CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors

Jyothi Krishna V S, Shankar Balachandran, and Rupesh Nasre

Abstract—Heterogeneous Multiprocessors (HMPs) are popular due to their energy efficiency over Symmetric Multicore Processors (SMPs). Asymmetric Multicore Processors (AMPs) are a special case of HMPs where different kinds of cores share the same instruction set, but offer different power-performance trade-offs. Due to the computational-power difference between these cores, finding an optimal hardware configuration for executing a given parallel program is quite challenging. An inherent difficulty in this problem stems from the fact that the original program is written for SMPs. This challenge is exacerbated by the interplay of several configuration parameters that are allowed to be changed in AMPs. In this work, we propose a probabilistic method named CHOAMP to choose the *best* available hardware configuration for a given parallel program. Selection of a configuration is guided by a user-provided *run-time* property such as energy-delay-product (EDP) and CHOAMP aspires to optimize the property in choosing a configuration. The core part of our probabilistic method relies on identifying the behavior of various program constructs in different classes of CPU cores in the AMP, and how it influences the cost function of choice. We implement the proposed technique in a compiler which automatically transforms a code optimized for SMP to run efficiently over an AMP, eliding requirement of any user annotations. CHOAMP transforms the same source program for different hardware configurations based on different user requirement. We evaluate the efficiency of our method for three different run-time properties: execution time, energy consumption and EDP, in NAS Parallel Benchmarks for OpenMP. Our experimental evaluation shows that CHOAMP achieves an average of 65%, 28% and 57% improvement over baseline HMP scheduling while optimizing for energy, execution time, and EDP respectively.

Index Terms—Asymmetric Multicore Processors, big.LITTLE, Optimal Hardware Configuration, Scheduling, Compiler Optimization

1 Introduction

Advancement in processor technology with symmetric multicore processor (SMP) scaling has resulted in large increase in power consumption per unit of chip area. The increasing importance of embedded systems has further emphasized the need for power-efficient computations while designing the CPUs. The quest to reduce power consumption has led to the fabrication of asymmetric multicore processors (AMPs). big.LITTLE [1] from ARM and Tegra-E1 [2] from NVIDIA are examples of AMPs.

AMPs are characterized by multiple types of cores. However, unlike in CPU-GPU heterogeneous multicore processors, all cores in AMPs generally implement the same instruction set architecture (ISA) and the processes can easily be migrated from one core type to another (similar to SMP).[1] In AMPs, a core type differs from another in terms of frequency bandwidth, micro-architecture, and cache size, among other aspects. These design choices are made based on power and performance constraints. The cores in AMPs can be logically divided into memory-centric cores (low-power cores) and compute-centric cores (high-power cores). The memory-centric cores have lower cache miss penalties (based on CPU cycles), lower static and dynamic power ratings, but at a cost of lower computational efficiency when compared to the compute-centric cores.

When the AMPs are designed with a considerable frequency difference between the powerful brawny cores and weaker wimpy cores, the selection of the optimal type of cores turns out to be a trade-off between execution time and power consumption [3]. One can choose the brawny cores to run the threads faster and the weaker cores to save on energy consumption. However, such a clear-cut favorite between these two cores is absent when a composite cost function such as Energy-Delay-Product (EDP) [4] needs to be optimized for arbitrary programs. The issue gets exacerbated due to variation in the frequencies the cores of a type are clocked at. For instance, as the frequency difference between different types of cores decreases, the weaker (simpler) cores exhibit better execution time than the powerful cores (with complex hardware optimizations) for heavily memory-bound programs [5]. In a similar manner, a powerful core may consume less energy than a weaker core depending upon their operating frequencies and the program characteristics. In a system with more than two types of cores, where the computational power difference between two adjacent classes of cores is not substantial, the choice of an optimal core type becomes obscure. The optimal solution point can easily shift to the next core type with slight changes in the input program. Our work in this article deals with identifying such an optimal configuration for a program by looking at the program characteristics, to be executed on a hardware with multiple core-types with varying execution and power behavior. The notion of optimality is also dictated by the user.

The application performance in a multicore environment degrades considerably in the presence of intensive inter-thread communication. In such cases, a power-efficient solution may discard using brawny (such as big) cores in favour of a more power-efficient core type (such as LITTLE) with

- *Jyothi Krishna V S is with IIT Madras. (jkrishna@cse.iitm.ac.in)*
- *Shankar Balachandran is with Intel Labs, Banglore.*
- *Rupesh Nasre is with IIT Madras.*

1. Heterogeneous-ISA AMPs also exist. But we do not deal with those in this work.

minimal or no performance penalty. On one extreme, for strong CPU-bound programs, the performance of big cores is twice as that of LITTLE cores [6]. On the other extreme, for memory-bound programs, it is now known that a large number of cores suffers from poor performance, with the big cores incurring a higher penalty. For such programs, an optimal number of cores is present [6] for each of the core types for which we can obtain an optimal execution time.

We anticipate most real-world programs to have a complex mix of characteristics (compute, memory, synchronization). Therefore, an optimal configuration for such programs would need to make a judicious use of cores of varying power (in our experimental setup, big and LITTLE cores). This motivates us to holistically model scheduling of parallel tasks, their assignments to various core-types, and configuration of each core. We illustrate that parallel systems exhibit wide variation in throughput based on the scheduling, and how our model helps achieve near-optimal schedule and configuration. While we illustrate the efficacy of our model using big.LITTLE system, we emphasize that our model is more general and applies to AMP configurations having capability of several different types of cores.

In this paper, we propose a holistic system named CHOAMP which makes use of a compiler, program analysis and transformation, regression analysis, and a runtime scheduler. A close-knit interaction of these system components helps us overcome the challenges arising due to unknown variable values during compilation, and unknown execution paths in the control-flow graph, and achieve a more precise modeling of the parallel program. Together, the components lead to a scheduling with appropriate core-configurations leading to near-optimal performance (based on the user-driven cost-function).

