Amphisbaena: Modeling Two Orthogonal Ways to Hunt on Heterogeneous Many-cores

Jun Ma†‡, Guihai Yan†, Yinhe Han† and Xiaowei Li†
†Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China
‡School of Computer and Control Engineering
University of Chinese Academy of Sciences, Beijing, China
{majun, yan_guanghai, yinhes, lxw}@ict.ac.cn

Abstract—Heterogeneous many-cores can deliver high performance or energy efficiency. There are two orthogonal ways to improve performance: 1) scale-out by exploiting thread-level parallelism, and 2) scale-up by enabling core heterogeneity. Predicting the performance of such architecture is increasingly challenging. We propose a comprehensive performance model Amphisbaena, or Φ, built from two orthogonal functions α and β. Function α describes the scale-out speedup and function β handles the scale-up speedup. The Φ model can clearly tell not only the overall speedup of a given multithreading and core mapping strategy, but also how to improve the multithreading and core mapping, hence should be a promising performance predictor for future heterogeneous many-cores. The results show that Φ model’s error rate is within 12%, which is lower than state-of-the-art methods. We demonstrate the application of Φ model by introducing a heuristic scheduling algorithm, which outperforms the baselines by 13% on average.

I. INTRODUCTION

Heterogeneous many-cores is a promising architecture to cater for increasing demand of computing. The successful stories have been witnessed in data centers and cloud computing [1][2][3]. Such architecture provides two ways to boost performance and energy efficiency: 1) highly thread-level parallelism supported by large core quantity, and 2) appropriate application-core mapping enabled by diversified core heterogeneity. Although such architecture promises attractive potential of performance/energy-efficiency, it is not easy to fully make the potential into reality. The main obstacle is the lack of high accurate and low complex performance model, which usually leads to sub-optimal runtime management.

To be an effective runtime management, one has to accurately predict and optimize the performance periodically. For heterogeneous many-cores, the Pareto Frontier of optimal performance depends on not only the number of active cores, but also the types of cores activated. However, given multithreaded applications, there are still no quick answers to the both questions. A large body of prior work only focus on homogenous architectures [4][5]. For example, CPR [6] models the performance of multicore processor, but it is unable to provide hints for heterogeneous cores. FDT [7] and TR [8] investigate the upper bound of thread number by detecting the synchronization primitives and hardware contentions. However, without considering core heterogeneity, their methods cannot apply to the heterogeneous many-cores.

In terms of heterogeneous architectures, many prior researches [9][10][11] only rely on memory related statistics to guide scheduling, and totally ignore the thread-level parallelism. The recently proposed PIE [12] model aiming to predict performance change between a pair of big and small cores. It claims that performance is determined by ILP and MLP; however, this model is still incomprehensive due to two reasons: 1) PIE cannot capture the performance impact from multithreading; 2) PIE is derived from only two type of cores and hence is unable to handle more complex heterogeneous many-cores with more than two types of cores.

Based on the above analysis, we conclude that prior work can not model the heterogeneous many-cores well. In this paper, we propose a novel, comprehensive analytical model, called Amphisbaena†, abbreviated as Φ in this paper. Φ model can simultaneously describe the performance speedup coming from two orthogonal ways: 1) large-scale multithreading, or simply called “scale-out” speedup, and 2) core-heterogeneity, or “scale-up” speedup. The rationale behind the above two ways is that an application’s performance is largely subject to either the number of threads spawned, or the heterogeneous cores’ capabilities for executing those threads. Unlike prior performance models, Φ is deliberately designed from two orthogonal functions, α and β, to model the performance of two independent ways. By doing so, Φ model not only gives an overall performance speedup, more importantly, clearly indicates where the speedup comes from: scale-out or scale-up, or both of them.

In the runtime management, scale-out speedup indicates how many threads of an application should be spawned, and scale-up speedup indicates which type of cores should be activated. We will demonstrate that Φ can serve as the online performance predictor for different runtime scheduling. Experimental results show that the average error of Φ is within 12%, which is less than that of three state-of-the-art baselines. Moreover, we propose a dynamic scheduling to demonstrate the application of Φ model; the performance can outperform the baselines by 13% on average.

