Power-Performance Modeling on Asymmetric Multi-Cores

Mihai Pricopi[*], Thannirmalai Somu Muthukaruppan[*], Vanchinathan Venkataramani[*], Tulika Mitra[*], Sanjay Vishin[†]

[*]School of Computing, National University of Singapore
{mihai,tsomu,vvanchi,tulika}@comp.nus.edu.sg

[†]Cambridge Silicon Radio
Sanjay.Vishin@csr.com

ABSTRACT

Asymmetric multi-core architectures have recently emerged as a promising alternative in a power and thermal constrained environment. They typically integrate cores with different power and performance characteristics, which makes mapping of workloads to appropriate cores a challenging task. Limited number of performance counters and heterogeneous memory hierarchy increase the difficulty in predicting the performance and power consumption across cores in commercial asymmetric multi-core architectures. In this work, we propose a software-based modeling technique that can estimate performance and power consumption of workloads for different core types. We evaluate the accuracy of our technique on ARM big.LITTLE asymmetric multi-core platform.

1. INTRODUCTION

Asymmetric multi-cores, also known as single-ISA heterogeneous multi-cores [16], are emerging as a promising solution that can achieve power-performance trade-off essential in high performance, energy constrained embedded systems such as tablets, smartphones, automotive telematics, and others. Asymmetric multi-cores integrate high performance, power hungry complex cores ("big" cores) with moderate performance, power efficient simple cores ("small" cores) on the same chip. The characteristic that distinguishes asymmetric multi-cores from heterogenous multiprocessor system-on-chips (MPSoC) prevalent in embedded platforms is that both the core types implement the same instruction-set architecture (ISA); that is, the same binary executable can be scheduled to run on either the big or the small core. Examples of commercial asymmetric multi-cores include ARM big.LITTLE [5], integrating high performance out-of-order cores with low power in-order cores and NVidia Kal-El [4], consisting of four high performance cores with one low power core. An instance of the former integrating quad-core ARM Cortex-A15 (big core) and quad-core ARM Cortex-A7 (small core) appears in the Samsung Exynos 5 Octa SoC driving high-end Samsung Galaxy S4 smart-phones.

A fundamental challenge in exploiting asymmetric multi-cores for power-performance trade-off arises from scheduling the workload to the appropriate core type. Initial proposals [7, 15, 18] employed a simple strategy of scheduling memory-intensive workloads on the small core and compute-intensive workloads on the big core. Recently [22] has shown that this strategy may lead to sub-optimal mappings and it is imperative to accurately estimate the power-performance characteristics of a workload on different core types.

The Performance Impact Estimation (PIE) mechanism proposed in [22] is a dynamic technique that collects profile information while executing the application on any one core type, and estimates the performance on the other core type. This estimation allows the scheduler to make appropriate adjustments to the application-core mapping at runtime. However, the PIE mechanism [22] has few shortcomings that renders it difficult, if not impossible, to be deployed on real hardware. First, the estimation is based on a number of simplifying assumptions such as the presence of identical cache hierarchy and branch prediction on both core types, which are unrealistic for commercial asymmetric multi-cores. Second, the PIE mechanism requires profile information, such as the inter-instruction dependency distance distribution, that cannot be collected on existing cores and requires specialized hardware support. Third, power estimation is completely missing as [22] focuses on throughput oriented server workload. Finally, and most importantly, the mechanism is evaluated using simulator where one has complete flexibility in choosing the core configurations.

In this work, we develop power-performance model for commercial asymmetric multi-core: ARM big.LITTLE. While an application is executing on ARM Cortex-A7 (alternatively ARM Cortex-A15), we collect profile information provided by hardware counters, and estimate power and performance characteristics of the same application on ARM Cortex-A15 (alternatively ARM Cortex-A7). *We evaluate the accuracy of our estimation on real ARM big.LITTLE hardware platform.*

Our modeling and estimation on real hardware are challenging in many ways. First, the big core and the small core are dramatically different, not just in the pipeline organization, but also in terms of memory hierarchy and the branch predictor — a reality that is ignored in all previous works [22, 15, 21]. These differences render the power, performance estimation from one core type to another considerably more difficult. Second, we are constrained by the performance counters available on the cores and their idiosyncrasies; for example, while the big core provides the *L2 cache write access* counter, it is unavailable on the small core. More importantly, in contrast to simulation based modeling work, we cannot rely on additional profiling information, such as inter-instruction dependency [22], that can only be collected by introducing extra hardware.

We overcome the challenges outlined above using a combination of static (compile time) program analysis, mechanistic model-

.

ing [13, 14], which builds analytical model from an understanding of the underlying architecture, and empirical modeling [17, 12], which employs statistical inferencing techniques like regression to create an analytical model.

Our performance model for any core centers around the CPI (cycles per instruction) stack that quantifies the impact of different architectural events (such as data dependency, cache miss, branch misprediction etc.) on the execution time. While we can obtain information about certain events (e.g., cache miss, branch misprediction) from the hardware counters, other information such as data dependency are not readily available. We rely on compile time static program analysis technique to capture the data dependency information and its impact on pipeline stalls.

Once we develop the CPI stack based performance model for each core, we proceed to estimate the CPI stack of the second core given the CPI stack of the first core. We employ regression modeling to estimate the architectural events (cache miss, branch misprediction) on the second core given information about the architectural events on the first core. These estimates of architectural events can be plugged into the CPI stack model of the second core to derive the CPI value and hence the performance estimate. Finally, our power model uses the CPI value along with additional information, such as instruction mix, memory behavior etc., to estimate the power behavior of the core.

Our concrete contributions in this work are the following.

- We propose a combination of static program analysis, analytical modeling, and statistical techniques to model the performance of individual cores and estimate power, performance across different cores on single-ISA heterogeneous multi-core platforms.

- Ours is the first work towards performance estimation across asymmetric cores on real hardware. Estimation on real hardware is challenging compared to simulation based studies [22] due to distinctly different configurations of the cores, memory hierarchy, and unavailability of some of the required hardware counters.

- Ours is the first work to model CPI stack on real out-of-order and in-order cores. [11] is the only existing work that models CPI stack for commercial out-of-order processors; but does not consider in-order processors. We demonstrate that our CPI stack model is more accurate as we combine the strengths of static program analysis and runtime analytical modeling.

- Ours is the only work to derive power estimation on the second core solely based on the execution profile on the first core. Existing works [9] require execution of the application on both cores to estimate power, an assumption that is unrealistic when migration cost from one core type to another is relatively high, as is the case in our setting.

2. RELATED WORK

Considerable number of prior works [8, 11, 14] have developed analytical performance models for processors. The two predominant approaches employed in building performance models are mechanistic modeling and empirical modeling. Mechanistic models are purely based on the insights of the target processor architecture. In [13, 14], the authors developed a simple interval based mechanistic model for out-of-order cores that assumes a sustained background performance level, which is punctuated by transient miss-events. The models from [13, 14] was further improved in [10] by weighing the dispatch stage in detail. Eyerman et al. [8] propose mechanistic model for superscalar in-order processors. In empirical modeling, the performance model is considered as a black box and typically inferred using statistical/regression techniques. Joseph et al. [12] use non-linear regression performance modeling. In [17], the authors employ spline-based regression modeling for performance and power across different micro-architectural configurations. The authors in [11] propose hybrid mechanistic-empirical modeling for commercial processor cores with few simplistic assumptions. However, our technique uses the combination of compile-time analysis, mechanistic modeling and empirical modeling to construct performance models for both out-of-order and in-order cores with better accuracy on a real platform.