This article makes the following contributions:

- We design a Predictor that can predict the optimal core-configuration, optimizing a user-provided runtime cost-function for running OpenMP programs. This involves an in-depth learning of all possible hardware configurations and its response to parallel programming elements. We have devised a supervised learning system where the runtime behavior is captured using different regression functions. We then use this knowledge-base to analyze the input program to predict its runtime behavior.
- We build a compiler to transform OpenMP programs to run with the predicted hardware configuration.
- We present an in-depth experimental study of thread-scheduling in asymmetric multiprocessing environment using a suite of NAS Parallel Benchmarks [7]. We observe that the predicted hardware configurations are mostly better than the base case and very often close to the best. We also show CHOAMP's ability to work well with dynamic scheduling using the HMP [8] scheduler for big.LITTLE and Compiler Enhanced Scheduling [9], a custom scheduler for OpenMP for AMPs.

The sequel is organized as follows. Section 2 briefly introduces HMP scheduling, CES and OpenMP. Section 3 explains our micro-benchmarking framework. Section 4 provides the theory behind CHOAMP. Section 5 provides detailed explanation of our proposed methodology. Section 6 discusses the implementation details including the compiler, regression analysis and the transformation, and illustrates the effectiveness of our approach using experimental evidence. Section 7 compares and contrasts with the related work, and Section 8 concludes.

TABLE 1: Comparison of big and LITTLE cores

	LITTLE core	**big core**
Core Types	Cortex-A7, A35, A53	Cortex-A15, A17, A57, A72
Pipeline	8-10 stage, in-order	15-24 stage, out-of-order
Frequency	400 - 1400 MHz (A7)	800 - 2000 MHz (A15)
Speed	1.9 DMIPS [10]	3.5-4.01 DMIPS
L1 cache size	32 KB	32-48 KB
ISA	Thumb-2	

TABLE 2: Peak and idle power consumption for different frequencies of big and LITTLE in Watts.

Cortex A15 (big)	**Single Core**		**Four Cores**	
Frequency	**Idle**	**Peak**	**Idle**	**Peak**
2.0 GHz	0.95	2.40	1.15	5.28
1.8 GHz	0.70	1.65	0.70	3.50
1.6 GHz	0.51	1.40	0.53	2.80
1.4 GHz	0.38	0.85	0.40	2.30
1.2 GHz	0.30	0.70	0.32	1.80
Cortex A7 (LITTLE)	**Single Core**		**Four Cores**	
Frequency	**Idle**	**Peak**	**Idle**	**Peak**
1.4 GHz	0.20	0.50	0.22	1.40

2 BACKGROUND

In this section we introduce big.LITTLE architecture, existing scheduling techniques, and OpenMP API.

2.1 big.LITTLE

In big.LITTLE from ARM, the powerful cores are called the big (brawny) cores and the weaker power-efficient cores are called the LITTLE (wimpy) cores. The number of big and LITTLE cores may vary across implementations. A comparison of big and LITTLE cores is given in Table 1.

Table 2 shows the idle and peak power consumption of big.LITTLE cores obtained by running several micro-benchmarks (*cf.* Section 3) on Odroid-XU3 board [11]. The power consumption readings are from the onboard power sensors. We collect the power consumption of a single core by switching off the other cores in the cluster. We observe that even for similar frequencies, the power consumption of big is much higher than that of LITTLE. big cores also perform better compared to LITTLE in case of compute-intensive programs while running at the same frequency [5].

Earlier implementations of big.LITTLE relied on cluster switching [12], which disallowed simultaneous usage of big and LITTLE cores. Global Task Scheduling / Heterogeneous Multicore Processor (HMP) scheduling [8] addressed this issue, allowing all the cores to be used at the same time. HMP scheduling is now integrated into the Linux kernel.

2.2 HMP Scheduling

HMP scheduler is a dynamic scheduler designed for big.LITTLE systems based on the core CPU-cycle utilization by a thread. Thus, threads with high CPU utilization are

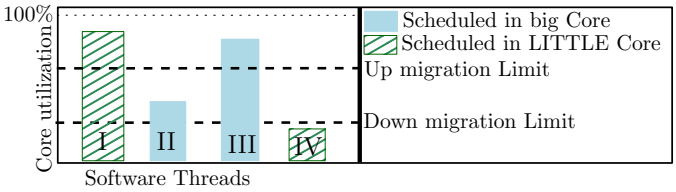


Fig. 1: Example of HMP dynamic scheduling.

executed on big cores, while others in LITTLE cores. A thread that is currently scheduled in LITTLE thread can be *up-migrated* to big core if the thread has a higher CPU utilization than the up-migration limit. Similarly, a thread that is currently scheduled in big can be *down-migrated* if the current CPU utilization of the thread is lower than the down-migration limit. The scheduler is invoked periodically to keep high-performing threads in big.

Figure 1 shows an example HMP scheduling system depicting percentage utilization of CPU cycles by different threads in big and LITTLE. Thread I is showing high CPU utilization (beyond a threshold) in a LITTLE core and the thread can benefit and run faster if it is scheduled in a more powerful core. Hence, the thread is scheduled for up-migration. On the contrary, thread II is currently scheduled in a big core with a low CPU utilization (below another threshold). Down-migrating to a LITTLE core can reduce energy consumption without much increase in its execution time. The HMP scheduling is integrated into the Completely Fair Scheduler (CFS) [13] in the Linux kernel.

2.3 CES

Compiler Enhanced Scheduling (CES) [9] is a *static* scheduler for OpenMP programs in AMPs. CES optimizes execution time by dividing the workload in the parallel region based on the task workload and the processing power of the core. The compiler estimates the relative speed-up of each core type for executing the threads and divides the parallel tasks accordingly such that the execution time of each thread in a parallel region is normalized. The big cores usually receive larger chunks of work compared to the LITTLE cores. There is a dynamic part to CES which attenuates the imbalance in the initial work distribution which might have sneaked in due to the compile-time assumptions.

