Does the Sharing of Execution Units Improve Performance/Power of Multicores?

RANCE RODRIGUES, ISRAEL KOREN AND SANDIP KUNDU, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst

Several studies and recent real world designs have promoted sharing of underutilized resources between cores in a multicore processor to achieve better performance/power. It has been argued that when utilization of such resources is low, sharing has negligible impact on performance, while offering considerable area and power benefits . In this paper we investigate the performance and performance/Watt implications of sharing large and underutilized resources between pairs of cores in a multicore. We first study sharing of the entire floating-point datapath (including reservation stations and execution units) by two cores, similar to AMD's Bulldozer. We find that while this architecture results in power savings, for certain workload combinations, it also results in significant performance loss of about 28%. Next, we study an alternative sharing architecture where only the floating-point execution units are shared, while the individual cores retain their reservation stations. This reduces the performance loss to 14%. We then extend the study to include sharing of other large execution units that are used infrequently, namely the integer multiply and divide units. Subsequently, we analyze the impact of sharing hardware resources in Simultaneously Multi-Threaded (SMT) processors where multiple threads run concurrently on the same core. It is observed that sharing improves performance/Watt at a negligible performance cost only if the shared units have high throughput. Sharing low throughput units reduces both performance and performance/Watt. To increase the throughput of the shared units we propose the use of Dynamic Voltage and Frequency Boosting (DVFB) of only the shared units that can be placed on a separate voltage island. Our results indicate that the use of DVFB improves both performance and performance/Watt by as much as 22% and 10%, respectively.

Categories and Subject Descriptors: C.1.3 [**Other Architecture Styles**]: Adaptable architectures

General Terms: Algorithms, Design, Experimentation, Management, Performance

Additional Key Words and Phrases: Multicore processors, resource sharing, Shared floating-point Execution Units (S_FP_X), Shared floating-point Queue and Execution Units (S_FP_QX), Shared Large Execution Units (S_FP_INT), Voltage Frequency Islands (VFI), Nominal Mode (NM), High Frequency Mode (HFM), High Voltage and Frequency Mode (HVFM), Dynamic Frequency Boosting (DFB), Dynamic Voltage and Frequency Boosting (DVFB)

ACM Reference Format:
Rance Rodrigues, Israel Koren, Sandip Kundu, 2013. Does the Sharing of Execution Units Improve Performance/Power of Multicores? *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Several studies have promoted sharing of large but underutilized resources between cores in a multicore processor [Dolbeau and Seznec 2002; Kumar et al. 2004; Butler

Submitted to Special Issue on Run-Time Management.
Author's addresses: R.Rodrigues, I.Koren and S.Kundu are with the Department of Electrical and Computer Engineering (Current address) 151 Holdsworth way, 306 Knowles Engineering Building, Amherst MA 01003.

DOI: http://dx.doi.org/10.1145/0000000.0000000

et al. 2011] to reduce the silicon area at a marginal loss of performance. For example, AMD in its BullDozer architecture [Butler et al. 2011] has implemented sharing of the entire floating-point unit including reservation stations and execution units. Several research publications, e.g., [Dolbeau and Seznec 2002; Kumar et al. 2004], go beyond FP units and also suggest sharing of caches, crossbars, branch predictors and large latency units. Most previous work only explores the performance impact of such sharing leaving the following questions unanswered.

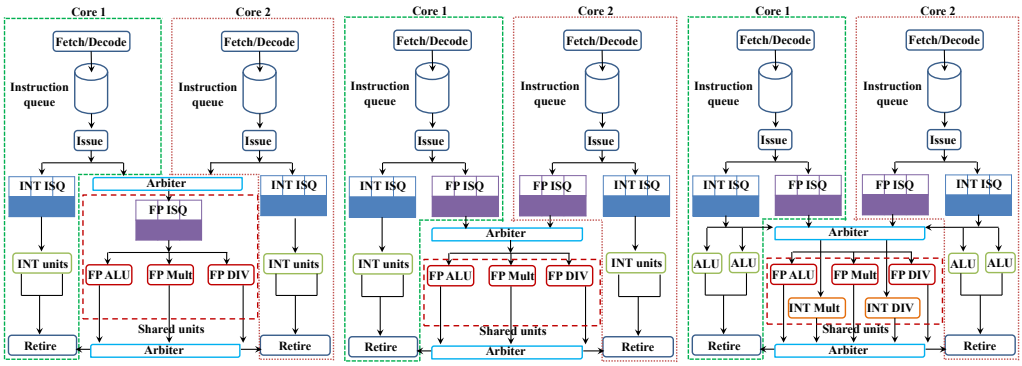
(1) What is the impact of sharing on performance and performance/power? While sharing clearly results in power savings, for certain workloads, performance loss may be too large.
(2) What are the most important parameters influencing performance and performance/power in sharing? We show that latency and throughput of the shared resources are dominant determinants of performance and performance/power, but most previous studies ignore them.
(3) How does sharing of resources play out for Big cores or Small cores? Mainstream computing can be broadly classified into performance efficient (Big cores) and power efficient (Small cores). It is thus necessary to study the impact of sharing resources in both such architectures.
(4) What is the impact of sharing in Simultaneously Multi-Threaded (SMT) processors? In particular, does sharing in SMT make performance or performance/power better or worse? Given that most mainstream cores are SMT capable[1], studying impact of increased resource utilization due to sharing is important.

In this paper, we investigate the performance and performance/Watt implications of sharing large and underutilized resources between a pair of cores in a multicore processor. At first, we study sharing of the entire floating-point datapath by two cores, similar to AMD's Bulldozer [Butler et al. 2011], where the issue queue (ISQ) and the FP execution units are shared. Using combination of workloads from various benchmarks, we study both the performance and performance/Watt when compared to the baseline architecture that does not involve sharing. Our findings show that while sharing results in considerable power savings, the performance penalty may be high ($\sim$28%) for certain workload combinations.

To mitigate the impact on performance, while still retaining some of the power benefits of sharing, we limit sharing to the underutilized execution units. For most workloads, FP instructions are not frequently encountered. Hence, we first explore sharing of just the FP execution units, while the individual cores retain their reservation stations. This modification yields higher performance compared to previous schemes. Still, a worst case performance loss of 14% is observed. Integer divide and multiply instructions are also encountered infrequently. Therefore, we extend our study to include the corresponding units. We find that sharing the integer divide and multiply units has only a small impact on both performance and performance/Watt. A summary of the resource sharing options explored in this paper is shown in Figure 1.

The utilization of the shared units depends on the width of the fetch and execution path. Accordingly, we target cores at opposite ends of the power/performance spectrum. On the higher end of the performance scale we consider a superscalar processor analogous in resources to Intel Nehalem/AMD K10 architecture (Big core). At the lower end of the power scale, we consider a processor similar in resources to Intel Atom/AMD Bobcat architecture (Small core). Our study includes both single threaded and SMT processor architectures. We also analyze the sensitivity to communication latency between the cores and the shared units. Our results show that while architectures that

[1]www.intel.com



(a) Sharing the issue queue and FP execution units (b) Sharing the FP execution units (c) Sharing the FP and integer divide and multiply units

Fig. 1. Overview of the studied resource sharing. ISQ = issue queue, FP = floating-point, INT = integer.

share execution units do provide power benefits at a negligible performance penalty (∼5% on average), such benefits hold only when the shared units have low latency and are highly pipelined. Performance and performance/Watt loss are observed for workloads that exhibit high contention for the shared execution units. To reduce the performance loss due to contention we propose to increase the throughput of the shared resources via Dynamic Voltage and Frequency Boosting (DVFB) which is controlled dynamically by the occupancy rate. Our results show that such dynamic boosting not only overcomes losses due to contention, but also results in significant increases in both performance (upto 13%) and performance/Watt (upto 14%), while realizing considerable savings in area (∼ 7-10% per core).

The following are the key contributions of this paper:

(1) We present a study on the performance and performance/Watt implications of three resource sharing alternatives for a dual-core processor.
(2) We study the performance and performance/Watt implications of resource sharing in SMT cores.
(3) We analyze the sensitivity of resource-sharing architectures to latency and performance of the shared resources.
(4) We show that while execution unit sharing has negligible impact on performance and positive impact on performance/Watt for most benchmark combinations, there are cases where resource contention results in a penalty as high as 22%.
(5) We present a Dynamic Voltage and Frequency Boosting (DVFB) scheme for the shared resources to mitigate the impact of resource contention, that not only compensates for the loss, but also increases the performance of most workload combinations.
(6) Finally, we describe a novel hardware-based feedback control mechanism for DVFB that automates the dynamic control process.

The rest of the paper is organized as follows. Recent work on shared resource architectures is reviewed in Section 2. An overview of the studied architecture is described in Section 3. The experimental set-up is presented in Section 4 which is followed by the results on static execution unit sharing in Section **??**. Results on the proposed DVFB are presented in Section 7. Finally, we present conclusions in Section 10.

2. RELATED WORK

The idea of sharing resources for performance or performance/Watt in a multicore has seen several manifestations. Simultaneous Multi-Threading (SMT) [Tullsen et al. 1995; Levy et al. 1996] was introduced more than a decade ago to improve the utilization of resources in microprocessors. In SMT, multiple threads are run on the same

core and threads share and compete for core resources. Dynamic resource sharing occurs naturally in SMT processors. [Dolbeau and Seznec 2002] explore intermediate design points between the 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 [Kumar et al. 2004] where the caches, crossbar and floating-point units were shared. Significant area savings at a minor loss of performance were reported. Both these schemes only focus on performance and do not consider performance/Watt. In addition, the impact of the shared resource access latency, or the effects on SMT processors were not studied.

[Watanabe et al. 2010] explore flexible sharing of a pool of "execution engines" among various processor cores. By ensuring that the producer and immediate consumers are sent to the same engine, efficient usage of the shared units was made possible. Still, each engine requires a queue and other data to keep track of producers and consumers which result in a complex design. In [Borodin et al. 2011], authors propose the sharing of functional units across cores in a 3D stacked die for online testing and/or performance improvement. A similar approach to 3D resource sharing was proposed in [Homayoun et al. 2012] where the Reorder Buffer (ROB), register file, instruction queue and the load/store queues were shared.

Dynamic exchange of execution units between pairs of cores was investigated in [Rodrigues et al. 2011; 2013]. Here, depending on the current workload characteristics, the cores may exchange execution units to maximize performance/Watt. The major advantage of such an architecture is that resource contention between the two cores does not take place but the design of the two cores is complicated. Further, this scheme will always incur the hardware and power overhead of two sets of execution units compared to the single set in our scheme.