The rest of this paper is organized as follows: Section II illustrates our motivation. Section III presents the Φ model. Section IV proposes the runtime management based on Φ. Section V validates the Φ model and the scheduling algorithm. Section VI describes the related work. Section VII concludes this paper.

†Amphisbaena is a mythological creature from Greek Mythology. It has a twin at the head end and one at the tail end, which metaphorically coincides with the two orthogonal ways in this paper.
The orthogonality between current state is 8-thread configuration on core type A (issue four representative PARSEC benchmarks [13]. Supposing the results in Figure 1 are called \( \Phi \) size: 96) and C (issue width: 8; ROB size: 128), respectively. of 16 and 32 threads on the core type B (issue width: 6; ROB

de % includes the performance of target multithreading configuration to the current configuration on the same type of cores. Scale-up speedup, modeled by \( \beta \), is determined by which type of cores is allocated to the application’s threads. We defined \( \beta \) as the ratio of performance on target cores to current cores under the same multithreading configuration.

Figure 1 illustrates the \( \alpha \) (X axis) and the \( \beta \) (Y axis) with four representative PARSEC benchmarks [13]. Supposing the current state is 8-thread configuration on core type A (issue width: 4; ROB size: 64), we then study \( \alpha \) and \( \beta \) under target of 16 and 32 threads on the core type B (issue width: 6; ROB size: 96) and C (issue width: 8; ROB size: 128), respectively. The detailed core configurations are listed in Table I. The results in Figure 1 are called \( \Phi \) space, which can profile how the performance responds to the scale-out and scale-up strategy respectively. Taking the fluidanimate for example, when the target state transits from current 8-thread with core A to target state of 16-thread with core B, the scale-out approach boosts performance by \( 2 \times \) while the scale-up speedup contributes to another \( 1.4 \times \). Its overall performance speedup then can be estimated as \( 2.8 \times \) (i.e. \( 2 \times 1.4 \)).

\( \Phi \) space can clearly indicate how to improve the overall performance of each application. First, spawning more threads not necessarily yields higher scale-out speedup, which is largely in line with the Amdahl’s Law [14]. For example, multithreading from 16-thread to 32-thread brings no speedup for ferret, fregmine, and even hurts the performance of streamcluster, hence should be avoided. Second, choosing more powerful core usually brings scale-up speedup, but reveals different application-specific extents because of the different boundness to computing or memory intensity [12].

III. \( \Phi \) Model

According to the definition of \( \alpha \) and \( \beta \), we can see that \( \alpha \) and \( \beta \) are orthogonal to each other, which makes the overall performance modeling a trivial issue: the resultant performance speedup, denoted by \( \Phi = \alpha \times \beta \), can be modeled by the product of \( \alpha \) and \( \beta \). \( \beta \) is the orthogonality between \( \alpha \) and \( \beta \) is confirmed by our experimental results shown in Section V.B.

A. Scale-out speedup: \( \alpha \)

We use execution time as the performance metric in \( \alpha \) modeling. According to the definition,

\[
\alpha = \frac{T_{current}}{T_{target}},
\]

where, \( T \) is the execution time on current and target threading configuration, respectively. Generally,

\[
T = T_{serial} + T_{parallel} + T_{penalty},
\]

where \( T_{serial} \) is the serial part, which determines the upper bound of scale-out speedup[15], \( T_{parallel} \) is the parallelizable part, which can be eaten away by spawning more threads. For online use, \( T_{serial} \) and \( T_{parallel} \) can be accurately obtained by training method [7], or instrumentation technique [16]. For example, we can insert specific instructions at the entry and exit of serial and parallel sections to record the elapsed cycles. As the distinctive part of our \( \alpha \) model, the multithreading overhead \( T_{penalty} \) is indeed the bottleneck of scale-out speedup and determined by both 1) software, such as inter-thread contention due to synchronization with locks and barriers, and 2) hardware, such as contention for shared resources of LLC, memory controller, and memory bus. Unfortunately, accurately modeling \( T_{penalty} \) is still an open question.

