Performance Modeling of Multithreaded Programs for Mobile Asymmetric Chip Multiprocessors

Ryan W. Moore
IBM
Email: rwmoore@us.ibm.com

Bruce R. Childers
Department of Computer Science
University of Pittsburgh
Email: childers@cs.pitt.edu

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Email: jingling@cse.unsw.edu.au

***Abstract*—Asymmetric chip multiprocessors (ACMPs) have multiple core types that are instruction-set compatible but optimized differently to trade performance and power in mobile devices. The challenge for ACMPs is to map the program to the best core type and thread count to achieve performance requirements under power constraints. This paper describes an empirical strategy, MONARCH, to automatically build estimation models that capture how a multithreaded program's performance scales with thread count and core type. We show that MONARCH's models are accurate and useful to find mappings that achieve performance goals while minimizing power.**

I. Introduction

Tablets, smartphones, and other mobile computers execute a range of workloads under varying quality-of-service (QoS) and power constraints. To satisfy these requirements, *asymmetric chip multiprocessors (ACMPs)* are emerging with multiple *core types* that use the *same instruction-set* but are optimized differently. ACMPs allow a balance between performance and power by mapping a workload to the most appropriate thread count and core type, given workload properties, performance goals and power requirements. Computationally demanding tasks (e.g., high-definition video playback) are mapped to the four main cores, while less intensive tasks (e.g., email) are mapped to the energy efficient core. ACMPs may be integrated in a system-on-a-chip (SoC) with other accelerators, which are not instruction-set compatible with the ACMP cores, such as a graphics processing unit. Mapping the application to ACMP cores is the focus of this work.

In an ACMP, a power-efficient core type has a low clock frequency and simple microarchitecture, while a performance core type has a high clock frequency and sophisticated out-of-order microarchitecture for instruction- and memory-level parallelism (ILP/MLP). Multiple cores of each type are usually available. For example, one variant of ARM's big.LITTLE has two types: a "big" Cortex-A15 for performance and a "little" Cortex-A7 for power efficiency [1]. An application binary can be executed by either type since they are instruction-set compatible. Cores are typically organized in clusters of similar types with a shared cache that is tailored to the cluster's core type (e.g., a small cache in a cluster of little cores). Caches are coherent, and clusters use the same memory subsystem.

The challenge for the operating system (OS) and compiler in an ACMP is to map the application to the right cores to meet performance goals (i.e., QoS) while minimizing power. Progress has been made on mapping for a fixed number of single-threaded programs. One approach executes a program on all core types to measure relative performance for picking the type that achieves performance and power goals [2], [3]. Recent work uses analytic and experimental models [4], [5], [6], [7]. Performance Impact Estimation is a state-of-the-art analytic model to select core type [5]. It uses performance counters gathered on one core type to predict program performance on another core type to choose between narrow in-order and wide out-of-order cores for single-threaded programs. This prior research has demonstrated the appeal of modeling to enable making complex mapping decisions with low overhead.

For single-threaded programs, the mapping problem consists of selecting one parameter, *core type*, which has been the focus of past work [8], [9], [10], [5], [6], [2], [3]. To get the full promise of ACMPs, however, multithreaded programs must also be considered. Multithreaded programming is increasingly common in mobile devices to allow using several cores for compute-oriented tasks, such as high-definition video.

Multithreaded programs make the mapping problem more complex than single-threaded programs due to an extra parameter, namely *thread count*. Techniques for single-threaded programs do not extend to multithreaded programs. In fact, thread count interacts with achieving performance under power constraints. For example, a program can be mapped with a few threads to a few big cores, or it can mapped with many threads to many small cores. Both choices may be performance equivalent, but have different power profiles. Complicating the problem is a multithreaded program's inherent scalability, ILP and MLP, which influence the best choice of core type and thread count. Because modeling is a proven technique for single-threaded programs, it is natural to consider this approach for multithreaded programs. Recent research suggests that profiling is useful to derive empirical heuristics to remap a multithreaded program by changing thread count on a conventional CMP with only one core type [11].

In this paper, we describe an empirical strategy, MONARCH, that automatically creates estimation models to guide mapping decisions for ACMPs. The models predict performance scalability curves that are parametrized by a mapping of thread count and core type. MONARCH addresses *compute-oriented* mobile multithreaded applications where dedicating a core to each thread is beneficial. Profile data is used to build the models. To minimize this data, a *projection function* transforms a scalability curve for one core type to another core type without full training. MONARCH is the first to handle both parameters (thread count and core type) for multithreaded programs executed on ACMPs. This paper contributes (1) estimation models to capture a multithreaded program's performance scalability for asymmetric core types (big and little cores); (2) a strategy (MONARCH) to create and use the models; (3) a technique to select best model and fit to data; (4) a technique to transform a model for one core type to another core type with minimal additional profile data; and, (5) an extensive evaluation of MONARCH's effectiveness.

