Mirage Cores: The Illusion of Many Out-of-order Cores Using In-order Hardware

Shruti Padmanabha
University of Michigan, Ann Arbor
shrupad@umich.edu

Andrew Lukefahr*
Indiana University
lukefahr@indiana.edu

Reetuparna Das
University of Michigan, Ann Arbor
reetudas@umich.edu

Scott Mahlke
University of Michigan, Ann Arbor
mahlke@umich.edu

ABSTRACT

Heterogenous chip multiprocessors (Het-CMPs) offer a combination of large Out-of-Order (*OoO*) cores optimized for high single-threaded performance and small In-Order (*InO*) cores optimized for low-energy and area costs. Due to practical constraints, CMP designers must choose to either optimize for total system throughput by utilizing many *InO* cores or maximize single-thread execution with fewer *OoO* cores. We propose *Mirage Cores*, a novel Het-CMP design where clusters of *InO* cores are architected around an *OoO* in a manner that optimizes for both throughput and single-thread performance. The insight behind *Mirage Cores* is that *InO* cores can achieve near-*OoO* performance if they are provided with the dynamic instruction schedule of an *OoO* core. To leverage this, *Mirage Cores* employs an *OoO* core as an optimal instruction schedule generator as well as a high-performance alternative for all neighboring *InO* cores. We also develop intelligent runtime schedulers which orchestrate the arbitration and migration of applications between the *InO* cores and the central *OoO*. Fast and timely transfer of dynamic schedules from the *OoO* to *InO* allows *Mirage Cores* to create the appearance of all *OoO* cores to the user using underlying In-Order hardware.

Overall, with an 8 *InO* per *OoO* configuration, *Mirage Cores* can achieve on average 84% of the performance of a CMP with 8 *OoO* cores, a 28% increase relative to current systems, while conserving 55% of energy and 25% of area costs. We find that we can scale the design to around 12 *InO*s per *OoO* before starvation for the *OoO* starts to hamper system performance.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; **Heterogeneous (hybrid) systems**; • **Hardware** → Chip-level power issues;

KEYWORDS

Heterogeneous multicores, Energy-efficient architectures, CMP scheduling

ACM Reference format:
Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2017. Mirage Cores: The Illusion of Many Out-of-order Cores Using In-order Hardware . In *Proceedings of The 50th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, MA, USA, October 14-18, 2017 (MICRO-50),* 14 pages.
https://doi.org/10.1145/3123939.3123969

1 INTRODUCTION

Growing energy and power dissipation concerns are impeding the rate of progress that has come to be expected in modern processors. To continue scaling performance, computer architects must fundamentally increase energy efficiency of processors across all platforms – from the data centers which draw megawatts of power, to mobile phones where battery life is precious to all users.

Decades of research pushing the limits of single-threaded performance have led to the sophisticated design of modern Out-of-Order (*OoO*) cores, albeit at the expense of high energy consumption. Shrinking transistor sizes with static threshold voltages gave rise to Chip MultiProcessors (CMPs) that allow parallel execution of multiple applications, further boosting system performance. In-order (*InO*) cores, on the other hand, are designed to be low-power and lean, but in the process compromise on performance. ARM's InO A7 core, for instance, consumes nearly 1/3 the energy of the OoO A15 core, while sacrificing 1/2 the performance[15]. Heterogeneous CMPs (Het-CMPs) allow for a combination of cores with different performance and energy characteristics. They can achieve energy-proportionality by executing the high performance applications/phases on the fast *OoO* cores, and running the low-utilization ones on the slower, more efficient *InO* cores.

Higher energy savings can be gained from increasing the utilization of *InO* cores. Additionally, more of them can fit in a given power and area budget of a CMP as compared to bulkier *OoO*s, potentially increasing system throughput. Yet, their inferior single-threaded performance forces general-purpose system designers to favor and maximize the number of *OoO* cores on Het-CMPs, while use of *InO* cores is limited to non-critical, background execution.

This work attempts to improve the adoption and utilization of *InO* cores in a Het-CMP by *reducing this performance disparity*. With *Mirage Cores* we envision a single-ISA Het-CMP with many small, *InO* cores and a few *OoO* cores, where the *InO* performance

*This work was done while author was at the University of Michigan

MICRO-50, October 14-18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9…$15.00
https://doi.org/10.1145/3123939.3123969

is increased substantially by using scheduling information from the *OoO* execution. Within the same power and area budget, *Mirage Cores*[1] proposes to replace some *OoO* cores with multiple, *nearly-as-fast InO* cores, thus increasing overall system-throughput and energy-savings while maintaining single-thread performance.

The performance advantage of an *OoO* comes from its complex instruction reordering backend. Using large instruction windows, the *OoO* can look past stalling instructions and extricate instruction parallelism by issuing subsequent independent instructions. This is an energy-intensive process requiring complex issue logic. However, as shown by previous research, [30, 34, 35], since a majority of an application is spent executing loops with predictable control and data flow, in the steady state, the *OoO* tends to schedule recurring instructions in the same program context in similar patterns. Moreover, if given a best-case *OoO* schedule for a set of instructions, with perfect control-flow knowledge, we found that an *InO* core with the same superscalar width and functional units as an *OoO* can attain up to 90% of its performance. This reveals a unique opportunity to utilize an available *OoO* for the purpose of creating highly optimized, dynamic schedules for applications executing on neighbouring *InO* cores in a Het-CMP, instead of merely accelerating critical sections as proposed previously [19, 45]. Capturing these schedules and running future iterations on *InO* cores, can provide the energy-savings of a leaner core while retaining all the performance advantage of an *OoO* core.

The process of remembering useful issue schedules on the *OoO* will henceforth be referred to as schedule *memoization* [34]. In many cases, running a short representative sample of millions of instructions on the *OoO* was enough to memoize execution phases of billions of instructions. *Mirage Cores* proposes a design wherein the *OoO* core functions as a schedule producer for its cluster of *InO* cores, when possible. This paper addresses challenges in its design and implementation, including the arbitration of the *OoO*'s time, the process of recording, transferring and replaying of schedules, and the architecture of the CMP.

While memoizing an application, the *OoO* finds repetitive schedules in its execution and records them in a *Schedule Cache* (*SC*), to be later consumed by an *InO* core. Since the producer *OoO* core is a scarce and costly resource, the benefits of *Mirage Cores* are dictated by how efficiently and intelligently it is employed. Its runtime must recognize memoizability within and across a diverse set of applications and determine the best usage of the *OoO*'s time. We develop a low-overhead runtime arbitrator that monitors consumer applications' executions and determines the best candidate for schedule optimization based on current performance, relative performance achievable on the *OoO* and Schedule Cache miss rates. The goal of the arbitrator can vary based on the requirement of the system. We build and analyze arbitrators optimized for both overall system throughput and equal resource allocation.

The *InO* cores are provisioned with a recently proposed *OinO* mode, which augments the core with the ability to interpret and correctly execute schedules from *OoO* cores [34]. They need fast access to these recorded schedules in order to achieve high performance. Each core has private L1 caches and a small Schedule Cache (*SC*), which are coherently connected to a shared L2 through the on-chip interconnect. The cost of migration of the application from the schedule producer to consumer includes transferring the contents of the *SC* and L1 caches, along with other state associated with the pipeline. These overheads limit the frequency of switching between cores to coarse periods of roughly a million cycles. The number of consumer cores that can be serviced by the producer core without compromising overall performance depends on the architecture of the cores themselves, migration overheads, and characteristics of the applications. To study these trade-offs, we evaluate several configurations for *Mirage Cores* on multi-application workload sets from SPEC 2006. Such a diverse set of application workloads stresses *Mirage Cores*'s arbitration schemes and architecture to reveal interesting observations regarding how heterogeneity among modern applications can be exploited. Although we focus on single-threaded benchmarks to evaluate our architecture in this paper, the concepts remain applicable to a multi-threaded environment. Section 6 qualitatively discusses the utility of *Mirage Cores* on multithreaded and server workloads.

This paper makes the following contributions:

- With the *Mirage Cores* architecture, we propose to replace many expensive *OoO* cores with one schedule producing *OoO* and many cheaper *InO* schedule consumers. A CMP with 8 *InO*s and 1 *OoO* allows us to trade off 15% performance for a 55% reduced energy consumption at 74% of the area as compared to a homogeneous CMP with 8 *OoO* cores.
- We propose a low-cost runtime arbitration mechanism that determines the best candidate for memoization using complexity effective metrics like *Schedule Cache* Misses Per Kilo Instruction (MPKI) and Instructions Per Cycle (IPC). We also compare against traditional homogeneous and heterogeneous CMPs with similar resources and show 30% improvement in overall performance while utilizing the *OoO* for 40% less time.
- We quantify the benefits of recording repeating issue schedules on the *OoO* core at coarse granularities of millions of instructions and replaying them on *InO* cores to achieve higher performance. We analyze configurations with various producer to consumer ratios and show that 1 producer core can create schedules for up to 12 consumer cores.

2 MOTIVATION

In this section we motivate how the *InO* cores in a *Mirage Cores* architecture can achieve near *OoO* performance at a lesser energy penalty, by leveraging memoized out-of-order issue schedules.

