Performance and Power Benefits of Sharing Execution Units between a High Performance Core and a Low Power Core

Rance Rodrigues, Israel Koren and Sandip Kundu
Depatment of Electrical and Computer Engineering
University of Massachusetts Amherst, MA 01003
Email: {*rodrigues,koren,kundu*}@ecs.umass.edu

***Abstract*—Several studies and real world designs have advocated the sharing of large execution units between pairs of cores in Symmetric Multicore Processors (SMP) for area and power savings. Such sharing was shown to have negligible impact on performance. Recently, a number of Asymmetric Multicore Processor (AMP) designs have become available. The objective of this paper is to investigate whether sharing of resources across AMPs offers similar benefits. Our study shows that while the area and the power savings remain similar, the performance of the smaller core in the AMP can improve significantly making sharing even more attractive for AMPs. Simulation results indicate that for certain workloads, the performance of the small core may improve by as much as 54% by sharing certain large execution resources of the big core, while affecting the performance of the big core by only ∼4%, resulting in an overall gain in system performance of 20%. The corresponding improvement in aggregate performance/Watt is 12% while the area savings is about 7%.**

***Keywords*—Asymmetric Multicore Processor (AMP), Symmetric Multicore Processor (SMP), resource sharing, performance, performance/Watt**

I. Introduction

Sharing underutilized resources has been investigated by several researchers [1], [2] and applied recently in the AMD Bulldozer architecture [3]. The general consensus is that such sharing has only a limited impact on the performance of SMPs. Apart from savings in area that results from such sharing, static power savings also yield an improvement in performance-per-Watt. In particular, sharing of large execution units such as the multiply and divide units has been reported to result in negligible performance penalty [1].

Previous studies have explored sharing in SMPs. These SMPs are designed such that reasonable performance is achieved for a wide variety of applications. Whenever these resources are not fully utilized, performance-per-Watt suffers. It is well-known that applications have diverse needs for compute resources. Furthermore, the resource needs for an application change over time, exhibiting distinct *phases* [4]. If the computing resources are underutilized during an execution phase of the application, power is wasted. AMPs have been proposed as a potential solution to this problem. Assigning a low demand application to a smaller core saves power without sacrificing performance [5], [6], [7]. Recently real world designs have emerged [8]. At a high level, AMPs consist of big (high performance) and small (low power) cores. Usually, workloads with high instruction-ievel parallelism (ILP) and memory-level parallelism (MLP) are best run on big cores while others run more efficiently on small cores.

The big cores are designed for performance and they are often over-provisioned with resources. For example, they have larger ROB, reservation stations and load/store queue to take full advantage of out-of order (OOO) execution. They also

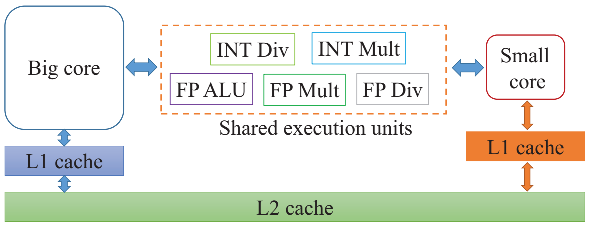

Fig. 1. A high level view of the resource sharing explored in this paper.

have multiple low latency execution units. In contrast, the small cores are designed for power efficiency and therefore do not warrant a similar provisioning of resources. These cores have smaller OOO execution resources, if any, and the execution units are often fewer in number with higher latencies. Occasionally, such cores may benefit from additional (or more powerful) execution resources, but such provisioning is expected to be inefficient in terms of performance-per-Watt for the more common use-cases of these cores.

Sharing execution resources in the context of AMPs has not been yet explored. By sharing resources in AMPs, the small core may benefit from getting access to the high performance resources of the big core. If the shared resources are not continously used by the big core, the impact on the big core's performance will be small. Consequently, overall system level performance is expected to improve. Performance-per-Watt will also improve due to savings in static power. Thus, unlike in SMPs where sharing only results in power savings, sharing resources in AMPs may also result in performance improvement for the small core. Sharing must, however, be limited to the relatively underutilized resources in the big core such that its performance is not compromised.