II. Motivation

Thread count and core type affect the performance of a multithreaded program in an ACMP, influencing whether QoS is met (e.g., performance within a specified range under a power cap). The relationship of these parameters is complex, depending on both software and hardware properties.

The best thread count is affected by communication and synchronization (software scalability) and the hardware resources (core type) allocated to the program. Core type can change the relationship between program behaviors, including the ratio of time in critical versus non-critical sections, time in computation versus communication, and wait time at synchronization points. For example, using a fast core may cause a program to spend less time in critical sections, scaling better from less lock contention. Thread count also has interactions with hardware factors, including cache and memory pressure. With more threads and better scaling, there can be more resource demand on the hardware. This increased demand may be satisfied by big cores with large caches. Alternatively, the demand may even limit inherent program scaling on little cores due to small caches. Similarly, a program that does not scale may be unable to fully use hardware resources of big cores.

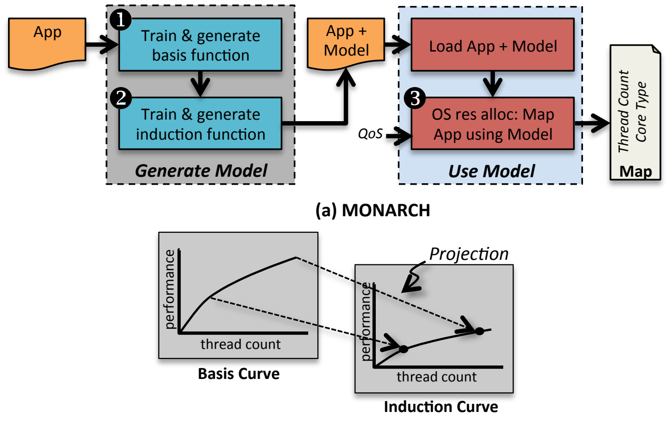
The way threads use core resources affects performance. Specifically, big cores are designed to extract instruction-level parallelism (ILP) and memory-level parallelism (MLP). For a thread to "run fast" on a big core, it needs both ILP and MLP—in fact, the thread may execute as fast on a small core as a big core, if it is memory-bound and lacks sufficient ILP to mask memory operations [5]. For a program with a large working set, a cluster of big cores with big cache might have better performance than a cluster with little cores and small cache. There can even be multiple configurations of thread count and core type that have equivalent performance, further compounding the challenge of mapping.

These factors should be balanced during mapping to an ACMP. For this purpose, we propose an approach, MONARCH, to build *estimation models* that predict how program performance scales with thread count and core type, enabling mapping to simultaneously consider both parameters. The models can be used to identify a *set* of performance equivalent configurations; external requirements on other factors, such as power consumption, can then drive selection of a *specific* configuration from the set of performance equivalent ones. For example, the models can be combined with power models to balance performance and power goals.

To ensure accuracy, MONARCH empirically constructs models from profiling data to expose the interaction between a program's properties (scalability, ILP and MLP) and core/cache architecture. From the profile data, models are generated using a new *adaptive regression analysis* that estimates performance of different configurations of thread count and core type for a program.

Similar to other profile-driven approaches, the time to profile (train) a model is a concern. To gather a full profile requires multiple program runs using different thread counts on each core type. The amount of profiling becomes problematic as the number of cores increase. Rather than gather full profiles, MONARCH builds a model of one core type using partial profiling and *projects* performance of that type to other core types. The approach collects partial profile data on the fastest core in the ACMP, which we call the *basis core*. The basis profile is augmented with selected data for each additional core type, which we call an *induction core* (or, "inductee"). A complete model is built by *projecting* performance on the basis to inductees using the profile data for the inductees.

III. Modeling Performance



(a) MONARCH

(b) Projection: Basis to inductee

Fig. 1: Overview of MONARCH

Figure 1(a) shows MONARCH's steps: 1) basis generation, 2) inductee generation, and 3) model use. The steps are fully automated, and our implementation has no additional programming burden beyond specifying data sets for profiling and invoking the tool. Steps 1 and 2 are done only once for a target device during normal software development. The model and program are then deployed in step 3 to user devices. A program's model is distributed as metadata that can be loaded by the OS for use on the user's device. If a model is not included with a program for a user's device, the OS can map the program conventionally without the model.

A. Basis

In step 1, a *basis function* is created for the fastest, largest core type. The highest speed speed setting is used when dynamic voltage frequency scaling (DVFS) is supported by the architecture. This function estimates a program's performance on the biggest core as thread count is changed. The basis function, $B(tc)$, has an input parameter, tc, for the thread count, and the function returns predicted performance of the application with that thread count on the basis core type. B must accurately estimate performance because it is used to make mapping decisions for big cores and to *approximate* scalability for inductee (little) cores. When DVFS is available, the basis is the "big" ILP/MLP core at the fastest speed setting; inductees are big or little cores at lower speed settings. Profile data about program execution on the basis core type is used to create B. The profile can be partial to reduce cost. Profiling gathers hardware counters (instructions retired and clock cycles) for multiple thread counts. From this data, MONARCH generates B using *adaptive regression analysis* to represent basis performance.