Asymmetric multi-cores pose significant challenges in scheduling [16, 9]. The online scheduling in heterogeneous multi-cores can be improved significantly by estimating performance across all the core types [22]. Craeynest et al. [22] propose a performance model to estimate the CPI stack across the big and small cores. Our power-performance model for asymmetric multi-cores has the following advantages over [22]: (a) our model was developed using real hardware (i.e., not any assumptions of the presence of additional counters mentioned in [22]), (b) we do not assume uniform memory hierarchy, and (c) we extend our performance model to estimate power consumptions. Cong et al. [9] propose an energy estimation technique on Intel's QuickIA heterogeneous platform. In [9], the estimation of energy-delay product requires the execution of the applications in all the core types. In comparison, our technique can estimate the power (also energy-delay product) without having to profile in all the core types.

3. ARM BIG.LITTLE ARCHITECTURE

We first describe the micro-architectural features of the ARM big.LITTLE asymmetric multi-core that we model for power, performance estimation. The single-ISA heterogeneous architecture consists of high performance Cortex-A15 cluster and power efficient Cortex-A7 cluster, as shown in Figure 1. The evaluation platform we use in this work contains a prototype chip with two Cortex-A15 cores and three Cortex-A7 cores at 45nm technology. All the cores implement ARM v7A ISA. The Cortex-A15 is complex out-of-order superscalar core that can execute high intensity workloads, while Cortex-A7 is a power efficient in-order core meant for low intensity workloads. While each core has private L1 instruction and data caches, the L2 cache is shared across all the cores within a cluster. The L2 caches across clusters are kept seamlessly coherent via the CCI-400 cache coherent interconnect.

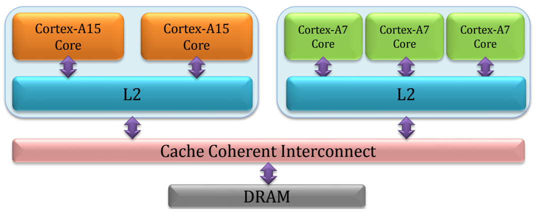


Figure 1: ARM big.LITTLE asymmetric multi-core.

Table 1 summarizes the micro-architectural parameters of Cortex-A15 and Cortex-A7, obtained from publicly released data. It should be evident that the cores are genuinely asymmetric in nature. The 2-way issue in-order pipeline of A7 containing 8-10 stages is dramatically different from the 3-way issue out-of-order pipeline of A15 containing 15-24 pipeline stages. Moreover, even the cache configurations and branch predictors are distinctly different in A15

Parameter	Cortex-A7	Cortex-A15
Pipe-line	In-order	Out-Of-Order
Issue Width	2	3
Fetch Width	2	3
Pipeline Stages	8-10	15-24
Branch Predictor	512-entry BTB 2-way	2K-entry BTB 2-way
L1 I-cache	32KB/2-way/32B	32KB/2-way/64B
L1 D-cache	32KB/4-way/64B	32KB/2-way/64B
L2 Unified-cache	512KB/8-way/64B	1MB/16-way/64B

Table 1: Architectural Parameters of Cortex-A7 and Cortex-A15

compared to A7. Most previous works [22, 15, 21] assume that the memory parameters are identical across different core types.

The architecture provides DVFS feature per cluster. The A7 cluster provides eight discrete frequency levels between 350MHz – 1GHz, while the A15 cluster also provides eight discrete frequency levels between 500MHz – 1.2 GHz. Note that all the cores within a cluster should run at the same frequency level. Moreover an idle cluster can be powered down if necessary. As our focus is on power, performance estimation across core types, we conduct the experiments by setting the same voltage (1.05 Volt) and frequency (1 GHz) for the two clusters. Estimating power, performance for different frequency levels is left as future work. We also consider execution of a sequential application on either A7 or A15, that is, we only use one core at a time and the idle cluster is powered down.

The heterogeneous cores exhibit different power and performance characteristics across workloads. Figure 2 shows the performance speedup, energy consumption ratio, and EDP (Energy-Delay product) ratio for 15 selected benchmarks on A15 in comparison to A7. Clearly, A15 has significant performance improvement compared to A7 (average speedup of 1.86); more importantly, the speedup varies significantly across benchmarks from 1.45 to 2.30. In terms of power, it is expected that A7 has lower average power compared to A15 for all the benchmarks. While average power on A7 is 1.44Watt, the average power on A15 varies from 4.20Watt to 5.15Watt. Even though A7 has worse performance, it can completely make up for it in terms of power to achieve far superior energy efficiency compared to A15 (1.78 times lower energy on average). A7 is also more energy efficient for all the benchmarks.

But in embedded systems, especially in interactive systems such as smartphones, we are more interested in the combination of energy and delay to decide on workload-to-core mapping because both battery life and response time are equally important. This metric is captured as Energy-Delay product (EDP). Interestingly, in terms of EDP, there is no clear winner: A15 is more efficient than A7 for 8 benchmarks due to faster execution that overcomes the power inefficiency, while A7 is superior for the remaining 7 benchmarks due to lower power consumption. Thus, the scheduler needs both power and performance behavior on a core type to decide on the appropriate mapping.

As observed in [22] and validated in our experiments, it is impossible to predict the power, performance characteristics of an application on different core types based on simple metrics such as memory access intensity. We also observe that the average migration cost across clusters is quite high: 2.10ms to move a task from A7 to A15, and 3.75ms to move from A15 to A7. This renders it unrealistic to first execute a workload on each cluster separately and then make the workload-core mapping decision as proposed in [9]. Thus it is essential to accurately estimate the CPI for performance and use the CPI to estimate power. We do so through power, performance modeling in the next section.

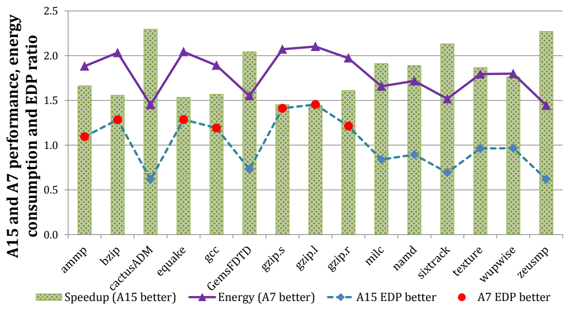


Figure 2: Performance improvement, energy consumption ratio and EDP ratio of A15 in comparison to A7.

4. PERFORMANCE MODELING

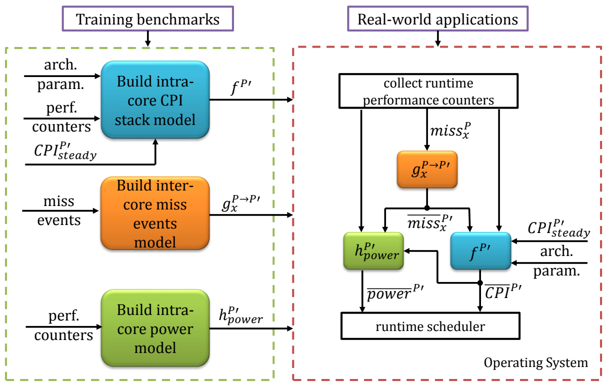


Figure 3: Inter-core performance, power estimation from P to P'.

