A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors

SPARSH MITTAL, Oak Ridge National Laboratory

To meet the needs of a diverse range of workloads, asymmetric multicore processors (AMPs) have been proposed, which feature cores of different microarchitecture or ISAs. However, given the diversity inherent in their design and application scenarios, several challenges need to be addressed to effectively architect AMPs and leverage their potential in optimizing both sequential and parallel performance. Several recent techniques address these challenges. In this article, we present a survey of architectural and system-level techniques proposed for designing and managing AMPs. By classifying the techniques on several key characteristics, we underscore their similarities and differences. We clarify the terminology used in this research field and identify challenges that are worthy of future investigation. We hope that more than just synthesizing the existing work on AMPs, the contribution of this survey will be to spark novel ideas for architecting future AMPs that can make a definite impact on the landscape of next-generation computing systems.

Categories and Subject Descriptors: A.2 [**General and Reference**]: Surveys and Overviews; C. [**Computer Systems Organization**]: Multicore Architectures; Heterogeneous (Hybrid) Systems

General Terms: Design, Performance

Additional Key Words and Phrases: Review, classification, asymmetric multicore processor, heterogeneous multicore architecture, big/little system, reconfigurable AMP

ACM Reference Format:
Sparsh Mittal. 2016. A survey of techniques for architecting and managing asymmetric multicore processors. ACM Comput. Surv. 48, 3, Article 45 (February 2016), 38 pages.
DOI: http://dx.doi.org/10.1145/2856125

1. INTRODUCTION

Modern computing systems have become more diverse than ever before. In terms of scale, they range from handheld systems to large datacenters, and in terms of utilization, they show interspersed periods of inactivity and peak resource demand. A wide range of workloads/tasks, such as multimedia, encryption, network, and cloud synchronization, run on these systems, each of which presents different resource demands. Different optimization objectives, such as per-thread performance, total throughput, energy, and area govern their design and operation. Diversity has thus become a rule rather than an exception.

In this era, even a highly optimized *monolithic* core cannot simultaneously meet the diverse and often conflicting requirements and goals. For example, an overprovisioned core will waste area and energy, whereas an underprovisioned core will fail to provide

Support for this work was provided by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research.
Author's address: S. Mittal, 1 Bethel Valley Road, Future Technologies Group, Oak Ridge National Laboratory, Building 5100, MS-6173, Tennessee, USA 37830; email: mittals@ornl.gov.

DOI: http://dx.doi.org/10.1145/2856125

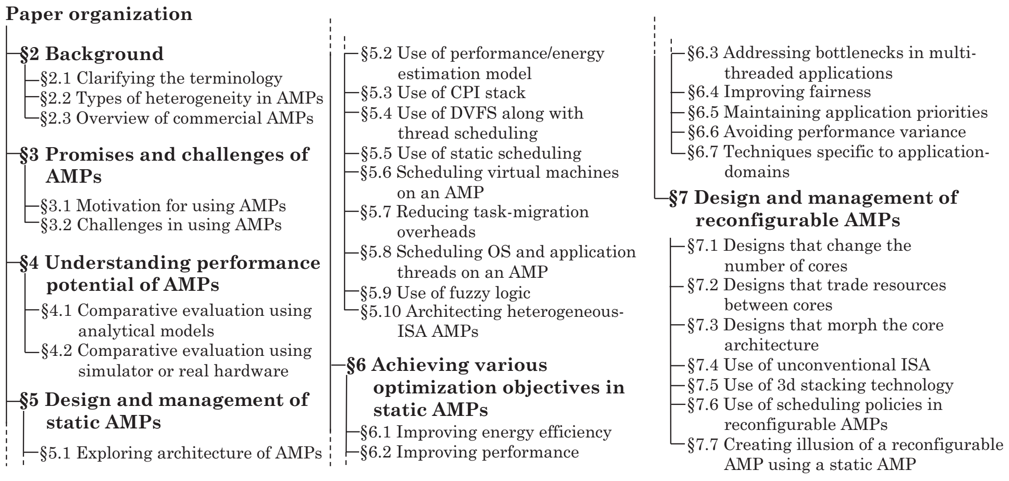
Paper organization
§2 Background
§2.1 Clarifying the terminology
§2.2 Types of heterogeneity in AMPs
§2.3 Overview of commercial AMPs
§3 Promises and challenges of AMPs
§3.1 Motivation for using AMPs
§3.2 Challenges in using AMPs
§4 Understanding performance potential of AMPs
§4.1 Comparative evaluation using analytical models
§4.2 Comparative evaluation using simulator or real hardware
§5 Design and management of static AMPs
§5.1 Exploring architecture of AMPs
§5.2 Use of performance/energy estimation model
§5.3 Use of CPI stack
§5.4 Use of DVFS along with thread scheduling
§5.5 Use of static scheduling
§5.6 Scheduling virtual machines on an AMP
§5.7 Reducing task-migration overheads
§5.8 Scheduling OS and application threads on an AMP
§5.9 Use of fuzzy logic
§5.10 Architecting heterogeneous-ISA AMPs
§6 Achieving various optimization objectives in static AMPs
§6.1 Improving energy efficiency
§6.2 Improving performance
§6.3 Addressing bottlenecks in multi-threaded applications
§6.4 Improving fairness
§6.5 Maintaining application priorities
§6.6 Avoiding performance variance
§6.7 Techniques specific to application-domains
§7 Design and management of reconfigurable AMPs
§7.1 Designs that change the number of cores
§7.2 Designs that trade resources between cores
§7.3 Designs that morph the core architecture
§7.4 Use of unconventional ISA
§7.5 Use of 3d stacking technology
§7.6 Use of scheduling policies in reconfigurable AMPs
§7.7 Creating illusion of a reconfigurable AMP using a static AMP

Fig. 1. Organization of the article in different sections.

high performance. Acknowledgment of this very fact by academia and industry is evident from two distinct categories of core designs in modern processors: processors such as Sandybridge and Power8 use a few wide out-of-order (OOO) cores, whereas processors such as Xeon Phi use many narrow in-order (InO) cores.

The next and natural step in this direction is an asymmetric multicore processor (AMP) design, which uses cores of different types in the same processor and thus embraces heterogeneity as a first principle instead of retrofitting for it. Different cores in an AMP may be optimized for power/performance or different application domains or for exploiting instruction-level parallelism (ILP), thread-level parallelism (TLP), or memory-level parallelism (MLP). Thus, by virtue of adapting to diversity, AMPs promise to be beneficial for a broad range of usage scenarios.

However, as the magnitude of heterogeneity in AMPs increases, either due to large number of core types or applications, diverse instruction set architectures (ISAs), or the range of parallelism they seek to exploit, the complexity of managing them escalates at an exponential rate. For this reason, conventional symmetric multicore processor (SMP) optimization techniques may not work well in an AMP, and hence novel techniques are required to architect AMPs. Recently, many techniques have been proposed to fulfill this need.

Contributions. This article surveys the techniques proposed to design and manage AMPs. Figure 1 shows the overall organization of this work. We first discuss the terminologies used in the research field to describe AMP-related concepts, with a view to clarify them and converge to use of standard terms (Sections 2.1 and 2.2 and Table I). To provide an application perspective, we discuss examples of commercial AMPs (Section 2.3 and Table II). We discuss the rationale behind the use of AMPs along with the challenges they present for system architects (Section 3).

To show the performance potential of AMPs, we discuss works that provide comparative evaluation of AMPs with SMPs, using analytical models, simulators, or real hardware (HW) (Section 4 and Table III). Section 4 also classifies the works based on the evaluation platform used by them. Further, we discuss techniques for architecting and managing static AMPs (Section 5 and Table IV) and then discuss techniques for optimizing various performance objectives in them (Section 6 and Table V). We then review techniques for designing and managing reconfigurable AMPs (Section 7 and Table VI). In these sections, we discuss the works in several categories, and although many of the works fall into multiple categories, we discuss them in a single category only. Section 8 concludes this article with a discussion of future perspectives. We hope that this survey will provide insights to researchers into opportunities and obstacles in use of AMPs and motivate them to propose even better techniques for making AMPs the first-class citizens of the computing world.

Scope. Since a large body of research work may fall within the purview of AMPs, we delineate the scope of the article as follows. We review techniques for multicore CPUs that have (micro)architecturally asymmetric CPU cores, either statically or due to dynamic reconfiguration (refer to Section 2 for more details). We do not include heterogeneous computing due to entirely different processors (e.g., CPU along with GPU/DSP/FPGA [Mittal and Vetter 2015]), cores merely having different circuit design (e.g., CMOS vs. Tunnel FET), or cases where heterogeneity between cores arises merely due to process variation. Although many papers reviewed in this survey do use dynamic voltage/frequency scaling (DVFS) to emulate an AMP, our focus in this work is not on per-core DVFS techniques but on processors where cores differ in crucial microarchitectural features, such as pipeline design, issue/fetch width, and cache hierarchy, and not merely in frequency/voltage.

2. A BACKGROUND ON AMP RESEARCH AND COMMERCIAL PROTOTYPES

Since previous works have used different terms to describe the same or similar concepts, in this section we first mention these terms to synthesize and clarify them. This discussion is expected to lead the researchers toward the use of standard terminology. We then classify types of heterogeneity in AMPs and discuss examples of commercial AMPs.

2.1. Clarifying the Terminology

Although asymmetric multicore processor is the term used most widely to describe processors with heterogeneous core types, several other terms have also been used, which are shown in Table I(a).[1] Additionally, note that most works reviewed in this survey discuss single-ISA AMPs, where all cores use the same ISA; such systems are specifically termed as *asymmetric single-ISA CMP* (ASISA-CMP) [Mogul et al. 2008]. Further, some papers discuss heterogeneous-ISA AMPs (refer to Section 5.10) and differentiate them from CPU-GPU heterogeneous systems; they are also termed as *OS-capable heterogeneous-ISA systems* [Barbalace et al. 2015], since unlike GPUs, CPUs can run full-fledged operating systems (OS).

Depending on the feature (e.g., performance, power, area, design complexity) that is emphasized, the asymmetric cores in an AMP can be termed differently, as we show in Table I(b). The processors that allow (or the techniques of allowing) reconfiguration of its microarchitecture to provide cores of different architectural features (e.g., issue width, instruction window) are referred to using various terms, which are shown in Table I(c). We refer to such processors as reconfigurable AMPs and other AMPs that do not provide such capabilities as static AMPs.

2.2. Types of Heterogeneity in AMPs

To get insight into the potential and scope of AMPs, it is important to understand and distinguish the types of heterogeneity that may be present in them. The classification suggested by Srinivasan et al. [2011] is shown in Figure 2. This survey reviews papers dealing with the last two categories, although as we show in Section 3.2.7, several papers use DVFS to emulate AMPs with physically asymmetric cores. Figure 2 also

[1]Note that the source included in Table I for a terminology is not necessarily related to its first use or origin.

Table I. Terminology Used in AMP Research

(a) *Different Terminologies for an AMP*
Asymmetric multicore (AMC) [Chen and Guo 2014], asymmetric multicore systems (ASYMS) [Sun et al. 2011], asymmetric multiprocessor systems (ASMPs) [Chitlur et al. 2012], asymmetric chip multiprocessors (ACMPs) [Suleman et al. 2007], heterogeneous microarchitectures (HMs) [Lukefahr et al. 2014], heterogeneous multicore processor (HMP) [Chen and John 2009], heterogeneous CMP (HCMP) [Navada et al. 2013], asymmetric cluster CMP (ACCMP) [Morad et al. 2006], big.LITTLE system [Jeff 2012]
(b) *Different Terminologies for Cores of an AMP*
Big/little (or big/small) [Jeff 2012], fast/slow [Becchi and Crowley 2006], complex/simple [Mogul et al. 2008], aggressive/lightweight [Ansari et al. 2013], strong/weak [Lin et al. 2014] cores, application/low-power processor (AP/LP) [Ra et al. 2012], central/peripheral processor [Lin et al. 2012]
(c) *Different Terminologies for Reconfigurable AMPs and/or Techniques for Architecting Them*
Reconfigurable [Pericas et al. 2007], configurable [Kumar et al. 2006], adaptive [Ansari et al. 2013], scalable [Gibson and Wood 2010], composable [Kim et al. 2007], composite [Lukefahr et al. 2012], coalition [Pricopi and Mitra 2012], conjoined [Kumar et al. 2004a], federated [Tarjan et al. 2008], polymorphous [Sankaralingam et al. 2003], morphable [Srinivasan et al. 2013], core morphing [Rodrigues et al. 2011], core fusion [Ipek et al. 2007], flexible [Pericas et al. 2007], dynamic [Hill and Marty 2008] and united [Chiu et al. 2010] processors

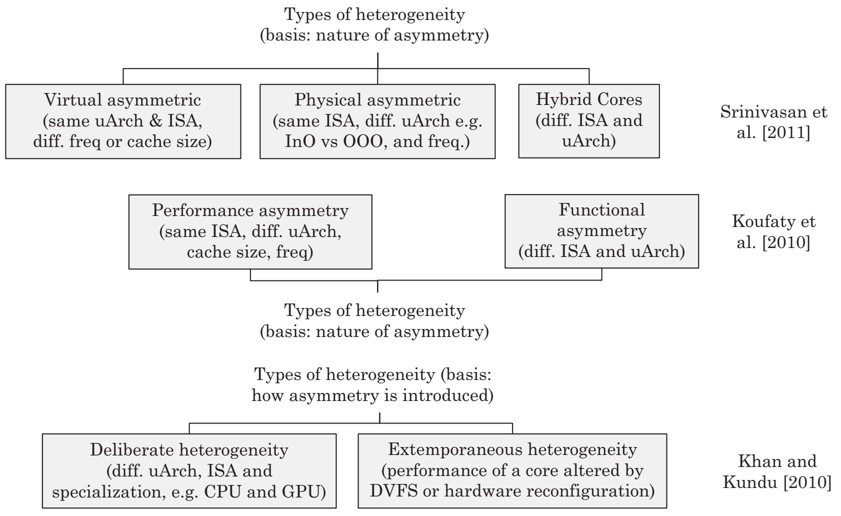
Types of heterogeneity
(basis: nature of asymmetry)
Virtual asymmetric (same uArch & ISA, diff. freq or cache size)
Physical asymmetric (same ISA, diff. uArch e.g. InO vs OOO, and freq.)
Hybrid Cores (diff. ISA and uArch)
Srinivasan et al. [2011]
Performance asymmetry (same ISA, diff. uArch, cache size, freq)
Functional asymmetry (diff. ISA and uArch)
Koufaty et al. [2010]
Types of heterogeneity
(basis: nature of asymmetry)
Types of heterogeneity (basis: how asymmetry is introduced)
Deliberate heterogeneity (diff. uArch, ISA and specialization, e.g. CPU and GPU)
Extemporaneous heterogeneity (performance of a core altered by DVFS or hardware reconfiguration)
Khan and Kundu [2010]

Fig. 2. Types of heterogeneity in AMPs (uArch, microarchitecture; freq., frequency; diff., different).

shows the classification suggested by Koufaty et al. [2010], where the first two categories mentioned by Srinivasan et al. are merged into the performance asymmetry category.

Figure 2 also shows another dimension of classification presented by Khan and Kundu [2010]. Taking extemporaneous heterogeneity to include reconfigurable AMPs, we review both types of architectures in this survey.