2.1 Core characteristics

Out-of-order cores are the result of decades of research to push the limits of single-core performance. Hardware structures like the Reorder Buffer, Reservation Stations, Load/Store Queues, and complex issue logic, allow them to overlook false dependencies and speculatively issue independent instructions out of program order while guaranteeing program correctness. This allows them to increase both Instruction Level Parallelism (ILP) and Memory Level Parallelism (MLP) and create dynamic schedules that are more optimized compared to those created by the best case static

[1] A *mirage* of a cluster of fast cores with underlying *InO* hardware

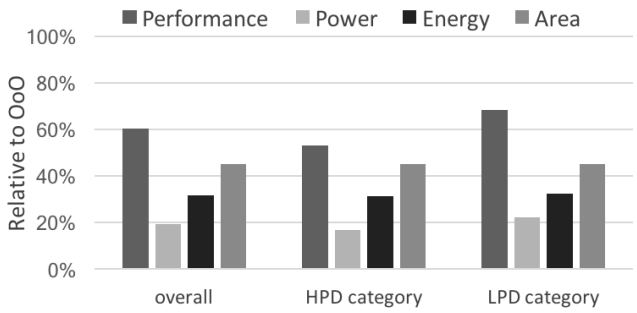


Figure 1: Performance, area and energy comparison between *OoO* and *InO* cores. Benchmarks are categorized into Low and High Performance Difference (LPD, HPD), based on their relative performances on *InO* and *OoO* cores.

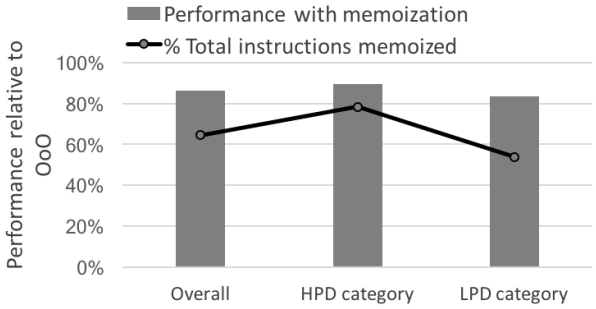


Figure 2: Fraction of the total execution that can be memoized, along with its effect on *InO* performance across benchmark categories.

compilers. Unfortunately, high performance comes at the cost of increased power consumption.

In-order cores on the other hand, are made of smaller, simpler logic that can only exploit the ILP statically and conservatively exposed by the compiler, and hence lose out on 40% of the *OoO*'s performance. Figure 1 quantifies the behavior of an *InO* core relative to an *OoO* core in terms of performance, power and energy, for the SPEC2006 benchmark suite. The *OoO* core is modeled as a 3-wide *OoO* core, while the *InO* core has an equal number of functional units and superscalar width, but is forced to issue instructions in program order. More details on the simulation infrastructure, benchmarks and energy models are given in Section 4. As seen in Figure 1, the *InO* consumes 1/5th the power of the *OoO*, thus operating ~3x more energy efficiently at less than 1/2 the area cost. Thus, a fixed area and power density budget can be met with a higher number of *InO*s than *OoO* cores.

To aid in analyses, we have divided the benchmark suite into two categories Low and High Performance Difference (LPD, HPD), based on their relative performances on *InO* and *OoO* cores (Section 4). The HPD category represents benchmarks that see higher relative performance on the *OoO* than the *InO* core, and hence significantly benefit from the *OoO*'s ability to reorder execution. Benchmarks like *hmmer*, with a high dynamic ILP and *mcf*, which benefits from parallel memory operations are examples in this category. The LPD category consists of benchmarks that could not take full advantage of the *OoO* core due to serialized or unpredictable code, like *gcc* and *gobmk*. Figure 1 also illustrates this classification. The HPD category performs faster on an *OoO* core, but also sees a greater power consumption because of higher utilization of the aforementioned core structures.

2.2 Benefits of Memoization

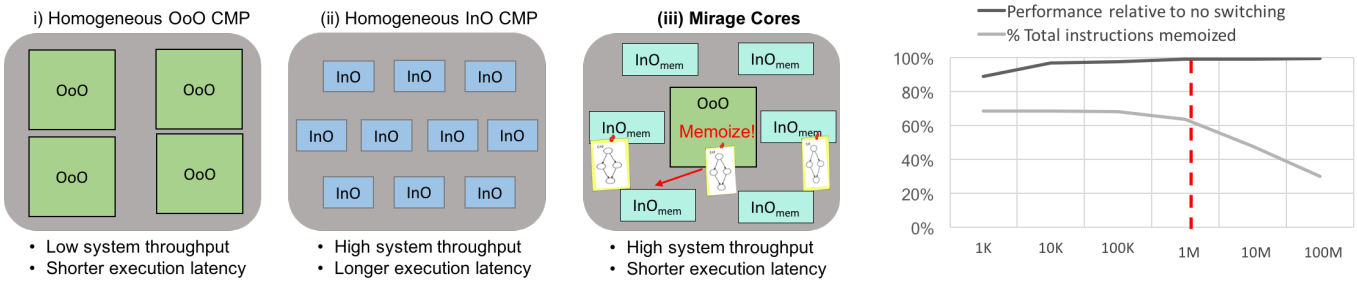
The previous section quantified the benefits of executing instructions out of order. When the instruction control and data flow is regular, as is the case in loops, the *OoO* core creates the same dynamic issue schedule for a given set of instructions (*a trace*) in the same context. When the trace's execution sees high variance and uncertainty, the *OoO* creates new schedules for the trace in that context to extract the most ILP. In fact, previous research[30] shows that only 19% of the *OoO*'s performance advantage is due to its ability to react to unexpected long latency events, by creating different issue schedules for the same trace. Since the majority of an application's execution is spent in loops, a substantial code fraction can be usefully memoized. Previous work [34] shows that 90% of all traces in a program's execution are scheduled similarly for 90% of their runtime. This exposes the wasteful nature of work done by the *OoO*. Lower energy consumption has been achieved on innovative architectures that enable storing such repetitive schedules and replaying them for future instances of the trace on the same or optimized cores [31, 34, 46].

Figure 2 also quantifies the fraction of the execution that can be memoized, for each category. These experiments were conducted under conditions with perfect control flow and an infinite Schedule Cache (*SC*), and thus represent the ideal scenario. The HPD category has a higher fraction of memoizable instructions. Most benchmarks in this category can efficiently exploit the *OoO* hardware because of their regular code structure, which translate to highly repeatable schedules. Exceptions include *mcf*, which exploits the *OoO*'s ability to dynamically recreate schedules around unpredictable and irregular long latency instructions to extract MLP. Once memoized, schedules are handed over to the *InO* core, which is augmented with an '*OinO*' mode to enable memoized schedule execution (details in Section 3) [34]. The performance boost achieved when such a mode is given the best-case memoized schedules is quantified in Figure 2. As expected, the LPD category gets a lower performance boost than the HPD category, because the benchmarks are both less memoizable and by definition gain minimal relative performance.

2.3 Concept of Mirage Cores

Figure 3a illustrates how *Mirage Cores*'s architecture can provide high throughput and single-thread performance with energy-efficiency. A CMP with four *OoO* cores (Figure 3a(i)) has the same area as that with ten traditional *InO* cores (Figure 3a(ii)). Although the *InO* CMP offers more system throughput and energy savings, it comes at the cost of unacceptable single-threaded performance. For the same area costs, three *OoO* processors can be replaced by six *InO* cores capable of achieving near *OoO* performance by executing memoized schedules (InO_{mem} cores in Figure 3a(iii)). This is the ncy suffers slightly, *Mirage Cores* offers higher system throughput, and also lowers power density and overall energy consumption.

An application is memoized by running on the *producer OoO* core for a fixed interval of time or the *memoize phase*, after which it transfers back to a *consumer* InO_{mem}. The interval length for



(a) Substitute bulky *OoO* cores with leaner *InO* cores capable of executing memoized *OoO* schedules.

(b) Memoizability reduces with increasing interval lengths, while migration overheads are unacceptable at smaller interval lengths.

Figure 3: Concept of *Mirage Cores*

Mirage Cores was empirically found as follows. Switching between cores incurs overheads of migrating not only the workload but also the penalty of cold L1 cache access. To measure performance lost due to migration, we execute two applications on three identical cores, with one application switching between two of them every *n* cycles, where n varies between 1000 to 10 Million cycles. As shown in Figure 3b, beginning with > 10% performance losses for a 1000 cycle interval, the migration penalty steadily reduces and is negligible (1%) beyond 1 Million cycles. To minimize migration costs, applications may therefore be allowed to migrate only at coarse intervals greater than millions of instructions.

Since the *OoO* is a scarce resource, it is imperative to run the application on the producer core just long enough, so as to satisfactorily capture all useful memoizable traces to be consumed in its subsequent execution. To observe the effect of phase length on memoizability, we run a similar oracle experiment as in Section 2.2 and measure the fraction of instructions usefully memoized, while changing how often the *OoO* could refresh the contents of an infinite *SC*. As shown in Figure 3b, for smaller intervals, a larger fraction of the execution was memoized, because an *OoO* that is allowed the opportunity to refresh its *SC* every 1000 cycles is more likely to capture relevant and updated versions of schedules in the cache. Traces might have schedules that change rapidly over time. As the interval length increases, the producer is more likely to encounter more than one schedule for a trace, and hence be unable to store one version to be used by the consumer. In order to have the best trade-off of low switching overheads with sufficient memoization, we choose **1 Million cycles** as our minimum memoize phase granularity. Applications have the opportunity to switch to the producer core for updating their *SC* at the boundaries of this phase. The arbitration mechanism of when the application is actually allowed to switch is described in Section 3.

