A Study on Polymorphing Superscalar Processor Dynamically to Improve Power Efficiency

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

***Abstract*—Asymmetric Multicore Processors (AMP) have emerged as likely candidates to solve the performance/power conundrum in the current generation of processors. Most recent work in this area evaluate such multicores by considering large (usually out-of-order (OOO)) and small (usually in-order (InO)) cores on the same chip. Dynamic online swapping of threads between these cores is then facilitated whenever deemed beneficial. However, if threads are swapped too often, the overheads may negatively impact the benefits of swapping. Hence, in most recent work, thread swapping decisions are made at coarse grain instruction granularities, leaving out many opportunities. In this paper, we propose a scheme to mitigate the penalty imposed by thread swapping and yet achieve all the benefits of AMPs. Here, a single superscalar OOO core morphs itself into an InO core at runtime, whenever determined to be performance/Watt efficient. Certain Intel processors already have a similar mechanism to statically morph an OOO core to an InO core to facilitate debug. We extend this existing capability to perform dynamic core morphing at runtime with an orthogonal objective of improving power efficiency. Results indicate that on an average, performance/Watt benefits of 10% can be extracted by our proposed morphing scheme at a very small performance penalty of 3.8%. Since this scheme is based on existing mechanisms readily available in current microprocessors, it incurs no hardware overheads.**

***Keywords*-Core Morphing; Asymmetric Multicore Processor (AMP); Out-of-Order (OOO); In-Order (InO)**

I. Introduction

Modern multicore processors provide increased transistor density which has enabled complex functionality and increased performance. This, however, has led to a power density problem. The increase in power density has become unsustainable at 100 W/cm^2 due to packaging limitation, resulting in packaging and microarchitectural changes in processor. The processor industry responded to this problem by incorporating multiple simple processors (Symmetric Multicore Processor (SMP)) on the same die [1]. Such processors are better suited for Thread Level Parallelism (TLP), but performance suffers whenever sequential applications are encountered [2].

Asymmetric Multicore Processors (AMP) were introduced as a potential solution to this conundrum. There have been a number of proposals made in literature [3], [4], [5], [6], [7] and recently, major corporations have released their own versions of AMPs [8]. Here, multiple cores of varying capabilities are included on the same chip. Usually, these cores are of two types; big (OOO) and small (InO). The big cores generally provide good performance while the smaller ones are power efficient. During runtime, whenever deemed beneficial, threads are swapped between the cores such that the objective function (energy, performance/Watt etc.) is satisfied. These multicores have been shown to significantly outperform the SMP counterparts, for a given area and power budget [3], [6], [9].

Thread swapping between cores incurs a performance overhead. This overhead can vary from a few thousand [6] to millions of cycles [10], [11] depending on the shared cache hierarchy and the algorithm used to make the swapping decisions. Hence, in most proposals, thread swapping decisions are made at the granularity of hundreds of thousands to millions of instructions [3], [10], such that the overhead associated with swapping threads is amortized over time. However, such an approach misses out on numerous opportunities that present themselves at a more fine grain instruction granularity [12]. This point is illustrated in Figure 1 where the IPC resulting from running the workload *mcf* on the OOO and InO cores is shown. In figure, the IPC is sampled at coarse grain instruction granularities of 50K instructions. Here, it can be seen that at no point is the IPC of the InO core comparable to that of the OOO core. However, when considering a more finer instruction granularity of 500 instructions (inset), it can be seen that not only are the IPCs of the two cores comparable, but at some points in the plot, the InO core outperforms the OOO core. The InO is the power efficient core and from the figure, it is clear, that at smaller instruction granularities, there is even more potential to make gains in performance/Watt by switching operation from OOO to the InO core. However, swapping threads at such a small granularity in current AMPs, will likely negate all benefits. Hence, there is need for a more fine grain switching mechanism that does not incur large thread swapping penalties.