2.4 OpenMP API

We provide an introduction to a subset of OpenMP parallel program features that are relevant to this paper. OpenMP API is designed for shared memory parallelism in C, C++ and FORTRAN. Our focus is on C and C++ programs. Special *#pragma* directives are used by the programmer to specify the intended OpenMP program behaviour to the compiler. A parallel program written using OpenMP API can have interleaved sequential and parallel regions. A parallel region, written inside *#pragma omp parallel* (omp_parallel), creates a team of threads, which may run in parallel to execute the code defined. The number of threads (`N_THREADS`) in a team can be set using environment variables or using OpenMP library calls (`set_omp_num_thread()`). For better performance, a parallel program needs to exhibit more parallelism and lesser synchronization.

The master thread is the initial thread which creates the team on encountering the omp_parallel region. In a team, thread id 0 is the master thread, while the remaining threads with ids 1..`N_THREADS`-1 are non-master threads. The code specified under omp_master (*#pragma omp master*) is executed only by the master. The remaining threads in the team skip omp_master and continue executing the parallel region. The parallel region is executed by master too.

Threads may have private variables, and the threads can communicate via shared variables. Shared variables can be synced to main memory using omp_flush (*#pragma omp flush*). The set of variables to be synced can be optionally specified as parameters to omp_flush. If the set is not specified, the entire local cache of the thread is synced with main memory before proceeding. A barrier_construct (*#pragma omp barrier*) is used to synchronize among all the team threads. Each thread waits at the barrier until all the other team threads reach the barrier. Barriers have implicit flush operations on all the threads. There is an implicit barrier at the end of every omp_parallel construct.

Apart from these, OpenMP provides omp_atomic (*#pragma omp atomic*) and omp_critical (*#pargma omp critical*) for updating and accessing shared variables without incurring a data race. omp_atomic is used to atomically read, write or update a single shared scalar variable at a time. The omp_atomic is applicable to only the immediately following statement, and a few binary operations are allowed to be executed using this construct. An omp_critical block is executed by all the team threads in a mutually exclusive manner. All unnamed omp_criticals are considered identical and maximum one thread can be in an omp_critical region at any time. Unlike in the case of atomics, there is no restriction on the operations or the data types allowed inside a critical region.

Work-sharing constructs define units of work distributed across team threads. The work sharing constructs have an implicit barrier at the end, which can be removed using *nowait* clause. There are three work-sharing constructs available in OpenMP C/C++. First, omp_for (*#pragma omp for*) implements a parallel for loop in `N_ITRS` iterations, where each iteration is executed once by one of the threads. The iterations can be executed in parallel with other iterations. The scheduling pattern of iterations to threads may be specified using static, dynamic, or guided clauses. In static scheduling, each thread is assigned equal-sized chunks of iterations in a round-robin fashion, until there are no iterations left. The size of a chunk, unless specified, is `N_ITRS / N_THREADS`. In dynamic scheduling, each thread is supplied with a chunk of iterations on demand, until there are no iterations left. Guided scheduling is similar to dynamic scheduling except that the chunk size (whose initial value can be specified using the optional parameter) starts off large and decreases in later steps to improve load-balancing. Dynamic and guided schedulings are often used for better load-balancing but have higher overheads compared to static scheduling. The chunk size plays an important role in the performance of dynamic and guided scheduling. If chunk size is very small, the scheduling overhead increases. On the other hand, if chunk size is very large, the load-imbalance is high. Therefore, a judicious setting of chunk size often affects parallel performance.

```
L#
1  #define M 50000
2  int f(int *s, int A[], int cumSum[], int L)
3  {
4   int MAX = 0, localSum = 0, temp = L/128;
5   int N = M - temp;
6   #pragma omp parallel
7   { int i, j;
8    #pragma omp for reduction(+:localSum)
9    for(i = 0; i<N ; i++) {
10     localSum += A[i];
11     cumSum[i] = 0;
12     for(j=0;j<N;j++) {
13       if(j<=i)  cumSum[i] += A[j];
14     }
15    #pragma omp critical
16     {  if(MAX < A[i]) MAX = A[i]; }
17   }
18  }
19  return MAX;
20 }
```

Fig. 2: Running example: An OpenMP code to find the maximum element and the cumulative sum of an array.

omp_reduction (reduction(identifier :list)), is often used with omp_for to update a shared scalar variable on an associative operation. The identifier is used to identify the reduction operation and list contains one or more variables on which the reduction operation might be applied inside the region. We can have multiple omp_reductions, one for each operation in a single omp_for.

In omp_sections (#*pragma omp sections*), we have multiple sections each encapsulated by #*pragma omp section* (single_omp_section), executed by one of the threads in the team. Section scheduling is arbitrary. A single_omp_section once scheduled to a thread will be executed by the thread.

omp_single (#*pragma omp single*) contains a block of code that is executed by only one of the randomly-selected team threads. The rest of the threads jump to the end of the omp_single block and wait at the implicit barrier at the end of the construct, unless *nowait* is specified.

Figure 2 provides an OpenMP code which computes the cumulative sum and the maximum element in the array A. Inside the omp_parallel region (from Line# 6 to Line# 18) we have an omp_for (from Line# 8 to Line# 17) which iterates over A. Location `cumSum[i]` stores the cumulative sum of A's elements from position 0 to i. Variable `MAX` stores the maximum element of A. To avoid race conditions, `MAX` is updated inside a critical region (from Line# 15 to Line# 16). Variable `localSum` stores the sum of A's elements and is updated using omp_reduction in Line# 10.

3 MICRO-BENCHMARKS

We now describe the generation of micro-benchmarks and training of the prediction model. Our micro-benchmark framework is inspired by EigenBench [14], a work on benchmarking different Transactional Memory (TM) systems.