In this paper, we analyze the sharing of large and underutilized execution units belonging to a big (high performance) core with a small (low power) core. Specifically, the integer (INT) multiply (Mult) and divide (Div) units, and the floating-point (FP) ALU, Mult and Div units of the big core are shared with the small core. A high level view of the resource sharing explored in the paper is shown in Figure 1. The small core that now has access to the fast execution units of the big core can benefit whenever workloads that demand such resources are executed on it. It also leads to static power savings since the small core no longer needs dedicated execution units. On the other hand, the big core may experience a performance loss. Design changes required to allow for sharing are also expected to add a small performance penalty due to an increased access latency of the shared resources. Still, our results indicate that sharing large and underutilized execution units between a big and a small core may result in performance improvement of as much as 54% for the small core and a performance

loss of at most 4% for the big core. Overall, the system performance is enhanced by as much as 20% in the best case and by 7% for the average case. A best case performance-per-Watt improvement of 12% and average improvement of 6% were observed over several multi-threaded and multi-programmed workloads considered in this study. The shared resources architecture also results in area savings of 7%.
The major contributions made in this paper are:

• Performance and performance-per-Watt analysis of sharing large and underutilized execution units of a high performance core with a low power core.
• Analyzing the sensitivity of the results to the shared resource access latency, and
• Demonstrating that sharing large execution resources in AMPs results in greater benefits than similar sharing for SMPs.

The rest of the paper is organized as follows. Recent related work is presented in Section II, followed by the details of the proposed architecture in Section III. The experimental set-up is discussed in Section IV. The simulation results are presented in Section V and the conclusions in Section VII.

II. Related work

The idea of resource sharing has long been in existence [1], [9], [10] for improving the performance-per-Watt of SMPs. Simultaneous Multi-Threading (SMT) [9], [11] was one of the first approaches for sharing idle resources. In an SMT processor, multiple threads are run on the same core and they share and compete for core resources. Such dynamic sharing improves resource utilization for certain combinations of workloads resulting in performance-per-Watt improvement. Dolbeau and Seznec [1] explore intermediate design points between the Chip Multiprocessor (CMP) and SMT architectures where the sharing of the caches, branch predictor and long latency execution units is explored. A similar study was presented in [2] where the caches, crossbar and floating-point units were shared. In these studies, significant area savings at a minor loss of performance were reported. However, the analysis was focused on the performance of SMPs and did not consider AMPs or performance-per-Watt. Further, the impact of access latency of the shared resources was not investigated. Often, an inverse relation between the shared resource access latency and the design complexity exists. Thus, this aspect warrants a more thorough investigation.
Other approaches have also been followed in the context of resource sharing. Watanabe *et al.* [12] explore flexible sharing of a pool of “execution engines” among various cores. By ensuring that the producer and immediate consumers are sent to the same engine, efficient usage of the shared units is made possible. However, each engine requires a queue and other data to keep track of producers and consumers resulting in a complex design. In [13], the authors propose the sharing of functional units across cores in a 3D stacked die. Depending on the need, execution units from a nearby core may be accessed to boost performance. A similar approach to 3D resource sharing was proposed in [14] where the re-order Buffer (ROB), register file, instruction queue and the load/store queues were shared.
The work most closely related to ours is that presented by Rodrigues *et al.* in [7], [15] where dynamic exchange of execution units between cores in an AMP was investigated. Depending on the current workload characteristics, two cores may exchange execution units to maximize performance-per-

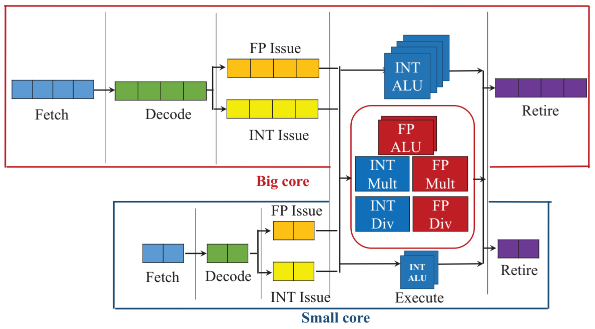

Fig. 2. Pipeline level view of the resource sharing explored in this paper. In the figure, FP = floating-point and INT = integer.