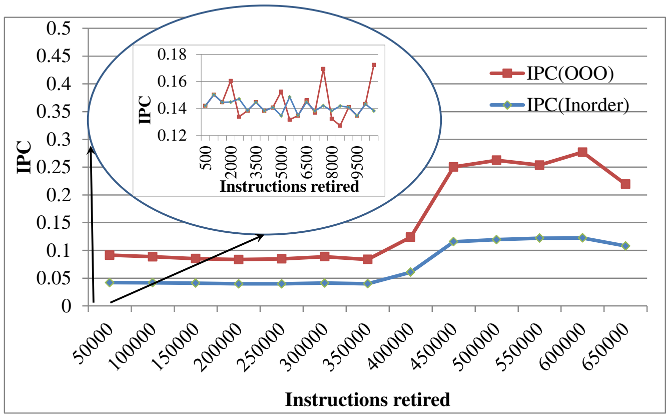


Fig. 1. IPC comparison between the OOO and InO cores when executing the workload *mcf*. In the main figure, each point on the horizontal axis represents 50K retired instructions. In the inset figure, IPC for the the instructions from 0 - 10K have been sampled at 500 instructions.

In this paper, we propose an architecture that reaps most of the benefits of AMPs and yet incurs almost no penalty usually associated with thread swapping. This is achieved by introducing heterogeneity within the same core by morphing it from OOO to InO and vice-versa. Intel's processors feature a special debug mode in which the OOO core turns into an InO core for debug purposes [11]. We propose to extend this mechanism for power efficiency by dynamic entry into and exit from the InO mode, which allows registers and the cache to retain their states, reducing overhead of morphing.

In our scheme, only a single core is considered. In the baseline mode, the core operates in the OOO mode providing high performance. However, during low IPC phases, the operation mode may

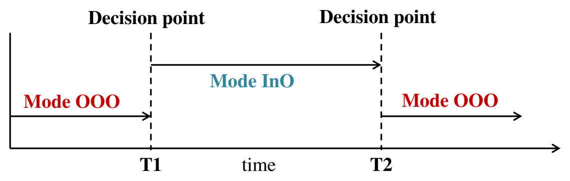


Fig. 2. Change in processor operating modes as a function of time. Initially, the core operates in OOO mode. At time instant T1, it is determined that InO mode is beneficial with respect to performance/Watt and hence the switch in operation mode. Finally, at time T2, the benefits of the InO mode go away and hence, the switch back to the OOO mode.

be switched to the InO mode for performance/Watt improvements. A similar switch is made from InO to OOO when these benefits are predicted to have diminished. The general operation of the proposed morphable core is shown in Figure 2. By switching operation modes on the same core, the proposed scheme takes advantage of heterogeneity and at the same time, incurs no overheads usually associated with thread swapping. It is thus capable of realizing the opportunities that exist at fine grain instruction granularities which results in significant performance/Watt gains.

Results indicate that when using an oracular decision making mechanism to determine thread swapping, such core morphing can result in average performance/Watt gains of 10% at a very small performance penalty of only 3.8% for a wide variety of workloads. Since the proposed scheme makes use of existing facilities in a processor, it has the advantage of being completely designed and verified in silicon and incurs no hardware overheads unlike several comparable schemes [12], [13], [14], [15], [16] We make the following contributions:

1) Quantification of the performance/Watt benefits of using in-built mechanisms designed for debug in modern processors [11].
2) Studies on the trade-off between performance loss and performance/Watt gain by switching between OOO and InO modes of operation on the same core.

The rest of the paper is organized as follows. In Section II, we cover some of the recent advancements in thread scheduling on AMPs. In Section III, the proposed architecture and implementation details are presented. Experimental setup is presented in Section IV which is followed by Results and Conclusions in Sections V and VI, respectively.

II. Related work

With AMPs gaining traction in recent times, there have been a number of proposals made on the subject. We cover some of the recent advances made in AMP thread scheduling and dynamic/morphable multicores.

A. Thread scheduling in AMPs

Several proposals exist employing offline regression based analysis techniques [17], [18], [19] for thread scheduling in AMPs. Here, the characteristics of the workloads that will be run on the AMP are learned offline which is then used online to schedule threads. Such approaches need prior knowledge and hence in some cases may not be practical.

Solutions that learn workload behavior online and based on this make thread scheduling decisions offer a more practical and generic solution to the AMP scheduling problem. Phase classification [20] and sampling techniques have been used to perform scheduling [3], [10], [21], [6]. Whenever an earlier detected phase is encountered again, information gathered from history is used to make the best thread to core assignment. However, sampling poses an overhead and hence such scheme may not be scalable with increasing core counts [7], [11], [22].