We tackle this problem by using the “univariate analysis”. First, we assume a contention-free scenario by configuring hardware resources over-provisioning in our simulator, then study the performance impact only from software synchronization which results in thread waiting. Figure 2(a) shows the relationship between thread number and time penalty. We choose applications with the most amount of software synchronization (locks and barriers). No matter what type of synchronization happens, the penalty can be tracked and returned by bottleneck-identifying instructions (BottleneckCall, BottleneckReturn and BottleneckWait) which have been used in [16]. Basically, the penalty correlates linearly with the number of threads, even though with different slope due to different intensity of locks and barriers. The more synchronization operations, the steeper slope. For example, both contention-intensive fregmine and fluidanimate exhibit steeper slope. Hence, we reckon that \( T_{penalty} \propto \prod \times n \times m \) is the number of thread. \( k_l \) represents software contention-intensity measured by synchronization waiting cycles per kilo-instructions (SPKI), which is accumulated by recording the penalties deriving from both locks and barriers. \( a_1 \) is a modulating constant.

Second, we tighten hardware resource to match the reality and log the stall cycles due to hardware contentions. Because we cannot hide the synchronization primitives of multi-threaded applications, we just regard the incremental penalty of hardware resource-unlimited experiment as the hardware contention-induced penalty. Unlike software contention, we choose applications with the most intensive hardware contention (with largest working set). We found that the hardware

---

**Fig. 1.** \( \Phi \) space: scale-out speedup (\( \alpha \)) and scale-up speedup (\( \beta \))

**Fig. 2.** The time penalty of software synchronizations (a) and hardware contentions (b)
contention-induced penalty increases much faster, roughly following a quadratic trend, as Figure 2(b) shows. Hence, we infer that $T_{penalty} \propto a_2 \times k_2 \times n^2$, where $k_2$ represents the contention-intensity measured by misses waiting cycles per kilo-instructions (MPKI). Since there are always misses happened before shared resource being accessed, MPKI just accounts for the hardware contention-intensity. MPKI is obtained by aggregating the penalty of to-LLC and to-MEM misses. Performance counters are needed to detect miss events at runtime. $a_2$ is also a modulating constant.

The overall penalty can be obtained by combining the above two components, i.e.

$$T_{penalty} = a_0 \times k_0 + a_1 \times k_1 \times n + a_2 \times k_2 \times n^2,$$

(3)

where $k_0$ is an application-specific bias derived from redundant computations [13], which is obtained on the fly and modulated by $a_0$.

B. Scale-up speedup: $\beta$

We use CPI as the performance metric in $\beta$ modeling. Based on the definition,

$$\beta = \frac{CPI_{current}}{CPI_{target}},$$

(4)

so given a thread, we need to accurately predict its CPI on various type of cores. Kenzo et al. correlated the CPI with two representative microarchitecture parameters: the frontend issue width ($W$) and backend ROB size ($R$) [12]. We also rely on the two parameters to differentiate heterogeneous cores, but build a more essential analytical function with them. We find that scale-up by core heterogeneity can be approximated with a power-exponent law, which is reasoned from the following observation. We first hold $W$ and study the sensitivity of CPI to $R$. As Figure 3(a) shows, the CPI improvement gets less significant with larger ROB capacity. Specifically, CPI initially reduces dramatically when ROB size increases, as Figure 3(b) shows. The CPI also decreases along with the issue width growing.

$$CPI = r + s \times R^{-t \times W}.$$  

(5)

Equation 5 is in accordance with prior knowledge: the marginal performance of scale-up speedup will decrease (lower slope) when running on more aggressive cores, because excessively large machine parallelism (larger issue width, ROB size, etc.) leads to mismatch with the available program parallelism (inherent instruction parallelism and concurrent memory access) [17]. Also, we find that the performance sensitivity to $W$ and $R$ shows very subtle difference. For instance, we find that when frontend capacity (represented by $W$) is too small, saying 1 or 2, then expanding backend will be ineffective. That’s because the pipeline becomes unbalance if the instructions issued in frontend underfeed the backend. So we put the $W$ in the exponent function in Equation 5, which can show higher gradient than power function in most cases, for modeling the frontend impact to performance.