B. Induction

In step 2, functions are created to estimate scalability on the induction core types from the basis. These functions are *projections* that transform B to fit expected scalability for inductees. A projection function, $P_{inductee}$, is derived from basis training with a small amount of additional profiling on the inductee. Figure 1(b) illustrates the concept. In this figure, the basis is transformed by a projection to match scalability for the induction core. There is one projection per inductee.

To get P for each inductee, MONARCH uses regression analysis between basis and inductee types. Thus, performance is estimated for each induction core by incorporating B into P. The analysis requires training on some profiled thread counts for inductees. We call these thread counts *inflection points*. For example, there are two counts used in Figure 1(b). This process minimizes profiling because only a small amount of data is needed for induction types. The number of inflection points can be controlled to trade accuracy versus training cost.

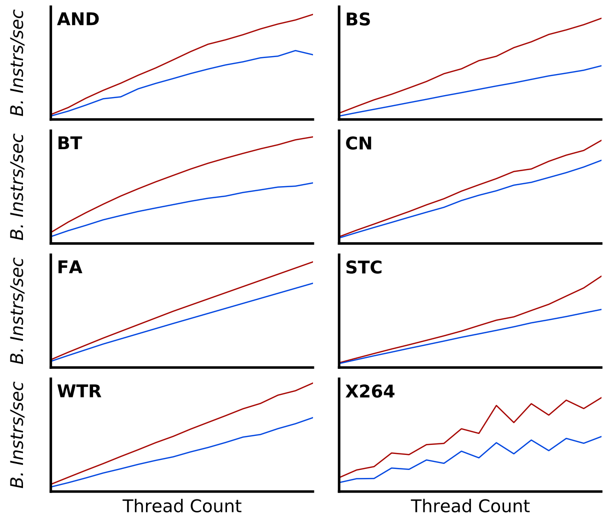


Fig. 2: Performance for 1-16 threads

This approach works because the scalability curve of an inductee often follows the basis. Figure 2 shows performance as thread count is changed for multithreaded programs on Grande and Petite core types (see Section V). The y-axis is observed performance at the thread count on the x-axis; the curves were generated by running the programs on each core type for 1 to 16 threads. Performance is reported in BIPS to account for differences in core clock speed. For our benchmarks, there is no excessive spinlock behavior that dilutes BIPS. Consequently, we use BIPS since it is easily measured and leads to similar conclusions as work throughput for our benchmarks.

The curve shapes for a program are similar across core type. In **BT**, each curve matches, except BIPS is shifted down from the bigger core to the smaller one. In **X264**, the curves have similar shapes, despite many "ups and downs" due to problems with work distribution at some thread counts. The shift from the big to little core is not simply a matter of clock speed differences, but reflects changes in architecture and interaction with program properties. For example, in **X264**, the little core smooths out local maxima and minima due to core architecture limitations to exploit ILP/MLP. For the big core, the swings are much wider. When the work is well distributed, the big core maximizes ILP/MLP extraction, achieving the highest performance. When the work is not well distributed, the distribution becomes the bottleneck, restricting the big core from taking full advantage of available ILP/MLP. Other programs have similar behavior. The figure also illustrates how many inflection points are needed. For **CN**, two points are needed to project from Grande to Petite. However, in some cases, more inflection points can be beneficial, such as **BT** where three points are useful.

The core types in an ACMP are points along a spectrum of ISA compatible pipelined microarchitectures. Thus, the scalability *trend* in the curves for different core types tend to reflect one another (i.e., they do not exactly match but show similar shapes), as illustrated in Figure 2. However, there can be local differences in absolute performance due to microarchitecture (e.g., **X264** and **STC**). The cache, in particular, can lead to larger relative differences, depending on whether the working set fits in one cache size or another. The curves in Figure 2 were gathered for cores with *different caches*, and have differences for this reason. When there are radical differences between core types (e.g., conventional pipelined core versus a GPU SIMD core), the performance curves are likely to be different. However, this is not the target for MONARCH.

Together, the basis and projection functions define an estimation model. The example, M_{BT}, below illustrates a model generated by MONARCH for **BT**:

$$M_{BT}(tc, ty) = \begin{cases} C_{Gra} \times B_{Gra}(tc) & ty = Grande, \\ C_{Pet} \times P_{Pet}(tc) & ty = Petite \end{cases}$$

$$B_{Gra}(tc) = \begin{cases} -0.064 \times tc^2 + 1.37 \times tc + \epsilon_0 & tc \in [1, 7), \\ -0.018 \times tc^2 - 0.836 \times tc + 1.644 & tc \in [7, 16) \end{cases}$$

$$P_{Pet}(tc) = 0.002 \times [B_{Gra}(tc)]^2 + 0.757 \times B_{Gra}(tc) + \epsilon_1$$

The model is parametrized by core type, ty, to select the function (B_{Grande} or P_{Petite}). The thread count parameter, tc, is used by the basis and projections to make a prediction.