Estimation based scheduling is an improvement over such schemes. Here, the performance and/or power of running a thread on another core in the AMP is estimated using statistics such as cache misses and pipeline stalls gathered on the host core [6], [7], [11], [12], [22], [23].

Thus, it can be seen that several techniques exist for online thread scheduling. In this paper, we assume the presence of performance and power estimation schemes for dynamic decision making since, our focus is the exploration of the benefits of core morphing. The implementation of an online estimation scheme by using performance counters and regression analysis is part of our future work.

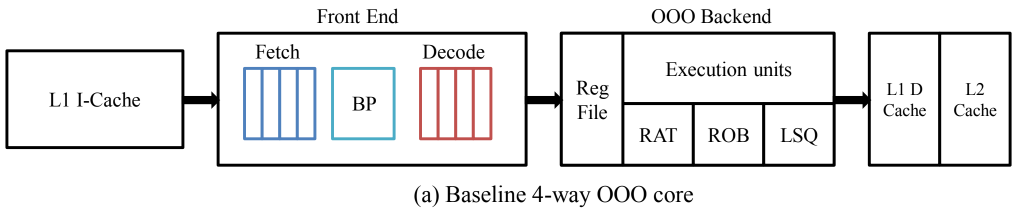
B. Morphable or dynamic multicores

There have been several proposals that advocate dynamic morphing of multicores or single cores such that performance and power efficiency is enhanced at run time.

In a number of proposals, the starting point is a multicore consisting of small cores which then fuse together into a large OOO core on demand [13], [14], [15]. Such approaches suffer from additional latencies that arise from combining resources from various cores. A different scheme was adopted by Khubaib *et al.* in [16] where they start with a baseline OOO core that morphs itself into Simulteneously Multithreaded InO core depending on the number of incoming threads. All such schemes require significant changes to the microarchitecture to be realized in practice.

Dynamic sharing of processor resources for power and performance benefits is also a well explored area. Kumar *et al.* [24] explore sharing of various large structures in the multicore for energy and area savings. In [6], Rodrigues *et al.* explored dynamic exchange of execution units such that performance/Watt is improved. However, all such schemes require extra circuitry that must be designed and verified.

In, [12], Lukefahr *et al.* make a proposal that is similar to ours. In their scheme heterogeneity is introduced into the same core by provisioning two execution backends to the same core. One backend is OOO while the other InO. Both backends share the same caches and fetch units. However, there are several differences between theirs and our proposal. Firstly, Lukefahr *et al.* employ two different backend pipelines (register file, execution units etc.) and decode unit while our scheme uses the same for both modes (OOO and InO). The additional units increase the core area, design/verification effort and time. More importantly, during operation mode switch, their scheme requires the architectural states to be transferred across the two pipelines which adds to the overhead. In contrast, the same register file is used by the two modes in our scheme. Finally, our scheme differs with respect to operations performed at the time of a mode switch (OOO to InO and vice versa). Whenever the scheme decides to switch from OOO to InO, the ROB is power gated and the subsequent instructions are re-fetched in InO mode. Unlike [12], our scheme does not delay the OOO to InO mode switch until all the other speculative instructions are drained from the ROB. Hence, we fully capitalize on the power benefits of moving to the InO mode while keeping the switching complexity and overhead at bay. When switching from InO to OOO mode, the ROB is powered back on and, the head and tail pointers of the ROB are re-initialized to point to the same slot. Hence, the ROB is presumed to be completely empty when the core is morphed back to the baseline OOO mode. Our scheme thus has simplicity and relies on existing mechanisms (used by Intel for debug [11]) for core morphing which makes it a very attractive and practical proposal.



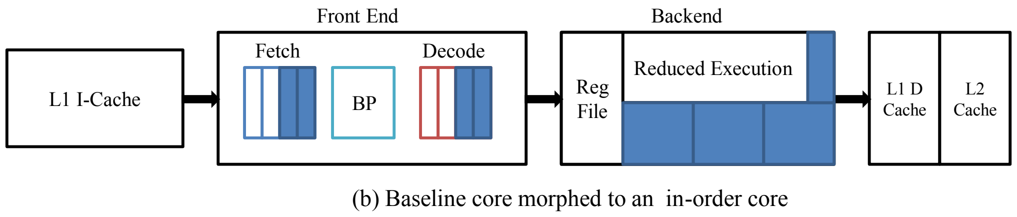


Fig. 3. (a) High-level view of the 4-way OOO baseline core. (b) The InO core obtained by morphing the baseline core. The blue shaded regions indicate the units that are power-gated during InO execution. BP - Branch Predictor.

