits of the proposed scheme over this scheme are higher than even what was obtained against the *Static* scheme (see Figure 5 and 6, and Figure 8). We did not find any case where this scheme performed better than the proposed scheme which is mainly due to the overheads involved during sampling. As the number of core types and workloads increase in the system, the number of phase changes and the number of sampling intervals increase significantly, which nullifies any benefits of this scheme. When ignoring the learning overhead, this scheme performs better than the proposed scheme by 5% on average, due to its more accurate predictions. This shows that even though the proposed scheme is slightly inaccurate in its

decision making, the decisions it makes are good enough and they do not incur any learning overheads. These results show that the proposed scheme is a more practical and scalable when compared to the sampling based learning scheme.

c) G_Oracle: This baseline also has the ability to swap threads between the cores but makes swapping decisions based on oracular knowledge. From Figures 7 and 8, it can be seen that in general, the proposed scheme performs worse than this baseline. This is expected, as this baseline makes perfect thread to core reassignments without incurring any overheads, which is not practical. What is interesting is that the proposed scheme does better than this oracular scheme in a few rare cases. The reason for this is that sometimes by taking a wrong decision (as is done by the proposed scheme), the opportunities that come up later, as compared to the case where always the right (greedy) decision is made, are different. Sometimes, these additional opportunities may provide even better benefits. Still, on an average, the proposed scheme performs worse than this scheme by 8%.

VII. Conclusions

We have presented a novel technique to assist thread scheduling in AMPs in order to maximize performance/Watt. The key idea is the use of program behavior on one core to predict the power and performance of the application on other cores in the AMP. We leverage the use of performance counters which are available in almost all processors for such a prediction. To illustrate our approach, an eight-core AMP was considered with two core types, one core designed to achieve high performance (HPerf) (two cores) while the other for low power (LP) (six cores). Detailed experiments on the choice of performance counters to estimate the performance and power on the HPerf core while the application executes on the LP core and vice versa have been presented. Approximate expressions based on the values of these counters were formulated to assist in the thread to core assignment so as to maximize performance/Watt. Phase classification was used to trigger the decision making process.

We compared our technique to a static baseline with best thread to core assignment, an online learning based scheme, and an oracular scheme with ability to swap threads between the cores. Our results indicate that the proposed scheme can achieve considerable performance/Watt benefits of about 20% and 200% on an average, over the static and online learning schemes, respectively. Moreover, the proposed scheme performs worse than the oracular scheme by only 8% on average.

References

[1] J. Held *et al.*, "White Paper From a Few Cores to Many: A Tera-scale Computing Research Review," 2006.

[2] R. Kumar *et al.*, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, ser. PACT '06, 2006.

[3] M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33 –38, july 2008.

[4] J. A. Winter *et al.*, "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*, ser. PACT '10, 2010.

[5] D. Koufaty *et al.*, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10.

[6] J. C. Saez *et al.*, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10, 2010.

[7] D. Shelepov *et al.*, "HASS: a scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, April 2009.

[8] R. Rodrigues *et al.*, "Performance Per Watt Benefits of Dynamic Core Morphing in Asymmetric Multicores," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, oct. 2011, pp. 121 –130.

[9] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd conference on Computing frontiers*, ser. CF '06, 2006.

[10] R. Kumar *et al.*, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, dec. 2003.

[11] O. Khan and S. Kundu, "A self-adaptive scheduler for asymmetric multicores," in *Proceedings of the 20th symposium on Great lakes symposium on VLSI*, ser. GLSVLSI '10, 2010.

[12] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09, 2009.

[13] T. Sherwood *et al.*, "Phase tracking and prediction," in *Proceedings of the 30th annual international symposium on Computer architecture*, ser. ISCA '03, 2003.

[14] S. Srinivasan *et al.*, "Efficient interaction between OS and architecture in heterogeneous platforms," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 62–72, February 2011.

[15] S. Srinivasan, R. Iyer, L. Zhao, and R. Illikkal, "HeteroScouts: Hardware Assist for OS scheduling in Heterogeneous CMPs," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 1, pp. 341–342, Jun. 2011.

[16] A. Fog, "The microarchitecture of Intel, AMD and VIA CPU," Copenhagen University College of Engineering, Tech. Rep.

[17] J. Renau, "SESC: SuperESCalar Simulator," 2005.

[18] P. Shivakumar *et al.*, "CACTI 3.0: An Integrated Cache Timing, Power, and Area Model," Tech. Rep., 2001.

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

[20] SPEC2000, "The Standard Performance Evaluation Corporation (Spec CPI2000 suite)."

[21] M. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, dec. 2001.

[22] C. Lee *et al.*, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 30, 1997.

[23] A. S. Dhodapkar and J. E. Smith, "Comparing Program Phase Detection Techniques," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36, 2003, pp. 217–.

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

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

[26] K. Singh *et al.*, "Real time power estimation and thread scheduling via performance counters," *SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 46–55, Jul. 2009.

[27] L. C. Heller and M. S. Farrell, "Millicode in an IBM zSeries processor," *IBM Journal of Research and Development*, vol. 48, no. 3.4, pp. 425 –434, may 2004.