Watt. With respect to design complexity, this approach is similar to ours. The major advantage of this architecture is that resource contention between the two cores does not take place. In our design, if the demand for the shared execution units from both big and small cores increases at the same time, performance will suffer due to contention. On the other hand, our design has the following advantages:

• To trigger a dynamic resource exchange, Rodrigues *et al.* monitor performance counters for estimating the utilization of the execution units. Based on these counters, decisions are made to exchange execution units. In contrast, our scheme does not require such monitoring.
• The decision making mechanism requires off-line training using a few workloads which is not always practical. Our scheme requires no off-line training.
• Execution units are exchanged at coarse grain intervals (2K - 10K instructions) to avoid the associated overheads. This may result in missing potential opportunities at finer granularities. In our scheme, cycle by cycle sharing of the execution units takes place and.
• Their scheme does not provide any hardware savings while our scheme results in 7% area savings.

To the best of our knowledge, this is the first work that explores resource sharing between AMPs to boost performance and performance-per-Watt.

III. Architectural details

The details of the proposed resource sharing architecture are presented is this section. Then, sources of performance penalties and estimated hardware savings are outlined.

A. *The shared resource multicore architecture*

In the proposed architecture, the large and underutilized execution units of the big core are shared with the small core in the AMP. The shared execution units are depicted in Figure 2 and include the floating-point ALU, Mult, Div and the integer Div and Mult units. In the figure, it can be seen that the big core is abundant with resources. It is deeply pipelined and can fetch, issue and retire up to 4 instructions per cycle. It features deeper issue and retire queues and has several fast execution units. The small core, on the other hand, has a fetch width of 2, smaller issue and retire queues, and its execution units are slow and fewer in number. Details of each modeled core may be found in Table I. Each core retains control of its own integer ALU execution unit since these units are small and so frequently encountered that sharing would result in significant performance penalties. Due to sharing, the small core no longer needs the corresponding dedicated execution units resulting in

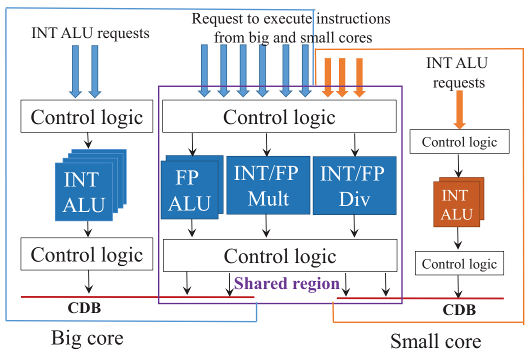

Fig. 3. The arbitration logic required to control access to the shared execution units.

area savings. However, additional hardware may be required to arbitrate access to the shared execution units.

B. Hardware details

A high level view of the control logic that grants access to the shared execution units is shown in Figure 3. In the baseline architecture (no sharing), each core has its own control logic that arbitrates access to the execution units. Whenever instruction dependencies are resolved, a request is made to the control logic for the appropriate execution unit. Depending on the opcode of the requesting instruction and whether the particular execution unit is busy or not, the request may or may not be granted. The instruction may then need to make another request in the next cycle. In the shared resource architecture, the hardware is different. Since each core retains its own INT ALU, dedicated control logic exists in each core to control access to these units. Any request made to one of the shared execution units must be arbitrated by common control logic that monitors requests coming from either core. The design of the logic is very similar to that in the baseline with two exceptions: (i) it must be designed to handle more requests than that in the baseline (up to 4 from the big core plus up to 2 from the small core), and (ii) there is need for instruction result tagging such that the result of the execution is forwarded to the correct bus. For example, result of an FP ALU instruction that belongs to the small core must be forwarded to the CDB of the small core. For this, a single bit flag may be used. With respect to the sharing policy while there are several approaches possible, we explore in this paper the first come first served policy for simplicity of analysis. We will consider more complicated approaches in the future.