EigenBench describes a tool (EigenTool) which uses a set of micro-benchmarks to analyze the efficiency of a TM system. The micro-benchmarks are designed to test a set of orthogonal application characteristics that form a basis for transactional behaviour. Precise extraction of the Eigen characteristics is a key step in designing the EigenTool.

3.1 Feature Identification

In CHOAMP, we identify a set of parallel program characteristics in OpenMP (from hereon termed as prime features), whose existence and intensity variations affect the runtime behaviour of the program in an AMP environment. The choice of prime features depends upon both the parallel programming platform (OpenMP) constructs and the hardware characteristics (in general, HMP; in our experiments, big and LITTLE). Similar to EigenTool, we use a collection of set of micro-benchmarks (one for each prime feature selected) to carefully study the behaviour of all the available hardware configurations and how each of the runtime cost functions of choice is affected. This helps us identify the efficiency of different hardware configurations of big.LITTLE. We use the data collected from these micro-benchmark runs to train a predictor, which is used to predict the optimal configuration O_{bl} for a given parallel program.

Each micro-benchmark is a valid OpenMP program generated with a known *intensity vector (ivector)* of the prime feature set. Each value in ivector represents the intensity of a prime feature as a fraction of the total ALU operations. By varying the value of a single feature in ivector over a range we create a set of micro-benchmarks whose run-time values indicate its effect in different hardware configurations.

The selection of the set of program features should be such that it covers all the important factors affecting the runtime behavior of our test programs. A sub-optimal set of features would result in an inaccurate predictor whereas choosing a lot of *non-influential* features would increase the training time of the predictor. The prime features can be divided into language-independent and language-dependent features. The language-independent features mostly revolve around dissimilarity in the hardware features of big and LITTLE. The language-dependent features vary across parallel programming models and across the parallel constructs supported by them.

The language-independent features represent the basic set of operations executed differently by different core types. These include memory operations, branch operations and data dependencies. Cache latencies and miss penalties of the big cores are much higher than those of the LITTLE cores. For instance, L2 latency for big is 21 cycles while for LITTLE it is 10 cycles [5]. Thus, as we increase the number of memory operations in the program the speed-up advantage of big cores over LITTLE cores reduces. The amount and density of the memory operations also play an important role in determining a core's performance. The big cores are usually stacked with hardware optimizations such as out-of-order execution, reorder buffers, and can withstand multiple cache misses. If the interval between two consecutive memory operations is adequate enough, the effect of a stall in big can be mitigated to some extent. Further, big cores are usually equipped with larger caches. LITTLE cores, on the other hand, have in-order execution and the pipeline stalls on every non-cached memory access. To capture this behaviour, we create multiple micro-benchmark programs with the same amount of memory operations but with different spreads of the load and store operations.

We also vary the virtual address spacing between the target address for consecutive memory operations from adjacent locations (to mimic cache-friendly array accesses) to purely random locations in a buffer larger than the L3 cache size (to mimic pointer based accesses which exhibit low spatial locality). This helps us capture the effect of memory accesses on different types of cores.

A branch misprediction can affect the ILP, and in turn, a core's performance. As we move from LITTLE to big, the complexity and the number of pipeline stages increase. This results in big cores incurring a higher penalty for every branch misprediction. Similar to the memory operations, we generate a set of micro-benchmarks with different spreads and intensities of branch operations. Our model also considers the possible effect of false-sharing. The effect of false-sharing becomes prominent when we consider using smaller chunk for scheduling the omp_for iterations having store operations on array locations. This increases the memory traffic, and thereby, memory latency.

The OpenMP specific features selected include barrier_constructs, omp_criticals, omp_flushes and omp_for for this work. The OpenMP constructs chosen as prime features are based on their impact on inter-thread communication and synchronization. As we increase the `N_THREADS` value, the synchronization cost also increases. So, for high intensities of barrier_constructs, a larger value of `N_THREADS` would suffer from higher execution time. In case of omp_criticals, more work done inside critical sections will be advantageous for big cores. This is because we will be able to execute the critical regions faster and thereby reduce the critical section waiting time. On the other hand, larger value of `N_THREADS` increases the waiting time penalty of the system, leading to lower energy-efficiency. omp_flush is costlier for big than a normal memory operation as the former blocks the pipeline until the flush operation is complete. The effect of omp_flush on LITTLE is not as severe due to inorder execution. omp_reductions have a similar effect on big and LITTLE as the reductions use local flushes at the end of the region to update values. CHOAMP also needs to train for different kinds of scheduling and sizes of omp_for. Static scheduling would result in equal work-distribution (assuming homogeneous workload in each iteration). big cores at high frequencies would execute their workloads faster and reach the barrier. This would lead to hardware under-utilization and thread migrations due to HMP scheduling.

3.2 Using Micro-Benchmarks

The code for micro-benchmark generators is available online [15]. We vary the intensity of the selected feature in the micro-benchmarks in a practical range (based on the analysis of various OpenMP benchmarks, namely, NPB [7], and FSU OpenMP programs [16]), and record various runtime parameters of interest. This is repeated for all the hardware configurations. The micro-benchmarks are semi-automatically generated to stress various performance aspects in the program. CHOAMP supports three cost functions: execution time, energy and EDP.

For each prime factor, a benchmark set is executed with the intensities of other prime factors being constant. We then combine the outputs of a benchmark set into the following equation. Since we have kept all other intensities constant and assume that the dependencies between prime factors are negligible we can generate the following equation

$$X * c_i = y_i^T \tag{1}$$

Here, each row in the matrix X represents the intensity of the prime factor in different powers starting from zero. c_i represents the coefficient vector for hardware configuration i. The y_i vector corresponds to the cost function values recorded using the micro-benchmark set for hardware configuration i. If we consider a linear relationship between the prime factor and the cost function, the matrix X will contain simply two columns ($c_0 * x^0 + c_1 * x = y$).

Using a regression tool we can estimate the coefficient vector. Such vectors are used by our Predictor to find y of an unknown intensity of the prime factor in the input program for each possible hardware configuration.