Another classification of AMPs can be done based on performance ordering between the cores. The cores in an AMP are called *monotonic* when they can be strictly ordered based on performance or complexity, and *nonmonotonic* when a strict ordering cannot be established, as different cores may be optimized for different instruction-level behavior and different application phases achieve highest performance on different cores. For example, an AMP with Alpha EV5 (21164) and EV6 (21264) is monotonic since EV6 has higher resources and provides better performance than EV5 for all applications [Kumar

Table II. Architectural Configuration of Four ARM Processors [ARM 2015a, 2015b; Pricopi et al. 2013]

	Cortex A15	Cortex A7	Cortex A57	Cortex A53
ISA	ARMv7 (32 bit)	ARMv7 (32 bit)	ARMv8 (32/64 bit)	ARMv8 (32/64 bit)
Frequency	1.7GHz	1.3GHz	1.9GHz	1.3GHz
Pipeline	Out-of-order	In-order	Out-of-order	In-order
Issue width	3	2	3	2
Fetch width	3	2	3	2
Pipeline stages	15 to 24	8 to 10	15 to 24	8 to 10
L1 I-cache	32KB/2-way/64B	32KB/2-way/32B	48KB/3-way/64B	32KB/2-way/64B
L1 D-cache	32KB/2-way/64B	32KB/4-way/64B	32KB/2-way/64B	32KB/4-way/64B
L2 cache (64B)	512KB-4MB/16-way	128KB-1MB/8-way	512KB-2MB/16-way	128KB-2MB/16-way
Perf. (MB/s)	99.69	77.93	155.29	109.36
Energy (mWh)	19.75	10.56	27.72	17.11
Perf./energy	5.04	7.38	5.60	6.39

Note: The last three rows show results on the XML parsing benchmark (BaseMark OS II Suite) [Frumusanu and Smith 2015] (Perf. = Performance).

et al. 2006]. However, an AMP with an Alpha core and an X86-64 core is nonmonotonic since neither of the two cores provides highest performance for *all* applications and phases [Venkat and Tullsen 2014].

2.3. An Overview of Commercial AMPs

To see the architecture of real-world AMPs, we now take a closer look at ARM processors that can be paired in big.LITTLE architecture. Table II shows architectural parameters of Cortex A15 (big) and A7 (little), which have same ISA but significantly different architecture and hence performance/power profiles. The same is also true for Cortex A57 (big) and A53 (little), and their architecture is also shown in Table II. All of these processors can have one to four cores per cluster.

In the same vein, the Qualcomm Snapdragon 810 has a quad-core Cortex A57 and a quad-core A53, whereas Snapdragon 808 has a quad-core A57 and a dual-core A53 [Whitwam 2014]. Similarly, Samsung's Exynos 5 Octa has a quad-core Cortex-A15 along with a quad-core Cortex-A7 [Samsung 2013], and Nvidia's Tegra X1 has a quad-core Cortex-A57 and a quad-core Cortex-A53 [Gil 2015]. Many other AMP products are also available [CNXSoft 2014; Ineda 2015; NVIDIA 2011], and this clearly shows the level of penetration of AMPs into commodity products.

3. OPPORTUNITIES AND OBSTACLES IN USING AMPS

To motivate this work, we now discuss both promises and challenges of AMPs.

3.1. Motivation for and Benefits from Using AMPs

3.1.1. Diverse Applications and Usage Patterns. As discussed previously, there exists large diversity between and within the workloads running on modern processors [Srinivasan et al. 2013; Zhong et al. 2007]. In addition, the objective function may change over time (e.g., due to operation conditions (plugged vs. unplugged system, thermal emergencies), priority inversion, or addition of new jobs in task queue). For example, in mobile systems, it is as important to optimize energy for prolonging battery life during idle periods as it is to optimize performance for multimedia applications and data processing during active use [Mittal 2014a; Zhu and Reddi 2013]. Similarly, although data centers show low average utilization, there exist occasional, brief bursts of activity that demand provisioning of a high amount of resources to meet service-level agreements [Mittal 2014b]. AMPs are natural choice for such systems and use cases.

3.1.2. Performance and Energy Factors. Although big and small cores generally provide better performance and energy efficiency, respectively, in several scenarios no single winner may be found on the metric of energy delay product (EDP) [Pricopi et al. 2013; Sawalha and Barnes 2012]. The big core may show better EDP in applications that are compute intensive and have predictable branching and high data reuse. By contrast, the small core may be better for memory-intensive applications and applications with many atomic operations and little data reuse, because for such applications, a negligible performance benefit provided by the big core may not justify its energy overhead. Similarly, cores optimized for certain operations can be highly useful for applications that use these operations intensely. Since energy efficiency has now become the primary constraint in scaling performance of all classes of computing systems [Vetter and Mittal 2015], an AMP design has become not only attractive but even imperative.

3.1.3. Benefits of Reconfigurable AMPs. Reconfigurable AMPs facilitate flexibly scaling up to exploit MLP and ILP in single-threaded applications and can scale down to exploit TLP in multithreaded (MT) applications. Thus, they may obviate the need of overprovisioning of cores, provide better HW utilization and resilience to errors, as a hard error in its HW may not disable the entire processor. They may also achieve better performance [Hill and Marty 2008] and energy proportionality [Watanabe et al. 2010] than static AMPs.

3.2. Challenges in Using AMPs

Several challenges must be addressed to fully realize the potential of AMPs.

3.2.1. Changes Required across the System Stack. The heterogeneous nature of AMPs demands complete re-engineering of the whole system stack. The cores of an AMP may have different supply voltages and frequencies, which presents manufacturing challenges. Further, the cores in an AMP should cover a wide and evenly spread range of performance/complexity design space; however, the complexity of scheduling on an AMP increases exponentially with an increasing number of core types and applications [Liu et al. 2013]. Further, since AMP HW is still evolving, continuing to change OS to account for them can lead to a fragile software (SW) system, and new policies may interfere with existing policies in the system, such as cache or main memory management policies [Gutierrez et al. 2014]. The techniques that use nonstandard ISAs or compiler support (refer to Table VI) demand a huge research effort and yet may not find wide adoption. Chitlur et al. [2012] discuss several other issues that system SW needs to address before running on an AMP.

3.2.2. Heterogeneity Complexity Trade-Off. Although single-ISA AMPs allow easier implementation, they may not fully exploit the potential of heterogeneity. This has motivated research on AMPs, where not only the HW but even the ISA, OS, and programming model of different cores may be different and may not share a HW-coherent memory [Lin et al. 2012, 2014]. Such systems present even more challenges (refer to Section 5.10), and hence they have not been explored to their full potential.

3.2.3. Thread Scheduling Challenges. Some techniques use offline analysis to perform static scheduling; however, they cannot account for different input sets and application phases and may become infeasible with increasing number of co-running applications. Other techniques collect runtime data to perform dynamic scheduling; however, they incur thread migration overhead and may be ineffective for short-lived threads, as the profiling phase itself may form a large majority of their lifetime.

To make scheduling decisions, the performance of a thread on different core types must be known. Some techniques *estimate* a thread's performance on a core type without actually running it on that core type (e.g., Koufaty et al. [2010], Lukefahr et al.

[2012], Nishtala et al. [2013], Pricopi et al. [2013], Saez et al. [2012, 2015], Srinivasan et al. [2011], and Van Craeynest et al. [2012, 2013]), whereas other techniques *actually run* threads on available core types to sample their performance on each core type (e.g., Becchi and Crowley [2006], Kumar et al. [2003], Lukefahr et al. [2014], Patsilaras et al. [2012], Sawalha and Barnes [2012], Shelepov et al. [2009], Sondag and Rajan [2009], Van Craeynest et al. [2013], and Zhang et al. [2014a]). The first approach is HW specific and error prone, whereas the second approach presents a scalability bottleneck due to its high profiling overhead.

3.2.4. Thread Migration Overheads. In static AMPs, thread migration may take millions of cycles [Becchi and Crowley 2006]. For example, in a big.LITTLE system with Cortex A15 and A7 processors, Pricopi et al. [2013] observe the latency of moving the task from the A15 to A7 and vice versa to be 3.75ms and 2.10ms, respectively. Flushing and warming of cache and other state variables present additional overheads, which restrict the migration to be performed on the order of thousands to millions of instructions, and preclude exploiting fine-grain scheduling opportunities.

3.2.5. Challenge to Maintain Fairness. With asymmetry-unaware thread schedulers, threads running on an AMP may be unfairly slowed down, which leads to starvation and unpredictable per-task performance. In fact, in a barrier-synchronized MT application, the performance advantage of big core may be completely negated if the thread running on it stalls waiting for other threads. Thus, ensuring fairness is important for meeting quality of service (QoS) guarantees and user-specified priorities.

3.2.6. Challenges in Reconfigurable AMPs. Although reconfigurable AMPs partially offset the thread migration overheads, they are not panacea either. Reconfiguration incurs latency and energy overheads [Ponomarev et al. 2001], such as I/D-cache flushes and data migration. Mitigating these challenges may require the use of a complex compiler [Zhong et al. 2007], custom ISA (refer to Table VI), or 3D stacking [Homayoun et al. 2012], along with changes to the OS and application binary. Reconfigurable AMPs present several trade-offs—for example, the use of centralized resources saves area but also presents a scalability bottleneck and vice versa. Similarly, providing high adaptation granularity to exploit different levels of ILP and TLP precludes specialization for accelerating specific applications.

Reconfigurable AMPs may also present practical challenges. For example, Salverda and Zilles [2008] show that a HW-only solution for fusion of *InO cores* is impractical. They show that to match the performance of designs that use dynamic scheduling, a reconfigurable architecture needs to handle many active dataflow chains simultaneously. When such an architecture uses InO cores, then either interleaving of active dataflow chains needs to be performed between the issue queues or a very large number of cores needs to be fused. The former requires dedicated HW for steering instructions, and the latter would incur very high overhead of fusion. They suggest that to avoid this, some OOO capability needs to be provided to the constituent cores, or alternative approaches (e.g., a compiler [Gupta et al. 2010]) need to be utilized.

3.2.7. Modeling Challenges. Since AMPs are not yet widely available, several works use DVFS (or clock throttling) to emulate an AMP (Table III). However, the use of DVFS as a proxy to emulate asymmetric cores oversimplifies the challenges of a real AMP and hence is likely to provide inaccurate conclusions [Koufaty et al. 2010]. Additionally, the use of DVFS does not allow realistic modeling of an AMP with nonmonotonic cores.

For their experiments, Koufaty et al. [2010] designed 40 microbenchmarks that test core microarchitecture, such as execution bandwidth, private cache latency, and branch misprediction penalty, but do not use any shared resource, such as last-level cache (LLC) or memory. As for big core, they chose a 2.8GHz OOO core that can retire up to four micro-ops (μOP) in each cycle. For small cores, they considered three options: (1) running the big core at 1.6GHz (i.e., the use of DVFS), (2) a 2.8GHz InO core, and (3) the same 2.8GHz big core, retiring up to 1 μOP/cycle. They show that with DVFS, the frequency ratio alone determines the core performance. Hence, for different microbenchmarks with different ILP, the relative performance of a big and small core (number 1) does not change. By comparison, the InO core (number 2) shows larger performance loss than the frequency-scaled core, and its performance varies with varying levels of ILP. The small core number 3 shows performance profile close to that of InO core and shows varying performance for different levels of ILP. Clearly, a frequency-scaled core does not realistically model a core with microarchitectural heterogeneity.

The remaining sections review several techniques to address these challenges.

4. UNDERSTANDING PERFORMANCE POTENTIAL OF AMPS

In this section, we first classify the works based on their evaluation platforms and then discuss several works that compare AMPs with SMPs to show that AMPs indeed provide performance advantage, which justifies their use.

To realistically evaluate AMPs, the evaluation platform should model cores of varying microarchitecture and OS-governed scheduling policies and should be fast to allow study of long execution periods. Due to the emerging nature of AMPs and complexity involved in their design, real HW platforms may not be widely available. In addition, they do not allow easy configurability and flexibility of experimentation as provided by the architectural simulators. By comparison, simulators may not accurately model real-world prototypes, may only provide user-space simulation capabilities, and may be too slow to allow full exploration of the design space. These factors make the choice of an evaluation platform crucially important, and hence Table III classifies the techniques based on whether they have been evaluated using a simulator or real HW.

Table III also shows those works that use theoretical models to gain insights and find limit gains from AMPs, independent of a particular HW or application. We now discuss several of these works.

4.1. Comparative Evaluation using Analytical Models

Hill and Marty [2008] extend Amdahl's law for modeling SMP, static AMP, and reconfigurable AMP by adding a simple HW model and assuming fixed chip resources. For each of these three processors, they develop a formula for computing speedup relative to using one single base core, as a function of an application fraction that is parallelizable, total chip resources (in terms of base cores), and core resources devoted to increase the performance of each base core. For simplification, they ignore scheduling/reconfiguration overheads and other processor components, such as cache. They show that AMPs increase sequential performance, and their maximum speedups are much greater than those provided by SMPs and are never worse. Further, assuming that all cores of reconfigurable AMPs can be fused or split as desired to accelerate sequential or parallel applications, respectively, reconfigurable AMPs provide larger speedup than static AMPs. Thus, their model points toward aggressively attacking serial bottlenecks and making globally optimized design choices even if they may be locally inefficient.

Juurlink and Meenderinck [2012] note that Amdahl's law assumes that when going from 1 to p cores, the parallelizable work remains constant (i.e., independent of p), whereas Gustafson's law assumes that parallelizable work grows linearly with p, and hence ignoring the different assumptions behind both of these laws can lead to incorrect predictions. Juurlink and Meenderinck generalize these laws to present a law that assumes parallelizable work grows as a sublinear function of p. Using this law, they

Table III. A Classification Based on Evaluation Platform

Category	References
Evaluation Platform/Approach	
Analytical modeling	[Eyerman and Eeckhout 2010; Grochowski et al. 2004; Gupta and Nathuji 2010; Hill and Marty 2008; Juurlink and Meenderinck 2012; Morad et al. 2006; Van Craeynest and Eeckhout 2013]
Real HW	[Annavaram et al. 2005; Barbalace et al. 2015; Cao et al. 2012; Chen and Guo 2014; Cong and Yuan 2012; Georgakoudis et al. 2013; Grant and Afsahi 2006; Kazempour et al. 2010; Kim et al. 2014, 2015; Ko et al. 2012; Koufaty et al. 2010; Kwon et al. 2011; Lakshminarayana and Kim 2008; Lakshminarayana et al. 2009; Li et al. 2007, 2010; Lin et al. 2012, 2014; Morad et al. 2006, 2010; Mühlbauer et al. 2014; Muthukaruppan et al. 2013, 2014; Nishtala et al. 2013; Panneerselvam and Swift 2012; Petrucci et al. 2012; Pricopi et al. 2013; Ra et al. 2012; Rafique et al. 2009; Saez et al. 2010, 2012, 2015; Shelepov et al. 2009; Sondag and Rajan 2009; Srinivasan et al. 2011; Sun et al. 2011; Takouna et al. 2011; Zhang et al. 2014b; Zhu and Reddi 2013]
Simulator	[Annamalai et al. 2013; Annavaram et al. 2005; Ansari et al. 2013; Becchi and Crowley 2006; Brown et al. 2011; Chen and John 2008, 2009; Chiu et al. 2010; Eyerman and Eeckhout 2014; Gibson and Wood 2010; Gulati et al. 2008; Gupta et al. 2010; Gutierrez et al. 2014; Homayoun et al. 2012; Ipek et al. 2007; Joao et al. 2013; Khan and Kundu 2010; Kim et al. 2007; Kumar et al. 2003, 2004a, 2004b, 2006; Lakshminarayana et al. 2009; Liu et al. 2013; Lukefahr et al. 2012, 2014; Luo et al. 2010; Lustig et al. 2015; Madruga et al. 2010; Markovic et al. 2014; Mogul et al. 2008; Mukundan et al. 2012; Najaf-Abadi et al. 2009; Najaf-Abadi and Rotenberg 2009; Navada et al. 2013; Padmanabha et al. 2013; Patsilaras et al. 2012; Pericas et al. 2007; Pricopi and Mitra 2012, 2014; Ra et al. 2012; Rodrigues et al. 2011, 2014; Salverda and Zilles 2008; Srinivasan et al. 2013, 2015; Strong et al. 2009; Suleman et al. 2007, 2009, 2010; Van Craeynest et al. 2012, 2013; Van Craeynest and Eeckhout 2013; Venkat and Tullsen 2014; Watanabe et al. 2010; Wu et al. 2011; Zhang et al. 2014a; Zhong et al. 2007]
Other Features	
Comparison of AMP and SMP	[Annavaram et al. 2005; Balakrishnan et al. 2005; Eyerman and Eeckhout 2010, 2014; Grant and Afsahi 2006; Gulati et al. 2008; Gupta and Nathuji 2010; Hill and Marty 2008; Juurlink and Meenderinck 2012; Kumar et al. 2004b; Lakshminarayana and Kim 2008; Li et al. 2010; Morad et al. 2006, 2010; Muthukaruppan et al. 2013; Najaf-Abadi and Rotenberg 2009; Patsilaras et al. 2012; Pricopi and Mitra 2014; Strong et al. 2009; Suleman et al. 2007, 2009; Sun et al. 2011; Van Craeynest and Eeckhout 2013]
Use of DVFS for emulating an AMP	[Annavaram et al. 2005; Balakrishnan et al. 2005; Cao et al. 2012; Grant and Afsahi 2006; Hruby et al. 2013; Kazempour et al. 2010; Ko et al. 2012; Kwon et al. 2011; Lakshminarayana and Kim 2008; Lakshminarayana et al. 2009; Li et al. 2007, 2010; Morad et al. 2010; Nishtala et al. 2013; Panneerselvam and Swift 2012; Shelepov et al. 2009; Sondag and Rajan 2009; Sun et al. 2011; Takouna et al. 2011]

predict multicore design trends and find fundamentally different predictions compared to Amdahl's law. For SMPs, Amdahl's law suggests the use of fewer but larger cores for highest performance, but the general law shows that fewer larger cores provide much smaller performance improvement, and only for smaller parallelizable fraction, compared to the predictions of Amdahl's law. Further, AMPs are found to provide better performance than SMPs, but their performance benefits are smaller, they take place for a smaller parallelizable fraction, and the big core has a smaller optimized size than that predicted by Amdahl's law. Similarly, the performance benefits of reconfigurable AMPs are found to be smaller than the predictions of Amdahl's law.