The aim of performance modeling is to estimate the performance of an application on a second core type (small/big) given its execution profile on the first core (big/small) type. Our model centers around CPI stacks. The basic observation behind the model is that the CPI follows a sustained background level performance CPI_{steady} punctuated by miss events that show up as temporary peaks. CPI_{steady} captures the cycles spent in the architectural events tightly coupled to the pipeline such as data dependency among instructions and structural hazards, while CPI_{misses} represents the cycles spent due to the external events such as cache miss and branch mispredicton.

$$CPI = CPI_{steady} + CPI_{miss} \qquad (1)$$

The performance estimation framework shown in Figure 3 comprises of three major steps. The first step is an off-line procedure where we build *intra-core CPI stack model* for each core type.

While CPI_{miss} can be expressed in terms of miss events and their latencies, computing CPI_{steady} requires presence of elaborate hardware mechanisms [22] that can collect inter-instruction dependencies and are not available in existing processors. We avoid additional hardware mechanism by observing that CPI_{steady} is an intrinsic characteristics of a program on a core type and is stable across different program inputs, whereas CPI_{miss} is highly dependent on the program inputs. For example, Figure 4 shows the estimated CPI_{steady} and CPI_{miss} values of *bzip* benchmark for different program inputs on A7 and A15. Note that expectedly CPI_{steady} is higher on A7 than A15 because A15 with out-of-order execution engine can better exploit instruction-level parallelsim in the presence of data dependencies and structural hazards. The estimated CPI is the summation of the estimated CPI_{steady}

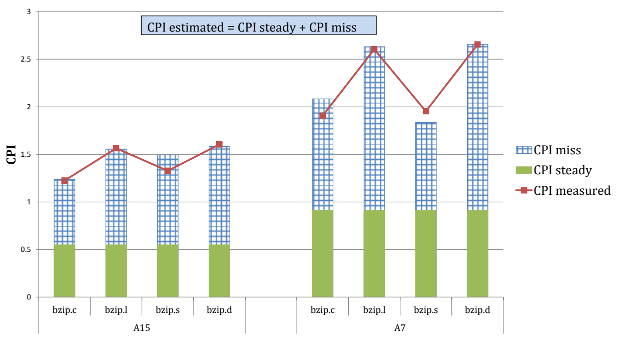


Figure 4: Estimated CPI_{steady} and CPI_{miss} of different inputs for the same benchmark on A7 and A15.

and CPI_{miss}. For reference, we have also plotted the measured CPI. Our assumption that CPI_{steady} of an application on a core type is stable across different program inputs is validated here as the variation in CPI has been captured accurately only through variation in CPI_{miss}.

We exploit this observation to estimate CPI_{steady} of a program on both core types at compile time (see Section 4.1) and encode this information with the binary executable. In other words, we estimate both CPI^{big}_{steady} and CPI^{small}_{steady} for a program at compile time. For applications with distinct phases, i.e., multiple computation kernels with different behavior, we estimate separate CPI_{steady} value for each phase.

To build the CPI stack on a core P, we collect the execution profiles of a set of training benchmarks through the hardware counters. We then combine analytical modeling with linear and non-linear regressions to derive the CPI stack model that accurately captures the contributions of the different events to performance. The CPI stack model can thus be expressed as the function f^P where

$$CPI^P = f^P(CPI^P_{steady}, miss^P_X, latency^P_X) \qquad (2)$$

where $miss^P_X$ and $latency^P_X$ are the number of occurrences and latency of each occurrence of the miss event X on processor P.

The second step is another offline procedure where we develop regression models that estimate the occurrence of different miss events on processor P' given the frequency of the miss events on processor P. These *inter-core miss event estimation models* are built by collecting and correlating corresponding miss events on both cores using a set of training benchmarks. The inter-core estimation model from P to P' for an event X can be expressed by a function $g^{P\to P'}_X$ where

$$\overline{miss}^{P'}_X = g^{P\to P'}_X\left(miss^P_X\right) \qquad (3)$$

where $\overline{miss}^{P'}_X$ is the predicted occurrence of miss event X on P'[1].

At runtime, when a new application is running on core P, the operating system collects the counter values at regular intervals to get information about the miss events on P. For each miss event X, it uses inter-core miss event estimation model to predict $\overline{miss}^{P'}_X$ on core type P'. Finally, it plugs in the estimated miss event counter values in the CPI stack model of P' to predict $\overline{CPI}^{P'}$.

$$\overline{CPI}^{P'} = f^{P'}(CPI^{P'}_{steady}, \overline{miss}^{P'}_X, latency^{P'}_X) \qquad (4)$$

4.1 CPI_{steady} estimation

Computing the CPI_{steady} value of a program on real hardware is challenging due to limited information that is exposed through the performance counters. While [22] proposes hardware counters that can count dynamic data dependencies and structural hazards for this purpose, the overhead of such counters is quite high due to the increased amount of book-keeping. An alternative is to simply assume $CPI^P_{steady} = 1/D$ where D is the dispatch width of processor P [11]. This assumption only holds true for perfectly balanced pipelines where the number of functional units for each type of operation is equal to the dispatch width and hence there is no structural hazards. It is not realistic as commercial processors do have unbalanced number of functional units. More importantly, the assumption completely ignores the dependency of CPI^P_{steady} on the characteristics of the program, in particular, inter-instruction data dependencies. We sidestep this problem by computing CPI^P_{steady} of a program on core P at compile time.

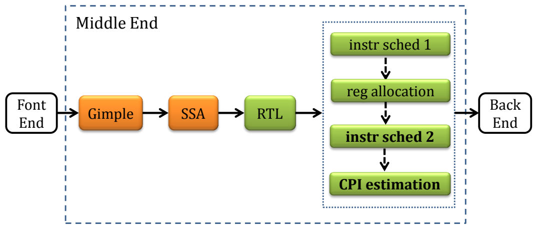


Figure 5: Estimation of steady state CPI of a program using gcc.

Most modern compilers have an optimization pass that takes care of instruction scheduling based on the hardware description of the processor pipeline. We use *gcc* compiler, where instruction scheduling optimization pass is performed twice, before and after register allocation pass (see Figure 5). When scheduling instructions, the algorithm uses a detailed description of the target processor pipeline. At this stage, the compiler is aware of the data dependencies among instructions and the structural hazards due to the limited number of the functional units in the processor pipeline.

We include our CPI_{steady} estimation pass after the second instruction scheduling pass. For each basic block B of the application, we extract the estimated number of cycles $cycle^P_B$, number of instructions $instr_B$, and the estimated frequency $freq_B$. Traditionally, the frequency values are obtained by profiling the application across different inputs but when the profile information is not available, the compiler can predict the behavior of each branch in the program using a set of heuristics and can compute estimated frequencies of each basic block by propagating the probabilities over the control graph. This estimate is used in our equation and it captures rather an average behavior of the application regardless of the input.

We define CPI^P_{steady} of an application $\mathcal{A}$ on core P as

$$CPI^P_{steady}\Big|_{\mathcal{A}} = \frac{\sum_B freq_B \cdot cycles^P_B}{\sum_B freq_B \cdot instr_B}\Bigg|_{\mathcal{A}} \qquad (5)$$

Note that only $cycles^P_B$ depends on the core type and leads to different steady state CPI values for different core types. The CPI_{steady} values thus computed for the small and big core are embedded into the application binary.