4 CHOAMP THEORY

CHOAMP employs a clustering-based approach for scheduling iterations to cores. Thus, iterations that are clustered together are scheduled together. The challenging aspect in applying clustering to parallel programs is that a single feature is often not sufficient to identify the cluster an iteration belongs to. The overall performance of an application is a complex interplay of several features such as ALU operations, memory instructions, and synchronization constructs. However, since an iteration needs to be executed by a single core in our setup, it is imperative to identify the most prominent feature which affects a given cost function. The challenge gets exacerbated when multiple features are prominent. Such scenarios demand prioritization over features to identify the final assignment of an iteration to a core. We first discuss our clustering approach in the context of various program features in isolation, and then explain how CHOAMP computes the final core-assignment combining individual recommendations from each feature.

4.1 Branch Instructions

Arbitrary branch instructions reduce cache effectiveness, and affect execution time. Executing branch instructions generally consume more energy than usual ALU operations (*cf.* Table 7). Thus, all our cost functions (execution time, energy and EDP) are directly affected by branches. Scheduling branch-heavy iterations to big cores can lead to their under-utilization and idling, and would also consume more energy. Therefore, whenever the percentage of branch instructions is higher than a threshold (determined by the predictor based on microbenchmark results), it is beneficial to cluster all such iterations to LITTLE cores.

4.2 Shared Data

Iterations accessing a lot of shared data should be scheduled together. Such a clustering leads to better cache efficiency and reduces inter-thread communication latency. A key concern here is to choose the type of core to schedule the iterations together. In our experience, it needs to be guided

by the frequency of memory operations (and other features such as synchronization). For instance, for low memory traffic, it is better to schedule iterations on big, and so on.

4.3 Memory Traffic Diversity

An interesting feature that affects scheduling is the diversity of memory traffic. We observed that scheduling the iterations with diverse memory demands together improves performance. Also, contrary to the usual belief, scheduling of iterations with high memory demand onto LITTLE leads to inferior performance (when the cost function is execution time). Instead, the performance is better if such iterations are scheduled on big but with lower frequency.

4.4 Synchronization Instructions

Synchronization instructions play a crucial role in the execution time as well as energy consumed by a parallel program. CHOAMP models the behavior of three synchronization constructs: atomics (omp_atomic), reductions (omp_reduction) and critical (omp_critical). The clustering behavior differs across each of them. For atomics, if the percentage (within an iteration) is beyond a threshold, the iterations are clustered onto LITTLE, otherwise big. When a medium type of core is present in an HMP system, the iterations can be clustered on it based on another threshold. The thresholds are suggested by the microbenchmarking (Section 3). On the other hand, reductions involve thread coordination. Hence, CHOAMP clusters the participating iterations onto the same type of core. For execution time, big is preferred. Critical section involves yet another consideration. If iterations involve many critical sections (beyond a threshold), CHOAMP prefers to schedule the corresponding iterations sequentially. Otherwise, depending upon the cost function, the iterations get clustered on different types of cores. In particular, for execution time, CHOAMP prefers big. For energy as the cost function, CHOAMP prefers LITTLE if the critical section involves shared data. For EDP, CHOAMP uses a combination of the above two rules. In case of barriers, the primary considerations are the synchronization cost and the barrier waiting time. When the number of barriers increases and the cost function is execution time, CHOAMP prefers a smaller number of cores. For energy as the cost function, CHOAMP prefers LITTLE for executing the barrier instructions. In the case of EDP, CHOAMP prefers a symmetric core system, where all cores have similar processing power, thus reducing the overall barrier waiting time.

4.5 Putting Features Together

Individual feature vectors lead to independent suggestions on the placement of iterations. However, depending upon the program, sometimes, the suggestions may lead to conflicting placements. For instance, if the number of branch instructions is high, the individual suggestion from the rules in Section 4.1 would be to schedule the iteration on LITTLE. However, in addition, if the memory traffic is high, based on the suggestion from the rules in Section 4.3, the placement would be on big. Therefore, it is imperative to combine individual suggestions by appropriately prioritizing them. This leads to devising a weight-function to reach

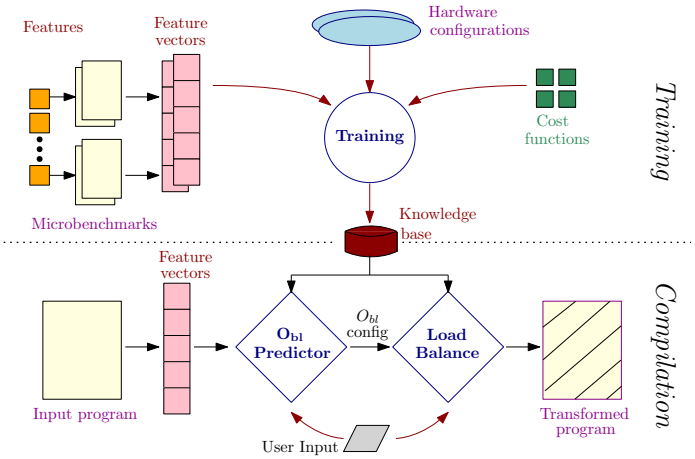


Fig. 3: Block diagram depicting the workflow of our framework. The top half shows the training phase and the bottom half shows various compiler stages in code-scheduling.

the final placement prediction. CHOAMP supports three types of weight functions: linear, quadratic and Gaussian (*cf.* Table 3), which we discuss next in more detail.

5 CHOAMP Implementation

We now discuss using micro-benchmarks for training the predictor, finding the optimal O_{bl} configuration and the corresponding transformation of the program. The workflow of our technique is depicted in the block diagram of Figure 3 and consists of primarily two phases. (i) Training Phase, where we generate micro-benchmarks to train the predictor for the selected program features and populate the knowledge base. (ii) Compilation Phase, where we analyze the input program and output the O_{bl} configuration for the input program. The compilation phase consists of an optional Load Balancing phase where we transform the input program to use existing load balancing techniques to distribute the parallel workload among the cores in O_{bl}. The user can provide input on the selection of Predictor kernel and the load balancing technique.