C. Performance implications of sharing

A performance boost for the small core is expected since this core now has access to the faster and more numerous execution units of the big core. On the other hand, a small performance penalty is expected for the big core whenever contention for the shared resources is high. Another source of increased latency is the slightly more complex control logic that grants access to the shared execution units to requests coming from the two cores and, the bus forward logic where an additional check must be made to forward the result to the correct core. Several authors [1], [2] claim this overhead to be around 0 to 1 cycle. For the sake of completeness, we analyzed the sensitivity of our results to the shared resources' access latency.

D. Area implications of sharing

We have estimated the area savings due to hardware resource sharing using McPAT [16] for a 45nm technology. This tool takes as input the dual-core configuration and outputs the

TABLE I. CORE PARAMETERS

Parameter	Small	Big	Parameter	Small	Big
Issue	2	4	INTREG	64	96
FPREG	64	80	INTISQ	16	36
FPISQ	16	24	LS units	2	4
LSQ	32	32	ROB	56	128
L1(I/D)	32K	32K	L2	2M shared	
Freq (GHz)	2.4	2.4	Type	OOO	OOO

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

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

estimated area for each block in the floorplan. For the considered dual-core (see Table I), the small core was estimated to occupy 23mm^2 while the big core around 35mm^2 excluding the L2 cache which was estimated to be around 15mm^2. Hence, the total baseline (no sharing) core area was estimated to be around 73mm^2. The large execution units occupy 5.11mm^2 and 9.78mm^2 in the small and big cores, respectively. Thus, the total area of the AMP with sharing is about 68mm^2 which results in area savings of approximately 7%. A small area overhead arises due to the control logic required to arbitrate requests to the shared execution units (see Figure 3). This overhead is expected to be lower than 1%. Thus, the area savings of the proposed architecture is expected to be slightly lower than 7%.

IV. EXPERIMENTAL SETUP

The parameters of the big and small cores modeled in our experiments are shown in Table I. The execution unit characteristics are shown in Table II. These parameters were obtained from [17]. SESC was used as the architectural performance simulator [18]. We made significant modifications to the simulator to enable shared resource execution with arbitration. Power was measured using Wattch [19] and Cacti [20] with modifications to account for static power.

For the experiments, we have selected 15 benchmarks: 7 benchmarks from the SPLASH-2 [21] (*barnes, cholesky, fmm, lu, radix, raytrace, water*) and 8 from the SPEC 2000 suite [22] (*fbench, flops, art, equake, gzip, ammp, mcf, gcc*). These workloads were chosen because they are diverse enough and thus enable a thorough evaluation of the architecture under study. For the SPLASH-2 workloads, two threads were spawned and run on the multicore. These threads are homogeneous in characteristics and hence are expected to test the studied architecture for cases where the threads execute instructions that must be forwarded to the shared units. We also created multiprogrammed workloads by combining two workloads each from the SPEC 2000 suite. For these workloads the threads are usually heterogeneous in characteristics. We thus tried to evaluate the studied architecture over a broad spectrum of potential workloads. The created workload sets and their characteristics are shown in Table III. Each workload was run until the sum of the instructions retired on the two core types equals 500 million instructions. The instruction distribution of the threads run is shown in Figure 4. In all Figures for experiments concerning two threads run on the two cores, the workload is denoted by $thread_1\_thread_2$ on the x-axis, with $thread_1$ executing on the big core and $thread_2$ on the small core.