3 MIRAGE CORES ARCHITECTURE

In this section we describe the architectural details of *Mirage Cores* along with the *arbitrator* policies and goals.

3.1 Architecture Overview

Practical constraints force CMP designers to pick one of two designs: a system made of many low-area and low-power *InO* cores that delivers high throughput and parallelism, or a system that consists of a few high-power, faster *OoO* cores to guarantee fast single thread performance. With *Mirage Cores*, we aim to find middle-ground by raising the performance of many *InO* cores with the support of a few *OoO* cores. Figure 4 illustrates a high level overview of *Mirage Cores*. Its description is as follows.

- This particular configuration consists of one *OoO* core partnered with a cluster of 4 *InO* cores. The *OoO* core not only provides high performance alternatives for the application to run, but acts as the main memoized schedule producer for its cluster of *InO* cores.
- The arbitrator is responsible for picking the best candidate to be transferred to the *OoO* core, if idle, and hence controls the extent of energy savings achieved. Factors like relative performance speedups and the potential for memoization contribute to this decision. We build and analyze arbitrators that can optimize for maximum system throughput or fair resource allocation, as described in Section 3.2.
- Both the *OoO* and *InO* cores' architecture need modifications that enable smooth recognition, storage and re-execution of memoized schedules. Section 3.3 describes these in detail.
- Each core has its own L1 Instruction and Data caches, and Translation Lookaside Buffers (TLBs). They all share access to a common L2 cache over a coherent bus. Migrating an application between the *OoO* and *InO* causes transfer of state between the cores over this bus. More details on this communication is given in Section 3.3.3.

3.2 Designing the Arbitrator

The mechanism that controls the scheduling decisions is perhaps the most important component of a Het-CMP, since it dictates whether the maximum efficiency is achieved. The scheduler or *arbitrator* is cognizant of the state of all the applications running on *Mirage Cores* and determines the most efficient usage of the lone *OoO*, subject to the goals set by the system designer. As shown in Figure 4, we design the arbitrator as a hardware extension, integrated in the *OoO* core and capable of polling performance counters from all active applications at predefined intervals. In Section 2.3, this interval was determined to be 1 Million cycles based on empirical studies.

In order to make the best decision for the oncoming interval, the *arbitrator* must estimate the future performance of applications in terms of some quantifiable metrics. Previous works make scheduling decisions by either relying on sampling all the heterogeneous

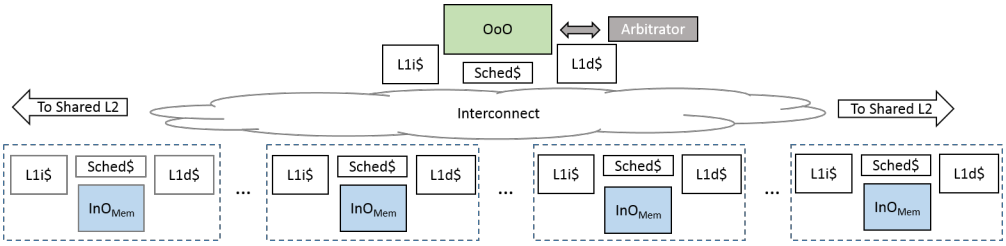


Figure 4: Overview of *Mirage Cores*'s architecture. In this configuration, 1 *OoO* is shared by 8 *InO* cores

core options periodically and assuming that past performance is an indication of future execution [4, 23] or by using static or dynamic models to estimate execution characteristics on the other cores [11, 33, 48]. In *Mirage Cores* we will use a combination of these methods, as described further. Scheduling decisions vary based on the overall goals set by the system designer such as maximizing energy-efficiency/system-throughput/fairness.

3.2.1 Energy-Efficiency Oriented Arbitration. *Mirage Cores* can provide the highest performance by ensuring that the underlying *InO* cores receive the most comprehensive set of dynamic schedules in their Schedule Caches (*SC*). At the same time, since *OoO* execution is expensive, the *arbitrator* aiming for maximum energy efficiency must activate the *OoO* with restraint, only when memoization exists. We found that the best metric to quantify the usefulness of memoization is the Misses Per Kilo Instruction (MPKI) observed on the *SC*(*SC-MPKI*).

The *SC* is populated by the *OoO* and only contains traces that both benefit from reordering and have repeating control and data flow. An *InO* core that executes the majority of its instructions from the *SC* (thus observing low *SC-MPKI*) is most likely experiencing near *OoO* performance, and therefore has no need to transition to the *OoO* yet. On the contrary, if it misses in the *SC* and is instead fetching program-order instructions from its L1 Icache (high *SC-MPKI*), it is either executing inherently non-memoizable instructions, or it has entered a new phase of execution and all traces in the *SC* are stale. Phase changes are prime opportunities for migration for schedule production. The *arbitrator* chooses an application with the highest Δ*SC-MPKI* metric above a fixed threshold for memoization, defined as follows:

$$\Delta SC\text{-}MPKI = \frac{SC\text{-}MPKI_{InO} - SC\text{-}MPKI_{OoO}}{SC\text{-}MPKI_{OoO}} \quad (1)$$

where $SC\text{-}MPKI_{OoO}$ represents the extent of memoizability of the application phase and is measured on the *OoO* during memoization. Applications like *bzip2* that are highly memoizable, measure a very low $SC\text{-}MPKI_{OoO}$. When transferred back to *InO*, they continue to hit in the *SC*, leading to a low Δ*SC-MPKI*. Such applications mainly see a rise in Δ*SC-MPKI* when they experience a phase change that renders the current *SC* stale.

Figure 5 illustrates this case using a timeline over 500 million cycles *bzip2*'s execution. Every point represents the Instructions Per Cycle (IPC) seen on the *OoO* core as well as the Δ*SC-MPKI* seen on the *InO* core for a 1 million cycle interval. The periods with constant IPC signify a regular loop, and hence the Δ*SC-MPKI* is near zero. Phase changes are evident when there is a shift in the level of IPC. The highlighted region shows that large changes in Δ*SC-MPKI* are seen in the immediate locus of a phase change, while the code transitions from one stable loop to another. The *arbitrator* migrates *bzip2* at this stage to the *OoO*, which refreshes the *InO*'s stale *SC* with more relevant schedules.

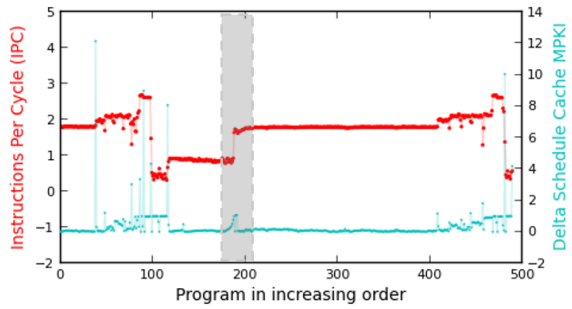


Figure 5: Relationship between Δ*SC-MPKI* and IPC for bzip2 where every point represents a 1M cycle interval across 500M cycles of its execution.

An *arbitrator* targeting maximum energy savings should only activate the *OoO* when there is scope for memoization. If none is observed, the *OoO* is powered down for that period to conserve energy. Applications like *astar* for example, have inherently no memoizability, and hence measure both high $SC\text{-}MPKI_{OoO}$ and $SC\text{-}MPKI_{InO}$. Because we use a ratio of these *SC-MPKI*s and not their absolute values to determine the next candidate for memoization, *arbitrator* will rightly avoid scheduling *astar* on *OoO*.

In some cases like *gcc*, due to high variability in control and data flow, schedules tend to repeat over only very short intervals of less than a million cycles. Although it observes a low $SC\text{-}MPKI_{OoO}$, its $SC\text{-}MPKI_{InO}$ tends to increase rapidly once it migrates back to *InO* and the execution moves to a non-memoized section of code. To prevent such an application from ping-ponging between the two cores, we divide Δ*SC-MPKI* by an additional decay factor determined by the duration since its last switch to *OoO*.

3.2.2 System Throughput Oriented Arbitration. Overall system throughput (STP) is the metric used by schedulers proposed in a substantial body of previous work [4, 23] to maximize throughput observed. It is defined as the mean of the speedups achieved by all applications.

$$speedup_i = \left(\frac{IPC_{InO(i)}}{IPC_{OoO(i)}}\right) \quad (2)$$

$IPC_{InO(i)}$ is the current IPC observed on *InO* by application *i*. $IPC_{OoO(i)}$ is measured on the *OoO* the last time application *i* ran there and is used to approximate its current IPC on *OoO*. The maxSTP *arbitrator* tries to maximize the STP at every interval boundary by migrating the application that incurs the most slowdown (lowest $speedup_i$) to the *OoO*. It also forcibly samples every application on the *OoO* at least once every 50 million cycles in order to avoid stale IPC_{OoO} values. In comparisons to existing Het-CMPs in Section 5, this *arbitrator* is employed on a traditional Het-CMP.