Amdahl's law assumes either completely sequential code or completely parallel code. Eyerman and Eeckhout [2010] extend this law for the applications in which part of parallel code is executed within critical sections (CSs). By developing an analytical model to find the execution time of a CS, they show that in addition to sequential code, synchronization through CSs also limit the parallel performance. Hence, designing one big core and multiple small cores for accelerating sequential and parallel codes, respectively, as per Amdahl's law [Hill and Marty 2008], harms performance, because in such a design, the sequential part due to CSs also runs on small cores. Thus, for a fixed HW budget, the best AMP design suggested by Amdahl's law is found to be inferior to an SMP design, because for the same HW budget, the small cores in an SMP will be relatively larger, faster, and hence more effective in accelerating CSs. For applications with higher probability of CSs and contention, the impact of sequential parts of CSs on overall performance increases further, and hence ignoring CSs leads to AMP designs that are inferior to SMPs. Given this, they suggest that in an AMP, the small cores should not be so tiny, but relatively larger (and may be fewer in number for a fixed area) so that by accelerating CSs, overall performance can be improved. Further, migrating CSs on big cores can yield substantial speedups.

Van Craeynest and Eeckhout [2013] compare AMP to SMP on two performance metrics: per-application performance and overall system throughput. They find the frontier of Pareto-optimal architectures that achieve optimum trade-off between the two metrics. They show that trading multiple simple InO cores for a few/single aggressive OOO core improves per-application performance but degrades system throughput and vice versa. Thus, different SMPs and AMPs exercise different trade-offs between the two metrics. Further, by virtue of mapping jobs to most suitable cores, some AMP configurations outperform specific SMP configurations on both metrics. However, some SMP configurations do yield optimal performance trade-offs, and thus AMP architectures do not necessarily outperform SMP architectures—in fact, some trade-offs are best achieved through SMP architectures. Further, two core types are found to be sufficient for providing most of the benefits from heterogeneity (also confirmed by Kumar et al. [2003]), as additional core types do not improve the performance significantly.

4.2. Comparative Evaluation Using Simulator or Real HW

Annavaram et al. [2005] present a technique to improve both sequential and parallel performance by adapting the energy consumed in processing each instruction based on the degree of parallelism available. Using the equation Power = EPI × IPS, (EPI, energy per retired instruction; IPS, total retired instructions per second on all cores), for a fixed power budget, a processor should spend larger EPI in phases of limited parallelism (low IPS) and vice versa [Grochowski et al. 2004]. Thus, to maintain a power budget constraint, for phases with large parallelism, less energy should be consumed in processing each instruction and vice versa. Based on this, they map high-IPS parallel phases to cores with low EPI, and low-IPS sequential phases to cores with high EPI. Compared to an SMP, an AMP provides a more suitable platform with cores of different EPI, and hence under the same power budget, using their technique on an AMP provides significant speedup compared to that on an SMP.

For an arbitrary application with both serial and parallel phases, Morad et al. [2006] use theoretical models for finding the upper bound of power efficiency (performance per watt) of an SMP. They compare this upper bound to an AMP's power efficiency, assuming that the big core executes the serial phase and all cores execute the parallel phase. From this, they show that the AMP achieves higher power efficiency than any SMP, which happens because AMPs optimize serial and parallel phases separately, whereas SMPs optimize both phases together. They further validate their findings by experimental evaluation over real HW.

Lakshminarayana and Kim [2008] study the effect of thread interactions on the performance/energy profile in AMPs and SMPs. They show that when threads have an equal amount of work and negligible interaction, an SMP with fast cores (configured to highest clock frequency) consumes a smaller amount of energy. With increasing thread interaction, an SMP with slow cores or an AMP consumes less energy. This is because when the processor frequently stalls waiting to acquire locks, slow cores may still provide reasonable performance. Further, an AMP optimizes energy when the work on different threads is different, as the threads performing longest task can be mapped to fast cores to achieve load balancing. Thus, depending on the characteristics of MT applications, an AMP or an SMP should be chosen for saving energy.

5. DESIGN AND MANAGEMENT OF STATIC AMPS

In this section, we reviews works on architecting and operating static AMPs. Table IV classifies these research works based on their key design features and management strategies. We now review several of these works.

5.1. Exploring Architecture of AMPs

Given the large configuration space of core types, finding the optimal number and types of cores is crucially important. Several works propose methods to explore this design space, as we discuss next.

Kumar et al. [2006] explore the architecture of cores for an AMP to achieve highest energy or area efficiency under varying TLP, area budget, and power budget scenarios. They discuss methodologies for arriving at scalable AMP designs and show that optimizing each core for a class of applications leads to the most efficient AMP. Additionally, for different applications, the cores exhibit different performance ordering. Hence, an AMP with monotonic cores (refer to Section 2.2) may not be optimal for a large range of workloads, and relaxing the monotonicity constraint leads to more efficient AMPs.

Patsilaras et al. [2012] present an AMP architecture consisting of both MLP and ILP customized cores, along with HW-level scheduling policies to make best use of it. Based on the L2 miss rate, their technique detects MLP phases; in phases of high MLP, an application is migrated to a core customized for MLP. In phases where ILP dominates, the application is migrated to a core that provides the largest IPS, and this core may change over the execution of application. Unused core is turned off for saving energy. They show that their technique improves both performance and energy efficiency.

Navada et al. [2013] study scheduling in AMPs with nonmonotonic core types. Their technique selects nonmonotonic core types at design time and schedules program phases to cores at runtime. They show that with N core types, the single-thread performance is optimized on an AMP with 1 "average core" type and $N - 1$ "accelerator core" types that mitigate resource bottlenecks in the average core. For example, for $N = 4$, the three accelerator core types could be one with larger window size, another with higher issue width, and a third with higher frequency, and they offset instruction window, issue width, and frequency bottlenecks, respectively. Thus, their technique provides average resource provisioning while avoiding distinct resource bottlenecks. They also propose a scheduling policy, which continuously monitors a running program and on observing a bottleneck on the current core migrates the program to a core that can relieve the bottleneck; in the absence of such a core, the program is migrated to an average core.

5.2. Use of the Performance/Energy Estimation Model

Effective scheduling on AMP requires estimating the performance/energy of threads on different core types, and several models have been proposed for this. In this section and in Section 5.3, we discuss these models.

Sondag and Rajan [2009] present a technique to improve performance of AMPs. Their technique performs offline analysis for detecting similarity between basic blocks and then detects phase transition boundaries using intraprocedural analysis. This information is instrumented in the binary. Using information provided by offline analysis,

Table IV. A Classification Based on AMP Architecture and Management Approach Used in Different Works

Category	References
Selecting core types in AMP	[Chitlur et al. 2012; Eyerman and Eeckhout 2010; Kumar et al. 2006; Najaf-Abadi and Rotenberg 2009; Navada et al. 2013; Patsilaras et al. 2012; Srinivasan et al. 2015; Van Craeynest and Eeckhout 2013]
DVFS with thread scheduling	[Annamalai et al. 2013; Grochowski et al. 2004; Kim et al. 2014; Lukefahr et al. 2014; Muthukaruppan et al. 2013, 2014; Zhu and Reddi 2013]
DVFS versus thread scheduling	[Grochowski et al. 2004; Kumar et al. 2003; Lukefahr et al. 2014]
Use of static scheduling	[Chen and John 2008, 2009; Chen and Guo 2014; Hruby et al. 2013; Kumar et al. 2003; Lakshminarayana and Kim 2008; Lukefahr et al. 2014; Ra et al. 2012; Shelepov et al. 2009]
Use of CPI stack	[Koufaty et al. 2010; Pricopi et al. 2013; Srinivasan et al. 2011; Van Craeynest et al. 2012]
Use of regression model	[Cong and Yuan 2012; Khan and Kundu 2010; Mühlbauer et al. 2014; Saez et al. 2012; Zhang et al. 2014a]
Use of fuzzy logic	[Chen and John 2008; Sun et al. 2011]
Scheduling VMs on AMP	[Cao et al. 2012; Kwon et al. 2011; Sun et al. 2011; Takouna et al. 2011; Wu et al. 2011]
Scheduling OS thread on a core	[Grant and Afsahi 2006; Hruby et al. 2013; Mogul et al. 2008]
Disjoint or overlapping (shared) ISA in different cores	[Barbalace et al. 2015; DeVuyst et al. 2012; Georgakoudis et al. 2013; Hruby et al. 2013; Li et al. 2010; Lin et al. 2012, 2014; Lustig et al. 2015; Srinivasan et al. 2011; Venkat and Tullsen 2014]
Binary translation	[DeVuyst et al. 2012; Georgakoudis et al. 2013; Wu et al. 2011]

the code sections showing similar runtime are grouped into clusters. Their technique chooses representative code sections from each cluster and evaluates their performance on different core types. Based on this, code sections in a phase are matched to a suitable core. When a satisfactory mapping for a phase type is determined, the same core scheduling decision is taken on future occurrences of that phase type, which reduces the runtime overhead. They show that their technique provides better performance than the Linux scheduler while maintaining fairness.

Khan and Kundu [2010] present a thread scheduling technique to improve performance and energy efficiency of AMPs. They detect the application phases using an empirical model based on basic block vector and instruction type vector. They also record other counters, such as committed instructions per cycle (IPC), speculative IPC, cycle count, and power. By using these as an input to an offline linear regression model, the application's power and performance profile on the other core is estimated. The dynamic scheduler uses these estimates, and on detecting a phase change, it finds and applies the optimal thread-to-core mapping that optimizes a desired objective, such as performance, power, or performance per power. They show that their empirical predictive scheduler performs very close to an ideal scheduler while incurring low overhead.

Cong and Yuan [2012] develop a regression model for estimating the energy consumption based on important parameters, such as execution cycles, retired instructions, and accesses to L1 D-cache and L2 cache. Their technique first uses static analysis to find the boundaries of the function calls and loops. Then the program is profiled to construct the call graph that captures the caller-callee relationship between function calls and loops. Then the major program phases are identified, and the selected functions calls and loops are instrumented. Afterward, using the regression model, the energy consumption and EDP are estimated. Based on this, a core is chosen for the execution of a function or loop that leads to the smallest EDP value.

Pricopi et al. [2013] present a SW-based modeling technique for AMPs. While an application is running on one core (either big or little), profiling information is collected, which is used to estimate power and performance characteristics of the application on another core (little or big, respectively). Since the microarchitecture of both cores are different, they use compiler-based application analysis, empirical modeling, and mechanistic modeling and thus build an analytical model from regression modeling and understanding of the underlying architecture. Their performance model builds on a cycles per instruction (CPI) stack that quantifies the impact of different microarchitectural events on execution time. While the information about events such as cache miss and branch misprediction is obtained from HW counters, the information about data dependency and its impact on pipeline stall is collected using compiler analysis. Using this, the CPI stack–based performance model is developed for both cores. By observing the microarchitectural events on the first core, these events on the second core are estimated using regression modeling. Using these and the CPI stack of the first core, the CPI stack and hence the performance estimate for the second core is obtained. Further, using these along with additional factors, such as memory behavior and instruction mix, the power model of the core is also obtained.

5.3. Use of the CPI Stack

Koufaty et al. [2010] present a dynamic scheduling technique called *bias scheduling*, which works by classifying a thread as big or small core *bias* when the speedup ratio of the running thread on a big versus small core is high or low, respectively. Their technique monitors the CPI stack and broadly divides the reasons of core stall into two, namely internal and external, stalls. Internal stalls happen due to the lack of a core's internal resources (an execution unit or a load port), contention on them such as a private cache, or TLB miss or branch mispredictions. The external stalls happen due to resources that are outside the core, such as I/O, memory, and shared LLC. Their technique estimates a thread's performance on each core type by analyzing the CPI stack and causes of internal and external stalls. They note that when the execution cycles dominate the CPI of a thread, the thread is a big core bias, and when the internal or external stall dominates the CPI stack, the thread is a small core bias. This happens because when external stalls dominate the CPI stack, the big core is expected to not do too much better in hiding this latency, and hence it would not provide large speedup. They implement a bias scheduling technique on top of the Linux scheduler and show that their technique performs load balancing and improves performance of the system.

Van Craeynest et al. [2012] present a model called *performance impact estimation* (PIE) to predict the best application-to-core mapping. They note that the ratios of MLP and ILP on big and small cores provide a good indication of the performance difference between those cores. Based on this, while an application is running on one core type, their technique collects ILP and MLP profiling information and the CPI stack, and uses this to estimate the ILP and MLP and hence application performance on the other core type. Overall, PIE tracks how a core exploits ILP and MLP, and using the CPI stack, it finds its overall performance impact. They make some simplifying assumptions such as presence of identical cache hierarchy on both core types. In addition, some HW counters (e.g., for recording interinstruction dependency distance distribution) required by this model may not be available in existing processors. They show that by making application-to-core scheduling decisions based on the PIE model, significant performance improvement can be achieved.

5.4. Use of DVFS along with Thread Scheduling

Using DVFS along with thread scheduling provides further opportunities to exercise the performance/energy trade-off, as shown by the following techniques.

Annamalai et al. [2013] present a technique for improving throughput/watt of an AMP by using both thread scheduling and DVFS. When an application runs on a core, their technique estimates the expected throughput/watt of the current application phase at different voltage/frequency levels on all available core types. By examining these values, the best thread-to-core mapping and core-operating conditions can be selected. Performance and power estimation is done using HW performance counters such as the number of fetched and retired instructions, cache hits/misses, branch misprediction, and IPC. They use a mechanism to detect phase changes and keep the thread migration overhead small; these decisions are made only when the application phase changes. They show that their technique provides higher improvement than static, DVFS-only, and thread swapping-only scheduling schemes.