We find that $s$ and $t$ are also two application-specific constants, and are highly correlated with memory intensity and computing intensity respectively. For example, Figure 3(a) shows that, when comparing to fluidanimate, the CPI of vips is more sensitive to $R$. That’s because vips is more memory intensive than fluidanimate. In Figure 3(b), swaptions, a financial analysis benchmark with more computing intensity, is more sensitive to $W$. Therefore, $s$ and $t$ can weight these kinds of sensitivities. Moreover, bodytrack, with approximate ratio of computing and memory intensity to fluidanimate, has a steady lower bias, which can be expressed by bias $r$.

These coefficients can be calculated as follows: given the $CPI$ comes from two components: $CPI_{base}$, the base CPI without any stalls from memory access, and $CPI_{mem}$, the penalty of data waiting or pipeline stalls [18]. Let $\eta = \frac{CPI_{mem}}{CPI_{base}}$, then $s$ can be calculated by $s = b_1 \times \eta$, which highlights the memory intensity. It implies that the memory intensity linearly correlates with the ultimate CPI, no matter which type of cores is used. $t = b_2 \times (1-\eta)$, which negatively weights the computing intensity on the exponent; this implies that the CPI of a computing intensive application can be reduced more by powerful frontend. The bias $r$ reflects the intrinsic CPI of an application on the assumed oracle core with infinite $R$ and $W$ [17][19], which can be approximated by $CPI_{base}$ in this paper. So $r$ is calculated by $b_0 \times CPI_{base}$. Also, $b_0$, $b_1$ and $b_2$ are modulating constants.

The $\alpha$ and $\beta$ model can be performed online, so does the $\Phi$ model. Based on any current phase of an application, we can use $\Phi$ model to predict its speedup of next phase on any other target configurations. The detailed model parameters are listed in Table II. In the next section, we will demonstrate the role of $\Phi$ model in runtime management.

IV. Runtime Management

A generic runtime management can be divided into three steps: 1) Predict performance speedup coming from scale-out and scale-up at the beginning of each management interval. Then the predicted values serve as the input of the next step. 2) Invoke scheduling algorithm to figure out the optimal configuration in terms of, for example, maximizing performance. 3) The operating system enables the specified multithreading and application-core mapping indicated by the optimal configuration. Our $\Phi$ model takes charge of the first step.

First, we describe the implementation issue of $\Phi$ model. Those application-independent modulating constants such as $a_0$, $a_1$, $a_2$, $b_0$, $b_1$ and $b_2$, can be obtained by offline regression. The other application-specific coefficients ($k_1$, $k_2$, $s$, $t$ etc.) can be calculated online, by following the detailed calculation routines discussed in Section III.A and III.B respectively. Table II summaries their values and implementations. Note that SPKI and MPKI are calculated based on the bottleneck-identifying instructions [16] and the built-in performance counters [18], respectively. $CPI_{base}$ and $CPI_{mem}$ are derived from the CPI stack calculation [18]. The same method has been also used by PIE [12]. The bias of redundant computing derives from the cycles of instruction duplication, and the bias of intrinsic CPI is replaced by $CPI_{base}$.

To demonstrate the application of $\Phi$ model, we have to feed its output to a management algorithm, as the second step.
Algorithm 1 How to scale-out \(D_{out}\) and scale-up \(D_{up}\)

1: Model \(\alpha\) and \(\beta\) for each application;
2: Calculate expectations \(E[\alpha]\) and \(E[\beta]\);
3: Allocate thread number \(n_i = \frac{E[\alpha_i]}{\sum_{i=1}^{N} E[\alpha_i]} \times N\);
4: Sort array \(E[\beta_1 ... \beta_M]\) and set priority;
5: repeat
6: \(E_{max}[\beta]\) chooses the fastest cores;
7: Pop \(E_{max}[\beta]\) from array \(E[\beta_1 ... \beta_M]\);
8: until array \(E[\beta_1 ... \beta_M] = \emptyset\);

where \(n_i\) is the thread number allocated for application \(i\), \(N\) is the number of all available heterogeneous cores, \(E[\alpha_i]\) is the scale-out expectation of application \(i\). It accounts for the average level of scale-out speedup based on Equation 7.

\[
E[\alpha_i] = \frac{1}{\left|U\right|} \sum_{j \in U} \alpha_{i,j},
\]

where \(\alpha_{i,j}\) is the scale-out speedup of application \(i\) at multithreading configuration \(j\). \(U\) is the set of all possible number of threads for scale-out. The size of \(U\) \((\left|U\right|\)) is 33 in this paper \((\{1\} \cup \{2 : 2 : 64\})\).

