Nucleus: Finding the sharing limit of heterogeneous cores

ILIAS VOUGIOUKAS, ARM Research and University of Southampton
ANDREAS SANDBERG and STEPHAN DIESTELHORST, ARM Research
BASHIR M. AL-HASHIMI and GEOFF V. MERRETT, University of Southampton

Heterogeneous multi-processors are designed to bridge the gap between performance and energy efficiency in modern embedded systems. This is achieved by pairing *Out-of-Order (OoO)* cores, yielding performance through aggressive speculation and latency masking, with *In-Order (InO)* cores, that preserve energy through simpler design. By leveraging migrations between them, workloads can therefore select the best setting for any given energy/delay envelope. However, migrations introduce execution overheads that can hurt performance if they happen too frequently. Finding the optimal migration frequency is critical to maximize energy savings while maintaining acceptable performance. We develop a simulation methodology that can 1) isolate the hardware effects of migrations from the software, 2) directly compare the performance of different core types, 3) quantify the performance degradation and 4) calculate the cost of migrations for each case. To showcase our methodology we run mibench, a microbenchmark suite, and show that migrations can happen as fast as every 100k instructions with little performance loss. We also show that, contrary to numerous recent studies, hypothetical designs do not need to share all of their internal components to be able to migrate at that frequency. Instead, we propose a feasible system that shares level 2 caches and a *translation lookaside buffer* that matches performance and efficiency. Our results show that there are phases comprising up to 10% that a migration to the OoO core leads to performance benefits without any additional energy cost when running on the InO core, and up to 6% of phases where a migration to the InO core can save energy without affecting performance. When considering a policy that focuses on improving the energy-delay product, results show that on average 66% of the phases can be migrated to deliver equal or better system operation without having to aggressively share the entire memory system or to revert to migration periods finer than 100k instructions.

CCS Concepts: •**Computer systems organization** →**Heterogeneous (hybrid) systems;** Embedded systems; •**Computing methodologies** →*Simulation evaluation;*

Additional Key Words and Phrases: HMP, Out-of-Order, gem5, migration, heterogeneous multiprocessing, simulation methodology, embedded systems

ACM Reference format:
Ilias Vougioukas, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2017. **Nucleus: Finding the sharing limit of heterogeneous cores**. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (October 2017), 16 pages.
DOI: 0000001.0000001

This work was supported in part by the Engineering and Physical Research Council (EPSRC) under grant number EP/K034448/1

Authors' addresses: Ilias Vougioukas, Andreas Sandberg and Stephan Diestelhorst, ARM Ltd., CPC1, Capital Park, Fulbourn Road, Cambridge, CB21 5XE, UK; Bashir M. Al-Hashimi and Geoff V. Merrett, Electronic and Software Systems Group, School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.

 1539-9087/2017/10-ART1 $15.00
DOI: 0000001.0000001

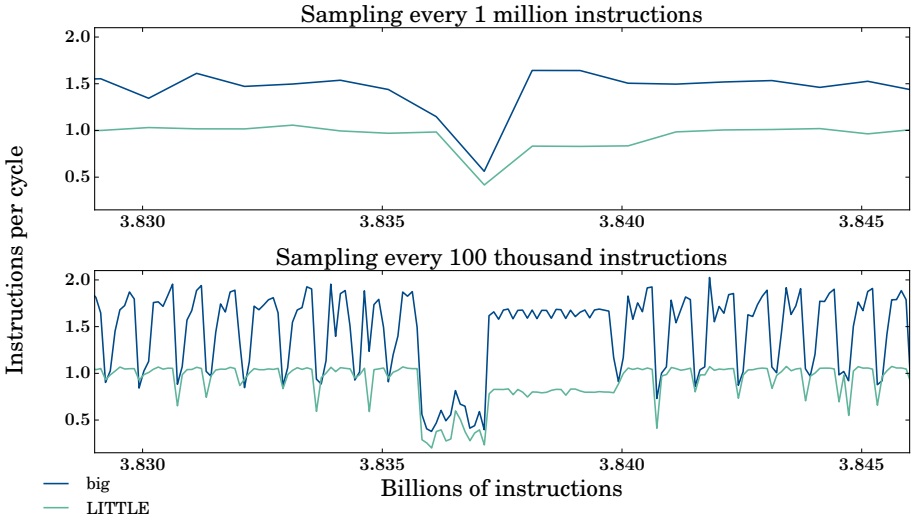


Fig. 1. An excerpt of execution of dijkstra running on two different core types. Sampling at a higher resolution reveals phases that can potentially be exploited to increase efficiency.

1 INTRODUCTION

In today's embedded devices, performance is always tied to power constraints and energy efficiency. This happens because, while process technology scaling is enabling larger transistor densities, power per area has shown to increase beyond a certain threshold in transistor size. This latter effect is commonly referred to as the breakdown of Dennard's scaling, which forces designs to operate only a fraction of the whole system, leaving the remaining design in a *dark silicon* state [8, 10]. To address this, *heterogeneous multiprocessors (HMPs)* have emerged over other designs, especially in mobile devices [17, 33], where keeping within a strict energy envelope while still being able to deliver performance on demand is crucial.

To be able to deliver high performance through *Memory Level Parallelism (MLP)* and *instruction level parallelism (ILP)*, an *Out-of-Order (OoO)* core is commonly used that has large caches, does aggressive speculation (branch predictors, prefetchers) and masks memory latency at the cost of significantly increased design complexity, area and power requirements. On the other hand, *In-Order (InO)* cores aim at conserving energy through a simpler and smaller design, at the expense of performance and lower operating frequency. A characteristic implementation of an HMP architecture is, for example, ARM's big.LITTLE [1], which uses two types of cores that share the same instruction set.

Workloads can be broken down into *phases*, some of which show high ILP potential and can therefore yield high performance while others heavily access memory and therefore deliver lower instructions per cycle (*IPC*). Choosing the most suitable core leads to an optimal performance and energy profile, while making a wrong decision can lead to unnecessary slowdown or energy waste.

Recent studies claim that *micro-phases* in the order of thousands of instructions exist [12, 30], as seen in Figure 1, and can be exploited to reduce energy consumption and boost performance through fine-grained migrations. In theory, perfect core matching should lead to optimal behaviour, however every time a migration is performed the working state needs to be transferred to the operating core, adding both a software and hardware execution *overhead*.

From the software side the *operating system (OS)* handles the scheduling to the appropriate core and sends the required signals to perform the migration. The complexity of the scheduler and the software context switch significantly affects the amount of overhead added to a system, which is on the order of *ten million instructions* [7, 19, 32, 35].