The first resource sharing architecture we study is similar to the AMD Bulldozer design [Butler et al. 2011]. In AMD's Bulldozer the fetch, decode and the entire FP execution (reservation stations and execution units) are shared between pairs of cores in a dual-core processor. In our study we also analyze a design that involves the sharing of the FP execution only.

3. SHARED RESOURCE MULTICORE ARCHITECTURE

We now present an overview of the target of our study – the shared resource multicore architecture. Hardware modifications necessary to support such an architecture are also described. A high level view of the studied architectures is shown in Figure 1. We consider the following three resource sharing alternatives.

3.1. Sharing the FP Issue Queue (ISQ) and execution unit (S_FP_QX)

Here the FP ISQ and FP execution units are shared between two cores. This architecture is depicted in Figure 1(a). This architecture is similar to AMD's Bulldozer but note that the Bulldozer design also shares the fetch and decode units. Sharing leads to contention for resources and the first point of contention here (in S_FP_QX) is the FP ISQ. Whenever FP instructions are ready to be scheduled, the control logic first checks to see if there is a slot available in the shared ISQ. Since the ISQ is shared, the number of entries available per core is reduced. Hence, whenever both the cores sharing the ISQ run FP intensive applications, the ISQ is expected to become a bottleneck in the design which may lead to pipeline stalls and performance loss. Another source of stalls is the shared execution units. Just like the ISQ, the effective number of execution units available is reduced in the dual-core architecture. Hence, a higher number of stalls is expected when FP intensive applications are run on the two cores that share the FP units.

Table I. Chosen core parameters

Parameter	Small	Big	Parameter	Small	Big	Parameter	Small	Big
Issue	2	4	INTREG	64	96	FPREG	64	80
INTISQ	16	36	FPISQ	16	24	LS units	1	3
LSQ	32	32	ROB	56	128	L1(I/D)	32K	32K
L2	512K	2M	Freq (GHz)	1.5	2.4	Type	OOO	OOO

Table II. Execution unit specifications for the cores. (P - Pipelined, NP - Not pipelined, PP - Partially pipelined, cyc = cycles)

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

Table III. Workloads considered for the experiments where each core runs only a single thread.

barnes_barnes	cholesky_cholesky	fmm_fmm	lu_lu	radix_radix	raytrace_raytrace
water_water	flops_fbench	equake_art	gzip_ammp	art_ammp	mcf_gcc

3.2. Sharing the FP execution unit only (S_FP_X)

In this instantiation, sharing of FP execution units only is explored and this architecture is shown in Figure 1(b). Unlike the previous case, the only source of contention here is the availability of the FP execution units. Hence, we expect a lower performance loss but also lower power savings compared to the previous scheme.

3.3. Sharing the FP execution units as well as the integer divide and multiply units (S_FP_INT)

In this instantiation, in addition to the FP execution unit, integer divide and multiply units are also shared. This architecture is shown in Figure 1(c). The number of stalls for this scheme is expected to be higher than for the S_FP_X architecture but greater power savings is expected.

Since resources are shared in all three architectures, there is a need for a centralized control mechanism that will grant access to the requester core. This is accomplished by means of an arbiter shown in Figure 1. The arbiter accepts requests and depending on the availability of the shared resource, grants access. Note that in all three cases, accesses to the shared execution units are independent and hence multiple requests may be sent to them at the same time. Once execution is complete, the execution result must be forwarded to the core that generated the request. This is accomplished by another arbiter that forwards the result to the rightful owner. We do not provide implementation details of the arbiter, which is fairly straightforward.

4. EXPERIMENTAL SETUP

To evaluate the idea of sharing infrequently used execution units for a wide variety of architectures, we considered processor cores at the two ends of the performance/power spectrum, i.e., a high-performance core (Big) and a low-power core (Small). These cores are representative of the Intel Nehalem/AMD K10 and the Intel Atom/AMD Bobcat architectures, respectively. In the rest of this paper, we will refer to them as Big and Small. Note that Big/Small refers to homogeneous dual-core processors.

In Tables I and II we describe the resource sizes and execution resource characteristics for the the two core types. The parameters were inspired by commercial architectures [Fog 2012].

SESC was used for architectural performance simulation [Renau 2005]. We made significant modifications to the simulator to enable shared resource execution with arbitration. Power was estimated using Wattch [Brooks et al. 2000] and Cacti [Shivakumar et al. 2001]. In the experiments we targeted 15 benchmarks: 7 from the SPLASH-2 [Woo et al. 1995] (*barnes, cholesky, fmm, lu, radix, raytrace, water*) and 8 from the SPEC 2000 benchmark suite [SPEC2000] (*fbench, flops, art, equake, gzip, ammp, mcf,*

Table IV. Workloads considered for the experiments where each core runs two threads. The + sign between workloads indicates that they are run on the same core and the _ is used as separator to indicate what is run on cores 1 and 2.

barnes+barnes_barnes+barnes	cholesky+cholesky_ cholesky+cholesky	fmm+fmm_fmm+fmm
lu+lu_lu+lu	radix+radix_radix+radix	raytrace+raytrace_raytrace+raytrace
water+water_water+water	equake+art_flops+fbench	mcf+gcc_art+ammp
equake+art_gzip+ammp	mcf+gcc_flops+fbench	equake+flops_art+fbench
mcf+art_gcc+ammp	equake+gzip_art+ammp	mcf+flops_gcc+fbench

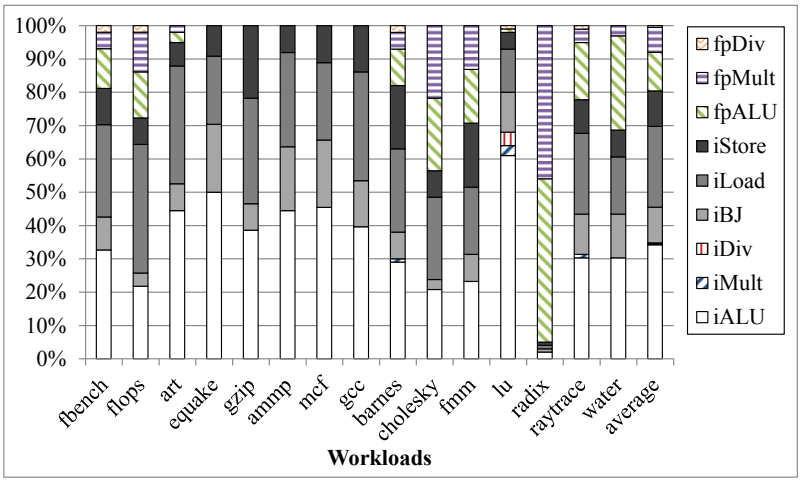


Fig. 2. The instruction distribution of the various workloads when run for 500 million instructions. The average over all workloads is also shown.

gcc). These workloads were chosen for their instruction distribution and performance diversity. Several combinations of workloads were considered for the two cores running single threads. We also considered the case of SMT, where each core runs two threads from a set of four. Homogeneous workload combinations were created using multiple threads from the SPLASH-2 workloads. We also created heterogeneous workloads by combining threads from the SPEC 2000 suite. The created workload sets are summarized in Tables III and IV for the single and SMT experiments, respectively. We thus tried to evaluate the studied architectures over a broad spectrum of potential workloads. Each workload was run until the sum of the instructions retired on the two core types equaled 500 million instructions. The instruction distribution of each individual thread run is shown in Figure 2.

5. ANALYSIS OF RESOURCE SHARING IN SINGLE THREADED PROCESSORS

We first present results and analysis for processors running single threads per core. Two cores share resources according to S_FP_QX, S_FP_X and S_FP_INT schemes described in Section 3. The workloads run in these experiments are shown in Table III. The performance and performance/Watt of the studied architectures relative to the one where no sharing takes place are presented. Sensitivity to the shared resource access latency is also analyzed. In the next section we study the effect of sharing in SMT processors where more than one thread runs on the same core. To compare the resource sharing architectures with the one that does not, three speedup metrics were used including the weighted, geometric and harmonic speed-up metrics. For the sake of brevity, only the results using the harmonic metric are presented. This metric also happens to be the most conservative of the three.

The harmonic speed-up metric for performance is calculated as follows:

$S_0 = (IPC_{thread0})_{new}/(IPC_{thread0})_{baseline}$
$S_1 = (IPC_{thread1})_{new}/(IPC_{thread1})_{baseline}$

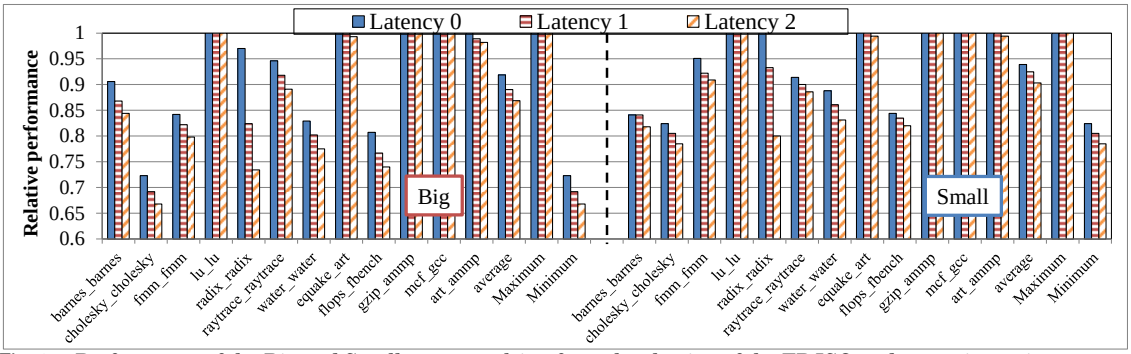


Fig. 3. Performance of the Big and Small cores resulting from the sharing of the FP ISQ and execution units (S_FP_QX) between the cores relative to a dual-core that does not share them for various communication latencies (between zero to two cycles).

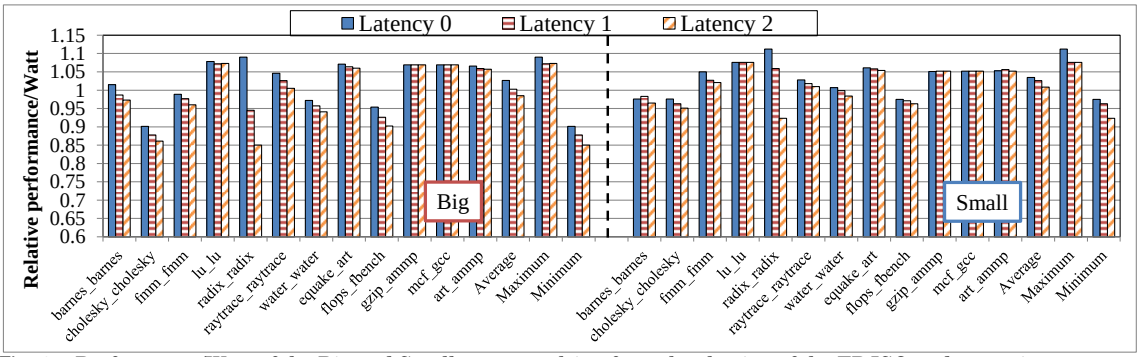


Fig. 4. Performance/Watt of the Big and Small cores resulting from the sharing of the FP ISQ and execution units (S_FP_QX) between the cores relative to a dual-core that does not share them for different (zero to two cycles) communication latencies between the cores and the shared units.