Muthukaruppan et al. [2013] propose a hierarchical power management approach for AMPs. Based on a control-theoretic approach, they utilize proportional-integral-derivative (PID) controllers of three types: the per-task resource share (RS) controller, per-task QoS controller, and per-cluster DVFS controller. The RS controller adjusts the CPU share of a task to meet its performance target. The QoS controller becomes active only when the thermal design power (TDP) constraint is violated, and in such a case, it throttles task performance to reduce the system load. On completion of a thermal emergency, it restores the performance to a user-defined level. For both big and small clusters, there are cluster controllers that apply DVFS to achieve utilization close to the target and achieve load balancing between cores. Migration of tasks between clusters is performed at much longer intervals than load balancing, as it incurs significantly larger overhead. In addition, when the TDP constraint is violated, a chip-level power allocator throttles the clusters' frequencies and signals the QoS controller to reduce the performance of tasks. Different controllers are invoked at different time granularities, and the output of one controller forms the input for another; thus, different controllers work in an interdependent manner. They show that their framework provides guarantees for safety, efficiency, and stability and also meets the QoS targets and TDP constraints.

Muthukaruppan et al. [2014] present a power management technique for AMPs that uses price theory from economics. The chip is organized into clusters of isofrequency cores. In terms of the price-theory model, the "virtual market" controls DVFS and so forth, and the processing unit (PU) of cores is the "commodity" that is traded using "virtual money." The PU on a core quantifies its computational resource, and one PU on a small core has a lower value than that on a big core. There are "agents" for chip, cluster, core, and task. Each agent aims to meet its goal (e.g., the chip agent ensures that the TDP constraint is not violated). Power management goals embody the "regulations" on the market and ensure its efficient functioning. Their framework has two parts: the supply-demand module (SDM) and load-balancing plus task-migration (LBT) module. The SDM aims to satisfy demands of all tasks while consuming minimal power, which is analogous to avoiding "inflation/deflation" to provide enough supply to fulfill the current market demand. The SDM elicits synergistic action from all agents and adjusts DVFS to avoid under/oversupply and reach to a stable price of the PU. The LBT module uses the concept of reduced spending, whereby it first seeks to fulfill the task demand and then performs load balancing within cluster and task migration across clusters to improve power efficiency. Load balancing reduces the voltage/frequency in a cluster, and task-migration exploits performance asymmetry of different clusters. To regulate the task migration overhead, only the constrained core from any cluster performs migration to the most oversupplied core in the target cluster. The authors show that their framework is scalable and minimizes energy while meeting task demands and TDP constraint.

Lukefahr et al. [2014] present a technique for improving energy efficiency using DVFS, AMP, or both. To find Pareto-optimal core-frequency choice for each epoch (of size 1K instructions) of the application for all possible performance constraints, they execute the application under all possible modes and collect performance/energy statistics for each epoch. To reduce the intractably large search space, they use approximation approaches for finding the subset of schedules that provide optimum performance-energy trade-off. They group multiple epochs into one segment, perform exhaustive search within each segment, and replace many schedules having similar energy and performance with two approximated schedules, namely worst case and best case, which bound the actual Pareto frontier. On detection of nonoptimal subschedules, enumeration of later schedules is avoided, which prunes provably nonoptimal schedules without evaluation. Thus, they explore the design space to find which architecture provides largest energy saving for a given performance level. They show that on using coarse-grain scheduling (10M instructions), the combination of DVFS and AMP proves advantageous, but on using fine-grain scheduling (1K instructions), the use of AMP alone provides large energy saving such that the use of DVFS becomes unnecessary.

5.5. Use of Static Scheduling

Shelepov et al. [2009] note that the architectural signature of an application summarizes its properties, such as its memory boundedness, sensitivity to variation in clock frequency, and available ILP. In addition, using these properties, the relative performance on different cores can be predicted. They propose a technique that uses architectural signatures (created using offline profiling and embedded into binary) to find the best application-to-core mapping. They show that their technique provides performance improvement very close to that of best static assignment that tests all possible scheduling options and finds the one with the best performance.

Chen and John [2009] present a scheduling technique that projects the core configuration and application resource demand to a unified multidimensional space and uses the weighted Euclidean distance (WED) between the two to schedule the application to cores. They show that WED and EDP are strongly correlated, and hence WED can be used as a proxy for deciding the execution efficiency of an application on a core. The smaller the value of WED, the better the execution efficiency. As for application characteristics, they examine data locality, branch predictability, and ILP, and for core configuration, they examine L1 data cache size, branch predictor size, and issue width.

5.6. Scheduling Virtual Machines on an AMP

Different VMs running on a host may have different resource requirements, and hence intelligent sharing of AMP resources among them is important for improving performance and fairness. We now discuss techniques proposed for addressing this need.

Kazempour et al. [2010] present a scheduling technique for hypervisors implemented in Xen. To ensure that all virtual CPUs (vCPUs) equally share the fast physical cores, the quota of a VM is decided depending on the number of vCPUs in it. For certain parallel applications, uniform distribution improves performance, as it accelerates all vCPUs. Their technique also supports assigning priorities to VMs. The fast-core cycles are first assigned to the high-priority VMs and then to the other VMs. Using this priority mechanism, sequential phases can be run on the fast core, which offsets the impact of a serial bottleneck on performance. Migrations are performed infrequently, which limits their overhead. They show that compared to the asymmetry-unaware default Xen, their technique provides significant improvement in performance.

Cao et al. [2012] note that managed languages such as PHP, Java, and C# use VM services to abstract over HW; however, due to these services, they also present differentiated performance and power workload and impose significant overhead. They propose the use of AMPs to boost VM services. Four key SW attributes, namely parallelism, asynchrony, noncriticality, and HW sensitivity, decide the strategy of mapping these services on cores of an AMP. For example, garbage collection (GC) does not happen on a critical path and can be performed asynchronously. Additionally, its main computation is parallel, and due to its memory-bound nature, it benefits from higher memory bandwidth. Hence, low-power InO cores with high memory bandwidth can be used for GC for improving system performance per energy (PPE). Similarly, a just-in-time optimizing compiler is noncritical and asynchronous, and it exhibits some parallelism; hence, it can be mapped to a small core for improving PPE. By comparison, an interpreter is not asynchronous and lies on critical path, and the parallelism of applications is manifested in that of the interpreter; thus, interpreter should be run on multiple cores. However, an interpreter also has low ILP, a small memory bandwidth requirement, and a small cache footprint; hence, executing it on low-power cores provides better PPE than executing it on a high-performance high-power core. Overall, their HW-SW co-design approach improves energy, performance, total power, and PPE significantly.

Takouna et al. [2011] present a scheduling technique for hypervisors that schedule VMs on an AMP. In their technique, the VMs with CPU-intensive applications are assigned to fast cores, as they show fewer CPU stalls due to infrequent memory accesses or I/O operations. The VMs with I/O-intensive applications are assigned to slow cores, which saves energy without losing performance. They showed that compared to an asymmetry-unaware round-robin scheduling technique, their technique improves performance and energy efficiency.

5.7. Scheduling OS and Application Threads on an AMP

Mogul et al. [2008] note that a big core does not accelerate OS to the same degree as it accelerates the application. Based on this, they suggest running OS kernel code, virtualization helper code (e.g., as in Xen), and device interrupts on an OS-friendly small core for improving energy and area efficiency. For applications with insufficient parallelism, OS code is migrated to a low-power core that performs interrupt handling in an energy-efficient manner. Additionally, high-performance cores can be selectively powered off for saving energy. For applications with sufficient parallelism, the OS load can be shifted to small cores, which frees high-performance cores for running application code.

Hruby et al. [2013] note that in many use cases, the performance of small cores is sufficient, and using them for OS code and the big cores for application code can provide higher total throughput. Their architecture enables using disjoint-ISA AMPs. For this, their OS provides a *live update* functionality by which a component can be replaced by that compiled for a different ISA, without shutting down the system. Recompilation alone can produce the desired version, as both versions are produced from a single code. To save energy, the workloads (e.g., system processes) can be consolidated on a few cores, and the remaining cores can be turned off. They show the advantage of their technique by using it in the network stack subsystem of the OS.

5.8. Reducing Task-Migration Overheads

Gutierrez et al. [2014] compare the impact of private and shared LLCs on the energy efficiency of AMPs. At the time of switching threads between cores, the working set of a thread needs to be saved in the cache; however, the cache of inbound core is expected to be cold with respect to the new thread. This leads to a performance penalty due to a large number of cache misses. To support working set migration, either a shared or private LLC(s) can be leveraged. Of the two, the shared LLC provides larger effective capacity but cannot be powered off to save energy; the private LLCs incur a larger number of coherence misses but may be easily powered off after working set migration is complete. They show that when the interval of switching threads between cores is reasonably large (e.g., 100K instructions or more), the performance difference between private and shared LLCs is negligible. This allows the use of private LLCs that can be optimized for the associated core. They also propose modifying the coherence protocol to enable ownership forwarding on read snoops, which reduces the writebacks with powered-off private LLCs and improves its energy efficiency.

Brown et al. [2011] present a technique to improve the performance of threads after migration between cores. Their technique records the access behavior of a thread during their execution. While forking or deactivating a thread, it summarizes the behavior to show potential future accesses to data and instruction and then migrates the summary with the remaining thread state. Finally, it prefetches suitable data and instruction into the cache of the destination core. They show that their technique improves performance both in the case of single-thread migration and on speculative MT systems. Similarly, Strong et al. [2009] discuss techniques to reduce the SW overhead of thread migration across cores.

5.9. Use of Fuzzy Logic

Chen and John [2008] note that the inherent characteristics of an application, such as instruction dependency distance (instruction count between the producer of a datum and its consumer), branch transition rate, and reuse distance distribution, determine its resource demand, which in turn decide the optimal core HW for it. They propose a technique that utilizes this fuzzy relationship between applications and cores to perform scheduling in AMPs. They use the application characteristics and the HW architectures to estimate the suitability degree for branch predictor size, cache size, and issue width. A fuzzy inference system is used to analyze these degrees, and by also integrating human knowledge, overall suitability degrees are obtained that have a strong correlation with EDP. Based on this, scheduling of applications on cores is performed.

Sun et al. [2011] present two fuzzy-control–based scheduling techniques for AMPs. One scheduler monitors IPC and migrates threads with higher IPC to faster cores. Another scheduler monitors LLC misses and migrates the threads seeing large number of misses to slower cores. The fuzzy controller has three engines, namely a fuzzification engine, an inference engine, and a defuzzification engine. The first engine converts a quantitative value into a qualitative one. Based on the human knowledge about the scheduling problem, the inference engine uses a set of rules (as mentioned earlier) to infer output fuzzy sets. These rules are created based on the performance heterogeneity of AMP cores. The defuzzification engine converts the output sets to a crisp number, based on which a thread is scheduled to a particular core. They implement their schedulers to form an asymmetry-aware Java virtual machine, which monitors every Java thread based on HW performance counters and makes suitable scheduling decisions.

5.10. Architecting Heterogeneous-ISA AMPs

Since the runtime state of an application is stored in an ISA-specific form, application migration across heterogeneous-ISA cores requires costly application state transformation. DeVuyst et al. [2012] propose a technique that minimizes the ISA-specific state to reduce migration overhead. Their approach ascertains the regions of application state whose form does not crucially affect performance. Using compiler analysis, they generate applications that store most of their state in an architecture-independent form, which minimizes the requirement of copying data and finding and fixing pointers at migration time. Since maintaining consistent state between different executables at all instructions incurs large overhead, they maintain equivalence only at function call points. To still support instantaneous migration, on any migration request, the application is moved to another core where binary translation runs until a function call is encountered, and then the application state is transformed and native execution continues on the core. They perform experiments using ARM and MIPS ISAs and show that their approach keeps the migration cost low. A recent work [Lustig et al. 2015] focuses on specifying and translating between different consistency models, in contrast to DeVuyst et al., who focus on cross-ISA migration under different microarchitectures.

Venkat and Tullsen [2014] show that choosing a diverse set of ISAs is critical for improving effectiveness of heterogeneous-ISA AMPs. They evaluate several key aspects in characterizing ISA diversity, such as code density, register pressure, decode and instruction complexity, SIMD (single instruction, multiple data) processing, and emulation versus native floating-point (FP) arithmetic. They evaluate three ISAs, namely Thumb (32 bit) and X86-64 and Alpha (both 64 bit). Since the 32-bit and 64-bit ISAs have different virtual address space and page table hierarchy, a common address translation scheme is required. For this, they use a memory management unit and a common page table structure based on that used in x86-64 for all three ISAs. A 64-bit computation on Thumb is done by SW emulation. To create target-independent data sections, a common intermediate representation is used based on the LLVM compiler framework. This acts as a bridge between ISAs and allows target-specific compiler optimization at the backend. They explore design space of three different ISAs and several architectural features under power and area constraints to find best designs for a homogeneous architecture, a single-ISA AMP, and heterogeneous-ISA AMP. They show that across applications and even across different phases of individual applications, the heterogeneous-ISA AMP outperforms the best single-ISA AMP.

Li et al. [2010] present an OS-based scheduling technique for AMPs where cores have different performance and overlapping but nonidentical ISA. For example, in the Cell processor, cores have the same virtual memory architecture and data types but differ in remaining features of the ISA [Gschwind et al. 2006]. Their technique runs applications on a core regardless of its ISA. When an unsupported instruction is encountered, a fault-type exception is generated. The fault handler migrates this thread to a core that supports the faulting instruction, and thus execution of the thread continues on the new core with re-execution of the faulting instruction.

Srinivasan et al. [2011] present a HW-assisted OS scheduling technique for AMPs with virtual/physical asymmetric or hybrid cores (refer to Figure 2). While an application is running on a core, a HW module monitors the compute and stall time. Using this, the performance of the application on remaining core types is predicted. The module also monitors time spent on general-purpose or special-purpose operations to find presence of code portions that can be offloaded to an accelerator. Based on this, the OS schedules the application to a core that provides the highest performance. They show that their technique improves performance in all three types of heterogeneity scenarios.

Lin et al. [2012] present a technique for mobile systems that enables the developers to easily program a peripheral processor (PP), which may have extreme architectural asymmetry compared to the central processor (CP). They provide an SW distributed shared memory (DSM) design that facilitates code on asymmetric processors to share data even in absence of a HW-coherent memory. Using an SW protocol to coordinate memory components of asymmetric processors for collaboratively serving memory accesses, DSM presents the abstraction of a shared memory, which is a more widely used abstraction than message passing. Using this, a developer can encapsulate code for PP as a special module and the compiler and runtime ensure that the encapsulated code runs efficiently on the PP and communicates with the remaining application as if running on the CP. To reduce the communication overhead, they use a lightweight consistency model that allows most communication to happen at synchronization points. Further, by serving most requests from the PP, the CP can be powered down to save energy.

Barbalace et al. [2015] present an SW architecture for running shared memory programs on heterogeneous-ISA AMPs. They assume a processor where multiple cores with the same ISA and a single memory coherency domain form a single processor island. Multiple processor islands may not support shared memory or cache coherency. Their technique provides SW DSM so that shared memory programs can run on heterogeneous-ISA AMPs without having to be rewritten. Their compiler framework uses offline analysis and profiling along with information of HW platform to produce a multi-ISA binary that can execute on their OS. At specific points, the runtime selects the most efficient island and migrates the thread to that island to run a code block on the most optimized ISA. Their experiments on a machine with Xeon and Xeon Phi show that their technique allows programs to run significantly faster than using OpenCL and the most performant native execution on either Xeon or Xeon Phi.

6. ACHIEVING VARIOUS OPTIMIZATION OBJECTIVES IN STATIC AMPS

In this section, we discuss research works on static AMPs in terms of the “figure of merit” they seek to optimize. Table V classifies these works based on their optimization objectives, such as performance, energy, and fairness. Several of these works are discussed next.