From a hardware perspective, the overheads depend on how much of the state needs to be transferred and how much is shared between the two migrating cores. The more state shared and preserved, the less *warm-up time* required, which is the period necessary to achieve maximum performance. Unlike software, hardware overheads can be significantly smaller, as low as thousands of instructions, depending on the amount of resources shared.

To reduce overall system slowdown when migrating, several studies propose the implementation of dedicated hardware to efficiently handle the transfer without any software intervention [12, 13, 18, 21, 26] and also fusing cores to share the front-end [20, 22, 24]. However, as hybrid designs are too complex to implement, it is worth investigating a simpler approach determining which components contribute most to the benefits and sharing only those.

The questions we aim to answer therefore are:

- Which core is more suitable for the current workload phase?
- Which components should be shared to improve migrations?
- How often can we switch before overheads suppress the benefits?

For the scope of this study we choose to focus *only on the hardware migration overheads for various cache and TLB sharing configurations*, as only through hardware acceleration it makes sense to explore fine-grained migrations. Our motivation is based on understanding migrations from an architectural standpoint and finding out how to improve future designs. Our contributions are:

- An architecture for HMPs which shares only the TLB and the last level cache. This effectively achieves the same performance as systems with merged cores [13, 18, 20–22, 24, 26], but is significantly simpler to design.
- Nucleus: A novel simulation methodology that allows the direct comparison of the behaviour of the exact same task on different cores, which is impossible on hardware platforms.
- Quantify the migration overheads for the OoO and InO core types under various degrees of sharing and find the migration period limit is roughly 100k instructions. Results show that full sharing does not realistically improve performance or reduce the overheads.

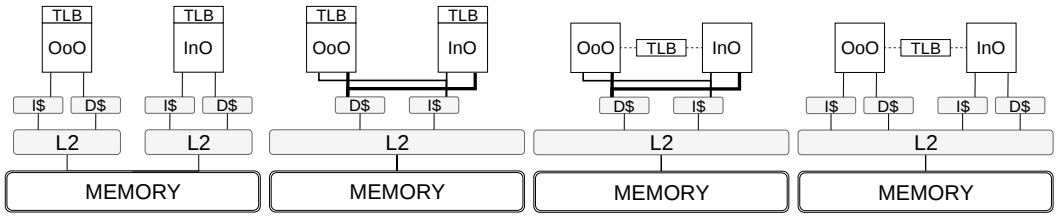
The rest of this paper is organised as follows: Section 2 describes the background theory behind resource sharing and migrations. In Section 3 we present the details of our simulation methodology. In Section 4 we perform a limit study stressing the capabilities of our methodology, which quantifies switching overheads and cases where migrations are beneficial. Section 5 puts our work in perspective with respect to related work in the field. Section 6 contains concluding remarks, describing the current limitations and our plans to extend the methodology and conclude the paper.

2 RESOURCE SHARING

2.1 Heterogeneous multiprocessor organization

The level of sharing in a system significantly impacts the ability to perform a context switch between cores without significant performance degradation. Even though this can notably reduce costs when migrating, it adds complexity, area and latency that can hurt the performance of the cores.

Based on this, conventional design mandates that heterogeneous cores usually share at best the *Last Level Cache (LLC)* or the main memory. To achieve high frequency switching, recent academic trends propose extreme attempts to bring cores closer together, usually by sharing instruction Level



(a) No sharing. (b) Share caches. (c) Share caches and TLB. (d) Proposal: Share L2 and TLB.

Fig. 2. Different types of sharing schemes examined for HMP systems

1 (*IL1*), data Level 1 (*DL1*) and Level 2 (*L2*) caches and in some cases even some internal components [13, 18, 20–22, 24, 26]. The feasibility of such designs however still remains questionable as some parts cannot be shared between drastically different designs, while others (IL1, DL1 caches) cannot be shared without lowering performance. A more realistic approach, instead of opting for full or no sharing, is to target the features cores can easily share to increase performance when migrating. With this in mind, we evaluate four different sharing schemes.

No sharing. The first case worth analysing is when cores have private caches and share no components except the main memory (Figure 2a). This sharing scheme is a common setup in chip multiprocessors and HMPs, with numerous industrial and academic examples[1, 6, 16]. Here every migration causes the system to transfer it's register file from the retiring to the incoming core before resuming normal operation. In addition the system also has to warm-up the empty or "cold" caches of the new core before it can achieve its maximum performance. The overheads for the no sharing ($T_{private}$) case can be broken down as:

$$T_{private} = \underbrace{T_{pip} + T_{BP} + T_{reg}}_{\text{non-shareable}} + \overbrace{T_{TLB} + T_{\$1} + T_{\$2}}^{\text{tied to core}} \tag{1}$$

Where T_{pip} and T_{BP} are the pipeline and branch predictor warm-up phases respectively, T_{reg} is the overhead to transfer the register file, T_{TLB} is the extra cost of warming up the TLB and $T_{\$1}$ and $T_{\$2}$ are the penalties induced when starting with cold L1 and L2 caches respectively.

Share caches. The next level of sharing we consider is one where the two cores partly or fully share the caches (Figure 2b). We consider this setup as an in between state, that can provide insight into system performance with and without sharing the TLB. This can potentially reduce the warm-up time after a switch. However, when workloads have a very small memory footprint, or thrash though data in the cache, the benefits of sharing are limited.

Another potential problem with this sharing scheme is that, even if caches contain useful data post-migration, empty TLBs can penalize the system significantly. This happens when, despite the fact that the desired data is still maintained in the cache, the virtual-to-physical translation is not in a TLB. TLB misses are slow, as they need to access main memory, to the point that a miss is much more expensive than an IL1 or DL1 miss, as the system ends up performing a page walk, requiring several memory accesses.

Share caches and TLB. The full sharing scheme (Figure 2c) assumes that both caches and TLBs are shared amongst heterogeneous cores, but still needs to warm-up the pipeline, branch predictors

and the register file. Sharing the register file is too complex design-wise, while its relatively small size permits ignoring the penalty, especially when compared to warming the branch predictors. The pipeline and branch predictors cannot be shared between different core types as they are functionally very different components. For example the state of the speculative, reordering, 4-wide and deeper pipeline in the OoO cannot be transferred to a 2-wide InO pipeline with half the depth in a conventional way. This design archetype has been a popular academic trend recently, with some studies claiming that even the branch predictor and pipeline can be re-purposed and shared to an extent [12, 13, 20–22, 24, 26].