Since we use BIPS in this paper as the metric, the functions are generated to predict instructions per cycle (IPC), which is scaled by clock cycles per second (clock frequency) of each core type to account for core speed differences. The basis function, B_{Grande}, in this example is piecewise with a degree two polynomial, to fit the changes in **BT**'s scalability. Because scaling on the inductees is similar to the basis, a linear function is effective for P_{Petite}.

C. Model Use

In step 3, the estimation model is used to identify performance-equivalent mappings of thread count and core type that meet QoS goals. A specific mapping is then selected to obey constraints and minimize power.

MONARCH is designed for the use case where the OS maps a program to different core counts and types due to changes in QoS at program launch. The OS uses an application's model derived by MONARCH to predict performance for different mapping choices. The mapping predicted to achieve QoS and minimize power consumption is selected. The mapping is fixed throughout program execution; the program has a single parallel region of interest that can be configured to execute with different degrees of parallelism. The mapping assigns one program thread to each core of the indicated type.

For mobile devices, MONARCH's use case covers the typical situation for a compute-oriented multithreaded program. In particular, QoS requirements can change from one program invocation to the next due to different user task priorities, battery reserves, etc. Consequently, during invocation, the program may need to be mapped in a new way to achieve the lowest power consumption with new QoS goals.

IV. Generating a Model

An estimation model has a basis and projection, as the example model, M_{BT}, illustrates. We use a regression analysis, described next, to generate these functions. For simplicity, we describe the process assuming a single speed for the big and little core types. This process can be extended to support multiple speeds by treating each core type–speed combination as an inductee, as previously suggested.

A. Adaptive Regression Analysis

Adaptive regression analysis generates the basis and projections to be accurate and minimize profiling. The analysis generates a piece-wise basis function, B, with n pieces. Each piece i in B is estimated by a polynomial of some degree, x_i, where $1 \leq i \leq n$, $x_i \leq$ MAXDEGREE, and MAXDEGREE is a fixed limit. A piece is defined by two inflection points, i.e., a range of thread counts, $[tc_{in,i}, tc_{in,i+1})$. $tc_{in,i}$ is the start inflection point for piece i.

There is a trade-off between the choice of n and x_i. A larger n implies B will better match sharp changes in scalability. However, with more pieces, there may be less available profile data (thread counts) to fit a higher degree polynomial. There may not even be "enough" profile points for x_i to build a certain degree polynomial, or the function may overfit. Further, with more pieces, more profiling is necessary to project from basis to inductees. Thus, the analysis tries to minimize n and select x_i to fit basis training data without overfitting. To find B, adaptive regression analysis varies n, $tc_{in,i}$, and x_i to generate candidate functions from which one is selected. The analysis proceeds in several steps:

1) Collect profile data for the basis core type by executing the program at different thread counts. This data may be incomplete but should uniformly reflect the expected range of thread counts for the program.
2) Select a portion of the profile data uniformly distributed among thread counts to train the basis.
3) Select n, $tc_{in,i}$, and x_i to maximize basis accuracy without overfitting, while reducing induction training. This step outputs the best B.
4) For each piece i in B, profile each induction core type at the start inflection point, $tc_{in,i}$. For the last piece, profile on the last inflection point. If a point is already profiled, then do not profile it again.
5) Using B and the induction profiles, generate projections to transform B to fit the profile data. This step outputs $P_{inductee}$ for each inductee.

Figure 3 shows the algorithm, SELECTBASIS, to build the basis (used in step 3). The algorithm inputs available profile data for a program. The profile is a database of tuples (tc, ty, hpc) that record hardware performance counters (*HPC*) for different core types (*ty*) and thread count (*tc*) combinations. We use counters to record total retired instructions, $instructions$, and clock cycles, $cycles$, of each thread. The counters are gathered for parallel portions of the program (the regions of interest, ROI). Aggregate performance is computed as the sum of the IPC values for the threads in the profiled thread count.

```
SELECTBASIS(profile)
 1 $B$ = NULL // at end, the best basis function
 2 $minPenalty = +\infty$
 3 $train$ = SELECT(profile, BASIS, TRAINAMT, UNIFORM)
 4 $test$ = SELECT(profile, BASIS, ALL)
 5 // iterate over pieces, splits of pieces and degrees
 6 for $n = 1$ to MAXSPLITS
 7   $splits$ = SPLITPIECES($train, n$) // inflection points
 8   $D = \{1, \ldots, \text{MAXDEGREES}\}^n$ // Cartesian exponentiation
 9   for $split$ in $splits$
10     for $degrees$ in $D$
11       $candidate$ = GENBASIS($train$, $split$, $degrees$)
12       if $candidate \neq$ NULL
13         $E$ = COMPUTEMSE($candidate$, $test$)
14         $C$ = NUMMISSINGPOINTS($split$, $profile$)
15         $penalty = E^2 \times (C + 1)$ // penalty function
16         if $penalty < minPenalty$
17           $minPenalty = penalty$
18           $B = candidate$
19 return $B$
```

Fig. 3: Algorithm to adapt and select basis function

Regression extrapolates performance scalability of a program for the basis core type from the profile data. On line 3 in the figure, profile data for the basis core is extracted as *train data* for regression. This portion of the basis profile (TRAINAMT) is selected to be uniformly distributed (UNIFORM) across available thread counts. To evaluate fitness of a basis candidate function, the full basis profile data is extracted as *test data* on line 4. The constant MAXTC is the largest thread count allowed for the application.