We explain the two phases in more detail below.

5.1 Training Phase

During the training phase, we run sets of micro-benchmarks in various available hardware configurations and collect the power, energy and runtime values. We use the Java Scientific Library [17] provided by M. Flanagan to estimate the coefficient values represented in Equation 1. The estimated coefficients form input to our Predictor.

The Predictor combines the different prime planes (listed in Section 3) of the source program using weighted Euclidean distance to generate the combined cost of all the prime features of the program.

Various regression techniques can be applied to Equation 1. For example, for linear regression, the equation boils down to $y = c_1 * x + c_0$ for each prime factor x. In CHOAMP, we have used three different regression functions for the prediction model: (i) Linear, (ii) Quadratic, and (iii) Gaussian. All three models are trained using the same

TABLE 3: Cost calculation for various Regression functions.

Regression	Cost Estimation
Linear	$\Sigma_{e=0}^{N} w_e\ c_e[1]{*}wl(e)$ + ($\Sigma_{e=0}^{N}\ w_e$ * $c_e[0]$)/ N
Quadratic	$\Sigma_{e=0}^{N} w_e$ * ($c_e[1]{*}wl(e)$ + $c_e[2]{*}wl(e)^2$ + $c_e[0]$) / N)
Gaussian	$\Sigma_{e=0}^{N} w_e\ Y_e$ / ($\sqrt{2 * \pi * sd_e}$*($e^{(wl(e)-X_e)/sd_e)^2}$))
N: Number of prime Features c[*]: coefficients from regression model using micro-benchmarks w_e: weight of prime feature $wl(e)$: intensity of the prime feature in the program Y_e: Y scale prime feature in the program X_e: Mean of prime feature in the program	

micro-benchmark set. From our experiments, we observe that different regression functions have different levels of accuracy based on how well they are able to fit each feature and the runtime intensity of the feature. CHOAMP with predictor trained with Linear (Quadratic, Gaussian) regression is termed linear-CHOAMP (quadratic-CHOAMP, gaussian-CHOAMP respectively).

The cost function calculation for different regression training functions is shown in Table 3. The cost function is computed as a weighted sum of values for each prime vector. The wl vector contains the intensities of the prime factors. N denotes the number of trained prime features. The regression functions estimate the relationship between the prime factors and the selected cost function. w_e provides differential weight for different prime features. Linear regression (c_l) uses a linear combination of ivector. The quadratic regression (c_q) is more powerful, as the cost function is modelled as a quadratic function of ivector. Though the function is more powerful, c_q is more susceptible to errors such as overfitting. The gaussian regression (c_g) assumes all the prime factors to be in a continuous domain. Using weighted sum, we can vary the importance of an individual prime feature by varying the weights. The choice of the predictor is guided by a global flag.

Table 4 shows the different prime features trained for CHOAMP. We show the study of memory micro-benchmark set in Table 5 for a subset of the hardware configurations trained. For instance, configuration 0L4b(2) contains zero LITTLE and four big cores, with big's frequency set to 2.0 GHz. The table presents the average values of the runtime cost function obtained, for the highest (1.0) and the lowest ivector (0.1) values for memory operations. The ivector values for the remaining prime vectors are kept constant. For memory operations, as we can see for execution time, big is advantageous over LITTLE; for energy, LITTLE is preferred; and for EDP, LITTLE is advantageous beyond a threshold.

Table 6 shows the average values of the runtime cost function obtained, for the highest (3.05E-5) and the lowest (4.77E-10) ivector values for the barrier micro-benchmark set. We can see that for higher intensities of barriers, using more cores is expensive on all fronts.

Table 7 shows the average values of the runtime cost function, for the highest (0.3) and the lowest (5E-4) ivector values for the branch micro-benchmark set. We see that for a very high percentage of branch instructions, LITTLE outperforms big in terms of execution time.

TABLE 4: Prime features chosen

Prime Feature	Description	ivector value
Branch operations	Percentage of Branch operations	varied from 0 to 30%.
Memory operations	Trained for multiple densities of Load and Store operations for same number of memory operations.	varied from 50 - 100%.
Atomic operations	percentage of omp_atomic operations	varied from 0 to 15%.
Barriers	Number of barriers in a parallel region. Trained for multiple densities of barriers for same number of barriers.	from 0 to 3.05E-03%
Critical sections	Percentage of operation inside critical section when compared to entire workload of a parallel section	from 0 to 6.10E-03%
False sharing	Taken as a percentage of all STORE operations in the parallel region.	Trained for different strides with potential false sharing.
Flush operations	Percentage of memory locations flushed using omp_flush.	varied from 0 to 5%.
Reduction operations	Percentage of reductions in a parallel region as a factor of all ALU operations	varied from 0 to 25%.

TABLE 5: Runtime values obtained for memory micro-benchmark set for intensity values: high=1.0, low=0.1.

Config.	Exec. time in s		Energy in J		EDP in Js	
	high	low	high	low	high	low
4L0b	5.70	2.01	5.82	1.82	33.31	3.67
0L4b(2)	2.71	0.62	15.71	2.88	42.57	1.81
4L2b(2)	2.33	0.67	13.43	1.46	38.56	2.55
4L4b(2)	1.80	0.47	11.39	2.43	20.93	1.15

5.2 Compilation Phase

First step in the compilation phase involves analyzing the input program to extract intensities of the prime features. A typical OpenMP program can have multiple parallel regions interleaved with sequential processing. In this work, we focus on optimizing only the parallel regions, which allows us to analyze each parallel region separately. CHOAMP creates a concurrent control-flow graph (CCFG) for the input program. All the basic-blocks in the CCFG are associated

TABLE 6: Runtime values obtained for barrier micro-benchmark set for intensity: high=3.05E-5, low=4.77E-10.