Our Proposal: Share L2 and TLB. The issue with full system sharing is that while the total slowdown of a system is reduced after a migration, sharing the level 1 caches and the TLBs between the cores adds extra latency that can dramatically reduce the overall performance. This is because components, like the IL1 cache, are very sensitive to latency (~4 cycles per hit) and even a single extra cycle of latency can lead to 25% performance decrease as it is constantly being accessed. Sharing the L1 has this effect as in order to connect two cores to the same cache extra capacitance from wires is introduced and a switching signal is added (e.g. a multiplexer), which add latency. For this reason we propose an alternate architecture (Figure 2d) that shares the L2 cache and the TLB.

2.2 Internal core components

While the sharing scheme plays a large part in determining system performance and energy efficiency, the different functionality of cores in heterogeneous systems also affects system behaviour. This is because, despite the inherent penalty to switching, migrating to a more suitable core can amortise the cost and boost performance compared to remaining on the less compatible core. For this reason, it is important for heterogeneous systems to be comprised of cores that cover a wide range of applications across a variety of energy/performance points.

For the OoO core, the deeper and wider the pipeline, the more cycles needed to warm it up. Furthermore, in order to approach steady state performance, the OoO core also needs to fill the re-order buffer and the branch predictor. Both these structures are vital for Out-of-Order execution and, perhaps more importantly, can significantly hinder performance when they are not warmed-up. The simpler branch predictor and the lack of a reorder buffer of the InO core, while limiting the ability to parallelise instructions and mask memory requests, make it much faster to assume steady performance. Additionally, the simplicity in functionality allows the core's design to be much smaller and consume less energy.

Overall comparisons between the two cores show that *the little core is 3x more power efficient when clocked at the same frequency* [21]. In our limit study, presented in Section 3, we will adhere to this power ratio to compare the big and little cores.

3 METHODOLOGY

In this section we describe a gem5 based [4] simulation methodology used to explore the cost of migrations. To allow for direct comparisons, we gather samples, where we compare the IPC of two simulations, one executing on a system that runs uninterrupted and and one on a system that commences after the migration is triggered for the exact same phases. Based on our methodology we devise two techniques, one to calculate the overheads for each core type for the sharing schemes in Section 2.1 and one to find potential phases where it makes sense to migrate to a different core.

To be able to measure the impact of switching cost in a system, it is necessary to directly compare execution with and without performing migrations. To achieve this, we propose *Nucleus*, a methodology where we fork the simulator creating two identical simulations, the main or parent simulation and the forked or child simulation. After the simulator splits, the main simulation

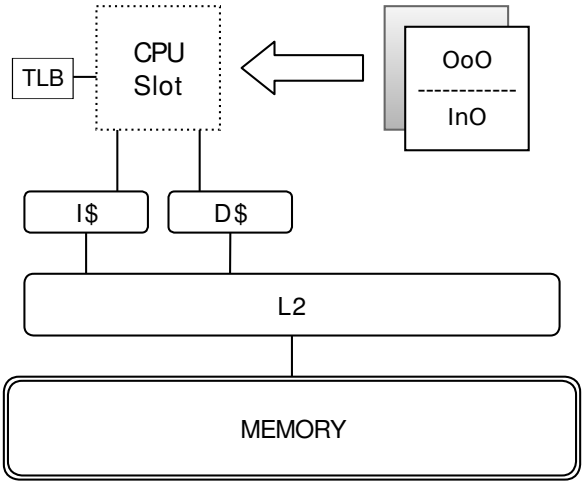


Fig. 3. The implementation with which the simulator "hot-swaps" the cores to emulate a system with varying shared resources.

continues unperturbed, while the fork performs a migration. Switching the core in the forked simulation can be achieved by signalling the operating system (*OS*) in the child to issue a migration. The OS can either trigger the migration itself, or the forked simulation cause a hardware interrupt to signal that a migration must be performed. Both of these methods however would add the OS migration overhead which, as described in the previous section, is undesirable as it will obfuscate the hardware effects.

Instead, we propose to simulate a single core system, where the forked simulations "hot-swap" the core leaving the caches and memory unaffected. To share the TLB, which for our simulator is tied to the core design, we just rewire the TLB to be connected to both cores, effectively detaching them from a specific core. When a hot-swap is performed, the system drains in-flight messages, flushes the operating core and afterwards swaps it with the inactive one.

One limitation from this approach is that the core operation is mutually exclusive. However, this is deliberate for our limit study as we are interested in single thread behaviour and it resembles the operation of the first commercial HMP systems [6] and merged cores that share the front end [20, 22, 24]. The infrastructure can easily be extended to accommodate beyond for more than one pair of coupled cores, however the decision tree of migration combinations is significantly more complex and is beyond the scope of this paper.

After the fork swaps the core designs, depending on the level of sharing explored, a writeback and invalidate is triggered for the components considered as private. The newly active core resumes operation with those structures empty and proceeds to warm them up before assuming steady state performance, which emulates the effects of a hardware migration.

On the software level, the system seamlessly continues to operate completely agnostic of the swap. This can be accomplished because the gem5 simulator can switch the micro-architecture of components in the system, such as the core design, without altering the system architecture state, such as the register values. This is exploited to single out the specific hardware component overheads during migrations. A graphical representation of the methodology infrastructure can be seen in Figure 3, where CPU slot is the placeholder for the desired CPU model, while the rest of the components remain unchanged.

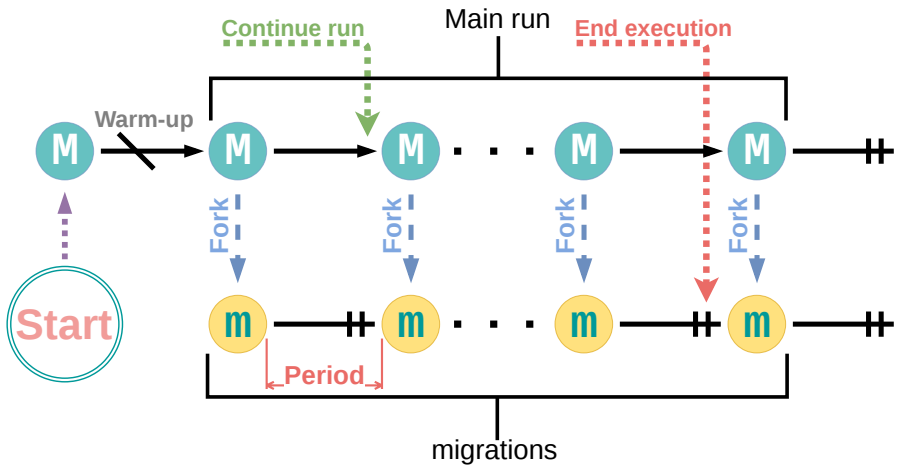


Fig. 4. A visual representation of Nucleus. The main simulation spawns periodic forks that perform a migration.