4.2 CPI stack model of big core

We extensively employ linear and non-linear regression models in our performance and power estimation framework. Our CPI stack model for the big core extends and adapts the mechanistic-empirical model proposed in [11] to Cortex-A15 core. Our model estimates the total number of clock cycles C required to execute an application on the big core as:

[1]We use $\overline{M}$ to indicate the estimated value of a metric across cores.

$$C^{big} = \beta_0 \cdot N \cdot CPI_{steady} + miss_{L1I} \cdot c_{L2} + miss_{br} \cdot (c_{br} + c_{fe}) + dmiss_{L2} \cdot \frac{c_{mem}}{MLP} \quad (6)$$

Once the total number of cycles are estimated, the CPI value can be easily computed by dividing the cycles by the total number of instructions N.

$$CPI^{big} = \frac{C^{big}}{N} \quad (7)$$

This parameterized model sums the number of cycles consumed due to internal and external events. The first term, CPI_{steady} is converted into the corresponding number of cycles by multiplying it with the total number of instructions N. The β_i parameters are unknown and are fitted through non-linear regression.

The next term represents the miss event cycles due to the instruction misses in first level of cache. The penalty paid for an instruction miss in L1 cache is c_{L2} and represents the number of cycles spent to access L2 cache and is micro-architecture dependant.

The next term of the equation quantifies the cycles spent during the branch misprediction events. The branch misprediction penalty is a function of the front-end length of the pipeline c_{fe} and the back-end of the processor where the branch is resolved in a branch resolution time c_{br}. The branch resolution time represents the number of cycles spent between the arrival time of the mispredicted branch in the dispatch queue and the moment when the branch is actually resolved in the execution unit. The branch resolution time is dependent on inter-instruction dependency, long-latency instructions and L1 data misses.

The next term of the Equation 6 represents the cycles spent due to the misses in the last level of data cache. The big core is an out-of-order core which takes advantage of the memory level parallelism such that part of an L2 cache miss latency overlaps with other independent L2 cache misses. Thus, we reduce the overall penalty by a factor MLP which is described as follows:

$$MLP = \beta_1 \cdot \left(\frac{dmiss_{L2}}{N}\right)^{\beta_2}$$

This equation assumes that the L2 data misses are uniformly distributed and the amount of parallelism that can be extracted has a power law relation with the window of misses per instruction from which the parallelism is to be extracted.

We recommend the reader to consult the work in [8] for more details about the intuition behind the presented equations.

4.3 CPI stack model of small core

Modeling the CPI stack for the small in-order core is simpler. We start from Equation 6 and remove the terms that are specific to out-of-order processors. The total number of cycles for the small core can be modeled using linear regression as follows:

$$C^{small} = \beta_3 + \beta_4 \cdot N \cdot CPI_{steady} + miss_{L1I} \cdot c_{L2} + miss_{br} \cdot c_{fe} + dmiss_{L2} \cdot c_{mem} \quad (8)$$

$$CPI^{small} = \frac{C^{small}}{N} \quad (9)$$

In case of in-order processor, the branch resolution time in the back-end pipeline is not relevant because there is no reorder buffer structure present in an in-order processor. Once there is branch misprediction, the entire pipeline has to be drained. Similarly, the MLP correction factor is not used as cache missses cannot overlap.

Parameter	Var name	Cortex-A15	Cortex-A7
Pipeline front-end	c_{fe}	4	13
L2$ access	c_{L2}	19	13
Main memory access	c_{mem}	140	100

Table 2: Estimated latency in cycles for miss events on A15 and A7

Parameter	Var name	Cortex-A15	Cortex-A7
Cycles	C	✓	✓
Instructions	N	✓	✓
Branch instr	N_{br}	✓	✓
Branch misses	$miss_{br}$	✓	✓
Load instr	N_{ld}	✓	✓
Store instr	N_{st}	✓	✓
Integer instr	N_{int}	✓	✓
Float instr	N_{fp}	✓	✓
L1I$ access	$access_{L1I}$	✓	✓
L1D$ access	$access_{L1D}$	✓	✓
L1I$ misses	$miss_{L1I}$	✓	✓
L1D$ misses	$miss_{L1D}$	✓	✓
L2$ data miss	$dmiss_{L2}$	✓	✓
L2$ write access	$dwaccess_{L2}$	✓	
L2$ write back	WB_{L2}	✓	✓
Power sensor	$Power$	✓	✓
Energy sensor	$Energy$	✓	✓

Table 3: Hardware Performance Counters on A15 and A7

4.4 Latency of miss events and performance counters

The performance models for both big and small core use a number of hardware performance counters and the latencies corresponding to each individual miss event. Table 3 enumerates all the performance counters that are used in our work and their availability on A7 and A15 cores.

While information about the pipeline structure and memory hierarchy configurations of A7 and A15 are available from publicly released data as well as processor internal registers, the cache miss and memory access latencies are not released. To estimate the access latencies to L2 cache and main memory we use the *lmbench* [19] micro-benchmark. Table 2 summarises the penalties in cycles for the different miss events used in our models for A7 and A15.

4.5 Contribution of CPI stack components

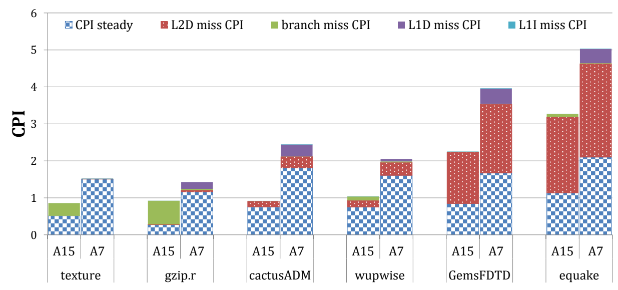


Figure 6: Estimated CPI stack components on A7 and A15 for a subset of benchmarks.

The miss events used in both the models are branch misprediction, L1 and L2 cache miss. We chose only these events as they contribute most to the overall CPI of a processor. In order to support our claim we conducted several experiments on a set of benchmarks that expose different computational behaviour. Figure 6 plots the estimated CPI stack on both small and big cores. We chose two compute intensive benchmarks (*texture* and *gzip.r*), two average compute intensive benchmarks (*cactusADM* and *wupwise*), and two memory bound benchmarks (*GemsFDTD* and *equake*). The benchmarks were selected from Vision [23], SPEC2000 and SPEC2006 [3] benchmark suits. In case of the memory bound applications, the impact of misses in L1 and L2 caches on the overall CPI is considerably higher compared to the compute intensive applications for which the CPI_{steady} and branch mispredictions are impacting the CPI mostly. Note that branch misprediction impacts A15 substantially more than A7, because A15 has an aggressive back-end pipeline that suffers more from squashing of instructions. We also observe that the speedup on A15 compared to A7 is dependent on a lot of factors, such as CPI_{steady}, branch misprediction cost, and cache miss cost.

5. INTER-CORE MISS ESTIMATION

The real challenge in inter-core performance and power estimation on asymmetric cores is that the memory hierarchy and the branch predictors may not be identical across different core types, as is the case in ARM big.LITTLE (see Table 1). The small cores are connected to a simpler cache system in order to increase the power efficiency, while the big cores are connected to a more complex memory that supports higher memory throughput, which increases the overall performance. Recent related works [22] assumed that the asymmetric systems have identical memory hierarchy. *The innovation in our approach is that we develop mechanistic-empirical models that can predict the occurrences of miss events $miss_X^{P'}$ on processor P', given their occurrences $miss_X^P$ on P obtained through hardware performance counters.*

In order to predict the CPI value of core P' while running on core P, we need to predict the values of the performance counters used in Equation 6 and Equation 8 depending on whether we are predicting the CPI of big core or small core, respectively. These counters are: number of first level data and instruction cache miss ($miss_{L1D}, miss_{L1I}$), number of last level cache miss ($miss_{L2}$) and the number of branch mispredictions ($miss_{br}$).