6.1. Techniques for Improving Energy Efficiency

Kumar et al. [2003] present an energy saving technique that works by dynamically estimating the resource requirement of a program and mapping it to a suitable core. For example, a wide-issue superscalar processor can issue several instructions in each cycle and hence is most suited for a program with high ILP. Mapping a program with low ILP on this processor leads to a waste of resources. Their approach allows optimizing for a different objective, such as performance and energy. They also show that chip-wide DVFS cannot provide as large energy gains as the intelligent use of AMPs.

Sawalha and Barnes [2012] present a dynamic scheduling technique to improve energy efficiency in AMPs. For each instruction window (of 10K instructions), their technique estimates its working set signature based on the blocks of instructions executed in this phase. By comparing the signature of consecutive windows, phases of execution are identified. For an unidentified phase, the thread EDP on each core type is computed for that phase. On repeated occurrence of a phase, its EDP is expected to be similar to that in its first occurrence. The EDP values of different phases are recorded, and based on that a thread-to-core mapping leading to the smallest value of EDP is chosen.

Nishtala et al. [2013] present a scheduling technique that accounts for both computation and memory demands of the application by monitoring retired instructions and LLC misses. Then, it maps the threads with high IPS and low miss rate to big cores and those with low IPS and higher miss rate to small cores. They show that for a general case where multiple threads are assigned to a core, their technique spreads contention for shared resources uniformly across the available cores and also provides significant improvement in energy efficiency.

Padmanabha et al. [2013] note that conventional thread migration policies cannot quickly react to fine-grain (at the level of hundreds of instructions) performance variations. They propose a technique that stores the current program context and execution history of each phase. Using this history information, their scheduler predicts phase changes and based on that preemptively migrates the execution to the suitable core in

Table V. A Classification Based on Study/Optimization Objective

Category	*References*
Energy	[Annamalai et al. 2013; Annavaram et al. 2005; Cao et al. 2012; Chen and John 2008, 2009; Chen and Guo 2014; Cong and Yuan 2012; Eyerman and Eeckhout 2014; Fallin et al. 2014; Gibson and Wood 2010; Grant and Afsahi 2006; Grochowski et al. 2004; Gupta and Nathuji 2010; Gutierrez et al. 2014; Homayoun et al. 2012; Hruby et al. 2013; Khan and Kundu 2010; Khubaib et al. 2012; Kim et al. 2007, 2014; Ko et al. 2012; Kumar et al. 2003; Lakshminarayana and Kim 2008; Lin et al. 2012, 2014; Lukefahr et al. 2012, 2014; Luo et al. 2010; Mogul et al. 2008; Morad et al. 2006; Mühlbauer et al. 2014; Muthukaruppan et al. 2013, 2014; Navada et al. 2013; Nishtala et al. 2013; Padmanabha et al. 2013; Patsilaras et al. 2012; Petrucci et al. 2012; Pricopi et al. 2013; Ra et al. 2012; Robatmili et al. 2013; Rodrigues et al. 2011, 2014; Sankaralingam et al. 2003; Srinivasan et al. 2013, 2015; Takouna et al. 2011; Venkat and Tullsen 2014; Watanabe et al. 2010; Wu et al. 2011; Zhang et al. 2014a; Zhu and Reddi 2013]
Deactivating unused core/ component to save energy	[Gibson and Wood 2010; Grochowski et al. 2004; Gutierrez et al. 2014; Hruby et al. 2013; Khubaib et al. 2012; Kim et al. 2014; Ko et al. 2012; Kumar et al. 2003; Lin et al. 2012, 2014; Luo et al. 2010; Mogul et al. 2008; Muthukaruppan et al. 2013, 2014; Patsilaras et al. 2012; Sankaralingam et al. 2003; Srinivasan et al. 2013; Watanabe et al. 2010]
Performance	[Annamalai et al. 2013; Annavaram et al. 2005; Ansari et al. 2013; Balakrishnan et al. 2005; Barbalace et al. 2015; Becchi and Crowley 2006; Cao et al. 2012; Chen and John 2008, 2009; Chen and Guo 2014; Cong and Yuan 2012; DeVuyst et al. 2012; Eyerman and Eeckhout 2010, 2014; Fallin et al. 2014; Georgakoudis et al. 2013; Gibson and Wood 2010; Gulati et al. 2008; Gupta et al. 2010; Gupta and Nathuji 2010; Gutierrez et al. 2014; Hill and Marty 2008; Homayoun et al. 2012; Ipek et al. 2007; Joao et al. 2012, 2013; Juurlink and Meenderinck 2012; Khan and Kundu 2010; Khubaib et al. 2012; Kim et al. 2007, 2015; Ko et al. 2012; Koufaty et al. 2010; Kumar et al. 2004a, 2004b, 2006; Kwon et al. 2011; Lakshminarayana and Kim 2008; Lakshminarayana et al. 2009; Li et al. 2007, 2010; Lin et al. 2014; Liu et al. 2013; Luo et al. 2010; Lustig et al. 2015; Madruga et al. 2010; Markovic et al. 2014; Morad et al. 2006, 2010; Mühlbauer et al. 2014; Mukundan et al. 2012; Najaf-Abadi et al. 2009; Najaf-Abadi and Rotenberg 2009; Navada et al. 2013; Nishtala et al. 2013; Panneerselvam and Swift 2012; Patsilaras et al. 2012; Pericas et al. 2007; Petrucci et al. 2012; Pricopi and Mitra 2012, 2014; Pricopi et al. 2013; Ra et al. 2012; Rafique et al. 2009; Robatmili et al. 2013; Rodrigues et al. 2011, 2014; Saez et al. 2010, 2012; Salverda and Zilles 2008; Shelepov et al. 2009; Sondag and Rajan 2009; Srinivasan et al. 2011, 2015; Suleman et al. 2007, 2009, 2010; Sun et al. 2011, 2012; Takouna et al. 2011; Van Craeynest et al. 2012, 2013; Van Craeynest and Eeckhout 2013; Venkat and Tullsen 2014; Watanabe et al. 2010; Wu et al. 2011; Zhang et al. 2014b; Zhong et al. 2007; Zhu and Reddi 2013]
Accelerating bottlenecks in MT applications	[Balakrishnan et al. 2005; Eyerman and Eeckhout 2010; Hill and Marty 2008; Joao et al. 2012, 2013; Kazempour et al. 2010; Lakshminarayana and Kim 2008; Lakshminarayana et al. 2009; Markovic et al. 2014; Morad et al. 2010; Suleman et al. 2009, 2010]
Fairness	[Kazempour et al. 2010; Kwon et al. 2011; Li et al. 2007, 2010; Markovic et al. 2014; Morad et al. 2010; Muthukaruppan et al. 2013; Pericas et al. 2007; Petrucci et al. 2012; Saez et al. 2015; Sondag and Rajan 2009; Sun et al. 2012; Van Craeynest et al. 2013; Zhang et al. 2014b]
Performance predictability	[Balakrishnan et al. 2005; Morad et al. 2010]
Domain-specific techniques	[Gupta and Nathuji 2010; Kim et al. 2015; Ko et al. 2012; Lin et al. 2012, 2014; Mühlbauer et al. 2014; Muthukaruppan et al. 2013, 2014; Ra et al. 2012; Zhu and Reddi 2013]
Reliability	[Gupta et al. 2010; Hruby et al. 2013; Ipek et al. 2007]

the AMP. Their technique allows execution on small cores for a large fraction of time to save energy with bounded performance loss.

Zhang et al. [2014a] propose a rule-set guided scheduling technique that works by creating “IF-ELSE” conditions on common performance metrics (stall and fetch cycles, cache misses). It maps programs to cores by comparing the runtime execution behavior to these rules at each scheduling interval. By checking these conditions, their technique decides whether the existing mapping or switching the programs on big and small cores will be more energy efficient.

6.2. Techniques for Improving Performance

Thread-level speculation leverages small cores to improve performance by allowing potentially dependent threads to run in parallel in a speculative manner. Inaccurate speculation, however, wastes power due to recovery and unproductive use of on-chip resources, and hence dynamic management of thread-level speculation is essential. Luo et al. [2010] present a mechanism for allocating speculative threads in AMPs where some of the cores support simultaneous multithreading (SMT), whereas other cores work as independent cores (CMP) with a private L1 cache. When the contention between speculative and regular threads is high, CMP provides higher energy efficiency than SMT and vice versa. Similarly, when TLP is high, CMP is beneficial since the threads can utilize multiple cores; however, when cache misses or synchronization lead to frequent stalls, CMP leads to a waste of resources, whereas SMT can partially hide cache miss and synchronization latencies. Based on these, at runtime their technique decides whether the speculative parallel threads may be allocated to CMP cores or to a single SMT-enabled core. Their technique also decides whether sequential/parallel threads should utilize a powerful wide-issue core or a less-powerful narrow-issue core. Additionally, when data reuse is small, the L1 cache is partially turned off for saving energy. When the advantages of thread migration and L1 turn-off do not justify their overhead, thread management operations are disabled. They show that their AMP with dynamic thread management performs better than the best SMP.

Liu et al. [2013] present a dynamic scheduling technique for improving performance under power constraint. They model generic mapping problem as a constrained 0-1 integer linear program that is an NP-complete problem. To solve this, they propose an iterative heuristic-based algorithm that works by first mapping threads to achieve highest possible throughput without consideration of power constraint. Afterward, swapping of threads is attempted between cores of nearly similar power consumption to meet the power budget. Compared to the optimal algorithm, their algorithm improves runtime by two orders of magnitude while providing results very close to the optimal algorithm. In addition, their algorithm scales well to hundred-core AMPs with little overhead and hence can be used as an online algorithm.

Becchi and Crowley [2006] show that in an AMP, dynamic thread migration policies provide larger performance improvement than the static policies. Their dynamic thread migration policy executes the threads for a small time duration on each core to measure their IPC. Based on this, a thread that achieves only modest performance improvement from running on a fast core is executed on a slow core, and a thread that benefits significantly from running on a fast core is executed on the fast core.

6.3. Techniques for Addressing Bottlenecks in MT Applications

A naive scheduler for AMP may execute bottlenecks in MT programs on a slow core, which harms performance severely (refer to Section 3.2.5). We now discuss several techniques that seek to avoid this situation and even benefit from asymmetric HW.

Suleman et al. [2009] present a technique that runs CSs on a big core to accelerate them and reduce thread serialization. On encountering a CS, a small core requests the big core to execute the CS. The big core acquires the lock, runs the CS, and notifies completion to the small core. Executing CSs exclusively on a big core obviates the need of frequent migration of lock and shared data between caches of big and small cores, as such data can always remain in the cache of a big core. This avoids extra misses due to data transfer. However, when several disjoint CSs contend for the big core, executing all CSs on a big core can lead to false serialization; to avoid this, their technique selects the CSs that should get executed on the big core and runs remaining CSs on the small core to avoid contention on the big core. In addition, to allow the big core to execute multiple CSs, SMT capability is provided to it.

Lakshminarayana et al. [2009] note that most threads, created together, execute for nearly-equal duration, because several parallel programs use multiple threads to execute a single piece of code and programmers generally perform static load balancing across threads. They propose a technique that uses this insight, along with knowledge of when threads were created, to estimate the remaining execution time of threads. Using this, their technique maps the thread with the largest remaining execution time (i.e., a lagging thread) to the fast core of an AMP to boost performance.

Saez et al. [2012] present a technique for scheduling single- and multithreaded programs in AMPs. They compute a utility factor (UF) based on both the TLP of program and the speedup of its threads. For N fast cores, the UF shows program speedup if N program threads are mapped to fast cores and remaining threads run on slow cores, compared to the case where all threads run on slow cores. They evaluate (1) AMP with cores of different frequency and (2) AMP with cores of different microarchitecture. To predict performance for (1), they decompose the completion time into compute time (estimated by assuming constant IPC and accounting for frequency difference) and stall time (estimated based on cycles taken in servicing the LLC misses in a given instruction window and memory latency). To predict performance for (2), they use a machine learning tool to build regression models for both fast and slow cores, which compute speedup from performance metrics, such as IPC, L2 and L3 (LLC) miss rate, execution stall cycles, and retirement stall cycles. Based on this, the scheduler computes the UF for programs, and if a program's UF is higher than those of other programs, its threads are preferentially mapped to fast cores; otherwise, they are mapped to slow cores.

Joao et al. [2013] note that in an MT application, performance may be hampered by both lagging threads (due to load imbalance or cache miss, etc.) and bottlenecks (code segments, e.g., contended CSs, that stall other threads). To combine mechanisms for accelerating both of these requires predicting their relative benefits, which depends on the application phase, input set, and processor HW. For this, they define a "utility of acceleration" metric that accounts for both: the code segment's criticality for its application (i.e., the fraction of application execution time consumed by code segment) and the anticipated acceleration from executing it on a big core (i.e., reduction in application execution time by accelerating the code segment). This metric *quantifies* the potential of acceleration of a code segment, and using it, the code segments from one or multiple MT applications, which should run on the big core can be decided.

Markovic et al. [2014] present a thread lock section-aware scheduling (TLSS) technique for improving performance by avoiding thread serialization. They assume that the OS scheduler sees identical logical cores to which threads are mapped at every SW-quantum, and thus the OS scheduling policies are not affected by the presence of asymmetric cores. TLSS maps threads running on logical cores to physical cores in every HW quantum. To identify CSs, they detect whether a core made a recent transition from executing kernel code to user code, which indicates a thread reaching a synchronization point and waiting on a lock (OS futex). This approach does not require

ISA extensions but has the limitation that it does not identify all CSs. If either none or any of the small core made a transition, then TLSS swaps the logical core running on that small physical core with the logical core running on the big physical core at the beginning of next HW quantum. This prevents a thread from occupying the big core for more than one HW quantum, unless it is the only runnable thread. In addition, when a thread reaches a CS and waits on a lock, upon acquiring the lock, TLSS attempts to schedule the thread on big physical core to accelerate it.

6.4. Techniques for Improving Fairness

Petrucci et al. [2012] propose a scheduling technique that monitors the performance of a thread on an existing core to estimate its performance on other cores using analytical models developed using offline analysis. Using this, the scheduling is done using a "lottery scheduling" mechanism [Waldspurger and Weihl 1994], in which each thread receives a dynamic number of tickets determined by the energy efficiency ratio of running the thread on a big core versus a small core. A higher number of tickets implies that a thread has higher priority to run on a big core. By dynamically changing the number of tickets given to a thread, its priority for running on the big core can be increased, which can be used to reduce unfairness and improve performance.

Van Craeynest et al. [2013] note that performance-oriented scheduling and pinned scheduling (i.e., binding threads to a core) can lead to large unfairness in an AMP. They present two fairness-aware scheduling techniques, which also provide high performance. The "equal-time" scheduling runs each thread (or workload) on each core type for an equal fraction of time. Since this may not lead to equal progress of all threads in heterogeneous workloads, they also propose an "equal-progress" scheduling that aims to get equal amounts of work done on each core type. This technique schedules the thread with the currently largest slowdown on the big core. To estimate the performance ratio of big and small core, their technique can run each thread on each core type for a small time duration or use the PIE model (refer to Section 5.2). By running threads on the big core for a fraction of time, their techniques also improve performance. They further propose a scheduling policy that optimizes performance when fairness is higher than a threshold and switches to optimizing fairness when fairness drops below the threshold. For both homogeneous and heterogeneous MT workloads, their techniques improve performance over pinned scheduling. Further, for multiprogram workloads, their technique significantly improves fairness while providing performance close to performance-oriented scheduling and better than pinned scheduling.

Saez et al. [2015] present a technique to improve fairness while ensuring acceptable throughput. Their technique performs scheduling and thread migrations between cores to ensure that programs experience equal slowdown. To monitor programs' slowdowns, their technique assigns a counter to each thread. For single-threaded programs, the counter tracks how much the program has progressed with respect to the expected progress from running it on a big core for the whole time. For MT programs, the counter tracks the relative progress that the thread has made in the AMP with respect to other threads of the same program. To achieve fairness, the threads are migrated between cores whenever the difference in counter value between threads exceeds a threshold. Their technique also allows trading off fairness with throughput by increasing the big-core share of programs with higher speedup at the cost of reducing this share of remaining programs.