$Speedup_{harmonic} = 2/(1/S_0 + 1/S_1)$

Here, baseline refers to the case where the cores do not share any unit. The performance/Watt speedup/slowdown is calculated similarly.

5.1. Sharing the FP ISQ and execution units (S_FP_QX)

5.1.1. Performance analysis. The performance of the Big and Small cores in the S_FP_QX configuration relative to the non-sharing architecture is shown in Figure 3. Shared resource access latencies of zero, one and two cycles were considered. The communication latency of zero cycles represents the ideal case where the design has been optimized to support sharing. It can be seen that even in this scenario, a significant performance loss is observed for both core types. Specifically, a worst case performance penalty of 28% and 18% (workload *cholesky_cholesky* when run on both the cores) is observed for the Big and Small cores, respectively. This architecture shares the FP ISQ and the FP execution units. Thus, two potential bottlenecks exist in the system yielding a large performance penalty. Increasing the communication latency results in an even larger performance penalty, as expected. This clearly shows the sensitivity of such a resource sharing architecture to communication latency. On an average, $\sim$5-10% performance penalty is observed for both the core types which increases with access latency. These results show that when sharing resources between cores, special consideration must be given to the resource access latency. The workloads that do not experience a slowdown are the ones with little or no FP instructions in the mix (e.g., *equake, art, gzip, gcc*). Interestingly, the Small core does not suffer as much as the Big

core with respect to performance. The Small core is moderately sized when compared to the Big core and consequently, the experienced bottleneck has a greater effect in the case of the Big core.

5.1.2. Performance/Watt analysis. The performance/Watt resulting from the sharing of the FP ISQ and the FP execution units (S_FP_QX) relative to the non-sharing architecture is shown in Figure 4 for both the core types. It can be seen that performance/Watt improvements are achieved for most workloads on both the core types, especially for the ones with no FP instructions. In general, FP instructions are not as frequently encountered as integer ones and hence, for a majority of the workloads this architecture will result in power savings. However, there are workloads where the performance/Watt degrades by as much as 10% (e.g., *cholesky_cholesky* when run on the Big core) even with access latency of zero cycles. This indicates that even though, in general, this architecture results in power savings, for workloads that contest for the shared resources, the performance/Watt will degrade. On an average, a 2.5% improvement for the Big core and a 3.5% improvement for the Small core were observed when the access latency was set to zero cycles. Increased latency reduces this improvement.

Even though the S_FP_QX architecture results in power savings in general, the experienced performance penalty can be very large ($\sim$28%). This results in poor performance/Watt and hence, we explored alternative sharing schemes to help mitigate the performance penalty.

5.2. Sharing only the FP units (S_FP_X)

5.2.1. Performance analysis. The performance of the S_FP_X architecture relative to the one where each core has its own execution units for the Big and Small cores are shown in Figure 5 for the various workloads considered. For zero cycle access latency, it can be seen that for all the workloads, there is no notable performance penalty for the Big core. Even for cases where both threads highly utilize the shared units, no performance penalty was observed (e.g., *cholesky_cholesky, radix_radix, flops_fbench*). This is because the Big core has large and fast execution units that are fully pipelined and unless contention takes place in the same cycle, no performance penalty will be experienced. This indicates that for a high performance core, contention related performance loss will rarely be a problem when the considered execution units are shared even when running workloads that include a large proportion of instructions that need the shared units. The worst case performance penalty has dropped to lower than 1% for the Big core, which is a significant improvement when compared to the S_FP_QX architecture ($\sim$28% performance loss in the worst case). This shows that in the Big core, the major bottleneck is the FP ISQ. Increasing its size may help mitigate the performance penalty but may result in power increase. However, such an analysis is out of the scope of this paper and is not presented. With an increase in access latency, there is a notable drop in performance. Still, for small latencies (one to two cycles), the performance penalty is well within reasonable limits (within 5% even for access latency of two cycles). Note that access latency of zero to one cycles is realistic. A similar assumption has been made in [Dolbeau and Seznec 2002; Kumar et al. 2004; Gupta et al. 2008]. Hence, for cores such as the Big core, for small shared resource communication latencies, the performance loss is acceptable if FP execution units are shared between pairs of cores. This is mainly attributed to the highly pipelined and low latency execution units.

The results obtained for the Small core do show notable performance penalty, even for the ideal case of zero access latency. This happens due to non-pipelined and relatively higher latency execution units present in the Small core (see Table II). Since not all the execution units are pipelined, there is greater chance for contention for the

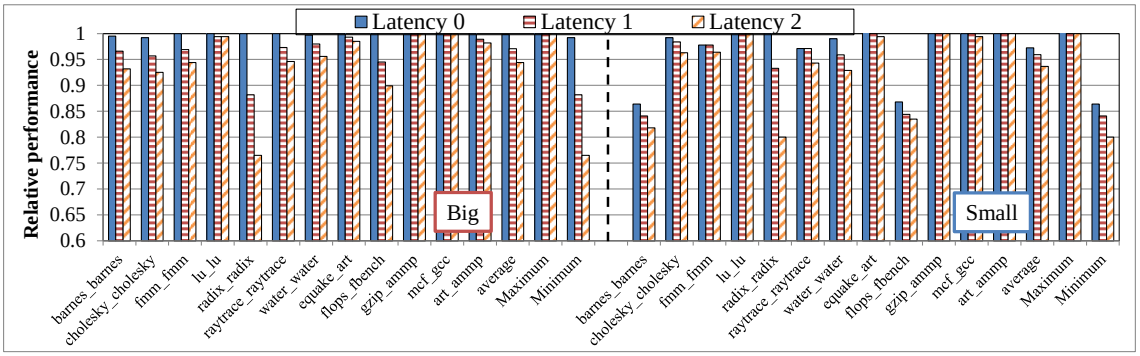


Fig. 5. Performance of the Big and Small cores due to sharing of the FP execution units (S_FP_X) relative to a dual-core that does not share them, for different communication latencies. The different bars correspond to various round-trip communication latencies (zero to two cycles) between the cores and the shared units.

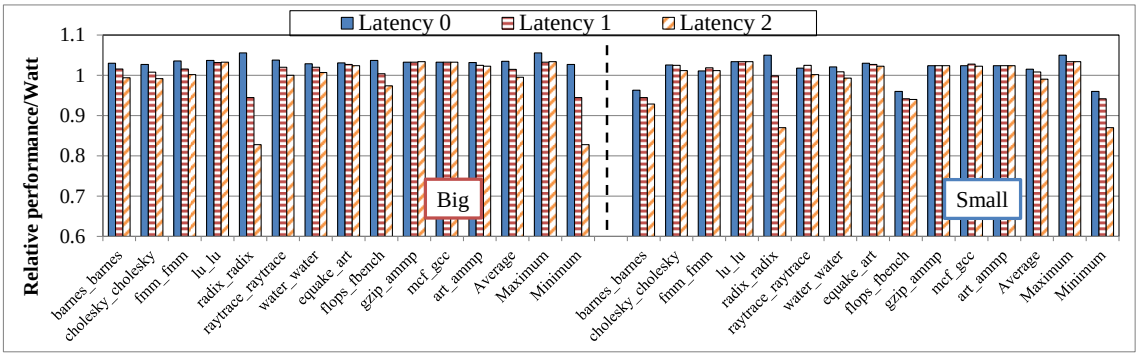


Fig. 6. Performance/Watt of the Big and Small cores due to sharing of the FP execution units (S_FP_X) relative to a dual-core that does not share them, for different communication latencies. The different bars correspond to various round-trip communication latencies (zero to two cycles) between the cores and the shared units.

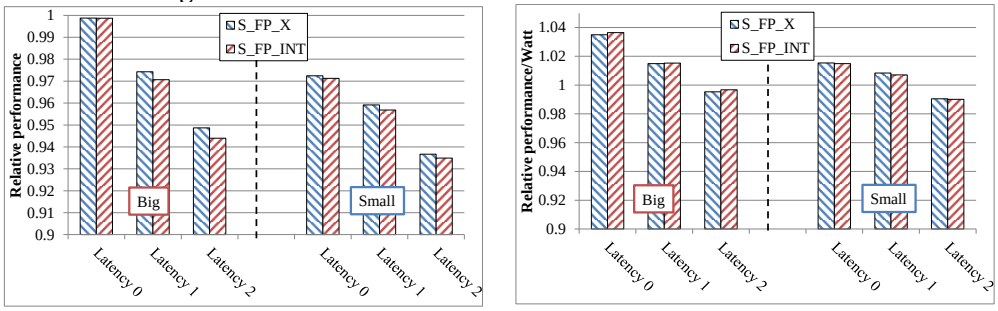
shared units. For example, for a non-pipelined multiplier with latency of 10 cycles, the execution unit cannot accept any more requests during the 10 cycles that follow this request. If this unit was pipelined, the next request to the same unit would be accepted one cycle later and hence there is a smaller chance for contention. In particular, the performance loss is the worst for *barnes_barnes* and *flops_fbench* (13-14%). In both these cases, the workloads running on each core exhibit significant proportion of FP instructions and as a result contention is very high for the shared resources. The average performance loss is within 8% for a two cycle access latency. It is thus clear that for cores with non-pipelined and large latency execution units, sharing may result in significant performance loss. When compared to the S_FP_QX architecture, for the Small core the average performance loss drops from the observed 7% (for a zero cycle communication latency) to around 3%. Hence, this architecture certainly results in lower performance penalty.

5.2.2. Performance/Watt analysis. Sharing the large and infrequently used execution units results in static power savings. This is expected to improve performance/Watt especially for the cases where no notable performance penalty is observed. However, power savings are not as large as that observed for the S_FP_QX architecture. The performance/Watt results obtained for both core types are shown in Figure 6. We have already seen that for the Big core there is no notable performance loss even for a communication latency of two cycles between the core and the shared units. Performance/Watt improvements of >1 were observed for the Big core with communication

latency of one cycle. It can be concluded that for the Big core, sharing of large execution units results in performance/Watt gains when considering realistic shared resource access latencies.