3.1 Nucleus

One important detail to note is that, irrespective of the sharing scheme, the register file is always considered to be shared as the simulator does not account for the cost of transferring it. Several hybrid core proposals claim that the state transfer is possible with negligible overhead [13, 21, 26]. While we do not consider this realistic for actual contemporary hardware, it helps to draw two useful conclusions from our study. First, it helps us assess a best-case scenario for each sharing scheme, and compare our results to many studies supporting fine grain migrations and merged designs [12, 13, 20–22, 24, 26]. Secondly, it allows a cleaner observation by singling out of the effects of specific components, and provides insight that could otherwise be masked.

Our methodology allows for certain experiments to be conducted which were not previously feasible. For instance, it is possible to compare the exact same phases in systems with different characteristics. Figure 1 shows how an excerpt of dijkstra is perfectly aligned. To contrast, as forked execution is not possible in hardware, two separate executions need to be triggered in parallel. However, as the executions are not fully deterministic, deviations between them will cause them to diverge and therefore the phases will end up misaligned. This is especially severe in the case of fine grain investigation as even a small skew can lead to wrong results. By forking from the same starting point Nucleus solves this problem as every sample is aligned, even at the finest granularity.

3.2 Comparing parallel simulations

To explore migrations, a main simulation is created that runs on the same core from beginning to end without performing any migrations. After a warm-up interval the main simulation starts spawning forks periodically until the workload terminates. The main simulation and forks are symbolised with M and m respectively in Figure 4. The period for each sample is measured in instructions instead of time. Using time to determine the sample period can cause the parallel simulations to diverge when there is a difference in performance, as the simulation with the higher IPC will execute more instructions in the allotted time. Sampling with instructions solves this as, irrespective to the performance or time they take, both runs will end up executing the same code.

The forked simulations examine *first-order migrations*, which effectively run only for a single period and then terminate. This way, for every period running on the main simulation, there is an equivalent simulation that examines how a system would be affected by a migration at that specific phase. The termination of the forked simulations after one period restricts our methodology to only one migration however, expanding to the more migrations would cause the state space to become too large to compute.

We devise two separate experiments with our methodology that answer two separate questions:

- **Migration Cost:** Taking into consideration the different sharing schemes, what are the warm up costs for each core?
- **Efficient Migrations:** What is the best setup (sharing scheme, switching frequency) for HMPs?

Answering the former question, we perform our methodology migrating always to the same core type, from OoO to OoO and likewise from InO to InO, for all the sharing schemes described in section 2.1. The migration frequency limitations for each core can be calculated by normalizing the performance of the migrations with the corresponding main simulations, averaged over all samples:

$$\text{Relative Performance} = \frac{\sum \frac{IPC_{mig.}}{IPC_{main}}}{N} \tag{2}$$

where N is the number of samples.

To determine what the optimal setup is for HMPs we devise another experiment, that switches the core type, from OoO to InO and vice versa, in the migratory simulations. The IPC comparison for this experiment reveals the performance disparity between the two cores in each phase. We explore two migration policies that determine beneficial switches, the *no trade-off policy* and the *energy delay product (EDP) policy*. The first is when migrating to the opposite core reduces power or execution delay without sacrificing performance or energy respectively, which can be useful in high response situations or when energy is a constrained resource. For the second policy we examine switches that lead to equal or better EDP. We quantify this by calculating the percentage of workload that a migration is beneficial:

$$\text{Beneficial Migrations} = \frac{\sum \left[\frac{IPC_{mig.}}{IPC_{main}} \geq \theta\right]}{N} \tag{3}$$

Beneficial Migrations refer to the amount of cases where switching from one core type to the opposite leads to an improvement. For instance, if the indicator is 0.6, this means that a system with an InO core as the main core could benefit from running on the OoO core for 60% of the workload. The [...] are the Iverson brackets [14]. $[P]$ is defined to be 1 if P is true, and 0 if it is false. The formula only counts the migrations that are better than the main run, using θ, which is the threshold we set to determine a beneficial migration. Depending on the policy desired the threshold θ is adjusted, for instance to find cases that lead to better EDP or to find the migrations that save energy without performance loss.

4 RESULTS

In this section we present our setup for both experiments, described in Section 3, and their results. We compare the sharing schemes from Section 2.1 to determine whether sharing the caches and TLB is worth the added complexity. We also present our experiments on efficient migrations and provide some insights into the results.

	OoO Core	In-Order Core
Pipeline Width	3-wide	2-wide
Pipeline Depth	15	8
Branch Predictor	Two-Level	Tournament
Operating Frequency	1GHz	1GHz

Shared Resources	Value
L1D Size	32KB
L1I Size	32KB
L2 Size	1MB
L2 Associativity	16-way
L2 Prefetcher	Stride

Table 1. The specifications of the system used in our experiments.

4.1 Experimental Setup

To evaluate Nucleus, we use the gem5 simulator [4] in *full system mode* and modify the forking mechanism [28] to be able to fork with instructions and switch the cores. The experimental setup simulates a heterogeneous system using big OoO and little InO publicly available ARMv8 models [27, 31], both operating at the same frequency using the classic memory subsystem. The simulated system uses the same cache sizes for both the OoO and the InO cores, so that a fair comparison between private and shared caches is drawn. The simulator is restored from a previous checkpoint that bypasses the boot-up and each benchmark is let to run until no warm-up transient effects occur. A detailed description of the system configuration is shown in Table 1.

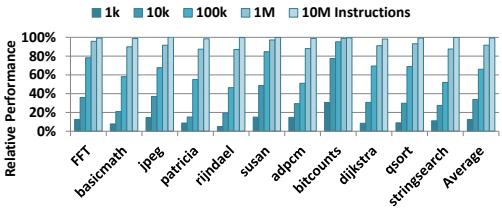
The sharing architecture simplifications we have made and the set of benchmarks have been selected to represent the best possible case for fine-grain migrations. While most sharing schemes are not feasible with today's implementation techniques, the hypothetical set-ups aim to answer whether it makes sense to pursue fine-grain migrations in HMP systems, *regardless of current physical limitations*.

4.2 Migration Cost

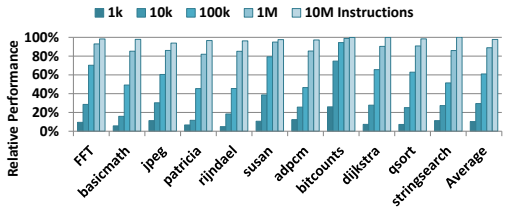
For the first set of experiments, the simulated system runs the mibench benchmark suite [15] using a minimal build of Linux Ubuntu 14.04.5 (kernel version 3.14.0) as the operating system. This features microbenchmarks that have very different characteristics and demonstrate high variation in branch, memory, and integer ALU operations. At the same time, the used datasets can fit in caches most of the time, which should work further in favour of the sharing architectures. The relatively small size of the workloads also helps with the computationally expensive process of forking the simulator.