6.5. Techniques for Maintaining Application Priorities

In this section, we discuss priority-aware scheduling techniques for AMPs.

Intel's QuickIA [Chitlur et al. 2012] is a platform for prototyping AMPs consisting of multiple Intel processors, such as Xeon E5450 (big) with Atom 330 (small). This platform can be used to perform experiments with AMP architecture, workload mapping, and OS heuristics. As case studies, they profile performance of different applications on big and small cores and use this to perform intelligent scheduling to improve throughput. In addition, based on the priorities of different applications, they can be scheduled on either big or small cores.

Zhang et al. [2014b] study QoS management for parallel programs that require multiple cores to fully exploit the TLP and hence can benefit from intelligent thread scheduling policies. Their experiments are conducted with four programs, where one has high priority and the remaining three have low priority. They study two types of policies. The first policy reserves identical cores (either big or small) for the high-priority program. Since mapping all high-priority programs to a big core may cause unfairness, the second policy reserves different types of cores for a high-priority program and uses the remaining cores for a low-priority program. A variant of this is partial-dedicated policy, where part of a core is shared among different programs, and thus the high-priority program runs on both dedicated and shared cores, whereas low-priority programs run on the shared cores and the remaining cores. They show that different policies are most effective in different execution scenarios. Overall, their policies improve the performance of high-priority program while also balancing system fairness.

6.6. Techniques for Avoiding Performance Variance

An asymmetry-unaware scheduler may schedule threads of heterogeneous workloads to fast or slow cores in different runs, leading to vastly different performance. Some techniques for addressing this performance variance are discussed next.

Balakrishnan et al. [2005] study the behavior of commercial applications on both SMPs and AMPs. By first running the applications several times on SMP, they ensure that on such systems, the performance is predictable. On repeating the same experiments on AMP, performance was found to be unpredictable due to reasons such as asymmetry-unaware scheduling, synchronization, and cache thrashing. To address this, they propose an asymmetry-aware OS scheduler that migrates an application from a slow core to an idle fast core to ensure that faster cores may go idle only after the slower cores. This scheduler eliminates performance instability. Since poor load balancing leads to instability, in applications with a large number of short-duration tasks, better load balancing and hence stability is observed, although the overhead of creating and destroying short-lived tasks also harms performance.

Li et al. [2007] present an OS scheduler that records the computing power of each core at boot time and at runtime assigns loads on each core in proportion to its computing power to achieve load balancing. Whenever the faster cores are underutilized, threads are migrated to them, which also improves fairness. Using memory resident sets, thread migration overhead is estimated, and in the case of high overhead, the migration is disallowed. They show that their technique improves performance and also maintains fairness and repeatability of performance under different runs.

Morad et al. [2010] note that for multiple co-running MT applications, the serial phases compete for processor resources with concurrently executing parallel threads. Hence, they become critical bottlenecks, and by accelerating them the resources can be used by many threads, which improves the throughput and fairness. Based on this, they propose a scheduling technique that gives higher priority to serial threads to run on faster cores. When multiple applications simultaneously have serial threads, only one of them receives faster cores, and the remaining serial and parallel threads share the remaining cores. To avoid unfairness, higher priority given to serial threads lasts only for a fixed time. They show that their technique improves throughput and fairness and reduces the variance in execution time over different runs of the application.

6.7. Techniques Specific to Application Domains

Several techniques for AMPs address the issues or exploit the properties of specific application domains, as we show next.

Ko et al. [2012] note that the applications running on mobile devices show varying resource demands. For example, multimedia-intensive applications show high CPU utilization and user interaction, whereas service daemons and other background processes show low CPU utilization. Based on this, in standby mode of the device, their technique assigns all background tasks to a small core and does not activate the big core. In active mode, foreground applications are assigned to a big, core and those with low CPU utilization are migrated to a small core. If no application runs on the big core for certain period of time, it is turned off. Thus, their technique meets the QoS requirement of resource-intensive tasks and saves energy by adaptively turning off the unused cores.

Ra et al. [2012] present a scheduling technique for improving energy efficiency of mobile AMPs composed of a high-performance application processor (AP) and a low-power processor (LP). Their technique accounts for performance and power consumption of applications on cores and migration overhead. Sensor sampling and buffering tasks are scheduled on the LP, whereas computationally intensive tasks are scheduled on the AP. They show the implementation of their technique for cases of single-task, complete applications consisting of multiple tasks and multiple simultaneously running applications.

Zhu and Reddi [2013] note that in mobile Web browsing, short Web page loading time is crucial for a satisfactory user experience, and the Web pages that do not load within a cut-off time (e.g., 2 to 3 seconds) tend to get abandoned by the users. At the same time, low energy consumption is necessary for maximizing battery lifetime. Since there exists a large variance in load time and energy consumption of different Web pages, they estimate these using statistical inference models, which are built based on detailed characterizations of different Web page primitives of the hottest 5,000 Web pages. These primitives include both the structural information (e.g., HTML) and style information (e.g., cascading style sheet (CSS)). They propose a scheduling technique that uses this model to map Web pages to the most suitable core and DVFS setting in an AMP to save energy while meeting stringent cut-off constraints.

Mühlbauer et al. [2014] note that for "parallelized query processing" workloads, big cores or small cores show better EDP, depending on whether LLC can or cannot (respectively) hold the entire working set. This is because for large working sets, LLC misses frequently stall the pipeline and hinder OOO execution, and thus reduce the performance advantage of big cores. Based on this, they build a multiple segmented multivariate linear regression model for estimating the performance and energy consumption of the workload for a given core type and workload-specific parameters. Using this, their job scheduler estimates energy consumption and response time on all core types and maps the job to a core that minimizes the EDP.

7. DESIGN AND MANAGEMENT OF RECONFIGURABLE AMPS

We broadly classify reconfigurable AMP architectures in three types (Table VI): those that dynamically fuse or partition the cores and thus change the core count (Section 7.1), those which share/trade resources between cores (Section 7.2), and those that transform the core architecture (Section 7.3). Some designs require unconventional ISAs (Section 7.4) or use 3D stacking technology (Section 7.5). In what follows, we discuss these reconfigurable AMP architectures.

Table VI. Classification of Reconfigurable AMPs

Category	References
Types of Reconfigurable AMP Architectures	
Designs that change number of cores	[Chiu et al. 2010; Gulati et al. 2008; Hill and Marty 2008; Ipek et al. 2007; Juurlink and Meenderinck 2012; Kim et al. 2007; Mukundan et al. 2012; Panneerselvam and Swift 2012; Pricopi and Mitra 2012, 2014; Robatmili et al. 2013; Salverda and Zilles 2008; Sankaralingam et al. 2003; Sun et al. 2012; Tarjan et al. 2008; Watanabe et al. 2010; Zhong et al. 2007]
Designs that trade resources between cores	[Gupta et al. 2010; Homayoun et al. 2012; Kumar et al. 2004a; Pericas et al. 2007; Rodrigues et al. 2011, 2014]
Designs that morph core architecture	[Fallin et al. 2014; Khubaib et al. 2012; Lukefahr et al. 2012; Srinivasan et al. 2013, 2015]
Other Features	
Use of unconventional ISA or ISA extensions	[Kim et al. 2007; Robatmili et al. 2013; Sankaralingam et al. 2003; Suleman et al. 2009, 2010; Zhong et al. 2007]
Use of compiler	[Barbalace et al. 2015; Chen and John 2009; Cong and Yuan 2012; DeVuyst et al. 2012; Georgakoudis et al. 2013; Gupta et al. 2010; Joao et al. 2013; Kim et al. 2007; Lakshminarayana and Kim 2008; Lin et al. 2012; Luo et al. 2010; Pricopi and Mitra 2012, 2014; Pricopi et al. 2013; Sankaralingam et al. 2003; Shelepov et al. 2009; Suleman et al. 2009, 2010; Venkat and Tullsen 2014; Watanabe et al. 2010; Zhong et al. 2007]
Use of 3D stacking	[Homayoun et al. 2012]

7.1. Reconfigurable AMPs That Change Number of Cores

Ipek et al. [2007] present a reconfigurable architecture, called *core fusion*, that allows formation of wider-issue OOO cores by fusion of neighboring OOO cores. In their architecture, up to four cores along with their D/I-caches can be merged at runtime to form cores having as much as 4× D/I-cache, branch target buffer, and branch predictor size and as much as 4× commit, issue, and fetch width. Their reconfigurable load-store queue (LSQ) and D-cache organization supports conventional coherence while executing a parallel program and produces no coherence traffic while executing a serial program in fused mode. When cores execute independently, LSQs and D-caches of individual cores are leveraged to avoid thread interference in L1 caches. A decentralized frontend and I-cache design leverages the frontend structures of individual cores to feed the fused backend without requiring extra resources in different frontends. Core fusion can provide lean cores to support fine-grain parallelism, fewer more powerful four-issue supercores to support coarse-grain parallelism, and one eight-issue supercore to support sequential execution.

The performance of fused configuration in core fusion architecture is significantly smaller than that of an isoarea monolithic OOO processor. To avoid this limitation, Mukundan et al. [2012] address two inefficiencies of core fusion design. The original design commits one fetch group, consisting of eight instructions at a time and inserts no operations (NOPs) in the reorder buffer (ROB) whenever a fetch group does not fill to its capacity. This, however, reduces the effective size of ROB, and to avoid this, they make the representation of NOPs compact in ROB. They also use a genetic programming–based improved instruction steering mechanism to reduce the overhead of remote operand communication. Thus, with improved performance for sequential codes, a core fusion–like design can obviate the need of wide-issue cores in future multicore processors.

Zhong et al. [2007] present a reconfigurable AMP with multiple simple homogeneous cores that can be organized in either decoupled or coupled mode. In decoupled mode, cores independently operate on different fine-grain communicating threads. Using a compiler, the program is subdivided into several communicating subgraphs that are initiated using the thread-spawn process. These threads exploit TLP and loop-level parallelism. In coupled mode, cores collectively function as a wide-issue very large instruction word (VLIW) processor and process multiple instruction streams in a synchronous manner. To minimize the overhead of thread spawning and synchronization barriers in execution of single-threaded branch-intensive programs, compiler-directed distributed branch architecture is used. Using this, multibasic block code regions can be executed by multiple cores. Memory dependences and memory communication are handled using flexible memory synchronization. The compiler identifies the types of parallelism present in a program phase and decides which one to exploit. The compiler partitions the code into threads, schedules instruction to cores, and directs communication among them. The requirement of a complex compiler is a limitation of their technique.

Tarjan et al. [2008] present an architecture where two scalar InO cores can be combined to achieve two-wide, OOO issue. They approximate OOO issue with area-efficient structures by using low-overhead lookup tables in place of broadcast networks and content addressable memories (CAMs). For example, instead of a CAM-based issue queue (IQ), a simple table is used where consumers subscribe to their producers by writing their IQ position in the consumer field entry of a producer's IQ. These lightweight OOO structures are placed between two scalar cores and utilize the decode, fetch, cache, register file, and datapath of scalar cores to provide an OOO core with performance close to that of a dedicated OOO core. For a single-threaded application, this federated core achieves double the performance of the InO core.

Watanabe et al. [2010] present an architecture that uses multiple InO execution units (EUs) to approximate OOO-issue capability on demand. They decouple the pool of EUs from instruction engines (IEs) that manage the thread context. This decoupled architecture allows versatile EU organization and to distribute instructions to EUs, they use instruction steering. Whenever possible, the dependent instructions are executed in the cluster formed by four adjacent EUs. Independent instructions are steered to free EUs and are executed simultaneously. By using free EUs, independent instructions can run ahead of the previous stalled instructions, which helps in exploiting ILP and MLP. An IE performs instruction steering for the EUs along with front- and backend pipeline functions of an OOO core. By adapting the active IE count and the number of EUs associated with each IE, the cores are dynamically customized. Unused resources are turned off for saving energy, and thus the number of cores are scaled. They show that by virtue of EU provisioning, their design dissipates power in proportion to single thread performance and thus achieves power proportionality.

Chiu et al. [2010] present an architecture that allows multiple scalar cores to be dynamically merged into a larger superscalar processor to benefit programs with both low and high TLP. When multiple cores are merged, the valid register values may be present in different register files of the merged cores. To address this, they use virtual shared register files (VSRFs). Using this, instructions of a thread executed in a merged core logically see a uniform group of register files. They use instruction analysis to identify dependence information and store it with the newly fetched instructions. Based on this, instructions in the merged cores can use VSRF to obtain their remote operands. They show that their two-core, four-core, and eight-core merged designs perform close to monolithic two-issue, four-issue, and eight-issue OOO superscalar designs while supporting a more diverse range of programs.

Pricopi and Mitra [2012] present a reconfigurable multicore architecture to exploit both ILP and TLP. Their architecture allows dynamically merging two or more simple cores to offer ILP to serial codes. For example, two to four two-way OOO cores can be merged to provide equal or better performance than a more complex four-way or eight-way OOO core. For scalability, they use a distributed execution framework that precludes centralized fetch/decode, instruction-scheduling, and dependency resolution. Using a compiler, the basic blocks and their register dependencies are identified. Using this, cores can fetch and execute different basic blocks at a time to boost the performance.

7.2. Reconfigurable AMPs That Trade/Share Resources between Cores

Pericas et al. [2007] present a processor architecture with runtime-adaptive instruction window size. The architecture contains a fast "cache core" that executes high locality code that depends only on cache accesses. Low-locality memory-intensive code is processed by a second core, called a *memory engine*. By dynamically removing or adding the memory engines, the instruction-window size can be changed. By sharing a memory engine network among threads, their technique adapts to application requirements and activates only the desired memory engines to improve performance within the power budget. Depending on the pending main memory accesses in groups of instructions, they have low or high execution locality, and based on this the corresponding threads are allocated to different cores. In their architecture, the use of several cores helps in hiding long-latency memory operations and achieving large effective instruction window size, by which ILP is exploited. Further, TLP present in parallel programs is exploited by assigning threads to cores based on performance.

Gupta et al. [2010] present a reconfigurable architecture that organizes the processor as a reconfigurable network of building blocks, which may be single- or multiple-pipeline stages. This architecture allows constructing logical processors from any complete combination of its constituent blocks at runtime. Many single-issue pipelines exploit parallel performance, whereas wider-issue pipelines exploit serial performance. For resilience, disabling is performed at fine-granularity of blocks instead of coarse-granularity of cores, and the remaining working blocks can form logical processors. Since reconfiguring distributed pipeline stages into a wide-issue processor presents high communication overheads, they use speculation of data and control dependencies across pipeline ways, and in the case of violation, they use a lightweight replay. Dataflow violations due to instructions being executed on two different pipeline ways harm performance. To avoid this, they use a compiler to statically identify data dependency chains. Using this, the issue stage steers instructions to the appropriate pipeline way.

Rodrigues et al. [2011] present a reconfigurable AMP architecture where each core has moderate overall resources with additional strength in a unique area. For example, in a two-core AMP, one core may handle integer operations well but provide poor performance on FP operations, and vice versa for the other core. When a thread needs strength in multiple areas, the execution resources of cores are realigned such that cores gain/lose strength in an area. For example, the weaker FP unit of core 1 is traded with stronger FP unit of core 2, and thus core 1 gets strength on both fronts, whereas core 2 loses strength in both the fronts. Thus, when the computation demand of threads change, their technique may either do morphing, swap the threads executing on the cores, or return to the original configuration, with the goal of improving energy efficiency.