III. Proposed Scheme

In this section, the proposed core morphing scheme which has the capability to switch between OOO and InO modes at fine-grained time intervals is described in detail. Figure 3(a) shows the considered baseline core which is a 4-way issue OOO complex superscalar core. To facilitate OOO execution and InO commit, the backend is provisioned with register alias table (RAT), load/store queue (LSQ) and ROB. The exact sizes of these resources are discussed in Section IV. A significant performance benefit is achieved by executing the thread on the baseline core (OOO) during high-ILP phases. However, when the processor stalls due to dependencies or is waiting for long-latency memory operations to complete, most of the core resources are idle wasting static power. An InO core with reduced fetch width and downsized core resources may be more power efficient during such phases. To that end, during low-ILP/memory intensive phases, we make use of the existing debug facilities such as that in Intel processors and switch operation from OOO to InO [11]. Further, we power off the ROB, RAT, LSQ and half of the decoders (since fetch width is also reduced to 2) completing the transition from the baseline OOO core to an InO core. The configuration of the processor in the InO mode is discussed in Section IV. The InO core (see Figure 3(b)) is thus more power efficient than the OOO core. While in InO mode, if the program moves to a high-ILP phase, the shut down units are powered on, reverting back to the baseline OOO execution.

We next explain in detail our scheme that decides when to morph at runtime.

A. *Oracular morphing scheme*

As we study the potential benefits of core morphing at fine-grained instruction granularities, we employ an oracular scheme to govern the control of morphing the core to adapt to the time-varying program behavior. Such an oracular scheme would provide an estimate on the upper bound on the performance/Watt benefits that could be achieved by morphing. Note that the implementation of a performance and power estimation scheme [7], [11], [12], [22] is part of future work.

The InO core with many of the core resources powered off consumes much less power at the potential cost of performance. Hence, it would be worthwhile to explore the potential performance/Watt benefits of morphing within a given performance constraint. In line with earlier proposals [12], we assume an overall performance loss of (5-10)% is tolerable relative to running the application completely on the baseline OOO core.

The program thread, by default, begins its execution in OOO mode. The performance and performance/Watt of executing fixed number of committed instructions on both operation modes, referred to as window, is assumed to be known to the oracle. For most of the windows, the OOO mode is expected to deliver higher performance while the InO mode is expected to achieve higher performance/Watt. At the end of each window, the oracle determines the best mode (OOO or InO) to execute the program phase. An OOO to InO switch is effected if the following conditions are met for that window:

1) Performance improvement achieved by the OOO mode relative to the InO mode is less than a predefined *threshold*.
2) Performance/Watt achieved in InO mode is greater than that in OOO mode.

It should be noted that the above threshold could be very different from our performance constraint goal (of overall performance loss of $<5\%$) as the thread may not be run in InO mode for majority of the windows. For lower threshold values, the first condition implies that no significant performance difference is observed between the two modes and hence, it would be better to execute the thread in InO mode to save power. The scheme morphs back from InO to OOO mode if the above two conditions are not satisfied for a window. Under such circumstances, the potential benefits of InO mode are deemed to have diminished.

The parameters of the scheme; namely the window size and the value of threshold need to be determined such that the performance constraint goal is met. As there is no analytical means to determine these parameters, they were obtained experimentally, the details of which are explained in Section V.

B. *Morphing overheads*

The overheads associated with earlier morphing [25], [12] or thread swapping [3], [7], [10] schemes are prohibitive, limiting core re-configurations (morphing/swapping) to occur at coarse-grained granularities. These overheads mainly stem from the communication latency to send/receive data operands/results [25], exchanging architectural states and the additional time required to warm up the dedicated caches and branch predictors [10], [11]. As the morphing happens within the same core in the proposed scheme, all the critical units (e.g., register file, caches and branch predictors) are already intact, completely avoiding all the above overheads. As shown in Figure 3(b), the only overhead associated with our scheme arises from partial powering off/on of the fetch, decode and execution units and complete shut-down/power-up of ROB, RAT and LSQ to switch

TABLE I

CONSIDERED BASELINE CORE PARAMETERS. THE VALUES WITHIN PARENTHESIS REPRESENT THE CHANGE WHILE IN INO MODE.