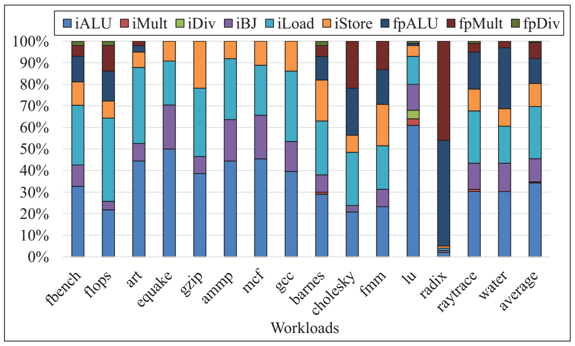

Fig. 4. The instruction distribution of the various workloads when run for 500 million instructions. The average over all workloads is also shown.

TABLE III. Workload characteristics

Workload	Characteristics
barnes_barnes	FP and INT intensive
cholesky_cholesky	FP intensive
fmm_fmm	Both FP and INT intensive
lu_lu	INT intensive
radix_radix	FP intensive
raytrace_raytrace	FP and INT intensive
water_water	FP and INT intensive
flops_fbench	Both FP intensive
equake_art	Both INT intensive
gzip_ammp	One INT and the other slightly FP
art_ammp	One INT and the other slightly FP
mcf_gcc	Both memory intensive

V. Results

We now present the performance and performance-per-Watt analysis of the proposed architecture. We first explore the upper bound on the performance enhancement possible for the small core when its dedicated execution units are replaced by the units it shares with the big core. We then present the results for the shared resource architecture. Both individual core (using the simple $Speedup = val(new)/val(old)$) and system level results are presented. System level results are based on the harmonic speedup metric which is calculated as follows:

$S_0 = (IPC_{thread0})_{new}/(IPC_{thread0})_{baseline}$
$S_1 = (IPC_{thread1})_{new}/(IPC_{thread1})_{baseline}$
$Speedup_{harmonic} = 2/(1/S_0 + 1/S_1)$

Here, *baseline* refers to the case where the cores do not share any unit. Shared resource access latencies of 0, 1 and 2 cycles were considered in the experiments, but we only show the results for 0 and 2 cycles since we found the results to be almost linearly dependent on latency.

To obtain the upper bound on the potential performance and performance-per-Watt for the small core with access to the shared execution units, we carried out an experiment where only a single thread is run on the small core and the core is assigned the INT Div, Mult and FP ALU, Div and Mult execution units of the big core. Note that such an architecture is not practical and is experimented with only to explore the potential upper-bound. The performance and performance-per-Watt results relative to the small core with its regular execution units are plotted in Figure 5 for the considered workloads. It has been observed that there are a few workloads that would benefit significantly with respect to both performance and performance-per-Watt if given access to faster execution units (e.g., *barnes,raytrace,equake,flops* and *fbench* show 52%, 54%, 30%, 30% and 40.5% improvement in performance and

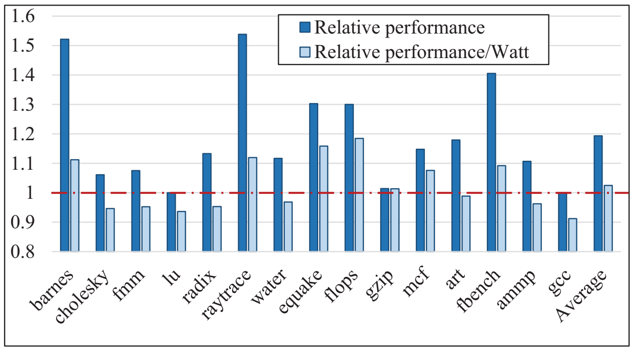

Fig. 5. The relative performance and performance-per-watt enhancement possible for the small core when its dedicated small execution units are replaced by those from the big core.

11%, 12%, 15%, 18% and 9% improvement in performance-per-Watt, respectively). However, there are also workloads such as *lu, gcc, gzip* where no performance improvement is observed while the performance-per-Watt is reduced. There are also workloads, such as *cholesky, fmm, radix, water, ammp*, where only a modest performance gain is achieved and the performance-per-Watt is decreased. The faster execution units come at the cost of larger dynamic and static power. On an average, the performance was found to increase by 20%, while the performance-per-Watt gain was around 2%. This experiment shows that not all workloads will benefit from faster execution units on the small core.