4.2.1 No sharing. In Figures 5a and 5b the performance of the migrating runs is presented, for the InO and OoO cores respectively, normalised to the main simulation's performance for every sample. A relative performance of 100% means that switches at that frequency would not cause a performance hit, whereas relative performance of 50% means that on average migrating with that frequency halves the IPC. This effectively allows for a fair comparison between a system performing a migration and one that does not. The performance drop at fine granularity suggests that with private caches fast switching is not feasible, as the overheads dominate. This represents fairly accurately conventional architectures we use today, and shows why migrations fall into the category of millions of instructions, even without considering the considerable software overhead.

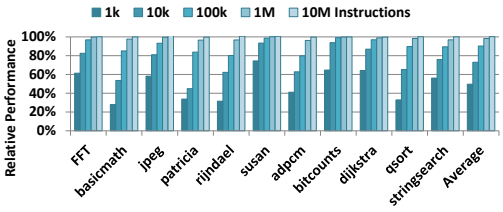
4.2.2 Share caches. When sharing all the caches but not the TLB, performance of the migratory samples visibly increases. One thing to note in Figures 5c and 5d is that the performance improvement of the InO core is slightly larger than that of the OoO one at 1k instructions migration period. The deeper pipeline and larger mis-speculation penalties seem to be causing a greater negative effect on the OoO core. Overall, the performance does not justify this complex design, especially



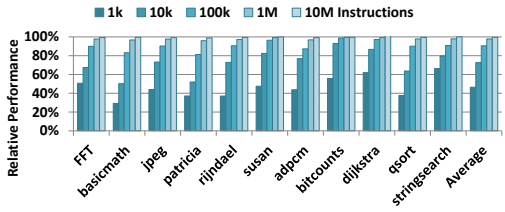
(a) InO performance with no sharing.



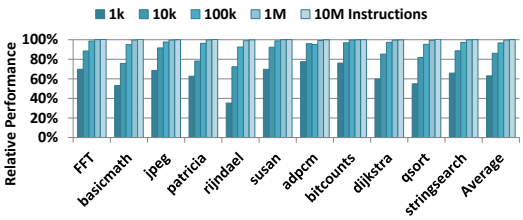
(b) OoO performance with no sharing.



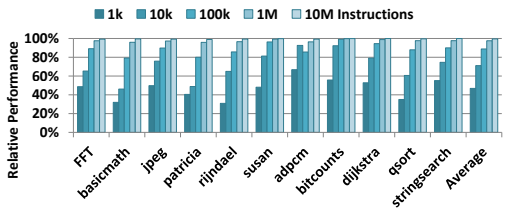
(c) InO performance sharing the caches.



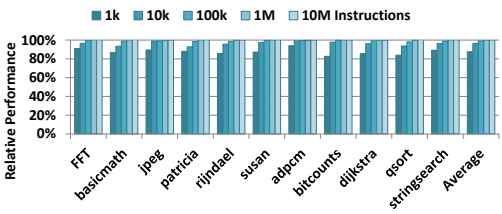
(d) OoO performance sharing the caches.



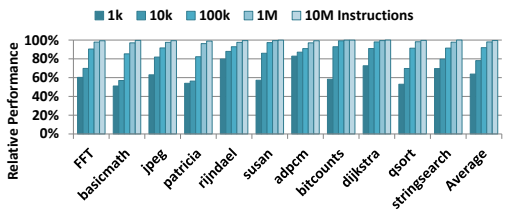
(e) InO performance sharing the L2 and the TLB.



(f) OoO performance sharing the L2 and the TLB.



(g) InO performance sharing caches and the TLB.

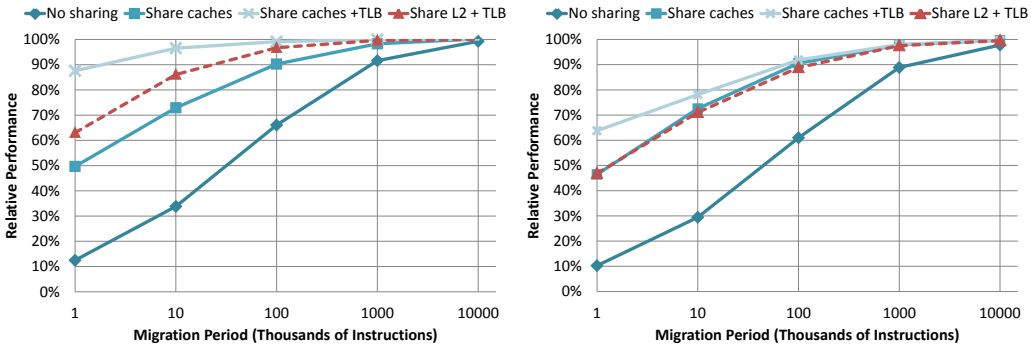


(h) OoO performance sharing caches and the TLB.

Fig. 5. Relative performance degradation in mibench for different sharing schemes.

for fine-grain switching. This is due to the fact that even though the caches are shared, the TLB misses cause the system to potentially access to main memory which adds huge penalties.

4.2.3 Share caches and TLB. When the system shares all memory components (caches and TLB) another interesting result is revealed. While the InO core eliminates almost all of the overhead even at the finest migration frequency, the OoO design still exhibits significant performance losses. This exposes the inability of the OoO core to achieve its maximum potential, due to the deeper pipelines



(a) Performance degradation for the InO core.

(b) Performance degradation for the OoO core.

Fig. 6. The average performance degradation for the two core type as a function of migration frequency.

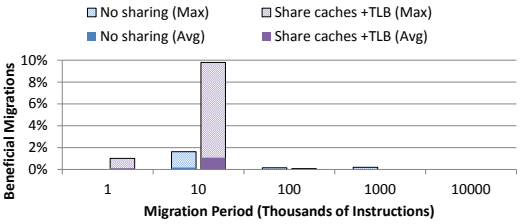
and the re-order buffer that need to be filled, as well as the fact that OoO execution heavily relies on speculation and branch prediction to achieve high IPC.