3.2.3 Fairness Oriented Arbitration. In some cases, especially with multiprogram workloads that care about Quality-of-Service (QoS), fairness of resource allocation is more important than STP [47]. Guaranteeing fairness, or making sure all threads get equal access to the *OoO*, will lead to a more balanced execution. To achieve this goal, the *arbitrator* migrates applications to the *OoO* in round robin order, such that every application has an equal time share. With memoization, an application can run almost as fast as on *OoO* hardware, and hence can meet the same QoS guarantees on the *InO*. The *OoO* could be powered down in such instances without causing a user noticeable performance degradation. The fair *arbitrator* assumes that the time an application spends executing memoized schedules on the *InO* counts towards *OoO* execution. It calculates the $Util_{(i)}$ metric to determine each application's *OoO* timeshare.

$$Util_{(i)} = \left(\frac{t_{OoO(i)} + t_{InOmemoize(i)} * speedup_i}{t_{overall}}\right) \quad (3)$$

where t is the time in cycles spent on each core for application *i*. $t_{InOmemoize(i)}$ is scaled by the speedup from Equation 2 for a fairer approximation of the time *OoO* would have taken. In round robin order, application *i* under consideration will be migrated to *OoO* only if either $Util_{(i)}$ is less than $1/(\#applications)$ or if Δ*SC-MPKI* falls below the threshold.

3.2.4 Arbitration in Software. The *arbitrator* described thus far is designed as part of the *OoO*'s hardware, giving it faster access to a core's performance counters and a shorter reaction time. A similar *arbitrator* built in the software layer will be restricted to coarser polling intervals of OS time slices (~10ms). Its effectiveness might be lower since memoizability in the code was shown to reduce sharply for coarser intervals (Section 2.3). The OS is oblivious to *Mirage Cores*'s current design but it can be leveraged to provide the runtime with higher system-level metrics, enabling more nuanced scheduling decisions.

3.3 Designing the Core Architectures

Several works [8, 31, 34, 49] propose methods that use *OoO* pipelines to create and record optimized schedules for another, more efficient pipeline. We will borrow the ideas most recently proposed by the DynaMOS system [34] to facilitate correct memoization in *Mirage Cores*.

In this paper, we refer to the issue sequence recorded by an *OoO* core for a trace as a *schedule*. A *trace* is a sequence of instructions between two consecutive backward looping branches. They're around 50 instructions long on average and capture the most recurring code paths in loops and functions; deeming them good candidates for memoization. Both traditional *OoO* and *InO* cores have to be supplemented by structures needed to enable efficient and accurate memoization.

3.3.1 Designing the OoO core. In order to memoize schedules, the *OoO* core must be able to recognize (a) when a trace is repetitive, and (b) if its instructions are scheduled in the same order. A naive way to find repeating schedules is to match the cycle-by-cycle schedule order of instructions in two trace iterations; which is expensive, both in terms of storage and computation. Instead we track performance metrics associated with a trace's schedule, like execution time, IPC, memory characteristics, branch misses and number of reordered instructions. We make the approximation that two trace executions with matching metrics are likely to have been issued in the same order.

Hardware tables proposed by Padmanabha et. al [34] maintain information about repeatability of schedules. Traces that are deemed memoizable are stored in a specialized Schedule Cache (*SC*) in a format that is useful to the *InO* core. Since the *SC* stores memoized schedules across millions of instructions, our algorithm is more conservative compared to prior work and only memoizes schedules that repeat with high levels of confidence. The overheads include an 8kB *SC* and 0.3kB of hardware tables.

3.3.2 Designing the InO core. To correctly execute memoized schedules on an *InO*, the *OinO* mode [34] was introduced. The modifications include:

Atomic Execution: Since the *InO* does not track correct program order for memoized code, it lacks the ability to detect and resolve unexpected events like branch misspeculations or memory aliases. To circumvent this issue, the *OinO* mode forces memoized schedules to execute atomically. On misspeculation, the whole trace is squashed and execution is restarted in original, non-memoized program order.

PRF: The *OoO* uses register renaming to eliminate false register dependencies. By renaming every architectural register (AR) to a unique physical register (PR), instructions with the same destination AR can be reordered on the *OoO*. To guarantee correct data flow across registers, the *OinO* must honor this register renaming. It is therefore supplemented with an expanded register file that can be used to store multiple versions of an AR and access them as per the memoized schedules. We allow every AR to map to at most 4 unique PRs, resulting in a 128 entry PRF. Bookkeeping adds an additional 28 bytes of storage. A bigger PRF and tables adds 14% dynamic energy to the *InO*.

LSQ: Loads and stores are reordered relative to each other in the memoized schedules generated by *OoO*. Memory alias errors could occur; for example, if a load is speculatively moved ahead of older stores that write to the same memory location. A specialized LSQ is added for the *OinO* mode, that tracks the original sequence of memory operations in a trace in order to detect an alias event. A fixed-size meta-data block, added to every recorded schedule, is used to store program-sequence ordering of memory operations. This allows loads and stores to be inserted into the LSQ in original program sequence, thus ensuring that aliases between them are detected correctly. In the event of no errors, all the stores are committed in order and the LSQ is flushed for the next trace. Misspeculations are handled by flushing the whole LSQ and restarting the trace. The added 32 entry LSQ contributes 5.5% overhead to the

dynamic energy of *OinO*, and the meta-data block adds 20B to each recorded schedule.

Schedule Cache: An 8KB specialized *SC* stores the schedules memoized and transferred from the *OoO* and specialized fetch and decode stages assist in interpreting them. It is designed similarly to Trace-caches [40], where the first block is pointed to by its set ID and a special End of Trace marker denotes its end. Its eviction policy first evicts existing traces that have been deemed unmemoizable and then falls back to LRU. A write to the *SC* is expensive, since each trace is compacted to prevent fragmentation and hence must be done conservatively. The *SC* contributes 10% towards increased leakage energy, but offers to reduce L1 Icache access energy by satisfying fetch requests from a relatively smaller cache.

Trace misspeculations are expensive since they incur the penalty of abort and replay and hence must be avoided. *Mirage Cores* employs a trace selection algorithm that is heavily biased against traces that mis-speculate thus maintaining this penalty to around 0.3% of execution time on average. Section 5 shows a break down of the power overheads added by all the components necessary to enable the *OinO* mode in comparison to *InO* and *OoO* cores.

3.3.3 Migration between cores. An important concern with Het-CMPs is the cost associated with migrating an application between cores. On migration, all of the active core's state, including the register file, program counter, control bits, store buffer entries, etc. must be explicitly stored into memory and its pipeline flushed. On resuming on the other core, there is a delay associated with rebuilding and warming up the stateful structures like L1 caches and branch predictor. With memoization, we introduce an added transfer of the 8KB *SC* contents to the consumer. As shown in Figure 4, a bus serves as a point of serialization for all communication outside the core and hence can contribute towards contention and slowdown. Preliminary studies (Section 2.3) showed that while the migrating benchmarks suffered slight performance losses, other benchmarks on the same bus saw negligible effects. This, coupled with the low migration frequency in our analysis, motivated us to re-use the shared high-bandwidth coherent bus interconnect between the L1/L2 caches for migrating contents of the *SC*s. If the bandwidth usage of benchmarks is expected to be high, an alternative is to include an interconnect exclusively for *SC* contents transfer, trading off increased area for higher performance. This interconnect can be simplified because it needs to support only unidirectional communication, from schedule producing *OoO* to consuming *InO* cores.

4 METHODOLOGY

Category	IPC Ratio	Benchmarks
High Performance Difference (HPD)	< 60%	cactusADM, bwaves, gamess, gromacs, h264ref, hmmer, leslie3d, libquantum, mcf, milc, povray, tonto, zeusmp
Low Performance Difference (LPD)	>= 60%	GemsFDTD, astar, bzip2, calculix, dealII, gcc, gobmk, namd, omnetpp, perlbench, sjeng, wrf, xalancbmk

Table 1: Classification of benchmarks by performance

Architectural Feature	Parameters
OoO	3 wide superscalar @ 2GHz 12 stage pipeline 128 entry ROB 128 entry integer register file 256 entry floating-point register file 8KB Schedule Cache
InO	3 wide superscalar @ 2GHz 8 stage pipeline 128 entry integer register file 128 entry floating-point register file 8KB Schedule Cache
Memory System	32 KB L1 iCache @ 2 cycles 32 KB L1 dCache @ 2 cycles 2 MB Shared L2 Cache with stride prefetcher @ 15 cycles 8192MB Main Mem @ 120 cycles 32 B L1-L2 bus @ 2Ghz

Table 2: Experimental Core Parameters

4.1 Workloads

We evaluate the *Mirage Cores* design on multi-application benchmark mixes consisting of 27 applications from SPEC 2006 benchmark suite[12] compiled for the ARM ISA. The benchmarks were split into two categories to understand how specific application characteristics affected *Mirage Cores*'s benefits. These categories, represented in Table 1, were picked based on their relative performance on *OoO* vs *InO* (Section 2.1). The aim behind *Mirage Cores* is to achieve high *InO* utilization with minimal performance losses for not only applications that are inherently slow (LPD category) but also for the HPD category of applications which would traditionally suffer great performance losses on *InO* cores.