After \(D_{out}\), \(D_{up}\) has to allocate available heterogeneous cores to the threads of each application. \(D_{up}\) follows the policy that application with largest \(E[\beta]\) (calculated similar to \(E[\alpha]\)) will be allocated the fastest type of cores. So we rank the \(E[\beta]\) of all applications in descending order, then follow the iterations: the application with maximal \(E[\beta]\) always gets highest priority to choose the fastest cores to its threads, until all cores have been allocated.

The runtime interval is on order of sub-seconds. It depends on not only the penalty of runtime management, but also the effectiveness of management affected by interval granularity [21]. In this paper, the runtime interval is one second. The first two steps in charged by OS hypervisor can be finished within 10 ms. OS triggers OpenMP for changing thread number dynamically, and operates thread migration for application-core mapping. The former is on order of 10ms [22] and the latter costs about from 1ms to 20ms [12]. Therefore, the overhead of runtime management is no more than 50ms, which can be amortized well at runtime.

V. EVALUATION AND RESULTS ANALYSIS

A. Experimental Setup

We evaluate our work with gem5 simulator [23]. Similar to PIE [12], we choose heterogeneous cores which are obvious different in frontend issue width \((W)\) and backend ROB capacity \((R)\). We also implement three types of in-order cores to bring more types of heterogeneous cores. The detailed configurations are listed in Table I. The cores are organized into clusters, and each cluster consists of 32 homogeneous cores. This layout method is similar to the baseline used in [3]. A distributed, cluster-level banked LLC is shared by all cores. Cache coherence is maintained by directory-based

---

**Table I. Microarchitectural Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>In-order</th>
<th>OoO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4(A), 6(B), 8(C)</td>
<td>—</td>
</tr>
<tr>
<td>ROB size</td>
<td>64KB, 2-way 64, 96, 128</td>
<td>32KB, DM</td>
</tr>
<tr>
<td>I/D cache</td>
<td>2ns/4ns</td>
<td>2ns/4ns</td>
</tr>
<tr>
<td>1/1D-1.1 access time</td>
<td>2ms/4ms</td>
<td>2ms/4ms</td>
</tr>
<tr>
<td>LLC access/miss time</td>
<td>12ns/54ns</td>
<td>12ns/54ns</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>hybrid 2-level</td>
<td>static, 2k</td>
</tr>
</tbody>
</table>

**Table II. Coefficient Details**

<table>
<thead>
<tr>
<th>Coefficients</th>
<th>Value</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_0)</td>
<td>1.837e-03</td>
<td>constant</td>
</tr>
<tr>
<td>(a_1)</td>
<td>0.0512</td>
<td>constant</td>
</tr>
<tr>
<td>(a_2)</td>
<td>2.025e-05</td>
<td>constant</td>
</tr>
<tr>
<td>(k_0)</td>
<td>BiasRedundantComputing</td>
<td>online</td>
</tr>
<tr>
<td>(k_1)</td>
<td>SPKI</td>
<td>online</td>
</tr>
<tr>
<td>(k_2)</td>
<td>MPKI</td>
<td>online</td>
</tr>
<tr>
<td>(b_0)</td>
<td>0.2837</td>
<td>constant</td>
</tr>
<tr>
<td>(b_1)</td>
<td>1.1675</td>
<td>constant</td>
</tr>
<tr>
<td>(b_2)</td>
<td>1.8427</td>
<td>constant</td>
</tr>
<tr>
<td>(r)</td>
<td>BiasIntrinsicCPD</td>
<td>online</td>
</tr>
<tr>
<td>(s)</td>
<td>CPImem/CPI</td>
<td>online</td>
</tr>
<tr>
<td>(t)</td>
<td>CPIbase/CPI</td>
<td>online</td>
</tr>
</tbody>
</table>

---

MOEL protocol which supports heterogeneous transactions [24]. We use PARSEC [13] as the basic workloads, given it targets the general purpose processors and shows diverse preferences to heterogeneous cores.