For the Small core, performance loss due to sharing even in idealized conditions (access latency of zero cycles) results in significant performance loss for several workloads. As a result, performance/Watt gains if any, are very modest with a few workloads experiencing performance/Watt loss. Still, on an average the performance/Watt gains are >1 for zero cycle access latency. on an average. Just like the Big core, increasing this latency to more than one cycle results in overall performance/Watt loss when compared to the baseline architecture. It is important to note that apart from two workloads (*barnes_barnes, flops_fbench*) all other workloads show a small improvement in performance/Watt. From Figure 5, it is observed that apart from those two workloads, there were also others such as *fmm_fmm, raytrace_raytrace* that showed performance loss but when considering performance/Watt, show improvements over the baseline. Hence, execution unit sharing architectures do in general improve performance/Watt.

Based on the results presented in this section, we can conclude that for Big cores, sharing FP execution units results in almost no performance loss but may result in small performance/Watt gains. In contrast, for Small cores, even though there is a small performance/Watt gain for low shared resource access latencies (between the core and the shared units), performance and performance/Watt losses observed for a few workload combinations, make the sharing of FP execution units between such cores questionable. This architecture provides slightly lower performance/Watt than the S_FP_QX architecture without considerable performance penalties which is a significant advantage.



(a) Performance (b) Performance/Watt

Fig. 7. Performance and performance/Watt of the Big core and Small core in S_FP_X and S_FP_INT configurations relative to a dual-core that does not share resources for various communication latencies.

5.3. Extending the sharing to include INT divide and multiply units (S_FP_INT)

Most prior work has explored the sharing of only the FP units between pairs of cores [Dolbeau and Seznec 2002; Kumar et al. 2004]. However, from Figure 2, it can be seen that apart from the workload *lu_lu*, no other workload shows any notable INT divide or multiply instructions. Thus, sharing these units in addition to the FP units, is a natural extension. We call the resulting architecture the **S_FP_INT sharing** architecture. We analyzed such additional sharing and the average results obtained over all workloads when run on each core type modeled as the S_FP_X sharing and S_FP_INT sharing architecture with respect to performance and performance/Watt are plotted in Figures 7(a) and 7(b), respectively. All results are shown relative to the architecture that does not share execution units. In general, it can be seen that for both the core types, with respect to performance, S_FP_X sharing is slightly better than the S_FP_INT sharing architecture and the opposite trend is observed with respect to performance/Watt. However, the differences are too small to prefer one architecture over

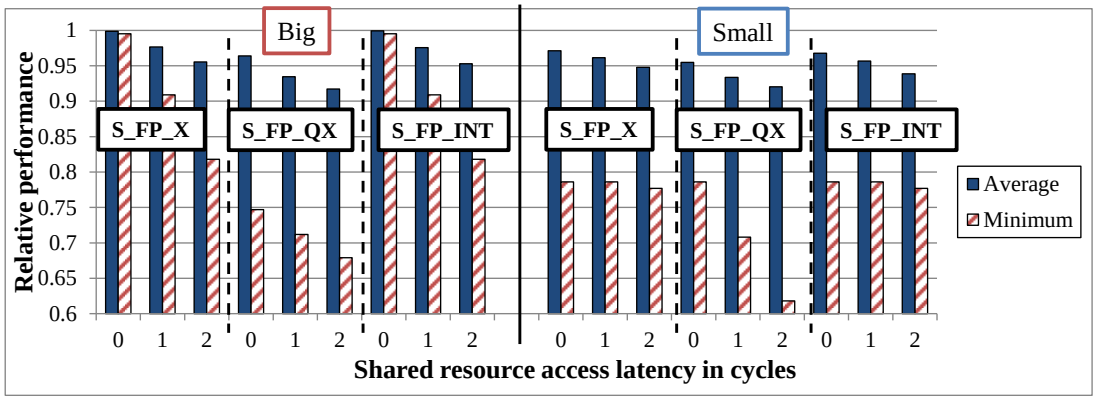


Fig. 8. Performance of the Big and Small cores in the S_FP_QX, S_FP_X, S_FP_INT configurations relative to the baseline for various communication latencies. Two threads were run on each core.

the other. But since INT divide and multiply are relatively large execution units, sharing them certainly yields area savings (details on area savings to soon follow). Hence, we conclude that S_FP_INT sharing enhances the benefits of S_FP_X sharing architectures.

6. ANALYSIS OF SHARING IN SMT PROCESSORS

We now present results on the effect of sharing resources in SMT processors. In these experiments, each core runs two threads. The various workload combinations considered are shown in Table IV. For the sake of brevity, only average and minimum speedup over all the considered workloads for each of the three resource sharing architectures relative to the baseline (where no sharing is implemented) are presented.

6.1. Performance analysis

The average and minimum performance achieved by the three resource sharing architectures relative to the one with no sharing is shown in Figure 8.

6.1.1. The S_FP_X and S_FP_INT architectures. In general, we found that the architectures that only share execution units result in more or less the same level of performance for both the Big and Small core types. Hence, we discuss both these architectures in this sub-section.

For the Big core, ignoring communication latency, a 1% performance loss is observed in the worst case and an even smaller penalty is seen on an average. This result is similar to that observed when running only a single thread per core. This indicates that even when up to four threads compete for the execution resources of the Big core, limited performance penalty will be experienced, which is mainly attributed to the large and fully pipelined execution units.

For the Small core, a larger performance penalty was observed when compared to the Big core. The worst case of 22% performance loss was observed for the workload *barnes+barnes_barnes+barnes* which constitutes an increase of 8% over the observed 14% when running the workload *barnes_barnes* in the earlier experiments. There were also some low IPC workloads such as *raytrace+raytrace_raytrace+raytrace* where performance penalty was smaller than that obtained while running *raytrace_raytrace*. For such workload combinations, stalls in the execution core mitigates the impact on performance of resource sharing. On an average, a 3% performance penalty was observed for the Small core.

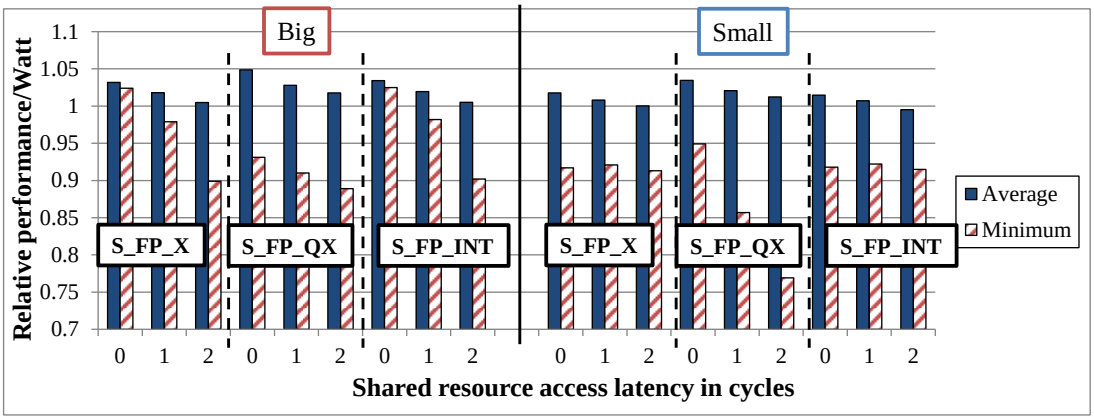


Fig. 9. Performance/Watt of the Big and Small cores in the S_FP_QX, S_FP_X and S_FP_INT configurations relative to a dual-core that does not share resources for various communication latencies. Two threads were run on each core.

In summary, we find that even in SMT processors, sharing execution resources between cores is expected to result in negligible performance penalty in Big cores and sometimes a notable performance penalty in Small cores.

6.1.2. S_FP_QX. From Figure 8 it is clear that the S_FP_QX architecture results in a larger performance penalty than S_FP_X and S_FP_INT for both core types. Ignoring access latency, we have observed that an average performance loss of 4% and 5% and a worst case loss of 22% and 25% were observed for the Big and Small cores, respectively. This performance loss increases with an increase in access latency as expected.

On the Big core, in the single threaded experiments, the workload *cholesky_cholesky* experienced the worst case of 28% performance loss. The loss was reduced to 16% when running the workload *cholesky+cholesky_cholesky+cholesky* in SMT mode. The reason for this drop in penalty is that in SMT mode, a system IPC of 0.35 was observed, which was a drop from the observed IPC of 0.5 in the single threaded experiments. Thus, additional stalls due to resource sharing does not have a high impact on the performance. A worst case performance loss of 25% was observed for the workload *water+water_water+water*. This constitutes a 8% increase in the observed 17% performance penalty when running the workload *water_water*. Hence, for the workload *water*, increasing the number of thread contexts per core results in an increased penalty for the Big cores. The performance loss is higher by 4% on average when compared to the S_FP_X and S_FP_INT architectures for the Big core.

For the Small core, just as for the S_FP_X and S_FP_INT architectures, the worst case was observed for the workload *barnes+barnes_barnes+barnes*. Another workload that exhibited a significant (17%) performance penalty was *radix+radix_radix+radix*. No performance penalty was observed for the same workload when running on the S_FP_X and S_FP_INT architectures. This workload suffers mainly from stalls in acquiring reservation station slots on the small core. Overall, the performance loss goes up by 2% on an average when compared to the S_FP_X and S_FP_INT for the Small core.

In summary, performance is expected to degrade for a few workloads in either of the sharing architectures. For the Big core, performance penalty is expected only in the S_FP_QX design. When compared to the experiments where only single threads were run on each core, performance penalty may sometimes be lower for SMT processors. The reason for this is that in SMT mode resource utilization is higher. Hence, if IPC is low, performance penalty due to sharing is also low.

6.2. Performance/Watt analysis

The performance/Watt of the various resource sharing architectures relative to the one with no sharing is shown in Figure 9.

6.2.1. S_FP_X and S_FP_INT. We have seen that for these architectures, little or no performance penalty was observed on the Big core. Consequently, power savings that result from sharing resources lead to performance/Watt gains. Such gains drop with an increase in the shared resource access latency. On an average, a performance/Watt gain of 3.1% and 3.5% were observed for the S_FP_X and S_FP_INT designs on the Big core. On the Small core we observed a significant performance penalty for some workloads. A worst case performance/Watt loss of 8% was observed for the workload *barnes+barnes_barnes+barnes*. However, on an average, a small performance/Watt gain of around 1.7% and 1.4% is observed for the S_FP_X and S_FP_INT architectures, on Small cores. Note that the performance/Watt gain does not drop below 1 for either configuration on both the Big and Small cores, even with a two cycle access latency.