A. Performance and Power analysis of resource sharing

The individual core level and system level performance observed due to resource sharing between the big and small cores are shown in Figures 6 and 7 for the various workloads. The access latency of 0 cycles represents the ideal scenario where the architecture and floorplan have been optimized for resource sharing. This latency also shows the effect that resource contention has on the results. We observe that the average performance increase of the small core drops by 2% from the observed 20% achieved when the small core has its own dedicated fast execution units (see Figure 5). The highest gain of 54% for the small core was observed when running the workload *raytrace_raytrace*. For the big core, contention related loss was found to be less than 1% on average with a worst case loss of 4% experienced when running the workload *barnes_barnes*. This shows that contention related losses are very small for both cores. In general, workloads where both threads run are FP intensive, e.g., (*barnes_barnes*, *raytrace_raytrace*, *equake_art* and *flops_fbench*), show improvement in performance. The workloads that do not show improvement are the ones that make no use of the faster shared execution units such as *ammp_gzip*, *mcf_gcc* and *lu_lu*. On an average, a system performance gain of 7% is observed when considering shared resource access latency of 0 cycles. The maximum system level gain of 20.2% was observed when running the workload *raytrace_raytrace*. As expected, an increase in the shared resource access latency reduces the benefits of sharing. The largest drop is observed for the workload *radix_radix* where a gain of 7% for a 0-cycle latency drops to a loss of around 18% for a 2-cycles latency. On an average, the system level performance gain drops from 7% to 1% going from a access latency of 0 to 2 cycles. This shows the sensitivity of the architecture to the shared resource access latency. Nonetheless, the fact that an improvement in

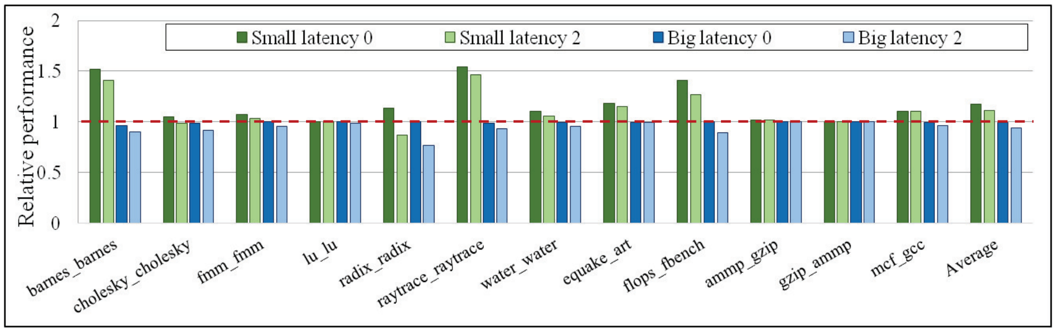

Fig. 6. The relative individual core performance of the big and small core in the shared resource configuration for various shared resource communication latencies. In the plot, $y = 1$ is highlighted with a red dotted line for ease of analysis.

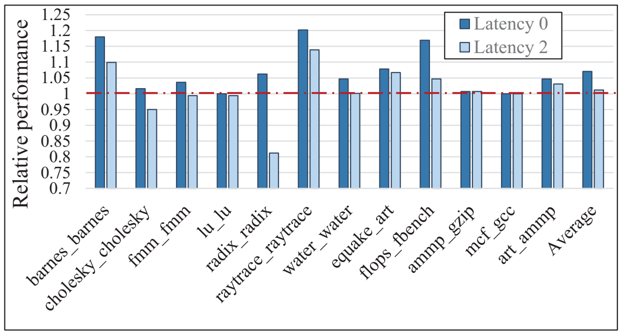

Fig. 7. The system level relative performance of the big and small dual-core in the shared resource configuration for shared resource access latencies of 0 and 2 cycles.

the average performance is observed even with a access latency of 2 cycles shows the effectiveness of resource sharing.