Inter-core branch misprediction estimation. The big core A15 has significantly more aggressive branch predictor compared to A7 to ensure sustained supply of instructions to the high-throughput back end. We observe that the branch misprediction rate on P' (big or small) is correlated to three metrics on P: the branch misprediction rate, the CPI, and the number of branches per instruction. The last metric signifies the rate of branch prediction — the higher the rate, the more is the benefit from a complex predictor. Similarly, the higher the instructions per cycle (or lower the CPI), the more is the need for aggressive branch predictor. Thus we define the inter-core branch misprediction estimation model as follows:

$$\overline{miss_{br}^{P'}} = \beta_5 + \beta_6 \cdot miss_{br} + \left(\frac{1}{CPI}\right)^{\beta_7} + \beta_8 \cdot \left(\frac{N_{br}}{N}\right)^{\beta_9} \Bigg|_P$$

Inter-core L1 instruction cache miss estimation. The L1 instruction caches on both cores have the same size and associativity. But the line size on A15 is 64 bytes, while the line size on A7 is 32 bytes. Thus A15 can exploit more spatial locality leading to reduced cold miss. But A7 has twice the number of sets compared to A15, which may lead to reduce conflict miss in A7. As we do not have information about cold and conflict miss, we attempt to estimate them. We assume that the number of cold misses $cold$ on processor P is the code size divided by the line size. To predict cold miss on P', the cold miss obtained from P is scaled by the average size of basic blocks $\frac{N}{N_{br}}$. The rationale is that the larger the basic block size, the more likely the cache benefits from larger line size due to spatial locality. Thus, our inter-core L1 instruction cache miss estimation model is

$$\overline{miss_{L1I}^{P'}} = \beta_{10} + \beta_{11} \cdot cold \cdot \left(\frac{N}{N_{br}}\right)^{\beta_{12}} + \beta_{13} \cdot conflict \Bigg|_P$$

$$cold = \frac{code_{size}}{line_{size}}\Bigg|_P ; \quad conflict = miss_{L1I} - cold\Bigg|_P$$

Inter-core L1 data cache miss estimation. The L1 data cache has the the same size but different associativity on A7 and A15. However, across a large range of benchmarks, we observe that there is very little difference between the number of L1 data cache miss on A7 and A15. So we employ a simple linear regression model for inter-core miss prediction of L1 data cache.

$$\overline{miss_{L1D}^{P'}} = \beta_{14} + \beta_{15} \cdot miss_{L1D}\Bigg|_P$$

Inter-core L2 cache miss estimation. The L2 is a unified data plus instruction cache on both A7 and A15. Even though both L1I and L1D miss filter down and access the unified L2 cache, the instruction accesses have higher spatial and temporal locality leading to negligible miss rate for instruction accesses in L2. Thus instruction miss in L2 does not influence the CPI stack on either A7 or A15 and can be safely ignored. This is fortunate because both A7 and A15 provide performance counters for only L2 data access miss and not L2 instruction access miss. We denote L2 data access miss as $dmiss_{L2}$ and it has significant influence on CPI stack as shown in Figure 6. Thus for accurate inter-core performance and power estimation, it is absolutely essential to predict $dmiss_{L2}$ correctly.

For our architecture, L2 cache is distinctly different in A15 compared to A7. Not only the L2 in A15 has twice the associativity of A7 (16-way versus 8-way); but also the size is doubled in A15 (1MB compared to 512KB). This also implies that the number of sets (1024) in L2 is exactly the same for both A7 and A15 and A15 is likely to have significantly less conflict miss due to higher associativity, whereas cold misses should be similar because the line size is identical.

How do we determine the number of conflict miss for L2 data access? We use the number of write backs to estimate conflict miss in L2. A write back indicates conflict miss because a memory line is being evicted from the cache due to conflict with another memory line. But not all conflict miss are captured via write backs. If the memory line being replaced in the cache is clean (i.e., contains read data), we cannot observe the conflict in terms of write back. We make the assumption that the rate of conflict miss is the same for both read data and write data. Thus we scale the write back by the fraction of write access to estimate the conflict miss $conflict$.

$$conflict = \frac{WB_{L2}}{(wfrac_{L2})^{\beta_{16}}}\Bigg|_P ; \quad cold = dmiss_{L2} - conflict\Bigg|_P$$

While predicting L2 data miss from the big core (A15) to the small core (A7), we have measured value for $wfrac$ from performance counters: number of L2 access (which is same as the sum of L1D and L1I miss) and number of L2 data write access

($dwaccess_{L2}$). Thus

$$wfrac_{L2} = \frac{dwaccess_{L2}}{miss_{L1I} + miss_{L2D}}$$

While predicting from the small core to the big core, however, we are challenged by the lack of performance counters for write access. So we estimate L2 write access as the L1D miss scaled by the fraction of store instructions over total memory instructions.

$$wfrac_{L2} = \frac{\frac{N_{st}}{N_{st}+N_{ld}} \cdot miss_{L1D}}{miss_{L1I} + miss_{L2D}}$$

We are now ready to predict $dmiss_{L2}$ across cores. We use linear regression of cold miss and conflict miss on P to predict the total miss on P'. We observe that while L2 instruction access miss is negligible, if the number of instruction access in L2 is high compared to total L2 access, there is higher chance of instructions evicting data through conflict in unified cache. Thus we scale the conflict miss by L2 instruction access fraction to obtain more accurate inter-core conflict miss prediction.

$$\overline{dmiss_{L2}^{P'}} = \beta_{17} \cdot conflict \cdot \frac{miss_{L1I}}{miss_{L1I} + miss_{L1D}} + \beta_{18} \cdot cold \Big|_{P}$$

Inter-core CPI estimation. Once we have estimated the miss events on core P', given the miss event information on core P, it is straightforward to obtain CPI estimate on P'. We simply need to compute $CPI_{miss}^{P'}$ by plugging in the estimated miss event values in the CPI stack of P' as defined by Equation 6 or Equation 8.

6. POWER MODELING

We now describe our modeling technique to estimate power on asymmetric multi-core. Unlike performance modeling, which required a combination of mechanistic and empirical modeling, power can be modeled purely based on regression analysis. We used a simple linear regression model to estimate the power consumption in terms of available performance counters.

Modeling power of small core. In big.LITTLE platform, the small cores are superscalar in-order, power efficient Cortex-A7 processors. We observe that the average power consumption of the small core is quite similar across all the benchmarks. The min and max power consumption measured across training benchmarks (from Table 4) are 1.385 watts and 1.506 watts respectively. Thus, there is no need to model power for the small cores.

Modeling power of big core. While power consumption on the small A7 core is stable across benchmarks, the big core (power-hungry, out-of-order A15) shows significant variation in power consumption within (due to phase behavior in programs) and across benchmarks. The observed min and max power consumption on A15 across training benchmarks (from Table 4) are 4.535 watts and 5.155 watts respectively. This is because complex out-of-order cores exhibit different access profiles of various micro-architectural components across the benchmarks. Thus, it is imperative to model application-specific power consumption on A15.