6.2.2. S_FP_QX. In general, performance loss on this architecture was larger than for the S_FP_X and S_FP_INT architectures. However, the power savings were far greater. Hence, even though the worst case performance/Watt loss of 8% was observed on the Big cores, an average gain of 5% and a maximum gain of 11% were observed for the workload *radix+radix_radix+radix*.

A similar result was observed on the Small core, where an average performance/Watt gain of 3.5% and a maximum gain of 7% were observed for the workload *raytrace+raytrace_raytrace+raytrace*.

In summary, this architecture results in better performance/Watt than the other two on an average. However, certain workload combinations may suffer significantly.

7. MITIGATING THE PERFORMANCE IMPACT OF SHARING LOW THROUGHPUT RESOURCES VIA DYNAMIC BOOSTING

So far, we have observed that some workload combinations experience a significant loss in performance due to resource sharing in Small cores. As indicated earlier, there are two reasons for this performance degradation. The first one is contention for the shared resources and the second reason is access latency between the core and the shared resources. Performance loss due to contention can be mitigated if the shared resources run faster. This may be achieved by replacing the existing high latency execution units by more powerful and small latency units [Dolbeau and Seznec 2002]. However, as was observed in Figures 3 and 5, the performance of most workloads does not degrade by sharing resources. Hence, increasing the strength of the execution units will result in power inefficiency for these workloads. Therefore, we propose the use of Dynamic Frequency Boosting (DFB) or Dynamic Voltage and Frequency Boosting (DVFB) where, depending on the currently executing workload characteristics, the voltage and/or frequency of only the shared execution units is increased. We only consider boosting of the shared execution units and not the shared ISQ in the case of the S_FP_QX configuration as accelerating the ISQ is not expected to yield any benefit.

Selective boosting of the shared execution units is achieved via Voltage and Frequency Islands (VFI) [Lackey et al. 2002; Garg et al. 2009; Jang et al. 2010; Semeraro et al. 2002]. In VFI designs, part of the processor core is operated at one voltage and/or frequency, while another part may be operated at a different voltage and/or frequency. For example, Ghosh *et al.* make use of voltage scalable hybrid arithmetic units in [Ghosh et al. 2010] for power benefits. Most previous work makes use of this concept for energy savings. Our objective is performance improvement of the shared resources only during periods of resource contention. This may potentially also result

Table V. The voltage and frequency levels considered for the two cores.

	High Voltage and Frequency Mode (HVFM)		High Frequency Mode (HFM)		Nominal Mode (NM)	
Core	Voltage	Frequency	Voltage	Frequency	Voltage	Frequency
Big	1.35V	3.4 GHz	1.1V	3.4 GHz	1.1V	2.4 GHz
Small	1.35V	2.13 GHz	1.1V	2.13 GHz	1.1V	1.5 GHz

in performance/Watt improvement. Given that the shared execution units are already separated from the cores (see Figure 1), placing them in an island is relatively simple. We did not consider full-chip voltage and frequency boosting due to its inherent power inefficiency.

Performance boosting may be achieved by increasing the frequency of the shared units. Often, power is the limiting factor that governs operating frequency. The frequency may be increased as long as package thermal limits are not exceeded and the circuit timing margins are not violated. Since the execution units are shared, increasing their operating frequency results in a much smaller power increase than full-chip boosting. Hence, if the circuits allow increasing the frequency of operation on demand, the implementation is simple. We call this mode the High Frequency Mode (**HFM**). For some circuits, voltage may also need to be increased to meet the timing requirements. We call this mode the High Voltage and Frequency Mode (**HVFM**) and this mode is expected to incur a higher energy penalty. Note that these two modes are mutually exclusive for a given design and are analyzed here for completeness of the evaluation. Either the circuit allows for HFM and HVFM is not needed or vice-versa. Thus, in the shared resource VFI, three modes are considered; the Nominal Mode (**NM**) with nominal voltage and frequency, the High Frequency Mode (HFM) and the High Voltage and Frequency Mode (HVFM). The voltage and frequency levels used for both core types in all the three modes are shown in Table V. These values were obtained from [Eyerman and Eeckhout 2011] and from data available on Intel's turbo boost technology[2,3]. The high frequency modes can potentially mitigate the performance loss due to resource sharing. On the other hand, power overhead is also expected. It is thus necessary to limit the use of these modes to only those instances when the shared resources are overwhelmed.

In order to model the high frequency modes in our experiments, the latency of the shared execution units was reduced proportionally to the gains provided by the increase in frequency. Latencies are set back to the usual values when the system returns to the NM. Cycles are always measured in the units of the NM frequency. Hence, we continue to use performance/Watt as the metric to measure relative speedup even though the shared resource island may switch between NM and HFM/HVFM.

We first present results on performance and performance/Watt when operating the Small cores in the HFM/HVFM throughout the execution. A dynamic scheme to switch between operation modes is then presented. We do not explore boosting the performance of the Big core since the shared execution units in such architectures are not expected to be a bottleneck.

7.1. Static Voltage Frequency Scaling

In this experiment, the shared execution units are always operated in the boosted mode (HFM/HVFM) irrespective of the workload characteristics. Such a scheme will result in increased power dissipation but is an interesting case to study as a potential upper bound on the performance mitigation possible by frequency boosting. The calculated harmonic performance and performance/Watt speedups for all the consid-

[2]http://www.intel.com/content/www/us/en/processors/core/core-i5-processor.html
[3]http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

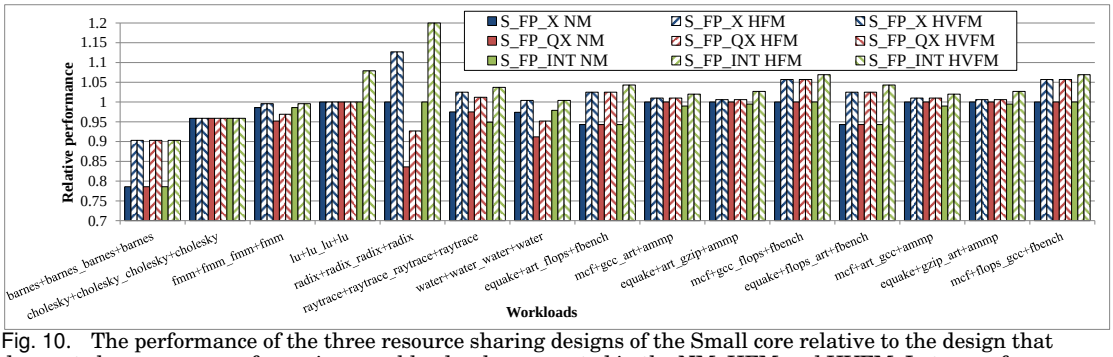


Fig. 10. The performance of the three resource sharing designs of the Small core relative to the design that does not share resources, for various workloads when operated in the NM, HFM and HVFM. Latency of zero cycles was considered.

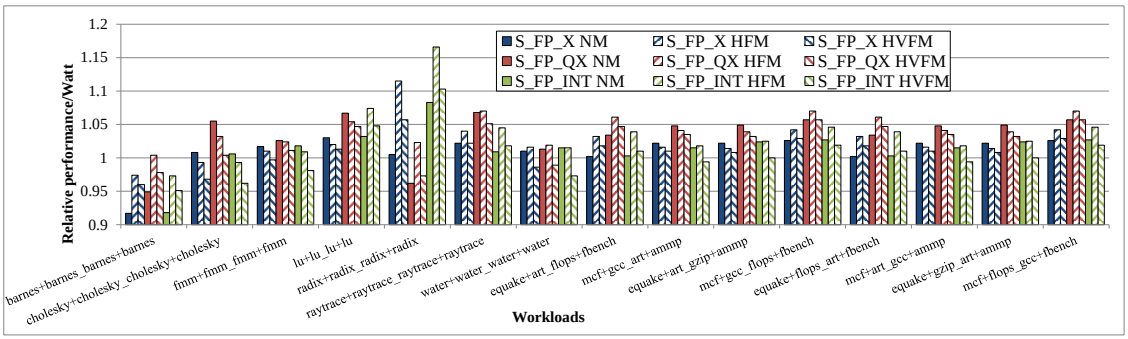


Fig. 11. The performance/Watt of the three resource sharing designs of the Small core relative to the design that does not share resources, for various workloads when operated in the NM, HFM and HVFM. Latency of zero cycles was considered.

ered workloads when executed on small cores in SMT mode for the NM, HFM and HVFM operating modes are shown in Figures 10 and 11, respectively. A shared resource communication latency of zero cycles was considered to get a representative picture without loss of generality.

7.1.1. Performance analysis. It can be seen that the performance is significantly improved in the boosted modes (HFM/HVFM) for several workloads. In particular, the workloads *barnes+barnes_barnes+barnes*, *radix+radix_radix+radix* and any workload running *flops* and *fbench*, show a considerable performance gain (7-20%) in the boosted modes of operation. There are also several workloads such as *cholesky+cholesky_cholesky+cholesky*, *fmm+fmm_fmm+fmm*, *equake+art_gzip+ammp*, and *mcf+gcc_art+ammp* where no notable improvement is observed. There is no difference between the HFM and HVFM modes with respect to performance as is evident from the figures. The boosted modes achieve a 4-5% on an average and a maximum of 20% improvement in performance over the NM mode. Clearly, from a performance stand point operating in the boosted mode is the best option.

7.1.2. Performance/Watt analysis. With respect to performance/Watt, it can be seen that there are workloads that benefit from the HFM/HVFM. Workloads such as *barnes+barnes_barnes+barnes*, *radix+radix_radix+radix* show a 6-7% improvement in performance/Watt. However, there are several workloads where performance/Watt in the NM mode is the highest. These workloads are *cholesky+cholesky_cholesky+cholesky*, *fmm+fmm_fmm+fmm* and workloads containing the combination *equake+art_gzip+ammp* and *mcf+gcc_art+ammp*. These were the

workloads where no notable performance improvement was observed (see Figure 10). Between the HFM and HVFM, the HVFM performs worse which is expected. This mode requires a higher voltage and hence results in larger power penalty than the HFM. These results clearly show that operating in the HFM or HVFM modes throughout execution is not desirable with respect to performance/Watt for several workloads. A dynamic scheme may yield better results.