4.2.4 Our Proposal: Share the L2 and TLB. When sharing the L2 and the TLB but keeping the L1 caches private to each core, the results of the InO core show a notable improvement over the previous sharing scheme. Our proposal shows a marginal decrease in performance for the OoO core, when performing fine grain migrations, but a significant improvement for the InO core. More importantly, sharing an extra TLB between the cores shows that even though the L1 caches are not shared, the overall system improves in performance over current architectures and performs on par with solutions that try to share all the resources. The improvement, as stated above, is due to the TLB sharing that helps to avoid accessing main memory to fetch data, taking care of the worst case when no pages are stored in the L2 cache. In the event where a TLB miss has occurred in the recent past, the page already exists in the L2 in which case the page walk for this system will be similar to the system that only shares caches.

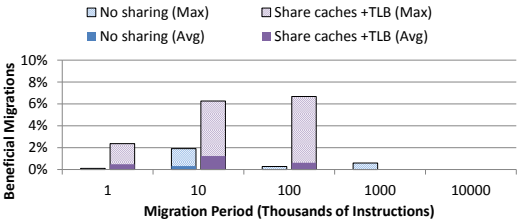
Figure 6 summarises the averages presented in Figure 5, which stresses the point that our proposal can achieve similar performance without resorting to sharing all resources. We see that the performance gap between our proposed method, depicted as the red dotted line with triangles, and the full sharing scheme is 17% for the OoO and 30% for the InO core. However, the design complexity of routing both cores to all the caches and the TLB is prohibitive. More importantly though, we observe that the migration frequency of the OoO core does not have significant slowdown above 100k instructions. At this granularity though, our proposal performs equally well without the added complexity. For HMP systems pairing OoO and InO, it makes more sense to have a simpler design sharing just the L2 and a TLB. Sharing the L1 caches adds more complexity, but does not allow the system to migrate any faster than 100K instructions, as the OoO overheads diminish any benefits beyond that point.

4.3 Efficient Migrations

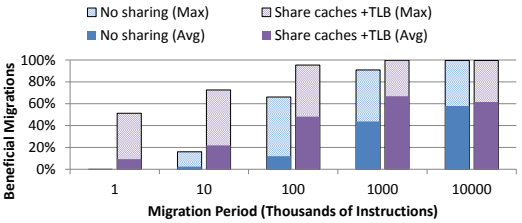
As introduced in Section 3.2, the critical question for HMP systems is whether migrating from one core to another yields any benefits. Based on equation 3, we find how much of each workload can be migrated to the opposite core, using the two policies mentioned in 3.2 to adjust the threshold value θ. We base our analysis on the fact that the InO core is 3x more energy efficient, an estimate in line with current HMP systems [21].



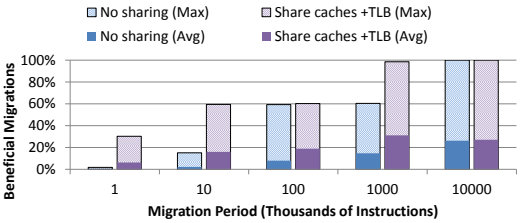
(a) Beneficial migrations from the InO to the OoO core with the No trade-off policy.



(b) Beneficial migrations from the OoO to the InO core with the No trade-off policy.



(c) Beneficial migrations from the InO to the OoO core with equal or better EDP policy.



(d) Beneficial migrations from the OoO to the InO core with equal or better EDP policy.

Fig. 7. Percentage of beneficial migrations when being energy and EDP constrained.

We present in Figure 7 the two extreme cases of sharing for the no trade-off and EDP policies mentioned in Section 3. The percentage in these plots signifies how much of the workload leads to a more desirable state. The solid coloured bars depict the benefits averaged across all workloads, while the shaded ones show the workload with the maximum benefit.

When using the no trade-off policy, a system on the InO core will migrate only when executing the same phase on the OoO core costs equal or less energy. For this to happen the big core must perform 3x better than the little one, so the threshold is adjusted to $\theta = 3$. In Figure 7a we see that, irrespective of how much hardware the cores share, the cases where this is beneficial are less than 1% on average and at most 10% of the total execution, for specific workloads.

When considering the switch from OoO to InO with the no trade-off policy, the samples are considered beneficial when the InO core performs equally as good or better than the OoO core ($\theta = 1$). This results in the same overall performance for only a third of the power. This case can happen when the OoO core cannot exploit any ILP or MLP and it is reduced to InO performance. Figure 7b shows that similarly to the previous result, the beneficial cases are at most 1% on average and 6% of the total execution for a few benchmarks.

While these cases are few, the no trade-off policy reveals that the benefits can only be exploited mostly at the migration periods of 10k and 100k instructions. This reveals that micro-phases most probably exist. However these remain latent at any granularity coarser than 100k instructions, as the phases average out, while migrating every 1k instructions yields no benefits as the overheads mask micro-phases.

For equal or better EDP our metric values performance more as $EDP = Power \times Delay^2$. The amount of beneficial switches when migrating from the InO to the OoO core for this policy is calculated using a threshold value of $\theta = \sqrt{3}$. The results show that the potential for switching and maintaining equal or better EDP in this case range between 48% and 66% when migrating

every 100K, 1M and 10M instructions. When performing the opposite switch with the EDP policy ($\theta = \frac{1}{\sqrt{3}}$), the results show that when aiming for the best EDP the OoO core is more suitable as the beneficial cases on average for all the workloads does not exceed 31% of the execution.

For both EDP experiments, migration periods larger 100K instructions are less desirable, as the gains are significantly smaller. Additionally, we observe that in most cases, sharing the caches and the TLB is not necessary as both sharing schemes show similar potential for useful migrations. Furthermore, in terms of the maximum of the beneficial migrations, we notice that for periods larger than 1M instructions both the InO and OoO core show that some workloads are better off running at the opposite core.

5 RELATED WORK

While the problem of maximizing efficiency in HMPs through core migrations has been researched in the past, the effect of migrations in the system is, to the best of our knowledge, still misunderstood especially in the realm of fine-grained migrations.

Some studies have explored the notion of migrating execution amongst heterogeneous cores that takes into account the effect of the memory system [9] [16]. Improvement is limited to relatively coarse-grained switching frequencies and modifications are required for the system to achieve them.

Other efforts aim at redesigning accelerator cores that can adapt better to the applications[23]. These accelerator type cores are determined through a performance study conducted at coarse instruction granularity. Furthermore, the redesigned cores also operate on coarse application phases rather than periodically deciding when to switch.

A similar approach [18] has been explored where a the core can morph to exploit MLP and ILP in the workloads. Furthermore, this HMP architecture has the potential for fine-grained migrations although this has not been fully explored, to the best of our knowledge. Other works [30] have recently examined this approach.