B. Model Accuracy Validation

The accuracy of \(\Phi\) model is determined by 1) the accuracy of \(\alpha\) and \(\beta\), and 2) the orthogonality between them. The following results are devoted to validate the accuracy and orthogonality, respectively.

Figure 4(a) shows the discrepancy between the predicted scale-out speedup calculated by \(\alpha\) model and the actual scale-out speedup by measurement. For each benchmark, there are 33 thread configurations (the threading is from 1, 2 to 64 in step of 2 threads, so the total space is \(A^6_{2:64} = 1056\)). We randomly generate 600 validation points from the 12 benchmarks (50 phases per benchmark) to make sure the time-consuming simulation time acceptable. The result shows that the predicted \(\alpha\) is fully competent to predict the scale-out speedup. The prediction error is under 5% on average, and the maximum error is no more than 8%. This merit mainly comes from the accurate model of penalty component \((\epsilon_{\text{penalty}}\) in Equation 3).

As a comparison, we plot the predicted scale-out speedup according to the Amadahl’s Law, which shows that the average error and maximum error reach up to 11.4% and 19.5%, respectively. Therefore, we believe the \(\alpha\) model can serve as a good amendment to the classical Amadahl’s Law.

Compared to the well-behavior \(\alpha\), predicting scale-up speedup \(\beta\) model from core heterogeneity is much harder. As Figure 4(b) shows (similar to \(\alpha\), 600 validation points are in the total space: \(12 \times 50 \times A^6_{2:64} = 180000\), where \(A^6_{2:64}\) covers all cases of thread migrations between two types of cores), the worst case’s prediction error reaches up to 25%. However, the average error is still kept below 8%, which makes \(\beta\) model still applicable in most cases. For comparison, we also study the prediction accuracy of recently proposed PIE model by applying it to the same samples, as shown in Figure 4(b). The result shows that PIE’s average error and worst error are 12.2% and 33.7%, which agree with the prior work [12]. This comparison confirms that our \(\beta\) model is more accurate when predicting the performance impact from core heterogeneity.

The validation of \(\alpha\) and \(\beta\) builds up high confidence in the
proposed Φ model. Figure 4(c) illustrates the cumulative distribution of prediction error for Φ model. Up to 1080 samples are randomly picked out from a huge space (633600 x 18000). The results show that for most benchmarks, applying Φ model always results in less than 10% prediction error, or for more than 80% samples of facesim, fluidanimate, and 90% samples of blackscholes, x264. Although Φ has shown to be more comprehensive and accurate than many previous models, we admit that Φ is not perfect. In some corner cases, the prediction error is up to 30% and even higher, such as dedup, swaptions. That’s largely because we make the tradeoff between model’s complexity and accuracy. In particular, we deliberately assume that α and β are completely orthogonal to each other. We find that α and β are weakly correlated rather than totally orthogonal in most cases, because different number of threads will result in different average CPI of an application.

Figure 5 evaluates the orthogonality between α and β. In this experiment, we measure three actual values in each interval, i.e. α_m, β_m, and Φ_m. If α and β are orthogonal, then \(|α_m × β_m − Φ_m|\) should be zero. Hence, we use the normalized value \(Cor = |α_m × β_m - Φ_m|/Φ_m\) to evaluate the orthogonality. First, we run one thread on baseline core A, and then T threads on core A. By measuring the running time, we can obtain the scale-out speedup α_m. Then, we run one thread on core B and measure the running time to obtain the scale-up speedup β_m. Lastly, measuring the speedup of T threads on core B can obtain the overall speedup Φ_m. Following this measurement, we build 2268 samples from the 12 benchmarks. The results are shown in Figure 5 with boxplot which can show central mark for median, interquartile range and possible outliers. These results can justify our orthogonality assumption. For most benchmarks, the Cor values are below 5%. The worst case happens on swaptions. This outlier reflects that scale-out speedup and scale-up speedup are not completely independent, because the thread CPI, or β can slightly change with different threads spawned. However, the results in Figure 5 confirm that our orthogonality assumption is a good approximation. As future work, it suggests a potential improvement to the current Φ model by involving the correlation between α and β.