Param	Value	Param	Value
Issue	4 (2)	INTREG	96 (NA)
FPREG	80 (NA)	INTISQ	36 (NA)
FPISQ	24 (NA)	LS units	3 (1)
LSQ	32 (NA)	ROB	128 (NA)
L1(I/D)	32K	L2	2M
Freq (GHz)	2.4	Type	OOO (InO)

TABLE II

EXECUTION UNIT SPECIFICATIONS FOR THE BASELINE CORE. (P - PIPELINED, NP - NOT PIPELINED, PP - PARTIALLY PIPELINED).THE VALUES WITHIN PARENTHESIS REPRESENT THE CHANGE WHILE IN INO MODE

FP DIV	FP MUL	FP ALU
1 unit, 21 cyc, P	1 unit, 5 cyc, P	2 (1) units, 3 cyc, P
INT DIV	INT MUL	INT ALU
1 unit, 23 cyc, P	1 unit, 8 cyc, P	4 (2) units, 1 cyc, P

between InO and OOO modes. Power gating or power up of all these blocks simultaneously may result in large power surge. Hence, we assume a staggered gating where only a single block is power gated every clock cycle totaling to 6 cycles for 6 blocks. Therefore, all our results assume an overhead of 10 cycles (with additional margin of 4 cycles) for each mode switch.

IV. EXPERIMENTAL SETUP

To evaluate the proposed core morphing scheme, we used a complex superscalar OOO processor as the baseline core. The list of the considered core parameters and execution latencies are shown in Tables I and II, respectively. Most of the core parameters and latencies were taken from [26]. It can be seen from Table I that the baseline core with large core resources (e.g., integer and floating-point registers, issue queues, L2 cache) is suited for high-end applications and is representative of modern superscalar processors. After mode switch to the InO mode, all OOO logic (ISQ, RAT, ROB, LSQ) is switched off. The resulting core configuration for the InO core is shown in Tables I and II in the brackets.

We used SESC for performance simulation [27] and employed CACTI [28] and Wattch [29] to calculate power with modifications to account for static power. The evaluation was carried out using 10 benchmarks from the SPEC2K [30] and Mediabench suites [31]. The benchmarks were carefully chosen to be diverse in nature and were run for 500 million instructions after skipping the initial 5 billion.

V. RESULTS AND ANALYSIS

We now present the experimental results of our proposed core morphing scheme. Determination of window size length (discussed in Section III-A) i.e. the number of retired instructions after which morphing decision must be made, is an important parameter of the scheme. Too small a window size may result in too frequent switching between operation modes, negating the potential performance/power benefits. On the other hand, too large a window may not realize any benefits at all. Once the window size is determined, results are presented on performance/Watt savings for varying performance loss thresholds. Finally, we present the overall performance/Watt improvement achieved using our scheme over the baseline core for all the considered workloads.

A. Determining the window size

To determine the window size, we make an assumption that a small performance loss (5-10)% for a substantial gain in performance/Watt is acceptable (25-30)%. We experimented with various window sizes

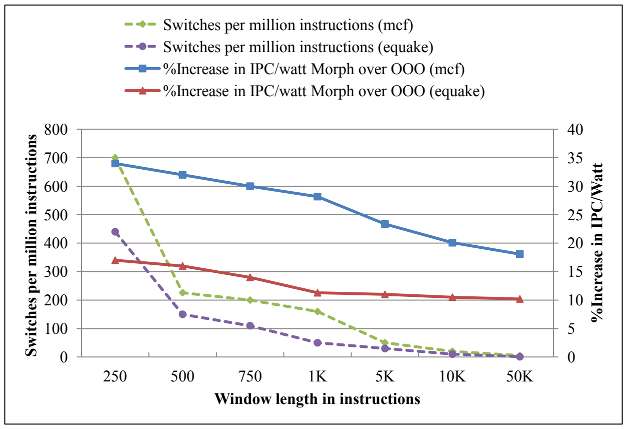


Fig. 4. Sensitivity analysis on window size vs. achieved performance/Watt improvement over baseline OOO mode and number of switches between operation modes.

varying from 250 to 100K retired instructions to explore the horizon. At the end of each window, our proposed scheme makes the decision regarding the best mode of operation (OOO or InO) based on the oracular knowledge.