7.2. Dynamic Voltage and Frequency / Frequency Boosting

A feedback control mechanism is needed in order to determine the best mode to operate in as a function of currently executing workload characteristics. To that end we developed a simple hardware scheme to enable switching between the NM and HFM/HVFM . The shared resources are expected to be a bottleneck whenever the contention for any one of the shared units increases. Occupancy or utilization of the shared execution units can potentially provide a good estimate of whether the bottleneck exists. Hence, we make use of this metric to switch between the NM and the boosted modes of execution.

Performance monitoring counters are available in most modern microprocessors [Contreras and Martonosi 2005; Singh et al. 2009]. For our purposes, we need as many counters as there are shared units to count the number of busy cycles for each execution unit. Whenever the occupancy for any shared unit exceeds a threshold (upper), the boosted mode is enabled. Switching back to the NM takes place when utilization reduces below a threshold (lower). As the occupancy of the execution units changes over time, it is necessary to keep checking for utilization within small intervals. At the end of each interval, all the counters are reset to zero so that counting for the new interval may begin afresh. Furthermore, to avoid too frequent voltage and/or frequency changes, a switch is initiated only if the decision to switch was observed for atleast 90% of the last *HisD* windows, referred to as history depth. For example, considering *HisD* = 10 a switch in operating mode is affected only of the decision to switch was observed for atleast 9 of the 10 recent windows. In the rest of this paper, we refer to the scheme that switches between NM and HFM as Dynamic Frequency Boosting (DFB) and the the scheme that switches between NM and HVFM as Dynamic Voltage Frequency Boosting (DVFB)

A simple illustration of the mechanism to control the mode switching is shown in Figure 12. There is a utilization counter for each shared execution unit. Control logic monitors these counters and accepts as input certain parameters that we call interval length, history depth, threshold upper and lower (soon to be introduced). The utilization for that window is then calculated and depending on the current operating mode of the VFI and the values of the input parameters, a change in operating mode may be affected. Note that utilization (proportion of busy cycles) is always measured with respect to the cycle time of NM. Since the execution units are accelerated in the boosted modes, this effectively reduces the utilization, potentially mitigating the bottleneck. The following four parameters of the dynamic mechanism need to be determined:

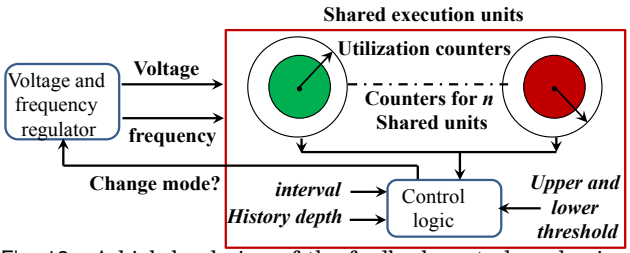


Fig. 12. A high level view of the feedback control mechanism that may be used to control the voltage and frequency of the VFI containing the shared resources.

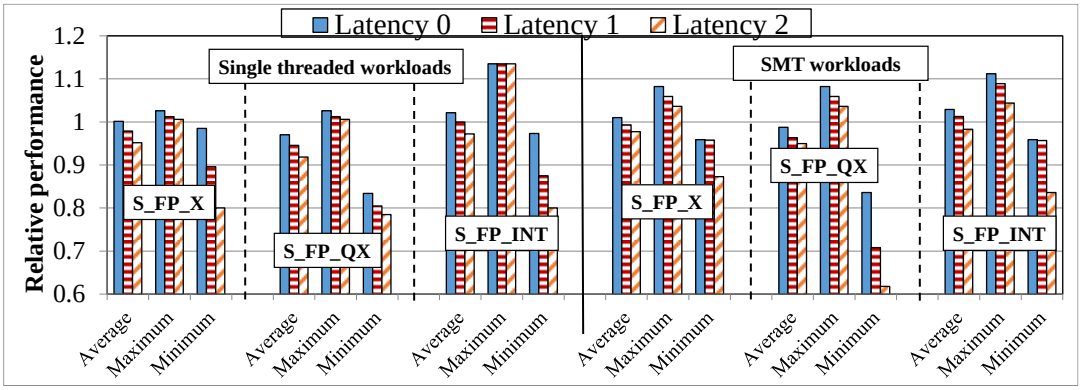


Fig. 13. Relative performance of the Small core in the S_FP_X, S_FP_QX and S_FP_INT configurations for various communication latencies when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

(1) The **window** or **interval length** (*IntLen*) in cycles after which the utilization counters must be sampled. Choosing too small a value may result in noisy behavior, while too large a value may result in missing potential opportunities.
(2) The **number of intervals** to wait until high confidence decisions may be made. This is called the **history depth** (*HisD*). A switch in mode is initiated only if the decision to switch was observed for 90% of the last *HisD* windows. Here as well, choosing too small a depth may result in frequent mode switches while too large a depth may result is missing opportunities to switch mode.
(3) The threshold to enter HFM/HVFM from NM. We call this **Threshold Upper** (*ThU*). A mode switch takes place only when the utilization of one of the shared execution units exceeds the *ThU*.
(4) The threshold to go back into NM from HFM/HVFM. We call this **Threshold Lower** (*ThL*). This mode switch takes place only when the utilization of **all** shared execution units goes below the *ThL*.

We carried an exploratory experiment to find the set of values for the aforementioned parameters. In these experiments, we used the workloads *barnes+barnes_barnes+barnes*, *raytrace+raytrace_raytrace+raytrace* and *equake+art_flops+fbench* for offline training experiments. Based on these experiments, the selected paramters are: *IntLen* = 20, *HisD* = 50, *ThU* = 85%, *ThL* = 50%.

7.3. Performance and performance/Watt analysis when using the proposed DFB/DVFB schemes

We now present the performance and performance/Watt achieved by the resource sharing architectures equipped with DFB and DVFB. Results for the Big core are not shown as the shared execution units were not found to be a bottleneck.

7.3.1. Performance analysis. The average, maximum and minimum relative performance of the DFB scheme over the baseline inwhich no sharing takes place in the S_FP_X. S_FP_QX and S_FP_INT configurations are shown in Figure 13 for communication latencies of zero to two cycles. Results are shown for both single threaded and SMT workloads. A comparison of the relative performance obtained in NM, DFB and DVFB modes are presented in Figure 14.

Considering the single threaded workloads, the worst cases observed in the NM for the S_FP_X and S_FP_INT configurations were for the workloads *barnes_barnes* with

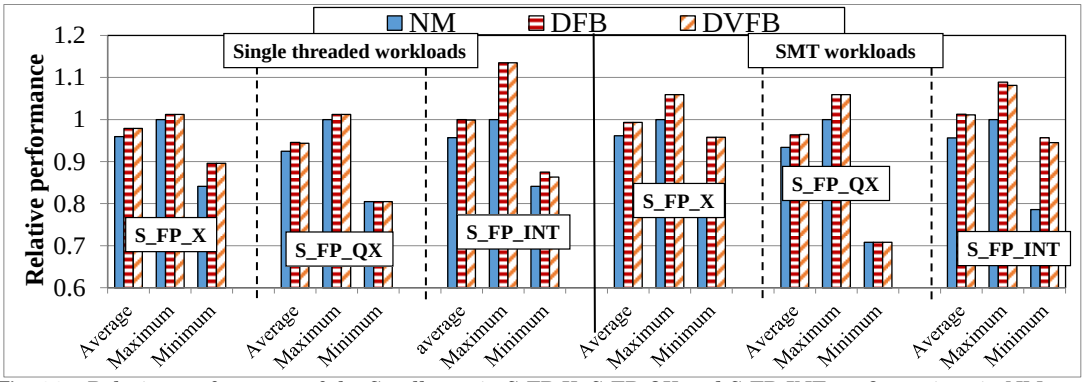


Fig. 14. Relative performance of the Small core in S_FP_X, S_FP_QX and S_FP_INT configurations in NM, DFB and DVFB for communication latency of one cycle when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

relative performance of 0.86 and *flops_fbench* with relative performance of 0.87. The performance of these workloads was significantly increased by 13-15% with an observed relative performance of 0.99 and 1.026 for these two workloads, respectively. On an average, performance was boosted by 3% for the S_FP_X configuration and by 4.5% for the S_FP_INT configuration when compared to the NM. Maximum improvement in performance of 3% and 13% were observed for the S_FP_X and S_FP_INT configurations, respectively over the baseline. There were instances where integer divide and multiply units were bottlenecks for a few workloads (containing *raytrace* or *lu*). Boosting the performance of these units resulted in significant performance gains of as high as 13% for *lu_lu*. For the S_FP_QX configuration, the worst case was observed for *barnes_barnes*, *cholesky_cholesky*, *water_water* and *flops_fbench* with relative performance of 0.84, 0.82, 0.88 and 0.84, respectively. Using DFB, the relative performance of these workloads was increased to 0.96, 0.83, 0.89 and 1.02, respectively, but not all workloads showed such notable improvement. The reason for this is that these workloads suffered more due to stalls in the ISQ and not the execution units. On an average, performance improvement of 4% was observed for the S_FP_QX configuration when compared to the NM. Increasing the latency of the shared resources results in a 2-3% drop in performance demonstrating the sensitivity of these architectures to the shared resource access latency.

With respect to the SMT workloads, for all the three configurations, the workload *barnes+barnes_barnes+barnes* showed worst case relative performance of 0.78. This was boosted to 0.96 in all three configurations representing a 23% improvement in performance. On an average, performance was improved by 4%, 3% and 5% for the S_FP_X, S_FP_QX and S_FP_INT configurations, respectively, relative to the NM. These architectures also compare well against the baseline architecture. The S_FP_X, S_FP_QX and S_FP_INT configurations achieve performance of 1.01, 0.98 and 1.029, respectively, relative to the baseline.

From Figure 14 we note that the benefits of the DFB and DVFB mechanisms are very similar although they differ in the overhead to switch between operating modes (DFB requiring 10 cycles vs. 20 cycles for DVFB).

7.3.2. Performance/Watt analysis. The performance/Watt results are summarized in Figures 15 for the DFB scheme, and in 16 for the NM, DFB and DVFB schemes. Just as was the case with performance, the DFB scheme significantly improves the performance/Watt.