The algorithm traverses the number of pieces (line 6), inflection points (line 9), and polynomial degree (line 10). The assignment of thread counts as inflection points defines the start and end points for the n pieces. The assignments, which we call "splits", are determined by SPLITPIECES on line 7. Similarly, all combinations of polynomial degrees for the splits are computed on line 8. The maximum number of pieces and degrees have limits (MAXSPLITS and MAXDEGREES). to constrain the iteration space and avoid overfitting to the training data. A small maximum degree polynomial with a moderate number of pieces works well in practice; we we set MAXDEGREE$=3$ and MAXSPLITS$=3$ to allow regression flexibility to pick a complex function without overfitting while limiting the search. These constants can be tuned to trade prediction accuracy and search cost.

On line 11, GENBASIS creates the candidate basis using least square regression for each piece. Because some parameter combinations for GENBASIS can lead to an invalid candidate basis, GENBASIS may return NULL (no basis generated). This situation happens when there is not enough points in the train data to allow regression on the function defined by the parameters. For instance, a large number of pieces (n) and high-degree polynomials for each piece ($degrees$) may result in an invalid function.

A candidate function's fitness is evaluated on lines 13 to 18 by determining a penalty of relative benefit to cost (line 15). Benefit is measured indirectly as mean squared error (MSE) on the test data; a larger MSE implies worse accuracy, and therefore, less benefit. Cost is how many inflection points in the candidate basis function have not been profiled. The thread count for each unseen inflection point on each inductee core is executed to get data for projection, and thus, the unseen points impose cost. In the penalty calculation, the error (E) is weighed more heavily than cost (C).

Once the basis function is selected, projections can be generated (step 5). A projection function is created for each induction by "reshaping" the basis to fit performance on an inductee. Similar to basis generation, inductee projections are generated using LSQ. A single piece is used and only polynomial degree is changed. We allow the polynomial degree to be selected to capture a potentially complex relationship between basis and induction performance.

To generate a projection, profile data is extracted for an induction core. The data are tuples of thread count and *HPC* values. B is used to predict the performance of the basis core type, which is mapped by regression to the profile data for the induction core type. The mapping gives the necessary (x, y) data required by regression. For a given thread count, B is used to compute aggregate IPC for the basis as x and the profile data is used to compute aggregate IPC for the inductee as y. Using the $(IPC_{basis}, IPC_{inductee})$ data, a high degree polynomial is tried first for the projection. If this degree cannot be used (due to missing data), then the next lowest degree is tried until a function is generated.

V. Evaluation

A. Methodology

We used Sniper [13] to simulate a many-core ACMP for future mobile computers (similar to ARM big.LITTLE), which we call **MOBI**. Table I shows **MOBI**'s architecture parameters. There are 16 cores of Grande and Petite types. The microarchitecture of each type differs in ILP and MLP aggressiveness. Cores of the same type are clustered in groups of 4 to share a last-level cache (LLC). A Grande cluster has a large LLC and a Petite cluster has a small LLC. Caches are kept coherent between clusters. We use 16 cores to stress MONARCH. With more cores, there are more configuration choices, exposing increased variability in performance that must be predicted correctly. Due to long simulation times, we consider only one speed per core type.

TABLE I: Architecture parameters for **MOBI**

Grande	1.86GHz,Dispatch=3,Window=64,LSQ=10
Petite	1.36GHz,Dispatch=2,Window=32,LSQ=2
Processor	16 cores of each type, 4 cores per cluster
L1 cache	private I&D, 32 KB, 4-way, 64 B block
Grande L2 cache	shared 2 MB, 8-way, 64 B block, 8-cycle hit
Petite L2 cache	shared 512 KB, 8-way, 64 B block, 8-cycle hit
L2 sharing	Each cluster of 4 cores shares one L2 cache
DRAM	45 ns DRAM, 1 controller w/1 channel

We used performance-oriented multithreaded programs from PARSEC and SPLASH-2 to cover a spectrum of MLP, ILP and scaling, including blackscholes (BS), bodytrack (BT), canneal (CN), fluidanimate (FA), streamcluster (STC), water (WTR), and x264. We also used Andersen's parallel analysis [14] for graph traversal (AS). The benchmarks are run to completion for 1 to 16 threads. One data set (*simmedium*) was used to generate the models, and another data set (*simlarge*) was used for evaluation. The models were generated using 50% of the thread counts (uniformly distributed between 1 and 16) on *simmedium*. Clang 3.2 (LLVM) with -O3 optimization was used as the compiler.