Figure 4 shows the percentage improvement in performance/Watt of the proposed core morphing scheme over the baseline OOO mode (see Figure 3(a)) and the number of switches/million instructions for 2 benchmarks (*equake* and *mcf*) for different window sizes. We show results for these workloads since benefits observed for these were the largest. In the figure, *Morph* core represents the core with the proposed core morphing capability where the execution switches between the OOO and InO modes depending on current operating conditions. For small window sizes, a substantial gain in performance/Watt over the baseline OOO mode for both *mcf* and *equake* is observed which starts to decrease with increasing window sizes. This is expected as smaller the interval, more are the opportunities presented [12]. Hence, theoretically in the absence of mode switching overheads, the smaller the interval the better. From the figure, it can be seen that increasing the window size from 250 to 500 results in less than 2% gain in performance/Watt for both workloads but the number of switches in operation modes between OOO and InO drops significantly (700 to 226 for *mcf* and 440 to 150 for *equake* per million instructions executed). This will result in smaller mode switch overhead. Increasing the instruction interval further results in significant performance/Watt loss ($\geq$ 4%) and no appreciable decrease in number of mode switches. The optimum instruction window length that would provide siginifcant increase in performance/Watt without adding too much overhead due to the number of switches required was thus found to be 500. Hence, in the rest of this paper, we use a window size of 500 instructions.

B. Analysis of performance threshold variation

Morphing to the InO mode of operation incurs a small overhead due to power gating the various unused structures. Further, in some cases it may make sense to run in the InO mode, even if the performance in that mode is smaller than but close (within a percentage threshold) to that in the OOO mode such that performance/Watt is maximized. To explore the potential benefits of such sacrifice in performance on performance/Watt gains, we conducted experiments with various performance thresholds. Decisions are made at the end of every 500 executed instructions as determined in the previous section. Our oracular scheme follows the greedy approach and tries to switch to the InO mode whenever possible. At the end of every interval, the oracle computes the performance difference between

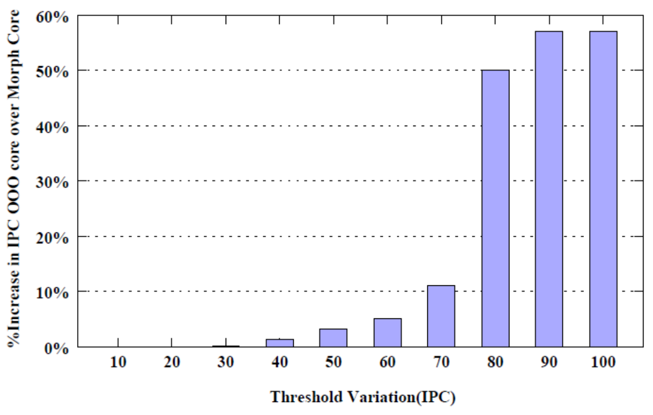


Fig. 5. Variation of performance(IPC) threshold vs percentage increase in IPC of OOO core over Morph core for *mcf*

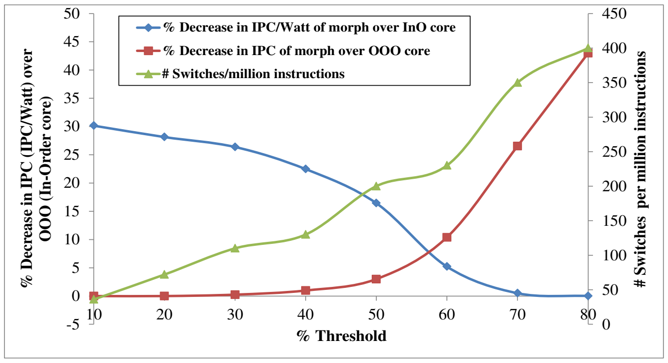


Fig. 6. Plot showing the variation of IPC, performance/Watt and number of switches of morphed core with performance threshold for mcf