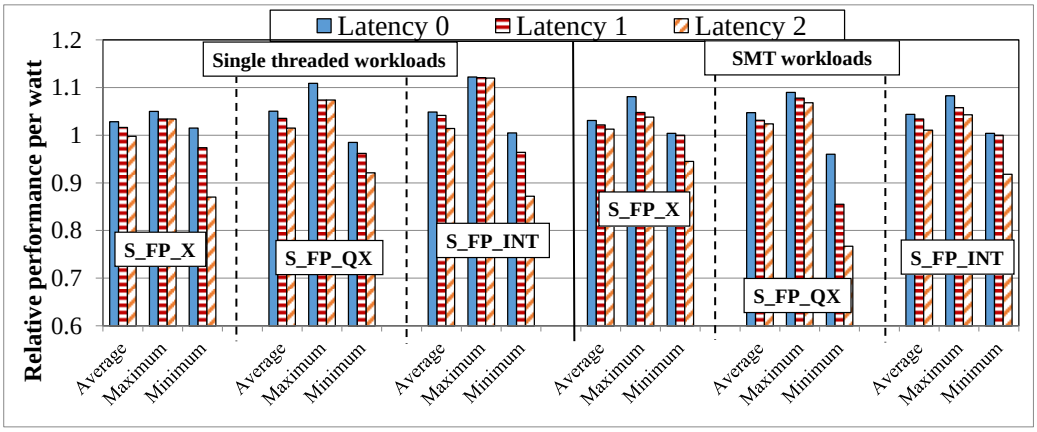


Fig. 15. Relative performance/Watt of the Small core in S_FP_X, S_FP_QX and S_FP_INT configurations for various communication latencies when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

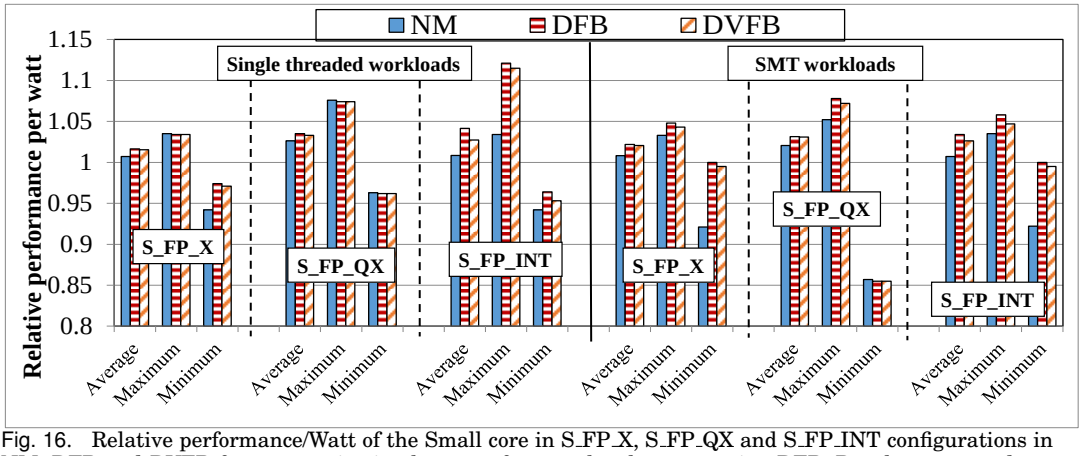


Fig. 16. Relative performance/Watt of the Small core in S_FP_X, S_FP_QX and S_FP_INT configurations in NM, DFB and DVFB for communication latency of one cycle when run using DFB. Results presented are summarized over all workloads for both the single threaded and SMT workloads.

For the single threaded workloads, the worst case workload combinations for the S_FP_X and S_FP_INT configurations were *barnes_barnes* and *flops_fbench* with relative performance/Watt of 0.96. This loss was mitigated with a 5% improvement in performance/Watt in the DFB mode. For the S_FP_QX configuration, the workloads *barnes_barnes*, *cholesky_cholesky* and *flops_fbench* have a relative performance/Watt of around 0.98. Among these, the relative performance of *barnes_barnes* and *flops_fbench* improved to 1.04 and 1.07, respectively, while that of *cholesky_cholesky* was only improved to 0.98. Once again, stalls in the ISQ was the reason for this. Maximum improvements of 5%, 11% and 12% and average improvements of 3%, 5% and 4.5% were observed for the S_FP_X, S_FP_QX and S_FP_INT, respectively, over the baseline. The corresponding average improvements were 2%, 2% and 3% for the S_FP_X, S_FP_QX and S_FP_INT, respectively, over the NM.

For the SMT workloads, worst case relative performance/Watt of 0.91 was observed when running the workload *barnes+barnes_barnes+barnes* on both S_FP_X and S_FP_INT configurations in the NM. This was improved to 1.01 and 1.005, respectively, by the DFB scheme. The worst case for the S_FP_QX was a relative performance/Watt of 0.94 running the same workload in NM. This was improved to 1.039 by running

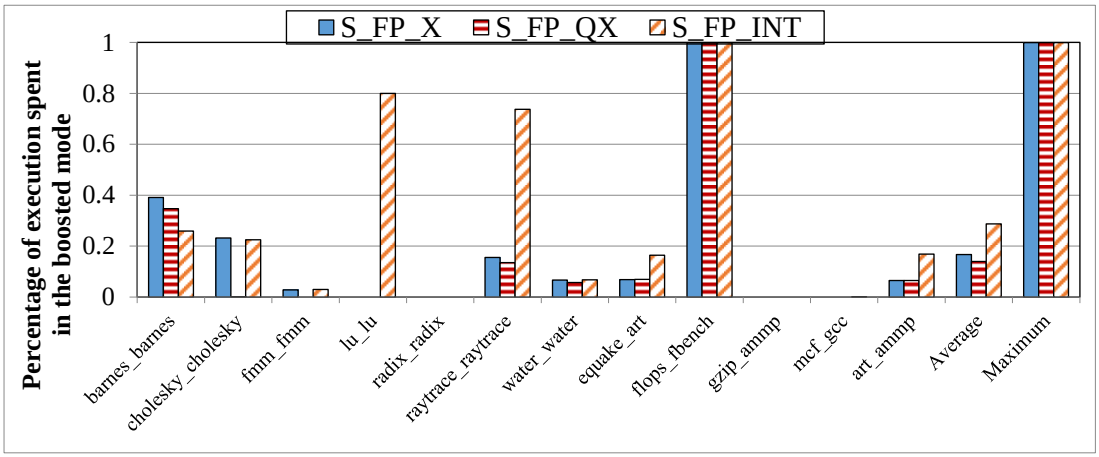


Fig. 17. Proportion of total execution time spent in boosted mode for the S_FP_X, S_FP_QX and S_FP_INT configurations running single threaded workloads. The Small core was run using DFB and communication latency was set to one cycle. The average is also shown.

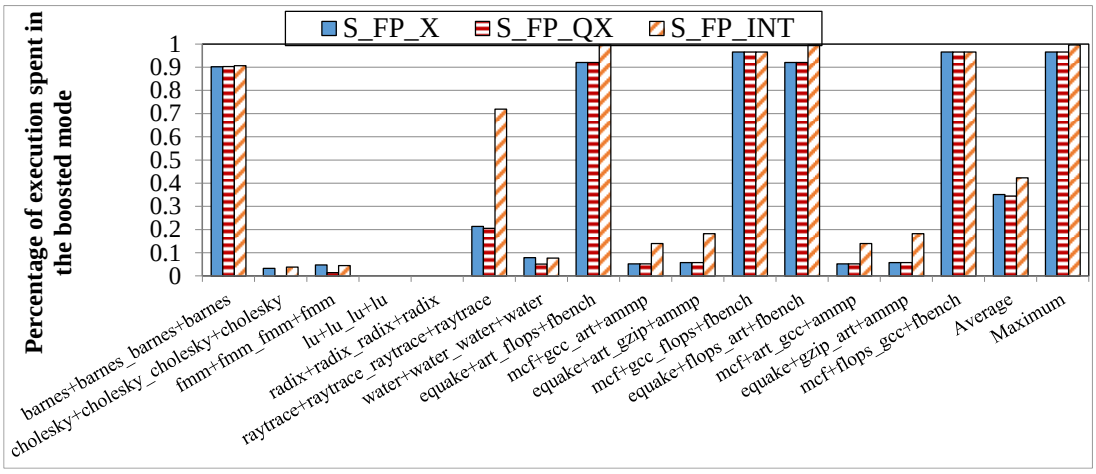


Fig. 18. Proportion of total execution time spent in boosted mode for the S_FP_X, S_FP_QX and S_FP_INT configurations running SMT workloads. The Small core was run using DFB and communication latency was set to one cycle. The average is also shown.

in DFB. A maximum improvement of 8%, 9% and 8.3% and average improvement of 3.1%, 4.7% and 4.3% in performance/Watt were observed for the S_FP_X, S_FP_QX and S_FP_INT, respectively, over the baseline. This yields an average improvement of 2-3% in performance/Watt when compared to the NM.

From Figure 16 we note that the benefits of the DFB and DVFB mechanisms are similar with DFB doing a little better (1-2%) since it does not incur the voltage regulator power overhead.

7.3.3. Percentage of execution time spent in the boosted modes. The boosted modes should not be used all the time. If this is the case, the processor was not properly sized and the results may be biased and misleading. In Figures 17 and 18 the percentage of time spent in the boosted mode in the DFB scheme is shown for the single and SMT workloads, respectively. Results are shown for all three sharing configurations for a shared resource communication latency of one cycle.

For the single threaded workload *flops_fbench*, all the three configurations run in the boosted mode for 100% of the time. This shows that for this workload, the shared execution unit was a severe bottleneck. Other workloads that were executed for most of the time (75-80%) in the boosted mode were *lu_lu* and *raytrace_raytrace* when run in the S_FP_INT configuration. These workloads resulted in contention for the integer multiply and divide operations. The DFB scheme detected this and accordingly operated in the boosted mode. The remaining 9 workloads operate in the boosted mode for 0-40% of the time. On an average, the Small core was operated in the boosted mode for 17-25% of the time for all the three configurations. Similar results were obtained for the DVFB mode.

For the SMT workloads, the number of workloads run in the boosted mode for all three configurations is higher. Nearly 6 of the considered 15 workload combinations run in the boosted mode for 70-100% of the time. In these experiments, contention for the shared resources is higher than that observed when running single threaded workloads. On an average, the Small core was operated in the boosted mode for 32-40% of the time for all three schemes and 9 of the 15 workloads operated in the boosted mode for less than 20% of the time. These results show that while some workloads prefer to run in the boosted mode for longer duration than others, there are also several workloads for which the NM suffices indicating that the target architecture was sized appropriately.

8. IMPLEMENTING THE DYNAMIC BOOSTING MECHANISMS

The proposed DFB and DVFB schemes have shown significant potential to not only mitigate performance loss, but in some cases result in both performance and performance/Watt improvements over the baseline. However, implementing such mechanisms may result in hardware and performance overheads. We now discuss these overheads and present the resulting area overhead in the next sub-section.