Prior work on the existence of phases at fine granularity and their potential micromanagement has shown that it is possible to exploit their benefits through a heterogeneous system with different back-ends sharing the same front end [12, 20–22]. Despite the claim, no experimental data is provided that reveals the actual existence of application phases that appear at a fine granularity nor are the circumstances under which they can be exploited explained.

In [13, 26], the authors have implemented a HMP system sharing the memory system and the register files amongst the heterogeneous cores, claiming under 100 cycle migration latency in actual hardware. They propose three dimensional stacked cores to deal with the wiring and routing problem. One interesting claim made is that the register file can be transferred with a very small overhead (one cycle), which points towards plausible future designs with high sharing. However, the system they propose has certain assumptions that we consider problematic. Firstly, the cores used for sharing are two OoO designs of different width, but using up roughly the same area. In their proposed system the similarity of the two cores make migrations pointless both in terms of energy efficiency and performance.

Additionally, no direct information is given about the operating frequency of the system however, the reported cycle time is 5.5ns which implies a frequency of 181MHz which would undeniably drastically degrade performance. We believe this is indicative of the performance degradation incurred when sharing sensitive components as like the register file, the L1 caches and the TLB as such approaches either add extra processing cycles or clock the system at extremely low frequencies to deal with the extra combinational logic and routing overheads.

In terms of HMP scheduling investigation, several studies examine whether it is possible to augment the architecture and make it more aware of the heterogeneity to exploit the benefits of fast and low overhead migrations [24, 29].

In the past, studies similar to those of fine-grained thread migrations focused primarily on levering homogeneous cores that operate at different energy and performance points by changing the voltage and frequency domains [5]. Significant evidence shows that fine-grained Dynamic Voltage and Frequency Scaling (*DVFS*) has received mixed results in terms of energy savings with some claiming notable gains in memory intensive phases [20, 25], while others refute this noting that, at fine-grain DVFS, the switching latency reduces performance and increases energy consumption [11, 34].

6 CONCLUSIONS

In this work we have proposed Nucleus, a full system simulation methodology using gem5, to study the effects of migrations in heterogeneous systems. Our methodology accomplishes this by forking the simulation, comparing a system that migrates with one that does not. We show that Nucleus can be used in various ways, exploring different aspects of migrations. For instance, it can be used to measure the performance degradation across different migration frequencies. Alternately, we propose using it to calculate how much of each workload can be migrated to a different core to gain in energy or performance.

We use our methodology to focus on identifying how much sharing of internal components (e.g. caches, TLB, register file) is most desirable for migrations as fine as 1k instructions. We show that our methodology is well suited to measure the hardware overheads, as it isolates them from the software overheads. Our results show that the two cores have asymmetric warm-up times and behaviour, where the OoO core needs larger instruction windows to amortize the cost of migration than the InO core. Overall this prohibits the beneficial use of migration frequencies finer than 100k instructions. Furthermore, we propose an architecture sharing only the L2 and the TLB to enable more efficient migrations. Compared to other sharing schemes, we find that our design performs equally as good as designs that share all the caches and the TLBs, but can be physically much simpler to implement.

To demonstrate the capabilities of Nucleus, we perform a case study running mibench, a suite of diverse embedded benchmarks, on a heterogeneous system. The results show that, even when the overheads are not prohibitive, migrating to save energy without sacrificing performance or vice versa is limited to at most 10% of the execution. Relaxing the policy to find equal or better EDP points of operation shows that migrations can be beneficial on average for 66% of the workload execution.

Currently our methodology is limited to exploring only first-order migrations. This prohibits us from performing studies using asymmetrical switching, where the little cores can migrate at higher frequencies than the big cores, as our study suggests. Our aim is to address this in future research, where some of the state can be retained on the big core while it is switched off. Additionally, we plan to expand to higher-order migrations which will enable our methodology to explore more of the state space for patterns in execution that, when identified, can lead to better scheduling of heterogeneous systems.

Finally, as the results of our proposed architecture show promise, we believe that it is worth investigating HMPs with hierarchical TLBs. Such systems can be more beneficial in terms of latency restrictions as the level 1 TLB can be private and the level 2 TLB be shared. Similar approaches have been mentioned in the past for homogeneous multiprocessors [2, 3], but need additional investigation in solving coherency issues in TLBs.

ACKNOWLEDGMENTS

This work was supported in part by the Engineering and Physical Research Council (EPSRC) under grant number EP/K034448/1 "PRiME: Power-efficient, Reliable, Many-core Embedded systems" (www.prime-project.org). Experimental data used available at DOI:10.5258/SOTON/D0161.

This research was supported by the people of ARM Research. We thank our colleagues for their valuable feedback that greatly assisted us in delivering a better version of this work.

REFERENCES

[1] ARM. 2013. big.LITTLE Technology: The Future of Mobile. *ARM white paper* (2013), 12. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Future_of_Mobile.pdf

[2] A. Bhattacharjee, D. Lustig, and M. Martonosi. 2011. Shared last-level TLBs for chip multiprocessors. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 62–63. https://doi.org/10.1109/HPCA.2011.5749717

[3] A. Bhattacharjee and M. Martonosi. 2009. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 29–40. https://doi.org/10.1109/PACT.2009.26

[4] Nathan Binkert, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. M.D. D Hill, David A. D.A. A Wood, Bradford Beckmann, Gabriel Black, Steven K. S.K. K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, A. Basil, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. M.D. D Hill, and David A. D.A. A Wood. 2011. The gem5 Simulator. *Computer Architecture News* 39, 2 (2011), 1. https://doi.org/10.1145/2024716.2024718

[5] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. 2005. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 24. 18–28. https://doi.org/10.1109/TCAD.2004.839485

[6] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. 2013. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM ® big.LITTLE ™ Technology. (2013), 1–8. https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf

[7] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, Andre Seznec, and Irisa Inria. 2005. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News* 33, 4 (nov 2005), 80–91. https://doi.org/10.1145/1105734.1105745

[8] Robert H. Dennard, Jin Cai, and Arvind Kumar. 2007. A perspective on today's scaling challenges and possible future directions. *Solid-State Electronics* 51, 4 SPEC. ISS. (2007), 518–525. https://doi.org/10.1016/j.sse.2007.02.004

[9] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM Press, New York, New York, USA, 261. https://doi.org/10.1145/2189750.2151004

[10] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark silicon and the end of multicore scaling. *IEEE Micro* 32, 3 (2012), 122–134. https://doi.org/10.1109/MM.2012.17

[11] Stijn Eyerman and Lieven Eeckhout. 2011. Fine-grained DVFS Using On-chip Regulators. *ACM Trans. Archit. Code Optim.* 8, 1 (2011), 1:1–1:24. https://doi.org/10.1145/1952998.1952999