The power consumption of A15 depends on the pipeline behavior and the memory behavior of the application. In particular, the instruction mix of an application is expected to influence the access profile of different architectural components such as ALU, floating-point unit, branch predictor etc, which in turn, determines the power consumed in the pipeline. The power consumption in the memory hierarchy is determined by the number of L1I, L1D, L2, and memory access. So we are looking for the function h^P in Figure 3 that models the power consumption

$$Power^P = h^P(N_X, miss_X^P, CPI^P)$$

where N_X is fraction of instructions of type X in instruction mix.

Given a set of training benchmarks, we first collect the performance counter values on A15 that captures the instruction mix and the access at different levels of the memory hierarchy. We also measure the power consumption on A15 (power measurement setup will be presented in Section 7). Next we employ correlation analysis to identify the important performance counter that are most related to power consumption. The total power consumption of the big core can be expressed in terms of the following linear regression model:

$$\begin{aligned} Power = \beta_{19} + \beta_{20} \cdot \frac{N_{int}}{N} + \beta_{21} \cdot \frac{N_{fp}}{N} + \beta_{22} \cdot \frac{1}{CPI} \\ +\beta_{23} \cdot \frac{access_{L1D}}{N} + \beta_{24} \cdot \frac{access_{L2}}{N} + \beta_{25} \cdot \frac{dmiss_{L2}}{N} \end{aligned} \quad (10)$$

The first three terms capture the power consumption in the pipeline, which is influenced by the proportion of integer instructions (N_{int}), the proportion of floating point instructions (N_{int}), and the instructions per cycle IPC (the inverse of CPI).

The power consumption is also linearly related to the rate of access to the various levels of the memory hierarchy, which is captured using the next three terms. Notice that we do not include L1 instruction cache access here because it is already included in terms of CPI. The higher the CPI, the lower the rate of access to L1 instruction cache.

Estimating power of big core from small core. The major challenge in estimating the power consumption of an application on the big core while running it on the small core is that we have to predict the access profile. In Equation 10, the instruction mix (N, N_{int} and N_{fp}) remains unchanged across cores. The inter-core miss event prediction model given in Section 5 estimates $\overline{CPI}$, $\overline{dmiss_{L2}}$, $\overline{miss_{L1D}}$, $\overline{miss_{L1I}}$ on the big core from the corresponding values on the small core (see also in Figure 3). We can then define

$$\overline{access(L2)} = \overline{miss_{L1D}} + \overline{miss_{L1I}}$$

These estimated values can be plugged into Equation 10 to estimate the power consumption on the big core.

7. EXPERIMENTAL EVALUATION

We now evaluate our power, performance estimation framework for asymmetric multi-core. We first present the experimental setup, followed by fitting errors of our model on training benchmarks, and finally a validation of our models within and across cores for a new set of test benchmarks.

Experimental setup. We use the Versatile Express development platform [5] comprising of a motherboard on which the big.LITTLE prototype chip is mounted as part of a daughter board. The motherboard handles the interconnection between the daughter board and the peripherals using an FPGA bus interconnection network. The board boots Ubuntu 13.02 Linaro with the Linux kernel release 3.7.0 for Versatile Express [6]. The platform firmware runs on an ARM controller, which is embedded on the motherboard. The Linux file system is installed on the Secure Digital (SD) card where all our benchmark applications are located.

We collect the hardware performance counter values using ARM Streamline gator kernel module and daemon [1]. We compile and configure Linux kernel to support the gator driver. The gator driver is a dynamic kernel module that interrupts the core at periodic intervals to collect the performance counters. The average CPU utilization of gator daemon is less that 0.5%, which indicates that the overhead of running gator daemon in the background is minimal. We use Matlab [20] to develop our regression models offline.

The prototype big.LITTLE chip consists of one A15 cluster and one A7 cluster at 45nm technology. The individual clusters are equipped with sensors to measure the frequency, voltage, current, power and energy consumption at the cluster level and not at the core level. Moreover, we can only power down a cluster; but not individual cores within a cluster. In our experiments, we utilize only one A15 core and one A7 core. The remaining cores in the clusters are logically turned off using system calls, such that no tasks are scheduled on them. Finally, we set the voltage and frequency for both the clusters at 1.05V and 1GHz, respectively.

Training Benchmarks	*ammp, cactusADM, equake, gcc GemsFDTD, gzip.s, gzip.l, gzip.r, milc namd, sift, sixtrack, texture, wupwise, zeusmp*
Test Benchmarks	*apsi, calculix, gamess, gzip.p gzip.g, h264, lbm, leslie3d, mcf, mgrid, mser, omnetpp parser, swim, tonto*

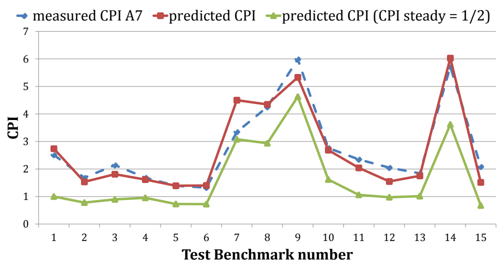
Table 4: Training and Test Benchmarks

Compiler setup. We implement our CPI_{steady} estimation pass in the GCC Linaro version 4.7.3. The GCC instruction scheduler [2] uses a very efficient pipeline hazard recognizer to estimate the possibility of issuing an instruction on a given core in a given cycle. The processor pipeline descriptions can be expressed in terms of a deterministic finite automaton, which in turn is used to generate pipeline hazard recognizer. The latest version of Linaro GCC compiler includes the processor pipeline descriptions for Cortex-A7 and Cortex-A15 cores. We exploit the hazard recognizer to estimate the data dependencies and structural hazards for a program on A7 and A15, which leads to the steady state CPI estimate as presented in Section 4.1.

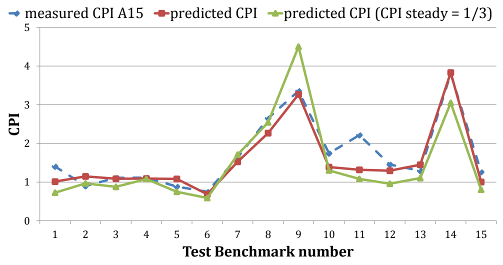
We compile all the benchmarks with -O2 optimization flag. This ensures that the instruction scheduling optimization pass and the CPI_{steady} estimation pass are invoked. We disable both the hardware and the software prefetcher in all our experiments. The Cortex-A15 cores comprise of Level 2 hardware prefetcher, while Cortex-A7 contains Level 1 data cache hardware prefetcher. All the hardware prefetcher are disabled by writing to the CP15 auxiliary control register. All the benchmarks are compiled with *-fno-prefetch-loop-arrays* flag to disable software prefetching.

Training and Test benchmarks. For our experiments, we use Vision [23], SPEC CPU2000 and CPU2006 [3] benchmark suites with reference inputs. Table 4 lists the set of benchmarks used in our training and tests set. We categorize all the benchmarks into three types based on the memory behaviour: memory intensive benchmarks, compute intensive benchmarks and intermediate benchmarks. The fraction of L2 miss per instruction is given by $frac_{L2} = (\frac{dmiss_{L2}}{N} \cdot 100)$. For memory intensive benchmarks, we chose the $frac_{L2} > 1.5\%$, while the compute intensive benchmarks have $frac_{L2} < 0.5\%$ and the remaining are classified as intermediate benchmarks. We randomly select five benchmarks from each category to capture diverse behavior in our training set. The training set is used to develop our regression models for power and performance, while the test set is used to cross-validate the model. As shown in Table 4, we keep the test benchmark set consisting of 15 benchmarks completely disjoint from the training set.