We also conducted an analysis of the power consumption in the non-sharing and sharing architectures. We do not include figures due to lack of space. It is observed that for a few workload combinations, the power consumption increases although in general, it is expected that resource sharing will reduce the power, in particular, the static power. This is not necessarily the case for dynamic power since the small core gets access to the execution units of the big core that consume more power. Power increases were observed for the workloads that make best use of the shared resources such as *barnes_barnes*, *raytrace_raytrace*, *equake_art* and *flops_fbench* for a shared resource access latency of 0 cycles. This reduces with an increase in the resource access latency. On the other hand, workloads such as *ammp_gzip*, *mcf_gcc* and *lu_lu* that do not make use of the shared resources result in power savings. On an average, the system level power increases by 1% for a 0-cycles latency and drops by 2% for a 2-cycles latency. The worst case power increase of 8% was observed for the workload *raytrace_raytrace* that makes the most use of the shared resources.

The individual core level and relative system level performance-per-Watt for each workload is shown in Figures 8 and 9, respectively. The performance increase results in a higher power consumption. Hence, the performance-per-Watt did not increase as much as the performance. Still, average improvements of 9% on the small core and 3% on the big core are observed for a 0-cycles access latency (see Figure 8). The best case for the small core, of almost 23%, was observed for *raytrace_raytrace*. For the big core, with 0-cycles latency, no workload shows losses. An increase in the latency, however, resulted in small (1% on average) loss. For *radix_radix*, at a 2-cycles latency, the small and big cores show 10% and 17% loss in performance-per-Watt, respectively, resulting in system level loss of 14%. This shows that if the shared resource access latency is high, sharing is not beneficial if the majority of the workloads that will be run on the multicore are sensitive to the latency. Still, system level performance-per-Watt improvements of 6 to 1% on an average and 12 to 9% in the best case, were observed for increasing shared resource access latencies. This shows that for generic workloads, improvements will be achieved even if the shared resource latency is as high as 2 cycles. The workloads that showed no performance improvement (*ammp_gzip*, *mcf_gcc* and *lu_lu*) show performance-per-Watt improvement due to static power savings. Sharing execution units in AMPs thus provides both power savings and performance improvement, something that sharing in SMPs cannot achieve.

VI. Scalability

So far, we have considered the sharing of large and underutilized execution units between one big and one small core and observed the potential for performance and performance-per-Watt improvements. In the future, we will consider sharing resources between more than 2 cores in the AMP.

VII. Conclusions

We have presented a detailed analysis of the performance and performance-per-Watt benefits of sharing execution units between a high performance (big) core and a low power (small) core. Specifically, the integer divide and multiply and, the floating-point ALU, divide and multiply units of the big core were shared with the small core. This architecture was analyzed for several multithreaded and multiprogrammed workloads. Since sharing resources may result in increased access latency, we analyzed the sensitivity of the performance and performance-per-Watt to the latency. Our results indicate that resource sharing between big and small cores has the potential to significantly enhance performance and performance-per-Watt of the small core by as much as 54% and 23%, respectively. For the big core a worst case performance loss of 4% was observed while the performance-per-Watt increases by 4%. On an average, the system level performance improvement ranges between 7 and 1% and the performance-per-Watt improvement ranges between 6 and 1% for shared resource