[12] Chris Fallin, Chris Wilkerson, and Onur Mutlu. 2014. The heterogeneous block architecture. In *2014 32nd IEEE International Conference on Computer Design, ICCD 2014*, Vol. -. 386–393. https://doi.org/10.1109/ICCD.2014.6974710

[13] Elliott Forbes, Zhenqian Zhang, Randy Widialaksono, Brandon Dwiel, Rangeen Basu Roy Chowdhury, Vinesh Srinivasan, Steve Lipa, Eric Rotenberg, W. Rhett Davis, and Paul D. Franzon. 2016. Under 100-cycle thread migration latency in a single-ISA heterogeneous multi-core processor. In *2015 IEEE Hot Chips 27 Symposium, HCS 2015*. IEEE, 1–1. https://doi.org/10.1109/HOTCHIPS.2015.7477478

[14] Ronald L Graham, Donald E Knuth, and Oren Patashnik. 1989. *Concrete Mathematics: A Foundation for Computer Science*. Vol. 2. xiii + 625 pages. https://doi.org/10.2307/3619021 arXiv:arXiv:1011.1669v3

[15] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *2001 IEEE International Workshop on Workload Characterization, WWC 2001*. IEEE, 3–14. https://doi.org/10.1109/WWC.2001.990739

[16] Anthony Gutierrez, Ronald G. Dreslinski, and Trevor Mudge. 2014. Evaluating private vs. shared last-level caches for energy efficiency in asymmetric multi-cores. In *Proceedings - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2014*, Vol. -. 191–198. https://doi.org/10.1109/SAMOS.2014.6893211

[17] H Homayoun. 2016. Heterogeneous chip multiprocessor architectures for big data applications. *2016 ACM International Conference on Computing Frontiers - Proceedings* (2016), 400–405. https://doi.org/10.1145/2903150.2908078

[18] Khubaib. 2014. *Performance and Energy Efficiency via an Adaptive MorphCore Architecture*. Ph.D. Dissertation. University of Texas Austin.

[19] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science - ExpCS '07*. 2–es. https://doi.org/10.1145/1281700.1281702

[20] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. 2014. Heterogeneous microarchitectures trump voltage scaling for low-power cores. *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14* (2014), 237–250. https://doi.org/10.1145/2628071.2628078

[21] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald Dreslinski, Thomas F. Wenisch, and Scott Mahlke. 2012. Composite cores: Pushing heterogeneity into a core. In *Proceedings - 2012 IEEE/ACM 45th International Symposium on Microarchitecture, MICRO 2012*. 317–328. https://doi.org/10.1109/MICRO.2012.37

[22] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M. Sleiman, Ronald G. Dreslinski, Thomas F. Wenisch, and Scott Mahlke. 2016. Exploring fine-grained heterogeneity with composite cores. *IEEE Trans. Comput.* 65, 2 (2016), 535–547. https://doi.org/10.1109/TC.2015.2419669

[23] Sandeep Navada, Niket K. Choudhary, Salil V. Wadhavkar, and Eric Rotenberg. 2013. A unified view of non-monotonic core selection and application steering in heterogeneous chip multiprocessors. In *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*. IEEE, 133–144. https://doi.org/10.1109/PACT.2013.6618811

[24] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2015. DynaMOS: Dynamic Schedule Migration for Heterogeneous Cores. In *MICRO'15*, Vol. -. 322–333. https://doi.org/10.1145/2830772.2830791

[25] Krishna K Rangan, Gu-Yeon Wei, and David Brooks. 2009. Thread motion: Fine-Grained Power Management for Multi-Core Systems. *Proceedings of the 36th annual international symposium on Computer architecture - ISCA '09* (2009), 302. https://doi.org/10.1145/1555754.1555793

[26] Eric Rotenberg, Brandon H. Dwiel, Elliott Forbes, Zhenqian Zhang, Randy Widialaksono, Rangeen Basu Roy Chowdhury, Nyunyi Tshibangu, Steve Lipa, W. Rhett Davis, and Paul D. Franzon. 2013. Rationale for a 3D heterogeneous multi-core processor. In *2013 IEEE 31st International Conference on Computer Design, ICCD 2013*. IEEE, 154–168. https://doi.org/10.1109/ICCD.2013.6657038

[27] Roxana Rusitoru. 2015. ARMv8 micro-architectural design space exploration for high performance computing using fractional factorial. In *PMBS@SC*.

[28] Andreas Sandberg, Nikos Nikoleris, Trevor E. Carlson, Erik Hagersten, Stefanos Kaxiras, and David Black-Schaffer. 2015. Full speed ahead: Detailed architectural simulation at near-native speed. In *Proceedings - 2015 IEEE International Symposium on Workload Characterization, IISWC 2015*. IEEE, 183–192. https://doi.org/10.1109/IISWC.2015.29

[29] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. 2009. Hass: A Scheduler for Heterogenous Multicore Systems. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 66. https://doi.org/10.1145/1531793.1531804

[30] Sudarshan Srinivasan, Nithesh Kurella, Israel Koren, and Sandip Kundu. 2016. Exploring heterogeneity within a core for improved power efficiency. *IEEE Transactions on Parallel and Distributed Systems* 27, 4 (2016), 1057–1069. https://doi.org/10.1109/TPDS.2015.2430861

[31] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver. 2013. A structured approach to the simulation, analysis and characterization of smartphone applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 113–122. https://doi.org/10.1109/IISWC.2013.6704677

[32] Dan Tsafrir. 2007. The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops) General. *ExpCS* (2007), 13–14. https://pdfs.semanticscholar.org/86f8/a42a44b82cf76dcfe023209cfa4cdc0c8981.pdf

[33] Violaine Villebonnet, Georges Da Costa, Laurent Lefevre, Jean-Marc Pierson, and Patricia Stolf. 2015. "Big, Medium, Little": Reaching Energy Proportionality with Heterogeneous Computing Scheduler. *Parallel Processing Letters* 25, 03 (sep 2015), 30. https://doi.org/10.1142/S0129626415410066

[34] Fen Xie, Margaret Martonosi, and Sharad Malik. 2005. Efficient behavior-driven runtime dynamic voltage scaling policies. *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES+ISSS '05* (2005), 105. https://doi.org/10.1145/1084834.1084864

[35] Kisoo Yu, Donghee Han, Changhwan Youn, Seungkon Hwang, and Jaechul Lee. 2013. Power-aware task scheduling for big.LITTLE mobile processor. In *ISOCC 2013 - 2013 International SoC Design Conference*. IEEE, 208–212. https://doi.org/10.1109/ISOCC.2013.6864009

Received May 2017; revised June 2017; accepted July 2017