The first 5 billion instructions in each benchmark were analyzed using Simpoints [44], and the highest weighted 1 billion instruction window per benchmark was used for all our experiments. We create 32 workload mixes, where a mix contains the same number of workloads as the number of *InO* in the configuration being studied. To analyze applications both within and across categories, we pick 10 mixes each exclusively within a single category, and 22 mixes with a random mix of both categories. Each workload mix is executed until every application within it completes a billion instructions. Since different applications can have different rates of execution, we restart applications that finish early while waiting for the rest to finish. Section 6 discusses *Mirage Cores*'s potential behavior on multithreaded applications.

4.2 Simulation Methodology

We use the cycle accurate Gem5 simulator [6] to model *Mirage Cores*, including the *InO*, *OinO* and *OoO* cores, caches and memory. The producer *OoO* core is a deeply pipelined, high performing out-of-order processor capable of issuing and committing three instructions per cycle. It boasts of large structures like the ROB and Reservation Stations that allow it to extract ILP and MLP from a large instruction window, creating the most efficient and optimized issue schedule for its hardware. The energy efficient *InO* is an stall-on-use in-order core that has the same superscalar width and FUs as the *OoO*. This configuration was chosen to allow the simplistic transfer of issue schedules between the *OoO* and *InO*. Each core within a cluster has private 32KB L1 instruction and data caches, and are each augmented with a 8KB schedule cache. This size was picked empirically by testing the effects of varying *SC* sizes on

performance and energy savings. The increases in relative STP as cache size increases is not as linear as the energy overheads, and was observed to plateau around 8KB. Since *Mirage Cores* aims at both area and energy-efficiency, we choose an 8KB cache for our final design as it provides the best performance per mm^2. They all share a 2MB per benchmark L2 cache over a 32B wide coherent bus interconnect. We use Gem5's relaxed consistency model (ARMv7a) and ensure that traces break at barriers and fences boundaries. Since a trace is atomic, its stores aren't revealed globally until correct execution. Table 2 describes the configurations used in detail. Our design is loosely inspired by clusters like ARM's Cortex A53 [2] consisting of 4 *InO* cores with private L1 caches that share access to a L2 over a high bandwidth coherent bus.

To model overheads of migrating an application between cores, we measure the costs incurred due to drain and refill of pipeline state and L1 caches and include that in our experiments. We generously approximate a 1000 cycle penalty to transfer an 8KB *SC* over the 32B coherent bus. Section 5 includes a breakdown of these costs.

We use the McPAT modeling framework to estimate area, static and dynamic energy consumption of the core and L1 caches [27]. We assume instantaneous power-gating of the *OoO* core when not in use. The energy overheads imposed on the *InO* core due to addition of the *OinO* mode, including the added leakage energy due to *SC*s is detailed in Section 5.

5 RESULTS

The benefits of *Mirage Cores* come from its runtime's ability to navigate the memoizability across diverse applications and exploit the available architecture.

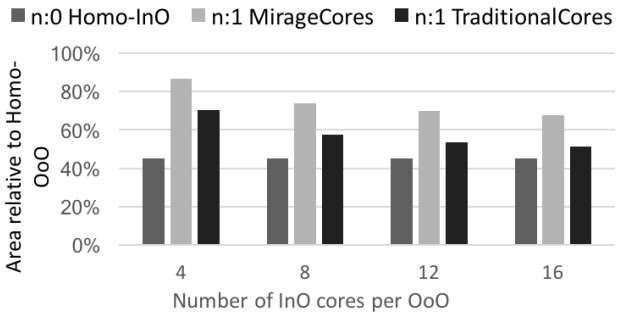


Figure 6: Adding an *OoO* core as well as *OinO* mode increases *Mirage Cores*'s area consumption relative to baseline

5.1 Architecture Configurations

Mirage Cores consists of 1 *OoO* schedule producer with a cluster of *n InO* schedule consumers, henceforth referred to as a *n*:1 configuration. Unless stated otherwise, the **baseline** for the following graphs is a homogeneous CMP (Homo-CMP) with *n OoO* cores (0:*n*). In addition, we also include comparisons to a Homo-*InO* CMP (*n*:0). This bar usually represents the lower limit in terms of performance, area and energy consumption. We compare *Mirage Cores*'s architecture to a corresponding *n*:1 traditional (no memoization) heterogeneous CMP (Het-CMP). Figure 6 compares the area of these architectures while varying *n* from 4 to 16. The traditional 4:1 Het-CMP, for e.g., introduces 1 *OoO* to the 4:0 Homo-*InO* CMP, increasing its area by 55%. The *OinO* mode adds structures described in Section 3.3.2, increasing area consumed by the 4:1 *Mirage Cores* by an additional 23%.

5.2 Throughput Aware Arbitration

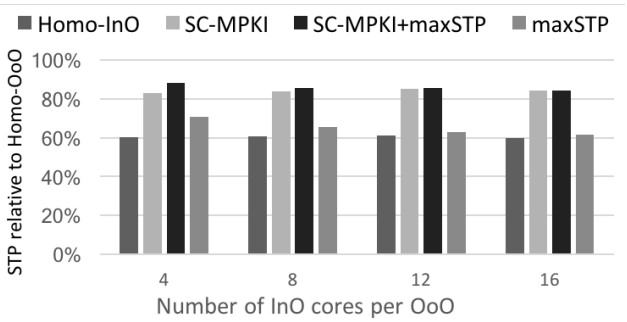


Figure 7: A traditional Het-CMP (*maxSTP*) achieves lesser speedup with a n:1 configuration than *Mirage Cores* (*SC-MPKI*, *SC-MPKI*+maxSTP)

The runtime *arbitrator*s dictate the overall performance and energy consumption of the above architectures. The proposed *SC-MPKI arbitrator* (Section 3.2.1) maximizes energy-efficiency by leveraging the *Mirage Cores* architecture and engaging the *OoO* to generate memoized schedules. As a comparison, the runtime for a traditional Het-CMP was described in Section 3.2.2 as per prior work[4, 23]. This scheduler maximizes STP at every interval boundary of 1 Million cycles, by reserving the *OoO* for the InO candidate with the highest expected speedup. Henceforth *maxSTP* will refer to the combination of this *arbitrator* acting on a traditional Het-CMP. The *SC-MPKI*+maxSTP *arbitrator* similarly aims to maximize throughput on *Mirage Cores*.

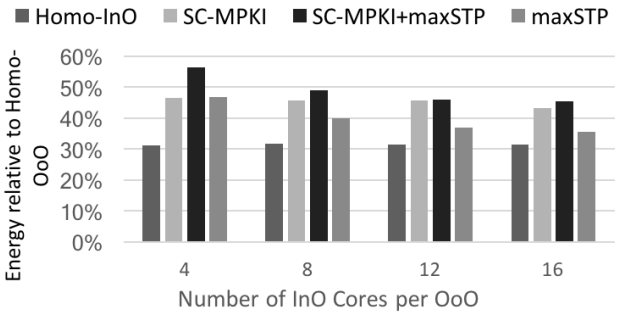
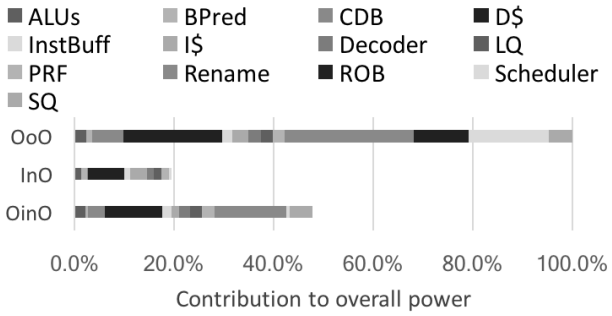
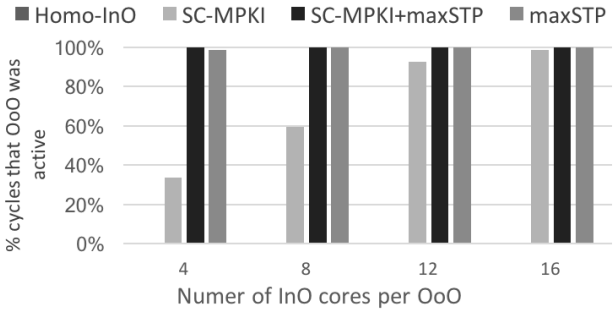


Figure 8: Relative energy consumption reduces as *n* increases

Performance: Figure 7 shows how these *arbitrator*s affect overall STP of a given system. In the 8:1 configuration, for instance, the *maxSTP arbitrator* gains 8% speedup over a 8:0 *InO* Homo-CMP. The *SC-MPKI* exploits the *OinO* mode to increase performance by 39% on the other hand, and provides nearly the same STP as the *SC-MPKI*+maxSTP *arbitrator*. Further sections will provide a deeper understanding of how the *SC-MPKI arbitrator* intelligently exploits



(a) Structures added for *OinO* mode incur higher dynamic power consumption than *InO*



(b) Lower *OoO* utilization conserves higher energy

Figure 9: The energy consumed by various CMPs depends on the constituent architectures as well their utilization