that obtained in the OOO mode and the InO mode. In Figure 5, the %increase in IPC of the OOO mode over the morph core is plotted for various performance thresholds for the workload *mcf*. Threshold of 60% indicates that the switch from OOO mode to InO mode may be made if the % increase in IPC of OOO mode to InO mode is less than or equal to 60%. If this condition is satisfied, the oracle then computes the resulting gain in performance/Watt after switching. A switch in operation mode is triggered only if both the above 2 conditions are satisfied for a particualr window. As shown in Figure 5, for low performance threshold between (10-40)%, the overall % increase in IPC of OOO core with respect to running in the morphed core is extremely small. This is expected, since for low performance threshold, number of switches from OOO mode to InO mode is quite small . For a performance threshold of greater than 50%, there is more opportunity for switching to InO mode. From Figure 5, we conclude that if the performance threshold is about 60%, the overall performance loss of morph core with respect to OOO core is less than 10% and hence may be within the acceptable limit when running the workload *mcf*.

More in-depth analysis for choosing the appropriate switching threshold is shown in Figure 6. We need to find an optimum performance threshold such that overall performance loss is tolerable (5-10)% and there is substantial overall gain in performance/Watt. As shown in Figure 6, at a threshold between (55-65)% we find that % decrease in IPC of the morph core over OOO core is less than 10%. Plot showing the % decrease in performance/Watt of morphed core over InO core is also shown in Figure 6. This plot is of significance, since it helps us understand the % performance/Watt that has been lost by running the core in morphed core as compared to running the application completely on the InO core. Again, at a threshold point between (55-60)%, the % decrease in performance/Watt is between (5-10)% and the number of mode switches per million instructions is between (200-250). For performance threshold greater than 60%, the number of switches increases significantly with not much of a gain in performance/Watt. Performance loss suffered by morphed core also increases steeply for threshold greater than 60%. Thus we conclude that optimum performance threshold lies between (55-60)% for the workload *mcf*. We add a guard band to prevent oscillation between operating modes. Hence, for the benchmark *mcf* the decision points are threshold of 55% for entering and 65% for exiting the InO mode. Different benchmarks have different charactersitics and we have conducted an extensive analysis for all the benchmarks similarly to find the appropriate performance threshold. We have found that there exists a different performance threshold for each workload. Our objective is to find the upper bound on the benefits of the core morphing scheme. Hence, in this paper we use the most optimum threshold for each workload. Exploring a common threshold over all workloads is part of future work. The threshold used for each workload is shown in Table III.

C. *Morphing benefits on performance/Watt*

The performance/Watt benefit obtained for different benchmarks is shown in Figure 7. Memory bound benchmarks like *mcf* and *equake* obtain significant improvement in performance/Watt when compared to the baseline scheme where the benchmarks are run completely in OOO mode. For *mcf* we achieve close to 35% improvement in performance/Watt while running in morphed core. This increase comes at the cost of a performance overhead of only 10%. For *equake* we achieve performance/Watt improvement of close to 15% with drop in performance of only 5%. For other benchmarks, we have achieved close to 5% improvement in performance/Watt without much loss in performance. On an average, the proposed scheme achieves a performance/Watt benefit of 10% at an overall performance loss of just 3.8% considering all the workloads. In Figure 7, we also show the percentage of time spent in the InO mode. More the time spent, more is the expected gain in performance/Watt and consequently more is the expected performance loss. It can be seen that in general this is the case. In particular, the InO mode is used for around 80% of the time when executing *mcf* which is a memory intensive application. This shows that for low IPC applications, the proposed scheme is capable of extracting significant performance/Watt benefits at a small performance loss.

In Table III, we also show the average number of mode switches per million instructions as made by the oracular mechanism. The number of switches is high for benchmarks that show significant increase in performance/Watt. Due to complete sharing of resources

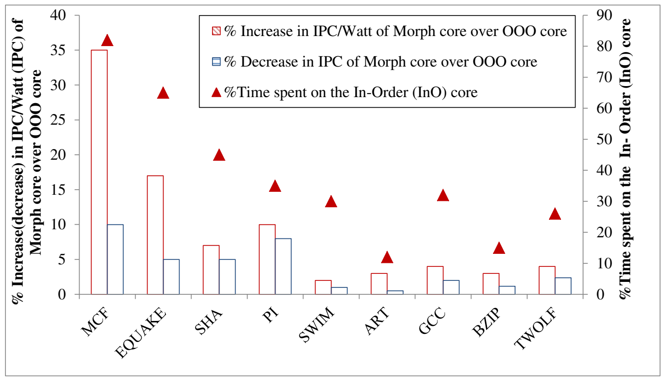


Fig. 7. Plot showing the maximum performance/Watt achieved still maintaing the overall performance loss within (5-10)%