C. Performance Comparison

We evaluate Phi scheduling by comparisons to three state-of-the-art algorithms. Bias [9] is indicated by memory related stalls, then schedules application that shows less stalls on big core or more stalls on small core. PIE [12] uses two analytical models to predict the performance impact between a pair of big and small cores, then moves the application with highest performance improvement to the big core. However, both Bias and PIE ignore the scale-out speedup. For fair comparison, we let their decision for thread number equal to Phi scheduling. Inversely, Static strategy allocates the same number of threads to each application, but lacks the indicator for heterogeneous core mapping. Therefore, we make its mapping policy the same to Phi. As a golden case, we also present an Oracle algorithm by exhaustively searching the huge solution space, though it is infeasible for online use.

In Figure 6, six workloads are selected and each workload is composed by four PARSEC benchmarks, as shown in Table III. The results are all normalized to Oracle. We can see that our Phi scheduling is the closest to Oracle. This result again justifies the efficacy of Φ model. Particularly, Phi averagely outperforms the other three baselines as high as 12.2% (Static), 13.3% (Bias) and 12.9% (PIE), even though Bias and PIE have the same multithreading configuration to Phi, or Static has the same heterogeneous core mapping to Phi.

VI. RELATED WORK

The performance model of heterogeneous architectures has been an active research topic for decades. Those methods are used to either facilitate early design stage to prune the huge design space [3], or guide runtime management [11]. Predicting performance becomes even more challenging with the advent of heterogeneous many-cores where performance is determined by not only scale-out, but also scale-up, or even...
the complex interactions between them.

Predicting multithreading performance initially is accomplished with the classical Amdahl’s Law [14]; however its beauty form fails to capture the inter-thread contentions such as data synchronization and hardware contention. Feedback-driven threading (FDT) [7] is designed to decide the upper bound of thread number. However, because FDT assumes a homogenous substrate, it becomes ineffective when applying to heterogeneous architectures.

Although heterogeneous architectures draw a lot of attentions [25][10][9], the work for performance prediction is far from mature. The recently proposed PIE [12] uses two models to predict performance change deriving from ILP and MLP. However, its cost of online implementation is very high. For example, PIE requires on-the-fly tracking a key parameter, called dependency distance, which involves a register table to track all the stream of dynamic instructions. By contrast, our model only needs limited runtime statistics which can be obtained from already built-in counters, and the results show that our model is more accurate than PIE. Furthermore, without considering performance impact from multithreading, PIE can not cope with the scale-out speedup.

Hill et al. [15] propose a performance prediction model for heterogeneous many-cores using the Amdahl’s Law. That model considers both multithreading and core heterogeneity. However, it does not provide any analytical function or algorithm. Morad et al. [26] give a more comprehensive equation; however, it correlates the scale-up speedup to core’s area, rather than micro-architectures, which results in large prediction error. Finally, both methods didn’t clearly split scale-out speedup and scale-up speedup, so are hard to guide runtime management.

VII. Conclusion

In this paper, we propose a comprehensive analytical performance prediction model $\Phi$ for heterogeneous many-cores. $\Phi$ model can simultaneously capture the performance speedup from multithreading (scale-out) and core heterogeneity (scale-up), which are described with function $\alpha$ and $\beta$, respectively. Finally, we find that $\alpha$ and $\beta$ are largely orthogonal, which greatly simplify the $\Phi$ model. The average error of $\Phi$ model is within 12%. We also present Phi scheduling to demonstrate the application of $\Phi$ model. The proposed $\Phi$ model highlights the tradeoff between scale-out speedup and scale-up speedup, hence we believe that it can serve as an ideal performance predictor for future runtime management.

REFERENCES