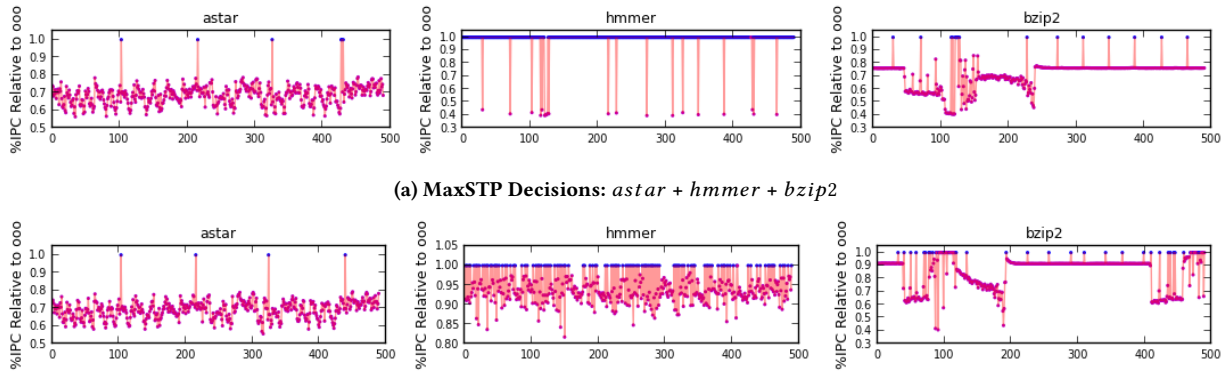
memoizability in and across applications. Overall, it achieves 84% of a 0:8 *OoO* Homo-CMP's performance with 26% less area. The area conserved can be recycled to include 2 more *OoO*s, or 3 more *OinO*s, or even uncore components like bigger caches, increasing STP even further.

Energy consumption: The energy consumed by a CMP, shown in Figure 8, depends on (a) the power characteristics of constituent cores as well as (b) the amount of time they were utilized for. Figure 9a shows that *OinO* mode additions like the bigger PRF and LSQs add a premium on traditional *InO* dynamic power. However, the lack of an ROB and an aggressive scheduler as compared to the *OoO*, allows it to retain most of the efficiency of an *InO*. Additionally, fetching trace blocks from a smaller *SC* allows it to maintain lower Icache and branch-prediction power. Although the *OinO* core increases the dynamic power of an individual *InO* core by 2.4x, its shorter runtime ensures a more conservative energy cost (60%). The *OoO* core continues to be the highest power burner, at 2.1x the power of an *OinO* core, and hence curtailing its utilization can lead to bigger energy savings. Figure 9b, displays the fraction of overall cycles that *OoO* was utilized on average for various configurations. The *SC-MPKI arbitrator*'s awareness of memoizability across applications affords it the highest conservatism in terms of *OoO* usage. For instance, it powers the *OoO* down for 40% of execution for the 8:1 configuration allowing energy savings of 54% relative to baseline. In contrast, both *SC-MPKI*+maxSTP and *maxSTP* schedulers aim to maximize throughput and hence always utilize the *OoO* to speed up the slowest application. Overall, we observe that the *SC-MPKI*+maxSTP provides very little benefits over *SC-MPKI*, since the latter exploits most of the high-return opportunities to migrate. Additionally, speeding up $1/n$ applications in a mix using *maxSTP* on a traditional Het-CMP is not nearly as beneficial as speeding up all n applications using memoization and simple heuristic based scheduling. As n increases, so does contention for the sole *OoO*, leading to tapering STP and energy wins. The *SC-MPKI* utilizes *OoO* at 100% after the 12:1 configuration. Without considering implementation complexities, we can conclude that the upper limit for designing a memoization-based cluster in *Mirage Cores* is 12:1. The next two sections will focus on understanding how the different *arbitrators* work in context to real applications and their effects on our workload mixes.

5.2.1 Case study. We study a workload mix of *astar*, *bzip2* and *hmmer* for a 3:1 configuration. Figure 10 portrays benchmark execution timelines over 500M cycles with each point representing the speedup observed during a 1M cycle interval relative to *OoO* execution. Points are marked in blue if they were scheduled on the *OoO* (and hence by definition have speedup of 1) and red otherwise. Figures 10a and 10b illustrate decisions made by the *maxSTP* and *SC-MPKI* schedulers respectively.

astar, a LPD category benchmark, has the lowest slowdown on *InO* and hence is not a prime candidate for maximizing STP as described in Section 3.2.2. Due to high variability in its control flow, it is not a good candidate for memoization either. Both schedulers in this study therefore, avoid migrating *astar* to *OoO*, except for periodic sampling, as illustrated by the majority of red points in its timeline. The other two benchmarks are more regular and show high memoization potential. *hmmer* experiences very high slowdowns (> 60%) on the *InO* and hence is chosen to migrate to *OoO* by the *maxSTP* scheduler for most of its execution, except around the 130M cycle mark (Figure 10a), where it competes with *bzip2*. For the rest of execution, *bzip2* is starved of *OoO* time. In contrast, the *SC-MPKI arbitrator* executes an application on *OoO* only when memoized execution and/or performance drops. Once memoized, *hmmer* can consistently achieve an average of > 90% of *OoO*'s performance while using the *OoO* a lot less, as evidenced by the majority of red points in its timeline in Figure 10b. This allows *SC-MPKI* to either schedule *bzip2* on the idle *OoO* more often to achieve a much higher average IPC or power it down to conserve energy. Thus, although *SC-MPKI* lowers the performance of *hmmer* by 7%, it increases STP by raising *bzip2*'s performance *and* conserves energy by powering down *OoO*.

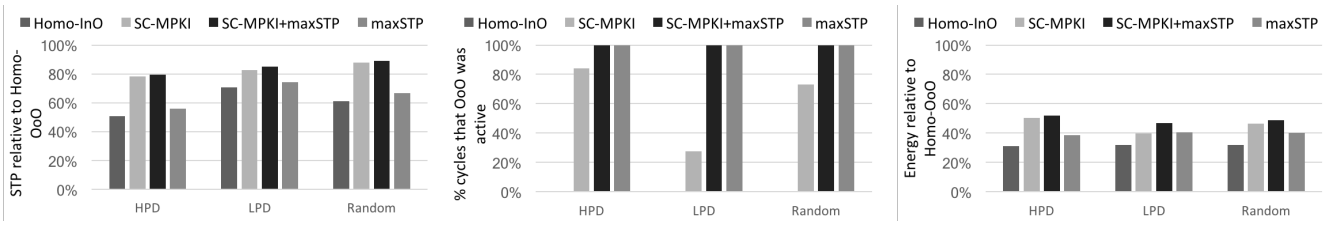
5.2.2 Analyses of Benchmark Categories. Diving deeper into the 8:1 configuration we evaluate *Mirage Cores* for the predefined benchmark categories. As was shown in Section 2.2, while the HPD category has inherently lower performance on *InO*s, its regular execution structure renders it more memoizable. Hence *SC-MPKI* achieves a 54% speedup compared to a Homo-*InO* CMP (Figure 11a) by engaging the *OoO* at 80% as a schedule producer for this category (Figure 11b). LPD benchmarks on the other hand, gain limited speedups on the *OoO* because of factors like high branch



(a) MaxSTP Decisions: $astar + hmmer + bzip2$

(b) *SC-MPKI* Decisions: $astar + hmmer + bzip2$

Figure 10: Case study marking the differences between the *SC-MPKI* and *maxSTP arbitrators*, for a 3 *InO* to 1 *OoO* core configuration. Every point in the timeline denotes the IPC observed for each 1M cycle interval over a 500M cycle period. If a particular interval executed on the *OoO* it is colored blue, while red represents intervals executed on *InO*.



(a) *Mirage Cores* benefits HPD benchmarks more

(b) HPD benchmarks find more opportunity to utilize the *OoO*

(c) LPD benchmarks consume less energy because of lower *OoO* utilization

Figure 11: Delving into the benefits of Het-CMP *arbitrators* and architectures based on application characteristics for a configuration of 8 *InO* with 1 *OoO* cores. *Mirage Cores* works best for a heterogeneous mix (Random) of HPD and LPD benchmarks.

misprediction rates (*gobmk*, *astar*) and lack of MLP (*gcc*) [7]. *SC-MPKI* barely sees opportunity to migrate them to the *OoO*, reflected in the 27% *OoO* utilization shown in Figure 11b. The speedup it gains from migration is only 12%, although its low *OoO* utilization saves it 10% more energy than the HPD category (Figure 11c). The *maxSTP* and *SC-MPKI*+maxSTP *arbitrators* on the other hand, maximize *OoO* utilization for LPD benchmarks despite minimal scope of speedup. The random category consists of benchmarks from either family, and hence represents the average case. Benchmarks that incurred *OoO* starvation in the HPD category due to high contention were allowed more access when paired with non-memoizable benchmarks in the random category, leading to higher performance. We can conclude that *Mirage Cores* works most efficiently with a heterogeneous mix of workloads, with varying *OoO* requirements.