TABLE III
PERFORMANCE THRESHOLD INDICATING THE ENTRY AND EXIT CONDITIONS FOR THE MORPH MODE AND NUMBER OF SWITCHES IN OPERATING MODE PER MILLION INSTRUCTIONS

Benchmark	Threshold (range) determined	Switches/million instructions
mcf	55-65	230
equake	55-60	180
sha	50-55	300
pi	50-55	100
swim	55-60	105
art	55-60	60
gcc	65-70	120
bzip	65-70	120
twolf	60-65	170

between OOO and InO operation modes, fine grain switching is possible at the cost of very low switching penalty. On an average, the overall impact in performance due to switching was found to be less than 1%.

VI. CONCLUSIONS

AMPs have emerged as the likely solution to the performance/power problem faced by the current generation of multicore processors. Prohibitive thread swapping overheads limit the potency of these architectures. In this paper, we have presented an architecture that provides the benefits of AMPs and does away with thread swapping. We propose to bring about heterogeneity within a core. This is made possible by using the debug mechanisms currently in use in the industry where operating mode of a core may be switched from OOO to in-order (InO). In the baseline mode, the considered processor operates in the OOO mode providing high performance. During low IPC phases, the core switches operation mode to InO such that performance/Watt is maximized. To explore the upper bound on the benefits of the architecture, an oracular decision making mechanism was employed to determine time instants of mode switch. Results indicate that core morphing can result in average performance/Watt gain of 10% which comes at a very small performance penalty of just 3.8%.

VII. ACKNOWLEDGEMENT

This research was supported in part by grants 0903191 and 1201834 from the National Research foundation.

REFERENCES

[1] J. Held *et al.*, "White paper from a few cores to many: A tera-scale computing research review," 2006.

[2] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33 –38, july 2008.

[3] 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.

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

[5] M. A. Suleman *et al.*, "Accelerating critical section execution with asymmetric multi-core architectures," *SIGPLAN Not.*, vol. 44, no. 3, pp. 253–264, Mar. 2009.

[6] R. Rodrigues *et al.*, "Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 1, pp. 5:1–5:23, Jan. 2013.

[7] ——, "Scalable thread scheduling in asymmetric multicores for power efficiency," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, 2012, pp. 59–66.

[8] P. Greenhalgh, "Big.little processing with arm cortex-a15 and cortex-a7," sep. 2011.

[9] E. Grochowski *et al.*, "Best of both latency and throughput," in *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE International Conference on*, oct. 2004.

[10] 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.

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

[12] A. Lukefahr *et al.*, "Composite cores: Pushing heterogeneity into a core," in *International Symposium on Microarchitecture (MICRO), 2012 IEEE/ACM International Symposium on*, dec. 2012.

[13] C. Kim *et al.*, "Composable lightweight processors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 381–394.

[14] D. Tarjan *et al.*, "Federation: Repurposing scalar cores for out-of-order instruction issue," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, june 2008, pp. 772 –775.

[15] M. Pricopi and T. Mitra, "Bahurupi: A polymorphic heterogeneous multi-core architecture," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 22:1–22:21, Jan. 2012.

[16] Khubaib *et al.*, "Morphcore: An energy-efcient microarchitecture for high performance ilp and high throughput tlp," in *International Symposium on Microarchitecture (MICRO), 2012 IEEE/ACM International Symposium on*, dec. 2012.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] V. Craeynest *et al.*, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 213–224.

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

[24] R. Kumar *et al.*, "Conjoined-core chip multiprocessing," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37, 2004, pp. 195–206.

[25] 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.

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

[27] J. Renau, "Sesc: Superescalar simulator," 2005.

[28] P. Shivakumar *et al.*, "Cacti 3.0: An integrated cache timing, power, and area model," Tech. Rep., 2001.

[29] 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.

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

[31] 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.