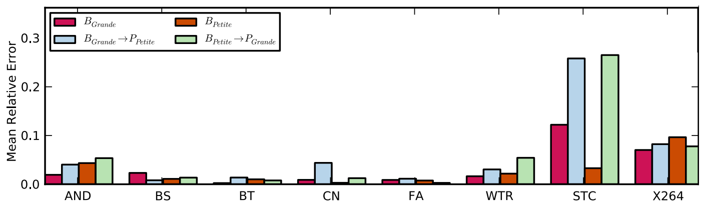


Fig. 4: Model accuracy (MRE for basis and projections)

To evaluate parameter choices for MONARCH, we use *mean relative error (MRE)*. This metric is the difference between a model prediction and actual performance reported by the simulator on *simlarge*. The parameter choice that minimizes MRE should be chosen since MRE reflects model quality for a previously unseen input data set. By selecting a parameter this way, a more accurate model will be constructed, which should also be better at predicting *trends* between core types to guide mapping. We also evaluate how well trends are predicted by examining the utility of the models from MONARCH in selecting mappings that achieve QoS with minimal power cost. These mappings are compared against an oracle that always picks the best mapping.

B. Model Accuracy

Figure 4 shows mean relative error for basis and projection functions, depending on the core type used as the basis. B_{Grande} and B_{Petite} give the error when Grande or Petite is used as the basis. Per program the choice of basis has an impact. For example, on **STC**, B_{Petite} has minimal error (0.035), while B_{Grande} has more error (0.14) because **STC**'s behavior varies between core types. On Petite, it scales linearly, while on Grande it scales with an upward curve. Curves are harder to predict than lines, resulting in the increased error. In **X264**'s case, the basis for Petite has higher error than the basis for Grande (0.10 versus 0.07). **X264**'s scalability (Figure 2) has many local maxima. On Petite the maxima are flatter due to lower performance. Therefore, MONARCH used a degree-2 polynomial to reduce inductee profiling. Despite these differences, the choice of basis actually has negligible impact. Because the fastest core (Grande), reduces profiling time, we use it as the basis.

Figure 4 also shows projection accuracy. $B_{Grande} \to P_{Petite}$ is the error when Grande is projected to Petite and $B_{Petite} \to P_{Grande}$ is the error when Petite is projected to Grande. The best basis and inductee depends on the program. Using the basis and projection together can result in less error than training on a single core type. For example, **AND** and **BS** for $B_{Grande} \to P_{Petite}$ have less relative error than B_{Petite} from capturing trends across core types and no overfitting. As expected, projection often has more error than the basis (**AND**, **STC**, **WTR** and **X264**).

STC has the most error in projecting between cores with approximately 0.28 MRE for both projections. From examination of execution behavior and scalability for the full runs of this program (Figure 2), we found that this error is due to differences in the curves at higher thread counts. On Grande, performance has an increasing upward swing, while Petite does not, which proves more difficult for regression to capture with few inflection points. In comparison, **X264** has better accuracy, despite "ups and downs" in scalability (Figure 2). For both core types, the same local minima and maxima appear, although the relative difference between them differs with

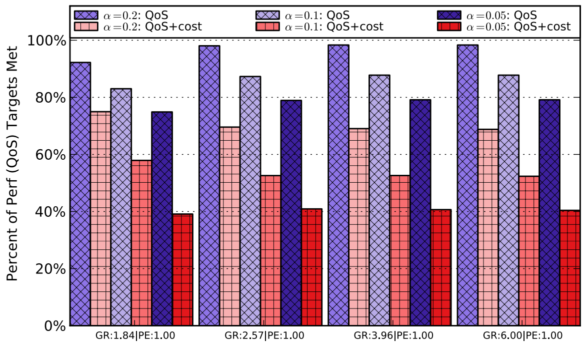


Fig. 5: Achieving QoS and cost

core type. Because the local minima and maxima appear at the same thread counts for both core types, projection can compensate for the relative differences. For this benchmark, adaptive regression analysis rejects the initial basis function generated due to negative predictions at low thread counts: An indication of overfitting. The check causes the analysis to try a different function that does better at these counts.

C. Using Models

Because MRE is useful only for comparing models, we also evaluate MONARCH's models to predict performance for different QoS goals and power costs. The mapping objective is to select a configuration that achieves a QoS target, while minimizing power. For evaluation, we use the models to select the mapping that is *predicted* to meet QoS and minimize cost. We use all actual performance values from the *simlarge* runs for all benchmarks at all thread counts as QoS targets.