8.1. Power overheads

With respect to power, a negligible power overhead is expected for DFB but for DVFB, power is lost during conversion. Assuming that the on-chip voltage regulator has a conversion efficiency of 90% [Kim et al. 2008], 10% of the power is wasted. We have found this power to be around 1% of the total power expended in the processor and is constitutes therefore, a very small overhead. This should be compared to the 12.5% power consumed by the execution units (measured during simulation) in conventional processors where no sharing takes place. Clearly, the overheads are far lower than the benefits provided by the boosting schemes.

8.2. Performance overheads

The dynamic boosting schemes affect a shift in voltage and/or frequency whenever deemed necessary. Two issues arise when employing such a dynamic control: (i) Lost cycles during the transition in voltage and/or frequency, and (ii) synchronization between the VFI's.

8.2.1. Cycles lost during operating mode transition. For the DFB scheme, only few cycles are lost during the frequency transition. IBM's PowerTune technology [Lichtenau et al. 2004] generates multiple frequencies which are selected using multiplexers. The overhead to switch between frequencies was reported to be one cycle. Even if a separate PLL is used to generate the additional frequency, the overhead to transition between the two frequencies is not expected to be significant. We have pessimistically assumed an overhead of 10 cycles for the DFB mode. For the DVFB mode, in addition to frequency transition, a voltage transition is also needed. In [Eyerman and Eeckhout 2011], it is reported that the dV/dT for on-chip voltage regulators is around $20mV/ns$. In our scheme, the cores transition between 1.1 and 1.35V. Hence, the time to tran-

sition between the two voltages is around 12.5ns. Considering that the Small core operates at 1.5GHz, the overhead in cycles for voltage transition is about 20 cycles. Note that during this period, the shared execution units are not accessible to avoid loss of signal integrity. Hence, whenever switches in mode are made, this penalty is always experienced.

8.2.2. Synchronization between the VFI's. Since the VFIs may sometimes operate at different frequencies and/or voltages, this may lead to synchronization problems between the islands possibly leading to loss of cycles. Note that synchronization problems will be avoided if buffers are inserted at the boundary of the two VFI's. In all the considered designs, buffers are already present in the design (ISQ). Furthermore, by making use of certain types of FIFO buffers [Semeraro et al. 2002] any penalty due to synchronization can be completely avoided. Hence, in our experiments, we do not consider any overhead due to synchronization.

9. AREA SAVINGS

In the target architecture, large and infrequently used resources are shared between cores. This certainly results in area savings. Kumar *et al.* in [Kumar et al. 2004] report that the area savings of sharing just the FP units is around 6.1%. Hence, the S_FP_X configuration is expected to result in around 6-7% savings in area per core. In [Shivakumar et al. 2003], Shivakumar *et al.* specify that the area occupied by the INT and FP execution units is approximately 12-13%. In Figure 19, the floorplan of the Intel Nehalem processor is shown[4]. The approximate area occupied by the execution units and the OOO scheduling logic (integer/FP ISQ and ROB) is also shown. The execution units occupy around 18% of the area of the core. Considering that ALUs account for a very small portion of the 18% occupied by the execution units, the S_FP_INT configuration is expected to yield around 8-9% savings in area per core. The OOO logic occupies 14% of the core area and assuming that half of that is occupied by the ROB and the other half by the integer and FP ISQ, the approximate area savings per core for the S_FP_QX configuration is around 9-10%. These savings in area are certainly expected to be considerably larger than the investment in real estate required for controlling access to the shared units.

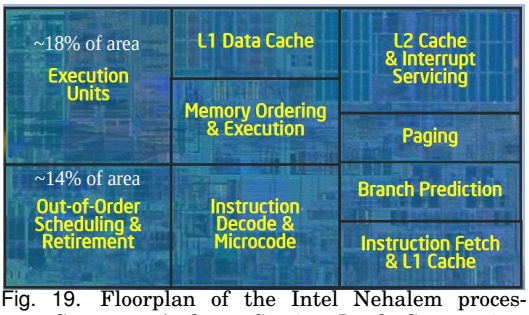


Fig. 19. Floorplan of the Intel Nehalem processor. Courtesy Andrew Semin, Intel Corporation. http://www.notur.no/notur2009/files/semin.pdf.

Next, we estimate the area requirement for an on-die voltage converter. [Hazucha et al. 2005] reports an area of 0.008mm^2 for an output power of 0.1 Watts in 90nm technology. We therefore, estimate an area of 0.16mm^2 ($20X$) for an on-die voltage converter with 2 Watts of output power. Considering that the die area of the Atom processor[5][6] is around 24-26mm^2, the area of the on-chip voltage regulator is negligible compared to the execution core area.

10. CONCLUSIONS

In this paper we have investigated the performance and performance/Watt of multicore processors that share infrequently accessed execution resources. Inspired by the

[4]http://www.notur.no/notur2009/files/semin.pdf
[5]http://vsevteme.ru/attachments/show?content=7591
[6]http://ark.intel.com/products/35635/Intel-Atom-Processor-230-512K-Cache-1_60-GHz-533-MHz-FSB

AMD BullDozer architecture, we studied the impact of sharing the floating-point (FP) execution unit and issue queue between two cores in a dual-core processor. We then expanded the scope of the study by considering a Big core that is akin to Intel Nehalem processor and a Small core that is akin to Intel Atom processor. A variety of multi-programmed and multi-threaded workload combinations were studied in single-threaded and Simultaneously Multi-threaded (SMT) modes. We found that this architecture can sometimes result in significant loss in performance ($\sim$ 28%). To mitigate this performance loss we limited the sharing to just the execution units including FP and integer divide and multiply units. This reduced the performance penalty to 14%. Sensitivity of the performance and performance/Watt of such architectures to shared resource access latency was also investigated. It was found that both performance and performance/Watt are highly sensitive to the shared resource access latency. Our sensitivity study has further indicated that as long as the cores share high throughput execution units, for most of the workloads, a small gain in performance/Watt is achieved at the expense of a small loss in performance. In order to mitigate such loss in performance, a dynamic voltage and frequency boosting (DVFB) scheme has been presented to accelerate execution in the shared resources. Such dynamic boosting was found to completely negate the performance losses and resulted in significant performance/Watt gains. The dynamic scheme improves the performance and performance/Watt of resource sharing architectures by as much as 22% and 10%, respectively. We also observed a performance and performance/Watt improvement of 13% and 14%, respectively, over non-sharing cores. Furthermore, the performance/Watt/area improves by as much as 26.2%, increasing the attractiveness of sharing.

REFERENCES

BORODIN, D. ET AL. 2011. Functional unit sharing between stacked processors in 3d integrated systems. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*. 311 –317.

BROOKS, D. ET AL. 2000. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*.

BUTLER, M., BARNES, L., SARMA, D., AND GELINAS, B. 2011. Bulldozer: An approach to multithreaded compute performance. *Micro, IEEE 31,* 2, 6–15.

CONTRERAS, G. AND MARTONOSI, M. 2005. Power prediction for intel xscale processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on Low power electronics and design*. ISLPED '05. 221–226.

DOLBEAU, R. AND SEZNEC, A. 2002. Cash: Revisiting hardware sharing in single-chip parallel processor. Tech. rep.

EYERMAN, S. AND EECKHOUT, L. 2011. Fine-grained dvfs using on-chip regulators. *ACM Trans. Archit. Code Optim. 8,* 1, 1:1–1:24.

FOG, A. 2012. The microarchitecture of intel, amd and via cpu. Tech. rep., Copenhagen University College of Engineering.

GARG, S. ET AL. 2009. Technology-driven limits on dvfs controllability of multiple voltage-frequency island designs: A system-level perspective. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*. 818 –821.

GHOSH, S. ET AL. 2010. Voltage scalable high-speed robust hybrid arithmetic units using adaptive clocking. *IEEE Trans. Very Large Scale Integr. Syst. 18,* 9, 1301–1309.

GUPTA, S. ET AL. 2008. The stagenet fabric for constructing resilient multicore systems. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*. 141 –151.

HAZUCHA, P., KARNIK, T., BLOECHEL, B., PARSONS, C., FINAN, D., AND BORKAR, S. 2005. Area-efficient linear regulator with ultra-fast load regulation. *Solid-State Circuits, IEEE Journal of 40,* 4, 933–940.

HOMAYOUN, H. ET AL. 2012. Dynamically heterogeneous cores through 3d resource pooling. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*. HPCA '12. 1–12.

JANG, W. ET AL. 2010. Voltage and frequency island optimizations for many-core/networks-on-chip designs. In *Green Circuits and Systems (ICGCS), 2010 International Conference on*. 217 –220.

KIM, W., GUPTA, M., WEI, G.-Y., AND BROOKS, D. 2008. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. 123–134.

KUMAR, R., JOUPPI, N. P., AND TULLSEN, D. M. 2004. Conjoined-core chip multiprocessing. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. IEEE Computer Society, Washington, DC, USA, 195–206.

LACKEY, D. ET AL. 2002. Managing power and performance for system-on-chip designs using voltage islands. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*. 195 – 202.

LEVY, H. ET AL. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Computer Architecture, 1996 23rd Annual International Symposium on*. 191.

LICHTENAU, C., RINGLER, M., PFLUGER, T., GEISSLER, S., HILGENDORF, R., HEASLIP, J., WEISS, U., SANDON, P., ROHRER, N., COHEN, E., AND CANADA, M. 2004. Powertune: advanced frequency and power scaling on 64b powerpc microprocessor. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*. 356–357 Vol.1.

RENAU, J. 2005. Sesc: Superescalar simulator.

RODRIGUES, R. ET AL. 2011. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. 121 –130.

RODRIGUES, R. ET AL. 2013. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Trans. Des. Autom. Electron. Syst. 18,* 1, 5:1–5:23.

SEMERARO, G. ET AL. 2002. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. HPCA '02. 29–.

SHIVAKUMAR, P. ET AL. 2001. Cacti 3.0: An integrated cache timing, power, and area model. Tech. rep.

SHIVAKUMAR, P. ET AL. 2003. Exploiting microarchitectural redundancy for defect tolerance. In *Computer Design, 2003. Proceedings. 21st International Conference on*. 481 – 488.

SINGH, K. ET AL. 2009. Real time power estimation and thread scheduling via performance counters. *SIGARCH Comput. Archit. News 37,* 2, 46–55.

SPEC2000. The standard performance evaluation corporation (spec cpi2000 suite).

TULLSEN, D. M. ET AL. 1995. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News 23,* 2, 392–403.

WATANABE, Y. ET AL. 2010. Widget: Wisconsin decoupled grid execution tiles. In *Proceedings of the 37th annual international symposium on Computer architecture*. ISCA '10. 2–13.

WOO, S. C. ET AL. 1995. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News 23,* 2, 24–36.