Config.	Exec. time in s		Energy in J		EDP in Js	
	high	low	high	low	high	low
4L0b	5.45	5.27	5.26	4.86	28.68	25.65
0L4b(2)	1.91	1.78	8.55	8.09	15.35	21.09
4L2b(2)	3.87	2.26	17.49	10.20	67.75	23.02
4L4b(2)	43.66	1.80	231.42	11.99	1.0E4	22.59

TABLE 7: Runtime values obtained for branch micro-benchmark set for intensity: high=0.3, low=5E-4.

Config.	Exec. time in s		Energy in J		EDP in Js	
	high	low	high	low	high	low
4L0b	0.84	0.98	1.23	1.05	1.03	1.03
0L4b(2)	1.13	0.63	4.83	4.62	5.24	3.05
4L2b(2)	0.68	0.52	4.65	4.35	3.12	2.80
4L4b(2)	0.61	0.42	5.88	4.45	2.49	3.30

with a workload (wl) vector. The wl vector is assigned the intensity of each program feature of interest in the block. Each basic block appends its wl to that of its parent block for computing the combined effect.

CHOAMP traverses the CCFG to compute the intensity of each prime feature in the input source program. For precise and effective modeling of OpenMP programs, we need to have an inter-procedural workload accumulation. CHOAMP creates a call-graph to store all the function calls with hooks to the call-sites. The call-graph analysis is a two-step process. The first step is intra-procedural, wherein individual functions are analyzed in isolation. The second step is inter-procedural which takes into account the calling context of a function. To handle the context, CHOAMP topologically sorts the call-graph and processes functions in that order. Use of recursion generates a potentially infinite number of contexts as the recursion depth is unknown at compile-time. CHOAMP currently does not handle recursion.

The intensity of each prime characteristic is stored as the fraction of the total number of prime characteristic operations performed in the code. After extracting features, CHOAMP passes the generated wl vector to the trained prediction model and calculates a cost function value for each selected hardware configuration for the runtime cost function of interest. The model then predicts a hardware configuration having the least cost function value to run the parallel region.

When multiple parallel regions exist, CHOAMP computes the O_{bl} configuration for each omp_parallel region separately. However, it is difficult in big.LITTLE to apply different hardware configurations to different parallel regions of the same program. Composing wl vectors of multiple parallel regions may result in different intensities of the prime characteristics, leading to suboptimal O_{bl} configuration. CHOAMP prioritizes the configuration associated with the most parallel region (dictated by wl vectors), with the assumption that it would dominate the cost function.

The example in Figure 2 contains a single parallel region with an embedded omp_for with `N` iterations. The wl vector values are collected as a function of `N` for some of the prime factors (such as memory operations and omp_critical) and are independent of N for some other prime factors (such as barriers). As a result, the accuracy of the Predictor would depend on the value estimation of N. If the runtime value of `N` is unknown, CHOAMP equates it to its largest known constant value in the program.

5.3 Estimating Values of Unknown Variables

The precision of the estimated wl vector is critically affected by the unknown variables. Especially in the context of our analysis in CHOAMP, knowing range of variable values can considerably improve prediction accuracy, leading to improved prediction. There are primarily two kinds of unknown variables that plague effectiveness of CHOAMP: loop bounds, and variables used in conditional statements. We handle these as follows. We first preprocess the source code to collect the program constants and the variables of interest; that is, loop bounds and variables used in conditional statements. To keep the method effective and practical, we choose to bucketize their values in exponential ranges. Thus, we define a set of variable ranges (R) bounded by powers of a selected base (such as 10) $R(i)$ representing the variable range to which the variable i belongs. A variable is expected to be pushed into the range which would bound the largest value the variable can attain during execution. The pre-processing happens as below. First, the program constants are pushed into a corresponding R with a flag to identify the constant. All the statements updating the variables of interest (loop bounds and those used in conditionals) are processed, and the unknown variables are pushed into the variable ranges based on their update values and the expressions they are used in. CHOAMP currently models binary expressions (which are more frequent) as shown in Table 8. Finally, the remaining (unclassified) variables are pushed into the largest non-empty variable range. After such a bucketing, for a non-constant variable i, it assumes the largest known value in the $R(i)$ as the value of i.

TABLE 8: Value ranges with prominent binary operations

Operation	Result range
i + j	range(MAX($R(i)$) + MAX($R(j)$))
i * j	range(MAX($R(i)$) * MAX($R(j)$))
i - j	range(MAX($R(i)$) - MIN($R(j)$))
i / j	range(MAX($R(i)$) / MIN($R(j)$))
i >> j	i - MAX($R(j)$)
i % j	$R(j)$
i = j	MAX($R(i)$, $R(j)$)
MAX($R(i)$): Known maximum value in range i range(k)): Range to which the value k belongs	

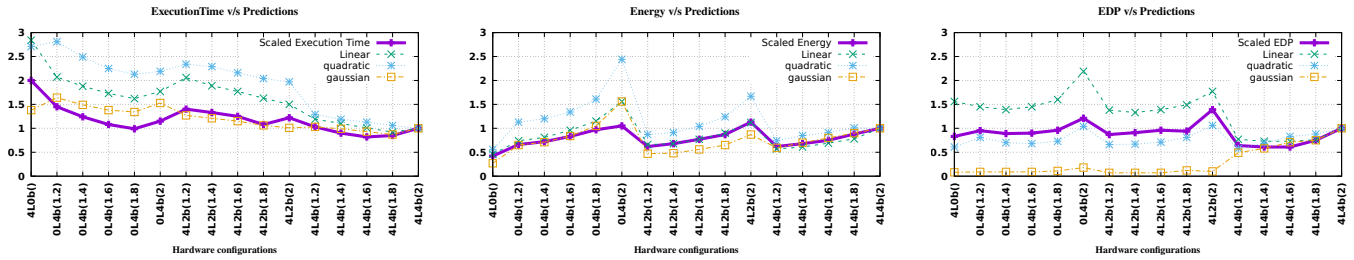
For example, consider Figure 2. We have `N`, the variable of interest which depends on variable `L`, which is unknown. The initial unknown value `L` is put into the current highest non empty bucket ($R_{1E4-1E5}$ containing constant `M` with value 50,000). The range of `temp` is estimated to be $R_{1E2-1E3}$ (range(MAX(R(`L`))), i.e., 50000/128)). Now, range for `N` is (range(`M` - MIN($R_{1E2-1e3}$), i.e., 50000-1000) as $R_{1E4-1E5}$. Hence the value of `N` is estimated as 50000 which is the maximum known value in the range of `N`.