Rodrigues et al. [2014] note that performance-optimized big cores are generally overprovisioned; however, they may remain underutilized. Additionally, due to their energy-optimized design, small cores do not have as many execution units. When the programs running on a small core require additional resources, the underutilized resources of the big core can be shared with the small core, which affects the performance of big core negligibly, but leads to area and power saving and significant improvement in performance of the small core. They propose an architecture where cycle by cycle, the integer and FP multiply and divide units, and FP ALU can be shared between cores. They show that their technique improves overall performance and performance per watt.

7.3. Reconfigurable AMPs That Morph the Core Architecture

Khubaib et al. [2012] present an architecture that uses a large OOO core as a base substrate to boost single-threaded programs and allows reconfiguring to the InO SMT core to accelerate parallel code. When the number of active threads exceeds a threshold, their technique reconfigures the core into a highly threaded InO SMT core that exploits TLP and provides high throughput by hiding long-latency operations. The SMT core is created using a subset of HW required for an aggressive OOO core (e.g., the execution pipeline of OOO core is used as such, the reservation station entries are used as the InO instruction buffer, and the physical register file of the OOO core is used as the architectural register files for many SMT threads). The structures that are not used in InO mode, such as OOO scheduling and renaming logic, are turned off to save energy.

Lukefahr et al. [2012] present a core architecture composed of a high-performance core (big μEngine) and an energy-efficient core (small μEngine). Only one μEngine is active at a time, and the execution switches dynamically between μEngines to adapt to the program characteristics. Using an online performance estimation scheme, the throughput of unused μEngine is predicted. If the unused μEngine has significantly higher throughput or better energy efficiency than that of active μEngine, the execution is switched. TLBs, fetch units, branch predictors, and L1 caches are shared between μEngines, and hence on a switch, only the register state needs to be transferred between the cores. To reduce this overhead even further, they transfer the register state in a speculative manner. The low-switching overhead enables switching at granularity of few thousands of instructions. By exploiting the phases where the performance gap between big and small μEngines is negligible, execution on a small μEngine can be maximized to save energy while limiting performance loss to a configurable bound.

Fallin et al. [2014] note that serial programs typically show heterogeneity at granularity of tens/hundreds of instructions, and this heterogeneity can be exploited by grouping contiguous regions of instructions into atomically executed blocks. Based on this, they propose a design that uses OOO, VLIW, and InO execution backends in each core. In addition, their technique decomposes the program into code blocks, finds the best backend for each block, and customizes the block for that backend. For a time, different blocks may use different backend types. They show that their technique improves energy efficiency compared to monolithic OOO, clustered OOO, and InO designs while maintaining performance.

Srinivasan et al. [2013] present an architecture that allows morphing a single superscalar OOO core to InO core at runtime whenever the latter is found to be more efficient on a given objective (e.g., power efficiency (performance/watt)). During high ILP phases, execution is performed using an OOO core. When the processor stalls due to dependencies or long-latency memory operations, most resources stay idle, and hence during low-ILP/memory-intensive phases, the operation is switched to an InO core with reduced fetch width and core resources to achieve higher power efficiency. In addition, unused processor structures, such as ROB and LSQ, are powered off. Since fetch width is reduced, the unused decoders are also powered off. While in InO mode, if a high-ILP phase is encountered, the execution is switched to OOO mode and the shutdown units are again activated.

Srinivasan et al. [2015] note that applications show different power efficiency (throughput/watt) for cores with different microarchitecture, such as different sizes of ROB, LSQ, and instruction queue; different fetch and issue width; and voltage/frequency. They explore the design space of such architectures to find the minimum number of distinct core types that benefit a wide range of workloads and allow fine-grain switching. These core types determine the number of modes of their morphable core architecture. Using HW counters, the most power-efficient mode for the current execution is found, and the core is dynamically morphed into that mode. Their architecture provides higher power efficiency than a static OOO core design.

7.4. Use of Unconventional ISA

Sankaralingam et al. [2003] present a polymorphous design that allows reconfiguration for multiple parallelism types and granularities, such as ILP, TLP, and data-level parallelism (DLP). Their processor architecture has four OOO, 16-wide-issue cores for boosting high-ILP single-threaded programs. These cores can be further partitioned to exploit fine-grain parallelism. Several resources in their architecture are polymorphous (e.g., register file banks, block sequencing controls, memory tiles, and reservation stations). For example, the memory tiles can be configured to work as scratchpad memory, L2 cache banks, or synchronization buffers. To exploit ILP, they treat the core's instruction buffers as a decentralized instruction issue window that leverages TRIPS ISA for achieving OOO execution. To exploit TLP using multiple threads, they provision space sharing the arithmetic and logical unit (ALU) array between threads. To exploit DLP, they note that such streaming programs show predictable control flow, and for them, multiple architectural frames (the group of reservation stations in different nodes with the same index) are fused together, and to fill the fused frames' reservation stations, the inner loops of streaming programs are unrolled.

Kim et al. [2007] present a reconfigurable architecture that enables simple cores to be dynamically composed into larger cores to optimize performance, energy, or area efficiency. To allow the processor to scale up to a large issue width (e.g., 64 wide), they completely avoid physical sharing of resources, which precludes the use of conventional reduced/complex instruction set computing (RISC/CISC) ISAs. Hence, they use nonstandard explicit data graph execution (EDGE) ISA, in which the dependence order of instructions within a block is explicitly and statically encoded, and thus instruction dependence relations are known a priori. This allows the I-cache capacity and instruction fetch bandwidth of the cores to linearly scale with increasing numbers of cores. Thus, the scalable reconfiguration achieved by their technique provides higher flexibility and performance than the techniques that centralize some of the processor resources.

7.5. Use of 3D Stacking Technology

Homayoun et al. [2012] note that a 2D design of reconfigurable AMPs leads to inefficient pooling of resources, which increases the pipeline depth and communication delays. They propose 3D design of reconfigurable AMPs, which allows optimized layout of the pipeline in 2D plane. Further, additional resources (e.g., LSQ, ROB, cache, registers, and instruction queue) are connected in the third dimension without perturbing the 2D pipeline layout. This allows resource sharing at fine granularity to exploit both ILP and TLP and achieving smaller communication delays due to 3D stacking [Mittal et al. 2014]. Further, the same design can be used to provide flexible resource pooling with different numbers of layers (e.g., 1, 2, or 4). They show adaptation in four key components, namely ROB, LSQ, register file, and instruction queue.

7.6. Use of Scheduling Policies in Reconfigurable AMPs

We now discuss techniques that seek to reduce scheduling overhead of reconfigurable AMPs and improve their performance and fairness.

Panneerselvam and Swift [2012] present a technique to extend Linux OS to support reconfigurable AMPs. When a reconfigured core is temporarily unavailable, a processor proxy for it is created on another core. Operations such as TLB shootdown require presence of the core, and to allow such operations to continue without waiting for unavailable core, their technique allows the proxy to access the per-CPU private data of the unavailable core. This reduces the reconfiguration latency from 150ms to 2.5μs and allows exploiting fine-grain reconfiguration opportunities. Further, to provide a logical abstraction of physical cores and HW threads, execution objects are introduced. The threads are scheduled to these execution objects, and thus the scheduler need not be aware of physical HW. Based on the requirements of parallel and serial applications, the scheduler decides when and what to reconfigure. For example, a high-priority serial code can be allowed to preempt other cores, or cores can be reserved for an important parallel application. Their technique allows serial codes to run on a resource-rich execution context and parallel codes on multiple cores and provides five orders of magnitude performance improvement over the Linux scheduler.

Sun et al. [2012] present a technique to improve performance and fairness in reconfigurable AMPs. Since collecting system-wide information about the most suitable core for each task incurs lower overhead with a centralized run queue (RQ) than with distributed RQs, their technique uses a centralized RQ to improve the effectiveness of scheduling and resource allocation decisions. The OS scheduler runs on separate cores than the reconfigurable cores, and hence the scheduling latency incurred by it is overlapped with that of performing useful computation on the reconfigurable cores, and this partially offsets the overhead of scheduling with centralized queues. Further, to ensure fairness, their technique makes scheduling decisions with the aim to provide similar tasks with performance in order of their priorities. It also seeks to ensure that when running simultaneously, dissimilar tasks of equal priority may get equal slowdowns.

7.7. Creating the Illusion of a Reconfigurable AMP Using a Static AMP

Some researchers propose techniques to use a fixed HW fabric and still provide the appearance of reconfigurability without incurring the overheads associated with it. We now discuss an example of this.

Ansari et al. [2013] note that in an AMP, the power and thermal budgets restrict the number of big cores and hence the number of threads that can be simultaneously accelerated. In their technique, instead of running individual threads for optimizing single-thread performance, big cores redundantly execute threads running on the small cores to provide execution hints. Using these hints, small cores achieve a higher cache hit rate and branch prediction accuracy. Due to the limited HW resources of a big core, it cannot simultaneously execute a large number of threads, and hence their technique runs a reduced, yet reasonably accurate, version of program on a big core. To obtain this version, code analysis is done using a dynamic compiler to remove all instructions that are not necessary for generating execution hints. Additionally, the program execution phases, in which the small core can benefit the most from execution hints, are selected. If hints are found unprofitable, hint gathering is disabled. Their technique provides better single-thread performance than using all small cores and better throughput than using all big cores.

8. CONCLUSION AND FUTURE OUTLOOK

In this article, we provided a comprehensive view of the research on designing and optimizing both static and reconfigurable AMPs. To promote the use of precise and standard nomenclature, we synthesized the terminology used in the literature. We highlighted the potential and challenges of AMPs. We believe that this article will be beneficial to researchers, processor architects, system designers, and technical marketing professionals and will open promising research avenues. We conclude with a brief discussion of the future outlook for this field.

AMPs include energy-efficient small core(s) for improving energy efficiency; however, the requirement of energy efficiency placed on future computing systems demands even more aggressive power management approaches. For example, exascale computing targets 10^{18} computations per second within a power budget of 20MW [Vetter and Mittal 2015], and mobile systems seek to process visually compelling graphics at the expense of a few watts. Evidently, the use of techniques like near-threshold computing, data compression, and low-leakage devices such as nonvolatile memory will be essential for AMPs to become even more economical in consuming energy.

With ongoing process scaling, the fault rates due to process variation and soft errors are increasing. Addressing these faults in AMPs poses unique challenges due to their extreme heterogeneous nature. Although most initial works have only studied performance and power optimization, moving forward, an in-depth exploration of reliability of AMPs is required to ensure their integration in commodity and especially mission-critical systems.

The heterogeneity of AMPs makes their design quite challenging, and these challenges are compounded by the presence of multiple management policies (DVFS, resource scaling such as cache reconfiguration, workload consolidation, etc.) and system-level objectives (per-application performance, throughput, energy, area, etc.), which are commonly seen in real-world systems. To transform the nascent, research-grade proposals for AMPs into mature and robust real-world commercial solutions, synergistic efforts are required from both academia and industry.

Most existing large-scale scientific and commercial applications are written for symmetric HW. Redesigning them to not just work on but benefit from asymmetric HW will demand major efforts. At the SW level, automating the task of profiling the performance of different cores and mapping suitable phases/applications to them will enable them to scale to large-size problems and a wide range of application domains. Simultaneously, architecture-level innovations are required to reduce task migration overhead in static AMPs and mode-switching overhead in reconfigurable AMPs. Much remains to be done before all layers of the computing stack will provide in-built support for underlying asymmetry.

REFERENCES

Arunachalam Annamalai, Rance Rodrigues, Israel Koren, and Sandip Kundu. 2013. An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. 63–72.

Murali Annavaram, Ed Grochowski, and John Shen. 2005. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05)*. 298–309.

Amin Ansari, Shuguang Feng, Shantanu Gupta, Josep Torrellas, and Scott Mahlke. 2013. Illusionist: Transforming lightweight cores into aggressive cores on demand. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'13)*. 436–447.

ARM. 2015a. big.LITTLE Technology. Retrieved December 29, 2015, from http://www.arm.com/products/processors/technologies/biglittleprocessing.php.

ARM. 2015b. Cortex-A Series Processors. Retrieved December 29, 2015, from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortexa/index.html.

Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'05)*. 506–517.

Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the programmability gap in heterogeneous-ISA platforms. In *Proceedings of the European Conference on Computer Systems (EuroSys'15)*. 29:1–29:16.

Michela Becchi and Patrick Crowley. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the Computing Frontiers Conference (CF'06)*. 29–40.

Jeffery Brown, Leo Porter, and Dean M. Tullsen. 2011. Fast thread migration via cache working set prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'11)*. 193–204.

Ting Cao, Stephen M. Blackburn, Tiejun Gao, and Kathryn S. McKinley. 2012. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the International Symposium on Computer Architecture (ISCA'12)*. 225–236.

Jian Chen and Lizy Kurian John. 2008. Energy-aware application scheduling on a heterogeneous multi-core system. In *Proceedings of the International Symposium on Workload Characterization (IISWC'08)*. 5–13.

Jian Chen and Lizy Kurian John. 2009. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the Design Automation Conference (DAC'09)*. 927–930.

Quan Chen and Minyi Guo. 2014. Adaptive workload-aware task scheduling for single-ISA asymmetric multicore architectures. *ACM Transactions on Architecture and Code Optimization* 11, 1, 8:1–8:25.

Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, Pragya K. Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. 2012. QuickIA: Exploring heterogeneous architectures on real prototypes. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'12)*. 1–8.

Jih-Ching Chiu, Yu-Liang Chou, and Po-Kai Chen. 2010. Hyperscalar: A novel dynamically reconfigurable multi-core architecture. In *Proceedings of the International Conference on Parallel Processing (ICPP'10)*. 277–286.

CNXSoft. 2014. ARM Cortex A15/A17 SoCs Comparison—Nvidia Tegra K1 vs Samsung Exynos 5422 vs Rockchip RK3288 vs AllWinner A80. Retrieved December 29, 2015, from http://www.cnx-software.com/2014/05/21/comparison-nvidia-tegra-k1-samsung-exynos-5422-rockchip-rk3288-allwinner-a80/.

Jason Cong and Bo Yuan. 2012. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'12)*. 345–350.

Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. 2012. Execution migration in a heterogeneous-ISA chip multiprocessor. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 261–272.

Stijn Eyerman and Lieven Eeckhout. 2010. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. 362–370.

Stijn Eyerman and Lieven Eeckhout. 2014. The benefit of SMT in the multi-core era: Flexibility towards degrees of thread-level parallelism. *ACM SIGARCH Computer Architecture News* 42, 1, 591–606.

Chris Fallin, Chris Wilkerson, and Onur Mutlu. 2014. The heterogeneous block architecture. In *Proceedings of the International Conference on Computer Design (ICCD'14)*. 386–393.

Andrei Frumusanu and Ryan Smith. 2015. ARM A53/A57/T760 Investigated—Samsung Galaxy Note 4 Exynos Review. Retrieved December 29, 2015, from http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-rev iew/6.

Giorgis Georgakoudis, Dimitrios S. Nikolopoulos, and Spyros Lalis. 2013. Fast dynamic binary rewriting to support thread migration in shared-ISA asymmetric multicores. In *Proceedings of the International Workshop on Code Optimisation for Multi and Many Cores (COSMIC'13)*. 4:1–4:10.

Dan Gibson and David A. Wood. 2010. Forwardflow: A scalable core for power-constrained CMPs. *ACM SIGARCH Computer Architecture News* 38, 14–25.

Lori Gil. 2015. NVIDIAs Tegra X1 Crushes the Competition. Retrieved December 29, 2015, from http://liliputing.com/2015/02/nvidias-tegra-x1-crushes-the-competition.html.

Ryan E. Grant and Ahmad Afsahi. 2006. Power-performance efficiency of asymmetric multiprocessors for multi-threaded scientific applications. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'06)*.

Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. 2004. Best of both latency and throughput. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'04)*. 236–243.

Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. 2006. Synergistic processing in Cell's multicore architecture. *IEEE Micro* 26, 2, 10–24.