7.1 Performance estimation accuracy



(a) CPI prediction on A7



(b) CPI prediction on A15

Figure 7: Intra-core model validation accuracy using CPI_{steady} obtained through compile-time analysis compared to the accuracy assuming $CPI_{steady} = 1/D$

We validate our performance and power estimation models using three sets of experiments. In the first experiment, we compute the fitting error for our regression models on the training benchmarks. It is important to get a good fitting in order to build an accurate model. However, with over-fitting, we run at the risk of large errors for new applications, for which the model was not trained. Thus, our second experiment computes error in intra-core performance and power estimation for the test benchmark set using the model derived from training benchmarks. This shows the robustness of the model, i.e., how well the model behaves for new applications. Finally, we validate the accuracy of our inter-core estimation models on test benchmarks. This challenging task requires both accurate CPI stack models and inter-core miss event estimation models to achieve good accuracy.

Fitting error in regression for training benchmarks. In Figure 8, the benchmarks are numbered in the same order as it is enumerated in Table 4. Figures 8a and 8d show the measured and estimated CPI for small and big core, respectively, on the training set. The average fitting errors observed are 8.2% for small core and 10.1% for big core, respectively. A7 CPI stack model has better accuracy because it is easier to build the CPI stack for in-order cores in comparison to complex out-of-order cores.

Given an application, we obtain CPI_{steady} on big and small core at compile time. This is in contrast to the technique proposed in [11] that assumes CPI_{steady} to be equal to $\frac{1}{D}$, where D is the dispatch width of the core ($D = 2$ for A7 and $D = 3$ for A15). In other words, the model in [11] completely ignores the impact of program characteristics on steady state CPI. Figure 7 shows the advantage of compile-time estimation on CPI prediction for both A7 and A15. Our technique reduces the prediction error by 33.3% on A7 and 8.1% on A15, on an average, in comparison to [11].

Intra-core validation for test benchmarks. In order to further evaluate the accuracy and robustness of our intra-core CPI stack

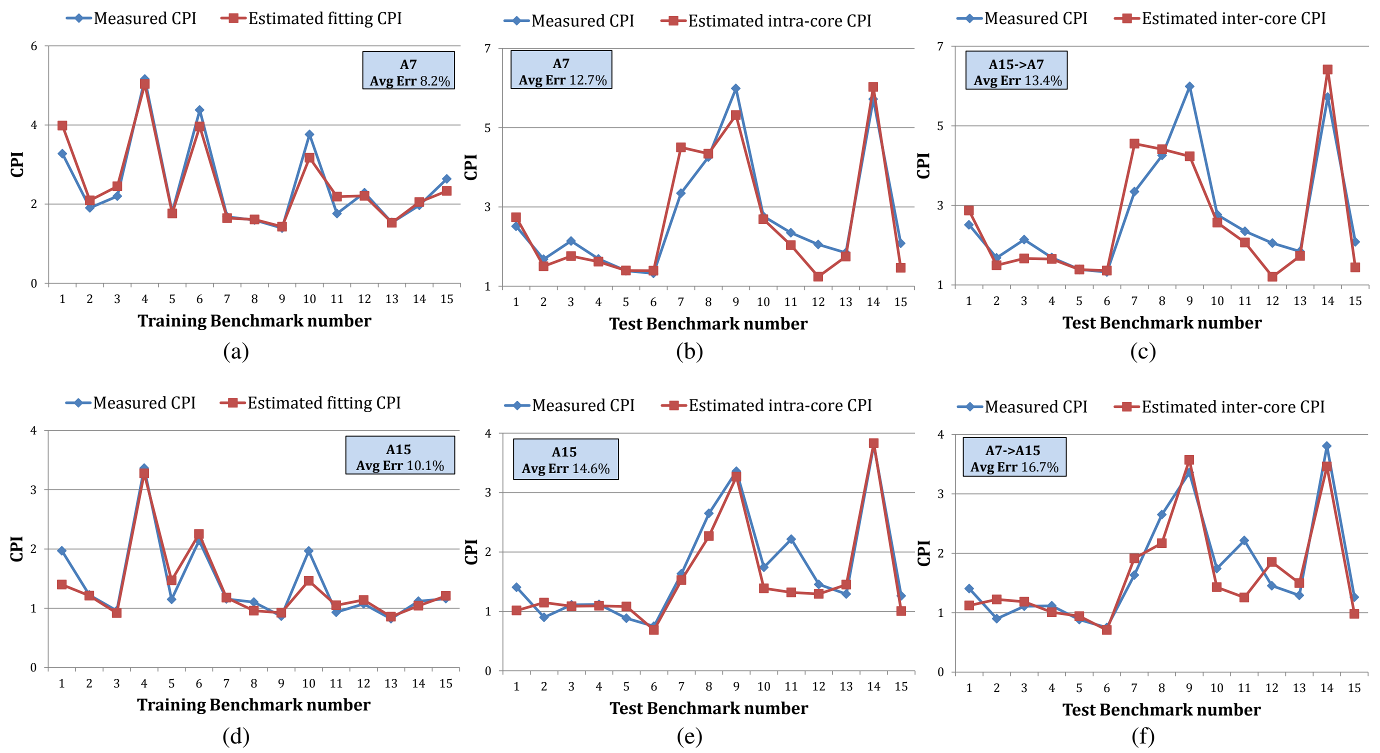


Figure 8: CPI stack model fitting error on training benchmarks, intra-core model validation error using test benchmarks and inter-core CPI estimation error for Cortex-A7 (top row) and Cortex A-15 (bottom row).

model, we compare the measured CPI and the estimated CPI for a completely new set of benchmarks (i.e., test benchmarks) from Table 4. From Figures 8b amd 8e, we observe that the average intra-core prediction errors are 12.7% for small core and 14.6% for big core, respectively. For both small and big cores, 80% of all test benchmarks have prediction error less than 25%. The error increases slightly compared to the fitting error, which is expected given the diverse characteristics of our test benchmarks. The intra-core validation with test benchmarks confirms that our CPI stack model is robust and we have avoided over-fitting.

Inter-core validation for test benchmarks. It is challenging to estimate the CPI for one core type, while executing on the other core type. The estimation is further exacerbated by the presence of highly dissimilar cache hierarchy. We perform *Inter-core validations* using the test benchmarks. For this set of experiments, we execute each test benchmark on A7 (alternatively A15), collect the execution profile, and estimate its CPI on A15 (alternatively A7) using our regression models explained in Section 4. We then compare the predicted CPI on the target core with the measured CPI to evaluate the accuracy of our estimation.

Figure 8c shows the measured CPI and the estimated CPI on small core using the performance counters from big core. The average *Inter-core validation* error in predicting small core CPI from big core is 13.4%; the maximum error is 43.2%. The comparison between the measured CPI and the estimated CPI on big core using the performance counters from small core is shown in Figure 8f. We observe average *Inter-core validation* error in predicting big core from small core is 16.7% and the maximum error is 41.3%. As inter-core estimation depends on both CPI stack model and inter-core miss event estimation models, this experiment validates the accuracy of both the models.