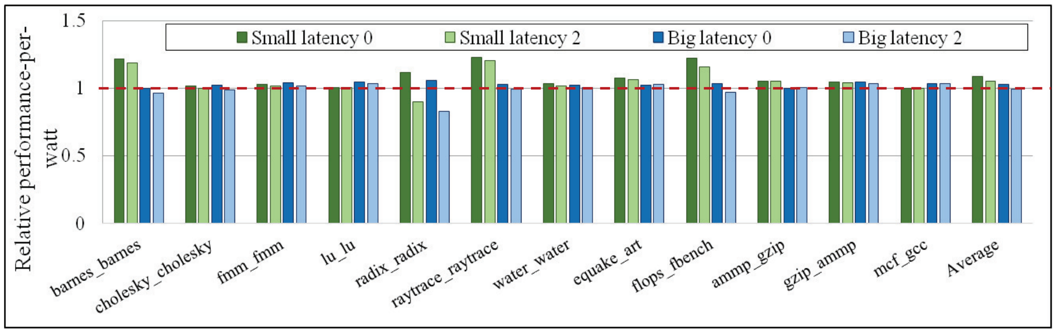

Fig. 8. Relative individual core performance-per-Watt of the big and small core in the shared resource configuration for various shared resource communication latencies. In the plot, $y = 1$ is highlighted with a red dotted line for ease of analysis.

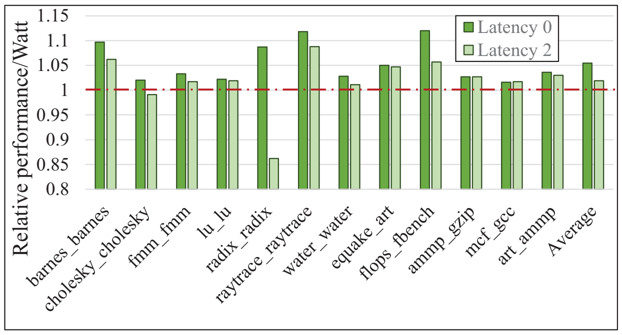

Fig. 9. The system level relative performance-per-Watt of the big and small dual-core in the shared resource configuration for shared resource access latencies of 0 and 2 cycles.

access latency ranging from 0 to 2 cycles. Resource sharing also enhances performance-per-Watt-per-area by as much as 18%, further demonstrating the benefits of such an architecture.

References

[1] R. Dolbeau and A. Seznec, "Cash: Revisiting hardware sharing in single-chip parallel processor," Tech. Rep., 2002.

[2] R. Kumar, N. P. Jouppi, and D. M. Tullsen, "Conjoined-core chip multiprocessing," in *Proceedings of the 37th annual IEEE/ACM Intern.l Symp. on Microarchitecture*, ser. MICRO 37, 2004, pp. 195–206.

[3] M. Butler, L. Barnes, D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," vol. 31, no. 2, 2011, pp. 6–15.

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

[5] R. Kumar *et al.*, "Single-isa heterogeneous multi-core architectures: the potential for processor power reduction," in *MICRO-36. Proceedings. 36th Annual IEEE/ACM Intern. Symp. on Microarchitecture*, Dec. 2003.

[6] 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.

[7] 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.

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

[9] D. M. Tullsen *et al.*, "Simultaneous multithreading: maximizing on-chip parallelism," *SIGARCH Comput. Archit. News*, vol. 23, no. 2.

[10] V. Cakarevic *et al.*, "Characterizing the resource-sharing levels in the ultrasparc t2 processor," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 2009, pp. 481–492.

[11] H. Levy *et al.*, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Computer Architecture, 1996 23rd Annual Intern.l Symp. on*, May 1996, p. 191.

[12] Y. Watanabe *et al.*, "Widget: Wisconsin decoupled grid execution tiles," in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA '10, 2010, pp. 2–13.

[13] D. Borodin *et al.*, "Functional unit sharing between stacked processors in 3d integrated systems," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, July 2011, pp. 311 –317.

[14] H. Homayoun *et al.*, "Dynamically heterogeneous cores through 3d resource pooling," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, ser. HPCA '12, 2012, pp. 1–12.

[15] 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.

[16] S. Li *et al.*, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42. 42nd Annual IEEE/ACM Intern. Symp. on Microarchitecture*, 2009, pp. 469–480.

[17] R. Rodrigues *et al.*, "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.

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

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

[21] S. C. Woo *et al.*, "The splash-2 programs: characterization and methodological considerations," *SIGARCH Comput. Archit. News*, vol. 23, no. 2.

[22] SPEC2000, "The standard performance evaluation corporation (spec cpi2000 suite)."