To specify QoS, we give a target BIPS, Q_{target}, and a threshold, α. To test many targets, we use all BIPS values for all benchmarks executed in all configurations of thread count and core type. The BIPS values are collected with full simulation on *simlarge*. The models are trained on *simmedium*. We say that a mapping "satisfies" QoS when a program's actual performance with *simlarge* under the mapping meets or exceeds $(1 - \alpha) \times Q_{target}$. Using MONARCH's models, a configuration is selected that best achieves QoS and power. This is done for every program and QoS target.

We assume cost is minimized when it falls within α of the lowest cost mapping. α is varied to adjust QoS and cost strictness ($\alpha \in \{0.2, 0.1, 0.05\}$). Four power cost ratios of Grande to Petite core types are used: 1.84, 2.57, 3.96, and 6.00. The ratio is the increased expense of using Grande over Petite. The ratios are derived from core frequency, microarchitecture, and supply voltage reported in the literature for big and little cores, including ARM big.LITTLE. To predict power cost, we use a linear model parametrized by number of cores (threads) and core type: $\textsc{Cost}(tc, ty) = tc \times \text{COST}(ty)$.

Figure 5 summarizes the result. The bars "QoS" show the percentage of mappings for each power cost ratio that satisfied QoS over all benchmarks. At $\alpha = 0.2$, the mappings chosen with the models typically satisfy QoS. On average over all power ratios, 97% of the mappings satisfied QoS. This high percentage happens because the models have MREs that are low enough to fall within the tolerance induced by α.

As α becomes stricter, the percentage of targets achieved declines. The average across all power cost ratios for $\alpha = 0.2$ decreases from 97% to 85% at $\alpha = 0.1$. At the strictest setting ($\alpha = 0.05$), 77% of the QoS targets are satisfied, down from 97%. In this case, model error is evident. Furthermore, this target is challenging: There are far fewer thread count and core type configurations that satisfy QoS with minimal cost. From examining the detailed results, we found the percentage reduction for $\alpha = 0.05$ is due to the models *overestimating* performance in a few cases. As a consequence, a configuration with less power cost than the optimal one is chosen, under the inaccurate estimation that performance for that configuration is above QoS. Our observations hold across all power cost ratios, except at the narrowest one (1.84) where there is less power difference between configurations.

The percentage of configurations selected satisfying QoS and minimizing power are shown by the bars "QoS+cost". When cost is considered, the percentage of targets achieved is less than QoS only due to two reasons. First, performance is *underestimated* in several cases, which causes more power-hungry configuration than needed to be selected. In turn, this leads to cost falling outside of α of the optimal mapping. Second, with strict α, there are few configurations within α in power and performance of optimal. Indeed, at $\alpha = 0.05$, only one configuration (optimal) usually satisfies the QoS and power cost target. Thus, percentage of QoS+cost targets achieved drops from an average of 75% (across all power cost ratios) at $\alpha = 0.2$ to 40% at $\alpha = 0.05$. A larger power ratio exacerbates the impact of the stricter α. For example, at a ratio of 1.84 and $\alpha = 0.2$, QoS+cost is achieved for 78% of the targets. This drops to 70% at a 6.0 ratio.

D. Comparison to PIE

To put the results for MONARCH in context to other approaches, we compared it to PIE [5], which predicts the performance of a running program on one core type using hardware performance counters from another core type. Because PIE was not designed for multithreaded programs, we extended it to aggregate performance across multiple threads. The extension, "MultiPie", extends PIE's equations that predict performance of the big core (Grande) from the little core (Petite) to consider a thread count, tc. We extended the equations in the original PIE paper to aggregate performance over multiple threads.

MultiPie determines scalability curves for Petite and Grande in two steps. First, it profiles the program at all thread counts $1 \leq tc \leq \textsc{MaxTC}$ using Petite with *simmedium*. This profile is over all thread counts, while MONARCH only partially profiles on any one core type. The full profile determines the scalability curve for Petite. Second, the scalability curve for Grande is determined from the profile data for Petite by applying the modified PIE equations. In both MultiPie and MONARCH, actual evaluation is done with *simlarge*.

We found that MultiPie can lead to inaccurate predictions due to overestimating performance (result not shown for brevity). Yet, despite potential inaccuracy, it usually reflects the trend in scalability (similar to MONARCH). We now examine whether this capability is sufficient to achieve QoS under power constraints. Using the same methodology as Section V-C, we determined how often MultiPie achieves QoS at minimum power. Figure 6 shows the results. MultiPie does a good job for QoS only, selecting configurations performing similarly or better than MONARCH's model. However, due to underestimation of performance, the MultiPie configurations are overprovisioned to meet QoS, leading to high power