7.2 Power estimation accuracy

Similar to the evaluation of our performance modeling, we evaluate the power modeling in terms of fitting error, intra core validation and inter core validation. As discussed earlier, we do not need to build a power model for the small core due to insignificant variance in power consumption across the benchmarks. Figure 9a shows the measured and estimated fitting power for the training set on big core. Similarly, Figure 9b and 9c compare the measured and estimated power for intra-core and inter-core validation on test benchmarks. The average prediction error is fairly low even for inter-core validation (3.9%) (y-axis is scaled to capture the small difference between the measured and estimated values). The power estimation across cores rely more on memory access behavior, which we predict fairly accurately leading to high acuracy.

7.3 Case Study

So far we have shown the accuracy of our power and performance estimation models for whole benchmarks. In reality, some benchmarks exhibit phase behavior in their execution. We envision that our estimation framework can be used in such contexts to continuously monitor the execution profile on one core and estimate the power, performance on the other core. This will allow the scheduler to migrate the task back and forth between the cores depending on which phase the program is currently in and the appropriate core type for that phase.

We conduct a case study experiment with *astar* benchmark to evaluate the accuracy and robustness of our model in detecting phase changes and accurately predicting the behavior on the target core for each phase. Figure 10 shows the estimated power, performance on A15 predicted from executing the application on A7. For references, we also show the measure power, performance on A15. The X-axis shows the number of committed instructions as time progresses. We set our sampling interval at 500ms, which roughly

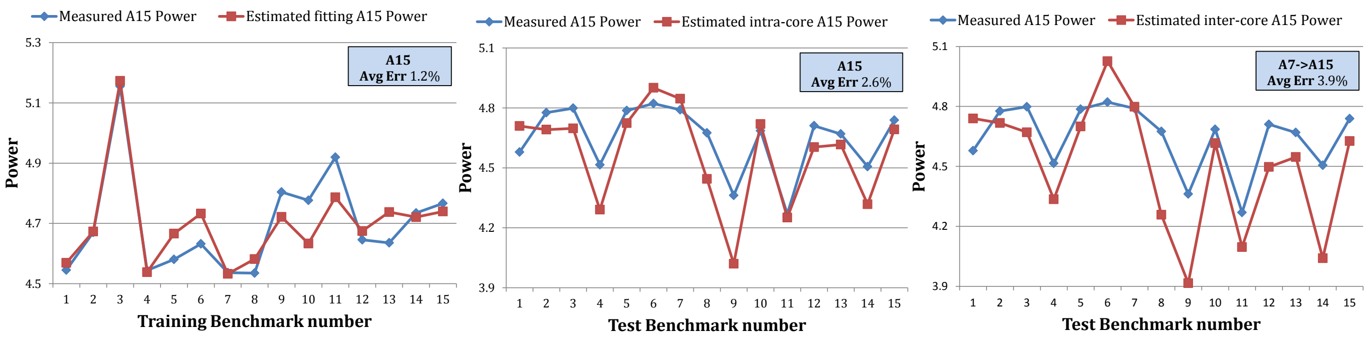


Figure 9: Power model fitting error on training benchmarks, intra-core model validation error using test benchmarks and inter-core power estimation error for Cortex-A15.

corresponds to 500 million instructions on A7 at 1GHz. The application demonstrates clear phase behavior. Our estimations are fairly close to the measured values. Thus we can track the phase changes accurately and present performance speedup and energy efficiency on A15 compared to A7 for each phase. How the scheduler can exploit this information depends on various policies and is not within the scope of this work.

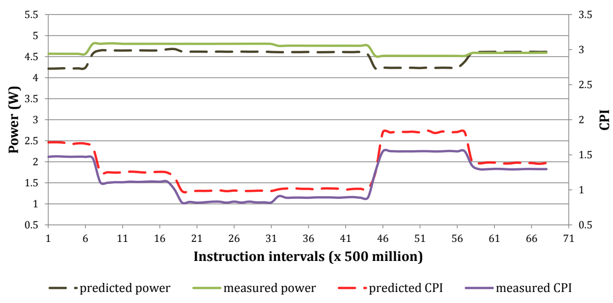


Figure 10: Contiuous CPI and power estimation from A7 to A15 for *astar* benchmark.

8. CONCLUSION

In this paper, we develop accurate models to estimate the power and performance on asymmetric multi-core architectures. Our aim is to predict, at runtime, the power, performance behavior of an application on a target core, given its execution profile on the current core, where the cores share the same ISA but has heterogenous micro-architecture. We overcome the challenges of distinctly different micro-architecture, memory hierarchy, and branch predictor on commerical asymmetric multi-cores through a combination of compile-time analysis, mechanistic modeling, and linear/non-linear regressions. One of the key contribution of our work is an accurate model that estimates the cache miss and branch misprediction rates on the target core, solely from the information available on the current core. Unlike almost all previous modeling works, we design and evaluate our estimation framework on a real asymmetric multi-core – ARM big.LITTLE.

ACKNOWLEDGMENTS: This work was partially supported by CSR research funding and Singapore Ministry of Education Academic Research Fund Tier 2 MOE2012-T2-1-115.

9. REFERENCES

[1] ARM infocenter. `http://infocenter.arm.com/`.
[2] Gcc processor pipeline description, http://gcc.gnu.org/onlinedocs/gccint/processor-pipeline-description.html.
[3] SPEC CPU Benchmarks. `http://www.spec.org/benchmarks.html`.
[4] Nvidia. the benefits of multiple cpu cores in mobile devices, 2010. `http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf`.
[5] ARM Ltd., 2011. `http://www.arm.com/products/tools/development-boards/versatile-express/index.php`.
[6] Linaro Ubuntu release for Vexpress, November 2012. `http://releases.linaro.org/13.02/ubuntu/vexpress/`.
[7] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Computing frontiers*, pages 29–40. ACM, 2006.
[8] M. Breughe, S. Eyerman, and L. Eeckhout. A mechanistic performance model for superscalar in-order processors. In *ISPASS*, pages 14–24, 2012.
[9] Jason Cong and Bo Yuan. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Low Power Electronics and Design*, pages 345–350. ACM, 2012.
[10] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A mechanistic performance model for superscalar out-of-order processors. *TOCS*, 27(2):3, 2009.
[11] Stijn Eyerman, Kenneth Hoste, and Lieven Eeckhout. Mechanistic-empirical processor performance modeling for constructing CPI stacks on real hardware. ISPASS, pages 216–226, 2011.
[12] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. A predictive performance model for superscalar processors. In *International Symposium on Microarchitecture*, pages 161–170, 2006.
[13] Tejas S Karkhanis and James E Smith. A first-order superscalar processor model. In *Computer Architecture*, pages 338–349, 2004.
[14] Tejas S Karkhanis and James E Smith. Automated design of application specific superscalar processors: an analytical approach. In *SIGARCH*, volume 35, pages 402–411, 2007.
[15] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Computer systems*, pages 125–138. ACM, 2010.
[16] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *MICRO*, pages 81–92, 2003.
[17] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *SIGOPS Operating Systems Review*, volume 40, pages 185–194, 2006.
[18] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *ACM/IEEE conference on Supercomputing*, 2007.
[19] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX*, pages 279–294, 1996.
[20] Matlab Nonlinear Models. http://www.mathworks.com/help/stats/nonlinear-regression.html.
[21] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Computer systems*, pages 139–152, 2010.
[22] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). ISCA, pages 213–224, 2012.
[23] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego vision benchmark suite. In *IISWC*, 2009.