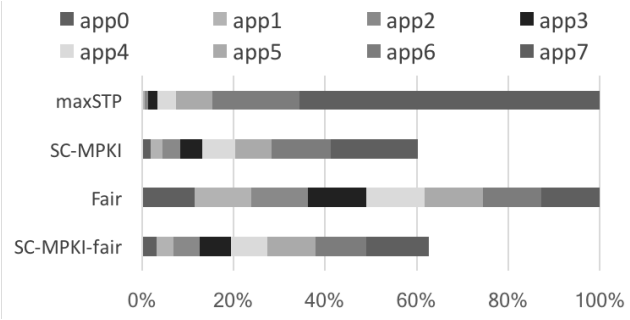


Figure 12: Utilization of the *OoO* per benchmark in a workload mix for the 8:1 configuration

5.3 Arbitrators for Equal Resource Sharing

The *arbitrators* in the previous section only allocated an application to *OoO* if it showed potential for either memoization or speedup. This could lead to unfair allocation of the lone *OoO* resource to individual applications. Figure 12 illustrates time spent by the *OoO* executing each constituent benchmark in a 8:1 configuration. The *maxSTP arbitrator* unfairly favors one application while starving others, which might be unacceptable in a given system. The *SC-MPKI* reduces overall *OoO* utilization, but is still prone to favoring some benchmarks over others. Under equal resource distribution in an 8:1 configuration, every benchmark should be allowed at least 1/8 time on *OoO*, or in other words, none should be allowed more than a 12.5% share. A *fair arbitrator* on a traditional Het-CMP will assign individual benchmarks in a mix to *OoO* in round-robin order, as shown in Figure 12. However, as shown in Figure 13,

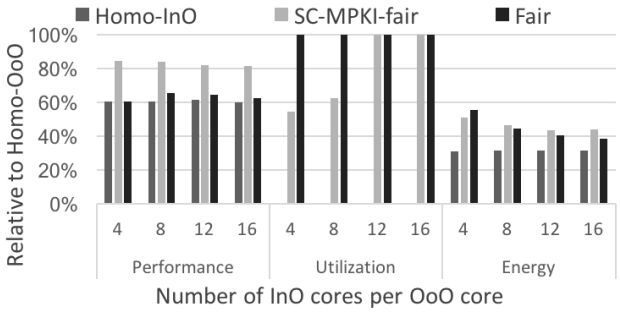


Figure 13: Overall evaluation for fair schedulers compared to Homo-*OoO* and *InO* CMPs

such an *arbitrator* incurs the energy penalty of utilizing the *OoO* at a 100% without the performance benefits. This is because some benchmarks inherently don't achieve speedups on an *OoO* as shown in previous sections. Additionally, migrating between cores at every interval boundary incurs very high performance and energy costs. The *SC-MPKI*-fair *arbitrator* minimizes this unnecessary migration, while maintaining a notion of fairness. As described in Section 3.2.3, it avoids migrating the next application in line to the *OoO* if it already experiences sufficient speedups from memoization. It thus achieves both performance and energy benefits (Figure 13) while ensuring that no benchmark occupies more than 1/8th of *OoO* time (Figure 12). Benchmarks experience <12.5% utilization of *OoO* with *SC-MPKI*-fair not because of starvation but because the *arbitrator* found them highly memoizable, and at their turn decided to conserve energy by turning off the *OoO* instead. Thus *Mirage Cores* can ensure energy-savings even while maintaining fairness between applications unlike a traditional Het-CMP.

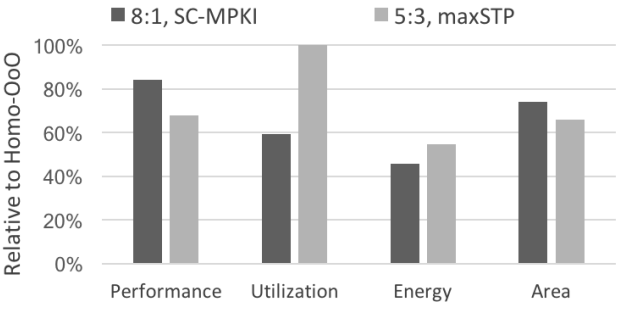


Figure 14: Area-neutral comparison of an 8:1 *Mirage Cores* configuration with a 5:3 traditional H-CMP

5.4 An Area Neutral Study

Hitherto, the *Mirage Cores* architecture was compared to an equivalent n:1 CMP of traditional cores. Figure 14 shows how the 8:1 configuration compares to an approximately area-neutral one with 3 *OoO* and 5 *InO* cores. Incidentally, this was picked by Kumar et.al [24] as the best Het-CMP configuration with traditional cores. We assume instantaneous transfer cost for this experiment. As observed, having 2 more *OoO* cores not only results in a 20% higher energy consumption for the 5:3 CMP, it is also slower by 23% as compared to using 1 *OoO* core as a schedule producer.

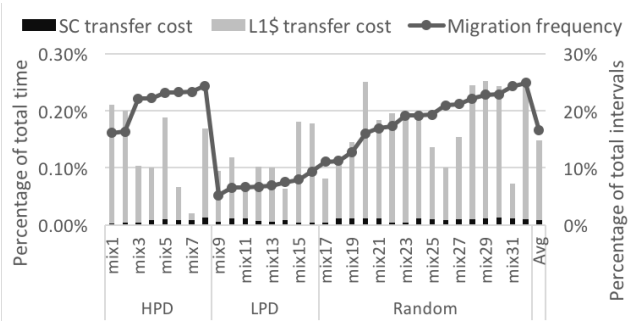


Figure 15: Transfer costs associated with *Mirage Cores*

5.5 Cost of Core Migration

Het-CMPs that allow an application to switch between cores incur the cost of draining and refilling its pipeline and cache state. In Section 2 we observed this overhead was dominated by L1 cache refill at 4 μseconds on average. *Mirage Cores* faces the additional penalty of migrating the contents of an 8KB *SC*, which on a 2GBPS, 32B bus is approximated at ~1000 cycles. Figure 15 breaks transfer cost per benchmark category into these components. It also shows the frequency of migration initiated by *SC-MPKI*. The HPD category of benchmarks are expectedly migrated to *OoO* more frequently for the sake of schedule production, while the LPD category do tend to remain on the *InO*. Overall we observe that the transfer overheads in *Mirage Cores*'s execution tend to be insignificant at 0.15%.

6 DISCUSSION

In this paper *Mirage Cores* was evaluated for SPEC 2006 as it represents a wide range of single-threaded applications that run on general purpose cores and stress a system's memory, processor and compiler. We consider multi-application workloads for our evaluation to showcase how *Mirage Cores* reacts to heterogeneity across application behaviors and more importantly, exploits it to gain higher energy savings, as proven in Section 5.2.2. *Mirage Cores* can potentially enable energy-efficient execution for multi-threaded programs as well. If threads perform homogeneous work, the *OoO* core can be used to memoize a single thread's repeatable phases and distribute it among all *InO*s in its cluster, thus speeding up all threads with one memoization attempt. As shown in previous work, the *OoO* can also be used to speed up bottlenecks, long running threads and critical sections [3, 25] as well as help provide QoS guarantees for long tail latencies [37]. The arbitrator could be augmented with thread criticality and runtime predictors [5, 10] to make better scheduling decisions. Although server workloads favor *Mirage Cores*' architecture style with less aggressive OoOs and more throughput [14], they tend to have much bigger instruction footprints, with more irregular fetch patterns. Currently in our design, memoization is dynamically detected based on repetitive fetch patterns and the architectural configurations are tuned for short, predictable phases. More evaluation would be required to judge these workloads' proclivity for memoization, and the most

suited core and cache configurations to serve them. A quantitative analysis on multithreaded workloads is in the scope of future work.

7 RELATED WORKS

This section surveys prior work in three broad areas relevant to *Mirage Cores*'s design.

7.1 Heterogeneous Cores Architectures

Heterogeneous processors designs range from migrating thread context across different architectures with same and different ISAs, to dynamically scaling voltage and frequency, and dynamically adapting hardware resources as needed.

Kumar et al. [23] wrote one of the seminal papers that proposed an energy-proportional multicore system with OoO and InO cores. Industry has successfully commercialized this proposal, with Qualcomm's Snapdragon [38] and Nvidia's Tegra [32] being a few notable examples. These systems boast of multiple ARM big.LITTLEs [15], a OoO-InO heterogeneous multicore system. Other industry products, like IBM's Cell [20] and Intel's vPro [17] and AMD's Accelerated Processing Unit [1] allow special code to be run on customized elements. To circumvent high switching costs associated with migrating between cores, works have proposed novel Het-CMPs that share stateful structures [13, 29] or reduce distance between them using 3d stacking [41, 50]. Another approach to heterogeneity is to reduce the voltage and frequency of the core (DVFS) to improve the core's energy efficiency at the expense of performance [18, 39, 51].

7.2 Scheduling Multicore Systems

Substantial research exists for scheduling on single ISA heterogeneous cores, both to achieve high throughput and fairness. Static approaches use offline profiling to leverage the relationship between inherent characteristics of a program and its resource usage, in order to determine the appropriate core to run on [11, 43]. Age-based scheduling[26] predicts age of programs and schedules the oldest thread on the big core. Becchi and Crowley [4] show that an IPC driven sampling based scheduling scheme outperforms a static based approach. Bias scheduling [22] is another scheme that determines a program's bias towards a specific core configuration based on dynamic internal and external stalls due to architecture variations and resource availabilities. The BIS scheduler exploits a similar clustered architecture like ours to reduce stalling on lock acquisition and bottleneck threads by running them on the *OoO* for multithreaded workloads. Rather than relying on sampling the performance on both cores, Van Craeynest et al. [48] propose prediction models which use IPC, MLP, and ILP to predict the performance on the inactive core. A regression model based approach [52] is used to schedule webpages with varying, diverse characteristics to underlying heterogeneous architecture. Fairness aware scheduling has been proposed for both heterogeneous core and cache architectures [28, 47]. All these above methods of scheduling on Het-CMPs are orthogonal to our work and can be combined with our memoization heuristic to provide more efficient application scheduling.