Divya P. Gulati, Changkyu Kim, Simha Sethumadhavan, Stephen W. Keckler, and Doug Burger. 2008. Multitasking workload scheduling on flexible-core chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 187–196.

Shantanu Gupta, Shuguang Feng, Amin Ansari, and Scott Mahlke. 2010. Erasing core boundaries for robust and configurable performance. In *Proceedings of the International Symposium on Microarchitecture (MICRO'10)*. 325–336.

Vishal Gupta and Ripal Nathuji. 2010. Analyzing performance asymmetric multicore processors for latency sensitive datacenter applications. In *Proceedings of the Workshop on Power Aware Computing and Systems (HotPower'10)*. 1–8.

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 of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'14)*. 191–198.

Mark D. Hill and Michael R. Marty. 2008. Amdahl's law in the multicore era. *IEEE Computer* 7, 33–38.

Houman Homayoun, Vasileios Kontorinis, Amirali Shayan, Ta-Wei Lin, and Dean M. Tullsen. 2012. Dynamically heterogeneous cores through 3D resource pooling. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'12)*. 1–12.

Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. 2013. When slower is faster: On heterogeneous multicores for reliable systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*. 255–266.

Ineda. 2015. Ineda Dhanush Wearable Processing Unit.

Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. 2007. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA'07)*. 186–197.

Brian Jeff. 2012. Big.LITTLE system architecture from ARM: Saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the ACM Design Automation Conference (DAC'12)*.

José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 223–234.

José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2013. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*. 154–165.

B. H. H. Juurlink and C. H. Meenderinck. 2012. Amdahl's law for predicting the future of multicores considered harmful. *ACM SIGARCH Computer Architecture News* 40, 2, 1–9.

Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. 2010. AASH: An asymmetry-aware scheduler for hypervisors. *ACM SIGPLAN Notices* 45, 7, 85–96.

Omer Khan and Sandip Kundu. 2010. A self-adaptive scheduler for asymmetric multi-cores. In *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI'10)*. 397–400.

Khubaib Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. 2012. MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Proceedings of the International Symposium on Microarchitecture (MICRO'12)*. 305–316.

Changkyu Kim, Simha Sethumadhavan, Madhu S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. 2007. Composable lightweight processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO'07)*. 381–394.

Jun Kim, Joonwon Lee, and Jinkyu Jeong. 2015. Exploiting asymmetric CPU performance for fast startup of subsystem in mobile smart devices. *IEEE Transactions on Consumer Electronics* 61, 1, 103–111.

Myungsun Kim, Kibeom Kim, James R. Geraci, and Seongsoo Hong. 2014. Utilization-aware load balancing for the energy efficient operation of the big.LITTLE processor. In *Proceedings of the Conference on Design, Automation, and Test in Europe (DATE'14)*. 223:1–223:4.

Byeong-Moon Ko, Joonwon Lee, and Heeseung Jo. 2012. AMP aware core allocation scheme for mobile devices. In *Proceedings of the IEEE Spring Congress on Engineering and Technology (S-CET'12)*. 1–4.

David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the European Conference on Computer Systems (EuroSys'10)*. 125–138.

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 *Proceedings of the International Symposium on Microarchitecture (MICRO'03)*. 81–92.

Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. 2004a. Conjoined-core chip multiprocessing. In *Proceedings of the International Symposium on Microarchitecture (MICRO'04)*. 195–206.

Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. 23–32.

Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004b. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *ACM SIGARCH Computer Architecture News* 32, 64.

Youngjin Kwon, Changdae Kim, Seungryoul Maeng, and Jaehyuk Huh. 2011. Virtualizing performance asymmetric multi-core systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA'11)*. 45–56.

Nagesh B. Lakshminarayana and Hyesoon Kim. 2008. Understanding performance, power and energy behavior in asymmetric multiprocessors. In *Proceedings of the International Conference on Computer Design (ICCD'08)*. 471–477.

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 (SC'09)*. 25:1–25:12.

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 ACM/IEEE Conference on Supercomputing (SC'07)*. 53:1–53:11.

Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'10)*. 1–12.

Felix Xiaozhu Lin, Zhen Wang, Robert LiKamWa, and Lin Zhong. 2012. Reflex: Using low-power processors in smartphones without knowing them. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 13–24.

Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. 2014. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 285–300.

Guangshuo Liu, Jinpyo Park, and Diana Marculescu. 2013. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *Proceedings of the International Conference on Computer Design (ICCD'13)*. 54–61.

Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski Jr., Thomas F. Wenisch, and Scott Mahlke. 2014. Heterogeneous microarchitectures trump voltage scaling for low-power cores. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT'14)*. 237–250.

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 International Symposium on Microarchitecture (MICRO'12)*. 317–328.

Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, and Antonia Zhai. 2010. Energy efficient speculative threads: Dynamic thread allocation in same-ISA heterogeneous multicore systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation (PACT'10)*. 453–464.

Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'15)*. 388–400.

Felipe Lopes Madruga, Henrique C. Freitas, and Philippe Olivier Alexandre Navaux. 2010. Parallel shared-memory workloads performance on asymmetric multi-core architectures. In *Proceedings of the Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'10)*. 163–169.

N. Markovic, D. Nemirovsky, O. Unsal, M. Valero, and A. Cristal. 2014. Thread lock section-aware scheduling on asymmetric single-ISA multi-core. *IEEE Computer Architecture Letters* 14, 2, 160–163. DOI:http://dx.doi.org/10.1109/LCA.2014.2357805

Sparsh Mittal. 2014a. A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Computer Aided Engineering and Technology* 6, 4, 440–459.

Sparsh Mittal. 2014b. *Power Management Techniques for Data Centers: A Survey*. Technical Report ORNL/TM-2014/381. Oak Ridge National Laboratory, Oak Ridge, TN.

Sparsh Mittal, Matthew Poremba, Jeffrey Vetter, and Yuan Xie. 2014. *Exploring Design Space of 3D NVM and eDRAM Caches Using DESTINY Tool*. Technical Report ORNL/TM-2014/636. Oak Ridge National Laboratory, Oak Ridge, TN.

Sparsh Mittal and Jeffrey Vetter. 2015. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys* 47, 4, 69:1–69:35.

Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. 2008. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro* 28, 3, 26–41.

Tomer Y. Morad, Avinoam Kolodny, and Uri C. Weiser. 2010. Scheduling multiple multithreaded applications on asymmetric and symmetric chip multiprocessors. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Programming (PAAP'10)*. 65–72.

Tomer Y. Morad, Uri C. Weiser, Avinoam Kolodny, Mateo Valero, and Eduard Ayguade. 2006. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters* 5, 1, 14–17.

Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Alfons Kemper, and Thomas Neumann. 2014. Heterogeneity-conscious parallel query execution: Getting a better mileage while driving faster! In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN'14)*. 2:1–2:10.

Janani Mukundan, Saugata Ghose, Robert Karmazin, Engin Ipek, and José F. Martínez. 2012. Overcoming single-thread performance hurdles in the core fusion reconfigurable multicore architecture. In *Proceedings of the International Conference on Supercomputing (ICS'12)*. 101–110.

Thannirmalai Somu Muthukaruppan, Anuj Pathania, and Tulika Mitra. 2014. Price theory based power management for heterogeneous multi-cores. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. 161–176.

Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. 2013. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proceedings of the Design Automation Conference (DAC'13)*. 174.

Hashem Hashemi Najaf-Abadi, Niket Kumar Choudhary, and Eric Rotenberg. 2009. Core-selectability in chip multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. 113–122.

Hashem H. Najaf-Abadi and Eric Rotenberg. 2009. Architectural contesting. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'09)*. 189–200.

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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 133–144.

Rajiv Nishtala, Daniel Mossé, and Vinicius Petrucci. 2013. Energy-aware thread co-location in heterogeneous multicore processors. In *Proceedings of the International Conference on Embedded Software (EMSOFT'13)*. 1–9.

NVIDIA. 2011. Variable SMP—A Multi-Core CPU Architecture for Low Power and High Performance. Retrieved December 29, 2015, from http://www.nvidia.com/content/PDF/tegra_white_papers/tegra-whitepaper-0 911b.pdf.

Shruti Padmanabha, Andrew Lukefahr, Reetuparna Das, and Scott Mahlke. 2013. Trace based phase prediction for tightly-coupled heterogeneous cores. In *Proceedings of the International Symposium on Microarchitecture*. 445–456.

Sankaralingam Panneerselvam and Michael M. Swift. 2012. Chameleon: Operating system support for dynamic processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 99–110.

George Patsilaras, Niket K. Choudhary, and James Tuck. 2012. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Transactions on Architecture and Code Optimization* 8, 4, 28:1–28:21.

Miquel Pericas, Adrian Cristal, Francisco J. Cazorla, Ruben Gonzalez, Daniel A. Jimenez, and Mateo Valero. 2007. A flexible heterogeneous multi-core architecture. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. 13–24.

Vinicius Petrucci, Orlando Loques, and Daniel Mossé. 2012. Lucky scheduling for energy-efficient heterogeneous multi-core systems. In *Proceedings of the USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*.

Dmitry Ponomarev, Gurhan Kucuk, and Kanad Ghose. 2001. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of the International Symposium on Microarchitecture*. 90–101.

Mihai Pricopi and Tulika Mitra. 2012. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Transactions on Architecture and Code Optimization* 8, 4, 22:1–22:21.

Mihai Pricopi and Tulika Mitra. 2014. Task scheduling on adaptive multi-core. *IEEE Transactions on Computers* 63, 10, 2590–2603.

Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. 2013. Power-performance modeling on asymmetric multi-cores. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'13)*. 1–10.

Moo-Ryong Ra, Bodhi Priyantha, Aman Kansal, and Jie Liu. 2012. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proceedings of the ACM Conference on Ubiquitous Computing (Ubicomp'12)*. 1–10.

M. Mustafa Rafique, Benjamin Rose, Ali R. Butt, and Dimitrios S. Nikolopoulos. 2009. Supporting MapReduce on large-scale asymmetric multi-core clusters. *ACM SIGOPS Operating Systems Review* 43, 2, 25–34.

Behnam Robatmili, Dong Li, Hadi Esmaeilzadeh, Sibi Govindan, Aaron Smith, Andrew Putnam, Doug Burger, and Stephen W. Keckler. 2013. How to implement effective prediction and forwarding for fusable dynamic multicore architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'13)*. 460–471.

Rance Rodrigues, Arunachalam Annamalai, Israel Koren, Sandip Kundu, and Omer Khan. 2011. Performance per watt benefits of dynamic core morphing in asymmetric multicores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 121–130.

Rance Rodrigues, Israel Koren, and Sandip Kundu. 2014. Performance and power benefits of sharing execution units between a high performance core and a low power core. In *Proceedings of the International Conference on VLSI Design (VLSID'14)*. 204–209.

Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. 2012. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Transactions on Computer Systems* 30, 2, 6:1–6:38.

Juan Carlos Saez, Alexandra Fedorova, Manuel Prieto, and Hugo Vegas. 2010. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the Computing Frontiers Conference (CF'10)*. 31–40.

Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. 2015. ACFS: A completely fair scheduler for asymmetric single-ISA multicore systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC'15)*.

Pierre Salverda and Craig Zilles. 2008. Fundamental performance constraints in horizontal fusion of in-order cores. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'08)*. 252–263.

Samsung. 2013. SAMSUNG Highlights Innovations in Mobile Experiences Driven by Components, in CES Keynote. Retrieved December 29, 2015, from http://www.samsung.com/us/news/20353.

Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA'03)*. 422–433.

Lina Sawalha and Ronald D. Barnes. 2012. Energy-efficient phase-aware scheduling for heterogeneous multicore processors. In *Proceedings of the IEEE Green Technologies Conference*. 1–6.

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, 66–75.

Tyler Sondag and Hridesh Rajan. 2009. Phase-guided thread-to-core assignment for improved utilization of performance-asymmetric multi-core processors. In *Proceedings of the ICSE Workshop on Multicore Software Engineering*. 73–80.

Sudarshan Srinivasan, Nithesh Kurella, Israel Koren, and Sandip Kundu. 2015. Exploring heterogeneity within a core for improved power efficiency. *IEEE Transactions on Parallel and Distributed Systems* PP, 99, 1.

Sudarshan Srinivasan, Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. 2013. A study on polymorphing superscalar processor dynamically to improve power efficiency. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI'13)*. 46–51.

Sadagopan Srinivasan, Li Zhao, Ramesh Illikkal, and Ravishankar Iyer. 2011. Efficient interaction between OS and architecture in heterogeneous platforms. *ACM SIGOPS Operating Systems Review* 45, 1, 62–72.

Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen. 2009. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review* 43, 2, 35–45.

M. Aater Suleman, Onur Mutlu, José A. Joao, Khubaib, and Yale Patt. 2010. Data marshaling for multi-core architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*. 441–450.

M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. 253–264.

M. Aater Suleman, Yale N. Patt, Eric Sprangle, Anwar Rohillah, Anwar Ghuloum, and Doug Carmean. 2007. *Asymmetric Chip Multiprocessors: Balancing Hardware Efficiency and Programmer Efficiency*. TR-HPS-2007-001. University of Texas, Austin, TX.

Hsin-Ching Sun, Bor-Yeh Shen, Wuu Yang, and Jenq-Kuen Lee. 2011. Migrating Java threads with fuzzy control on asymmetric multicore systems for better energy delay product. In *Proceedings of the International Conference on Computing and Security*.

Tao Sun, Hong An, Tao Wang, Haibo Zhang, and Xiufeng Sui. 2012. CRQ-based fair scheduling on composable multicore architectures. In *Proceedings of the International Conference on Supercomputing (ICS'12)*. 173–184.

Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel. 2011. Efficient virtual machine scheduling-policy for virtualized heterogeneous multicore systems. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)*.

David Tarjan, Michael Boyer, and Kevin Skadron. 2008. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the Design Automation Conference (DAC'08)*. 772–775.

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 International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*. 177–187.

Kenzo Van Craeynest and Lieven Eeckhout. 2013. Understanding fundamental design choices in single-ISA heterogeneous multicore architectures. *ACM Transactions on Architecture and Code Optimization* 9, 4, 32.

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 International Symposium on Computer Architecture (ISCA'12)*. 213–224.

Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*. 121–132.

Jeffrey Vetter and Sparsh Mittal. 2015. Opportunities for nonvolatile memory systems in extreme-scale high performance computing. *Computing in Science and Engineering* 17, 2, 73–82.

Carl A. Waldspurger and William E. Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'94)*.

Yasuko Watanabe, John D. Davis, and David A. Wood. 2010. WiDGET: Wisconsin decoupled grid execution tiles. In *Proceedings of the International Symposium on Computer Architecture (ISCA'10)*, Vol. 38. 2–13.

Ryan Whitwam. 2014. Qualcomm Unveils 64-Bit Snapdragon 808 and 810 SoCs: The Apple A7 Stop-Gap Measures Continue. Retrieved December 29, 2015, from http://goo.gl/v4ywMW.

Youfeng Wu, Shiliang Hu, Edson Borin, and Cheng Wang. 2011. A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'11)*. 236–245.

Ying Zhang, Lide Duan, Bin Li, Lu Peng, and Srinivasan Sadagopan. 2014a. Energy efficient job scheduling in single-ISA heterogeneous chip-multiprocessors. In *Proceedings of the International Symposium on Quality Electronic Design (ISQED'14)*. 660–666.

Ying Zhang, Li Zhao, Ramesh Illikkal, Ravi Iyer, Andrew Herdrich, and Lu Peng. 2014b. QoS management on heterogeneous architecture for parallel applications. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'14)*. 332–339.

Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. 2007. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'07)*. 25–36.

Yuhao Zhu and Vijay Janapa Reddi. 2013. High-performance and energy-efficient mobile web browsing on big/little systems. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA'13)*. 13–24.

Received April 2015; revised August 2015; accepted November 2015