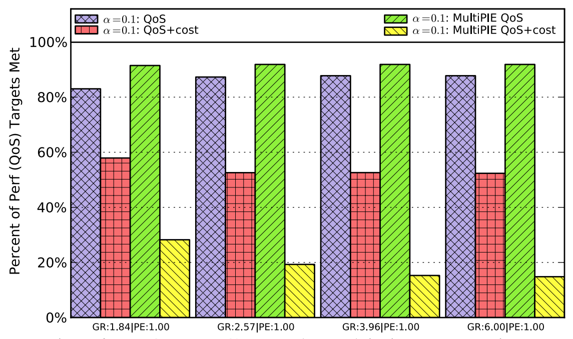


Fig. 6: MONARCH and MultiPie comparison

cost, as shown by the percentages for "QoS+cost". Because MultiPie underestimates performance, the more powerful core type (Grande) is usually selected to meet the QoS target. This powerful core has a high clock speed and advanced microarchitecture, easily meeting QoS with fewer threads than the less capable core. The selection of the big Grande core type comes at the cost of more power consumption. Thus, MultiPie's performance underestimation manifests in the QoS+cost results.

The insight from these results is that accuracy in both absolute performance and in predicting performance trends are important to mapping, particularly when a power constraint is imposed. We conclude that the models generated by MONARCH achieve an effective balance between these two aspects.

VI. Related Work

ACMPs can trade performance and power using multiple instruction-set compatible core types [1], [3]. New processors are adopting this architecture to allow a close mapping between workload and hardware capability. The challenge is to map the workload to the right resources. Researchers have shown the importance of mapping applications for QoS with co-runners in CMPs [12]. Co-runners introduce runtime asymmetry, which must be managed. This work addressed workloads made of single-threaded programs for symmetric CMPs. Wang et al. studied remapping for multithreaded programs using models derived with machine learning [11]. This approach built a program-agnostic model to determine thread count, but it did not consider core type.

Techniques have been proposed to allocate ACMP resources to an *a priori* amount of work, i.e., the mapping is only core type. Several approaches integrate performance estimation and online monitoring to predict performance on core types, including direct [3] and indirect measurement [15], [5] of run-time and power. These approaches focused on multiprogrammed workloads. Our approach builds an independent model of multithreaded program scalability.

Statistical models, similar to our approach, have been used to predict performance [9], [10], [16], [4]. Using HPC, analytic models have been developed to guide migration between core types [5]. A program's "signature" on architectural resources can be used to select models for different core types [17]. Recent work extended PIE with compiler assistance to estimate cycles per instruction [4]. Unlike MONARCH, these efforts did not address scalability and core bias of multithreaded programs. MONARCH uses regression to model scalability of thread count and core type mapping. The approach adapts underlying functions to fit program behavior. Minimizing training is also a key aspect of MONARCH.

VII. Conclusion

ACMPs have multiple core types to trade performance and power. This paper describes MONARCH, an empirical strategy, to construct estimation models that predict a multithreaded program's performance for different thread count and core type mappings. The approach automatically derives an estimation model with low training cost. MONARCH is the first technique to consider both thread count and core type to predict program scalability for ACMPs used in mobile devices. Our results show that model predictions are accurate and beneficial for mapping multithreaded programs to asymmetric cores.

VIII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0811352, CNS-1012070 and CCF-1422331.

References

[1] P. Greenhalgh, "Big.LITTLE Processing with ARM CortexTM-A15 and Cortex-A7," ARM, ARM, 2011.
[2] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Conf. on Computing Frontiers*, 2006, pp. 29–40.
[3] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Int'l. Symp. on Microarchitecture*, 2003, pp. 81–92.
[4] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Int'l. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2013, pp. 15:1–15:10.
[5] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Int'l. Symp. on Computer Architecture*, 2012, pp. 213–224.
[6] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu, "An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs," in *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2013, pp. 63–72.
[7] S. Srinivasan, L. Zhao, R. Illikkal, and R. Iyer, "Efficient interaction between OS and architecture in heterogeneous platforms," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 62–72, 2011.
[8] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *European Conf. on Computer Systems*, 2010, pp. 125–138.
[9] O. Khan and S. Kundu, "A self-adaptive scheduler for asymmetric multi-cores," in *Great Lakes Symp. on VLSI*, 2010, pp. 397–400.
[10] J. Cong and B. Yuan, "Energy-efficient scheduling on heterogeneous multi-core architectures," in *Int'l. Symp. on Low Power Elect. and Design*, 2012, pp. 345–350.
[11] Z. Wang, M. F. P. O'Boyle, and M. K. Emani, "Smart, adaptive mapping of parallelism in the presence of external workload," in *Int'l. Symp. on Code Generation and Optimization*, 2013, pp. 1–10.
[12] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Int'l. Symp. on Microarchitecture*, 2011, pp. 248–259.
[13] "Sniper multi-core simulator," http://snipersim.org, 2013.
[14] Y. Su, D. Ye, and J. Xue, "Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems," in *Int'l. Conf. on High Performance Computing*, 2013, pp. 149–158.
[15] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *European Conf. on Computer Systems*, 2010, pp. 139–152.
[16] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Design Automation Conf.*, ser. DAC, 2009, pp. 927–930.
[17] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: A scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.