7.3 Optimizing Instruction Schedules

Storing instructions in the order they execute helps aid the efficiency of fetch [40, 42]. Compilers use profile-based static scheduling mechanisms [16] or run-time binary optimization [36] to create optimized instruction schedules. Even the best compilers fail to achieve the high performance of schedules created by the hardware, however, due to lack of dynamic runtime information. Nvidia's Project Denver [9] is a similar endeavor to ours, where a dynamic code optimizer (DCO) in software extracts ILP for recurring code and stores it in an optimization cache to be used by an *InO*. Since this is a software process, the program is stalled when it is being optimized. By using an on-chip core to memoize schedules, forward progress can continue to be made.

Several other works propose caching OoO schedules, and replaying them either on the same core with structures turned off [46, 49] or on different pipelines [8, 13, 31, 34]. DynaMOS proposes the *OinO* mode of the *InO* core that we use for interpreting memoized schedules. While architectures like HBA[13] and DynaMOS [34] use one core's frontend to feed into OoO and InO backends, we use 2 separate, self-sufficient cores. This shared frontend gives their architecture the ability to allow frequent migration every few hundred instructions allowing the OoO to constantly update the schedule cache with new schedules. *Mirage Cores* observes memoizability over coarser intervals, while having to arbitrate between multiple candidates for the *OoO* resource. The concept of sharing a front-end to optimize instruction fetch for many cores at the backend is also adopted in the Confluence architecture [21] for server-workloads.

8 CONCLUSION

Although there are more transistors on chip today, practical power and thermal constraints limit the deployment of homogeneous multicore systems with many big *OoO* cores. More small *InO* cores can fit in a given area, but their low performance limits their widespread usage. With *Mirage Cores*, we build a heterogeneous multicore system that combines the performance advantages of an *OoO* with the area and energy benefits of an *InO*. *Mirage Cores* pairs a few *OoO*s with a cluster of many *InO*s. By allowing the *OoO* to memoize schedules for the *InO* cores in its cluster, *Mirage Cores* raises the performance of the system, giving the impression of a core with all *OoO*s. Overall, with an 8 *InO* per *OoO* configuration, *Mirage Cores* can achieve on average 84% of the performance of a Homo-CMP with 8 *OoO* cores, while conserving 55% of energy and 25% of area costs.

9 ACKNOWLEDGEMENTS

This work is supported in part by ARM Ltd and by the National Science Foundation under grants SHF-1217917, XPS-1628991, SHF-1527301 and CAREER-1652294. The authors would like to thank the fellow members of the CCCP research group, and the anonymous reviewers for their time, suggestions, and valuable feedback.

REFERENCES

[1] AMD. 2014. AMD Compute Cores. (2014). https://www.amd.com/Documents/Compute_Cores_Whitepaper.pdf.
[2] ARM. 2015. Cortex-A53 Processor. (2015). http://www.arm.com/products/processors/cortex-a/cortex-a53-processor.php.

[3] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. 2005. The impact of performance asymmetry in emerging multicore architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE Computer Society, 506–517.

[4] Michela Becchi and Patrick Crowley. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers*. ACM, 29–40.

[5] Abhishek Bhattacharjee and Margaret Martonosi. 2009. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 290–301.

[6] N. Binkert and others. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.

[7] Sarah Bird, Aashish Phansalkar, Lizy K John, Alex Mericas, and Rajeev Indukuru. 2007. Performance characterization of SPEC CPU benchmarks on intel's core microarchitecture based processor. In *SPEC Benchmark Workshop*.

[8] Bryan Black and John Paul Shen. 2000. Turboscalar: a high frequency high IPC microarchitecture. *ISCA27* (2000), 36–44.

[9] Darrell Boggs, Gary Brown, Nathan Tuck, and KS Venkatraman. 2015. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro* 35, 2 (2015), 46–55.

[10] Qiong Cai, José González, Ryan Rakvic, Grigorios Magklis, Pedro Chaparro, and Antonio González. 2008. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 240–249.

[11] Jian Chen and Lizy K John. 2009. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference*. ACM, 927–930.

[12] Standard Performance Evaluation Corporation. 2006. SPEC 2006. (2006). http://www.spec.com/cpu2006/.

[13] Chris Fallin, Chris Wilkerson, and Onur Mutlu. 2014. The Heterogeneous Block Architecture. SAFARI Technical Report No. 2014-001 (March 2014).

[14] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[15] Peter Greenhalgh. 2011. Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7. (Sept. 2011). http://www.arm.com/files/downloads/big_LITTLE_Final.pdf.

[16] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, and others. 1993. The superblock: an effective technique for VLIW and superscalar compilation. *the Journal of Supercomputing* 7, 1-2 (1993), 229–248.

[17] Intel. 2008. 2nd generation Intel Core vPro processor family. (2008). http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf.

[18] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. 2006. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. of the 39th Annual International Symposium on Microarchitecture*. 347–358.

[19] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. *ACM SIGPLAN Notices* 47, 4 (2012), 223–234.

[20] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. 2005. Introduction to the Cell multiprocessor. *IBM journal of Research and Development* 49, 4.5 (2005), 589–604.

[21] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 166–177.

[22] David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European conference on Computer systems*. ACM, 125–138.

[23] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. 81–92.

[24] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st annual international symposium on Computer architecture*.

[25] Nagesh Lakshminarayana, Sushma Rao, and Hyesoon Kim. 2008. Asymmetry aware scheduling algorithms for asymmetric multiprocessors. In *In Proc. of the Fourth Annual Workshop on the Interaction between Operating Systems and Computer Architecture*. Citeseer.

[26] Nagesh B Lakshminarayana, Jaekyu Lee, and Hyesoon Kim. 2009. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 25.

[27] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 469–480.

[28] Tong Li, Dan Baumberger, David A Koufaty, and Scott Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 53.

[29] 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 of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 317–328.

[30] Daniel S McFarlin, Charles Tucker, and Craig Zilles. 2013. Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 241–252.

[31] R. Nair and M. Hopkins. 1997. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups. In *Proc. of the 24th Annual International Symposium on Computer Architecture*. 13–25.

[32] NVidia. 2011. Variable SMP - a multi-core CPU architecture for low power and high performance. (2011). http://www.nvidia.com/content/PDF/tegra_white_papers/ Variable-SMP-A-Multi-Core-CPU-Architecture-for-LowPower-and-High-Performance-v1.1.pdf.

[33] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2013. Trace based phase prediction for tightly-coupled heterogeneous cores. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 445–456.

[34] Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2015. DynaMOS: Dynamic Schedule Migration for Heterogeneous Cores. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM.

[35] Oscar Palomar, Toni Juan, and Juan J Navarro. 2009. Reusing cached schedules in an out-of-order processor with in-order issue logic. In *Computer Design, 2009. ICCD 2009. IEEE International Conference on Computer Design*. IEEE, 246–253.

[36] Sanjay J Patel, Tony Tung, Satarupa Bose, and Matthew M Crum. 2000. Increasing the size of atomic instruction blocks using control flow assertions. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 303–313.

[37] Vinicius Petrucci, Michael A Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. 2015. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 246–258.

[38] Qualcomm. 2011. Snapdragon S4 Processors: System on Chip Solutions for a New Mobile Age. (2011). https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age.

[39] Krishna K Rangan, Gu-Yeon Wei, and David Brooks. 2009. Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 302–313.

[40] Eric Rotenberg, Steve Bennett, and James E Smith. 1996. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 24–35.

[41] 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. *migration* 100, 1K (2013), 10K.

[42] Amirali Sharifian, Snehasish Kumar, Apala Guha, and Arrvindh Shriraman. 2016. Chainsaw: Von-neumann accelerators to leverage fused instruction chains. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–14.

[43] 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 heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 66–75.

[44] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 45–57.

[45] M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 253–264.

[46] Emil Talpes and Diana Marculescu. 2005. Execution cache-based microarchitecture for power-efficient superscalar processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 13, 1 (2005), 14–26.

[47] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 177–187.

[48] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE). In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA '12)*. 213–224.

[49] Carlos Villavieja, Jose A. Joao, Rustam Miftakhutdinov, and Yale N. Patt. 2014. Yoga: A Hybrid Dynamic VLIW/OoO Processor. HPS Technical Report No. 2014-001 (2014).

[50] Randy Widialaksono, Rangeen Basu Roy Chowdhury, Zhenqian Zhang, Joshua Schabel, Steve Lipa, Eric Rotenberg, W Rhett Davis, and Paul Franzon. 2016. Physical design of a 3D-stacked heterogeneous multi-core processor. In *3D Systems Integration Conference (3DIC), 2016 IEEE International*. IEEE, 1–5.

[51] Qiang Wu, Margaret Martonosi, Douglas W Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. 2005. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 271–282.

[52] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. *Network* 8000 (2013), 6000.