Using the variable ranges, CHOAMP proceeds to find the cost function value for each hardware configuration trained, using the functions described in Table 3.

5.4 CHOAMP Example

We revisit our example in Figure 2 to demonstrate the efficiency of our Predictor. Figure 4 shows the normalized runtime characteristics with the normalized cost function values calculated by the three regression functions. We use *i*L*j*b(*k*) notation to represent a hardware configuration with i number of LITTLE cores, j number of big cores, where each big core has a frequency of k GHz. The LITTLE core frequency is kept constant at 1.4 GHz. All the values are normalized with respect to those of the 4L4b(2) configuration. We observe from the figure that the values predicted by CHOAMP mostly follow the same pattern as the scaled runtime values, which helps in identifying the best configuration.

Table 9 shows the runtime values of the example for the O_{bl} configuration predicted by each Predictor along with the O_{bl} configuration predicted by each Predictor and the best hardware configuration. We can see that the Predictors are



(a) Normalized execution time compared to different regression functions.

(b) Normalized energy compared to different regression functions.

(c) Normalized EDP compared to different regression functions.

Fig. 4: Runtime characteristics of the example in Figure 2 for different hardware configurations.

TABLE 9: The runtime values of O_{bl} configuration predicted for different cost functions for the example given in Figure 2.

	Base case	Manual	CHOAMP predictions		
	4L4b(2)	Best case	Linear	Quadratic	Gaussian
Time in s	6.34	5.18 4L4b(1.6)	5.36 4L4b(1.8)	6.34 4L4B(2)	5.36 4L4b(1.8)
Energy in J	30.16	12.52 4L0b	12.52 4L0b	12.52 4L0b	12.52 4L0b
EDP in Js	191.13	116.91 4L4b(1.4)	116.91 4L4b(1.4)	121.85 4L4b(1.2)	166.33 4L2b(1.2)

able to predict hardware configurations that are close to the optimum configuration.

5.5 CHOAMP Scope

CHOAMP continues to be under active development. It currently supports almost all of the OpenMP features including various parallel sections, scheduling strategies, and synchronization constructs. However, there are a few non-trivial aspects which CHOAMP does not support: nested parallelism, recursion, and OpenMP tasks.

OpenMP allows nested parallelism where each member in the team can create a new set of threads, where the creator thread assumes the role of the master in the new team. The newly created team will have its own local barriers and pragma constructs. Even though nested parallelism theoretically increases the effective number of threads, it is going to be limited by the hardware configuration.

As discussed earlier, recursion poses issues with respect to a potentially infinite number of calling contexts. In our study with NPB, none of the nine benchmarks uses nested parallelism or recursion. Therefore, we leave recursion handling as a future work.

We restrict the discussion of CHOAMP to the subset of synchronization constructs that are prominent in the benchmarks we have chosen for initial learning [16], [18] and NPB [7]. Constructs such as OpenMP tasks introduce new challenges. With tasks, all the work in a parallel region is not available at once. Also, the tasks can switch threads (*untied* tasks). These are dependent on runtime which makes it difficult to estimate and model the ivector for prediction.

While considering a different AMP hardware, we need to update the language-independent prime features. For example, consider a distributed AMP (i.e., with NUMA architecture). The stalls induced by the shared-data (including false-sharing) can be costlier than say, branch misprediction stalls. Also, barrier synchronization may have a different cost for the same base configuration, but with a different set of cores chosen based on the selected cores' locations (due to NUMA effect). We can address these issues by incorporating core locality as a part of the hardware configuration. This can increase the number of possible configurations to be trained for. However, CHOAMP can be useful as a starting point for designing a Predictor for the AMP.

We use regression model as our learning algorithm as a simple model that can capture the effects of the prime factors, with low modelling and training costs. We are able to capture the essence of the program features with the regression model, though more complex and costly machine learning kernels such as support vector machines and neural networks can replace the regression model with better efficiency while learning for more complex prime features.

Even though the model has been explained using OpenMP, the underlying CHOAMP methodology is applicable in a wider setting: (i) identifying the prime features, (ii) learning a Predictor over the prime features based on the parallel programming model and AMP, and (iii) using the trained Predictor for predicting the best possible configuration available with the AMP. For example, when using pthreads, the possible set of language dependent prime features could include pthread_create, pthread_join and pthread_exit. These features pose similar challenges as OpenMP tasks and would require more exhaustive sets of micro-benchmarks to train a good Predictor.

5.6 Handling Imbalanced Workload

The inherent power imbalance in Asymmetric Multicore Processors makes it difficult to balance running times of threads in parallel programming platforms such as OpenMP. Current compilers are tuned for optimizing the code for SMP hardware with an equivalent amount of workload given to each thread between two barriers.

Since the big cores can usually execute threads much faster than the LITTLE cores, often the threads that are initially scheduled in big cores reach the barrier earlier than the threads that are scheduled in LITTLE core. Once a thread running on big reaches the barrier, its CPU utilization reduces and the HMP scheduler would eventually down-migrate it to LITTLE. On the other hand, a thread running on LITTLE and yet to reach the barrier may be up-migrated. This can lead to overall hardware under-utilization. The as-

TABLE 10: Test environment description

Hardware	Odroid XU3 [11]
Processor	Samsung Exynos 5422
Microbenchmark scripts	Python version 2.7
Parallel platform	OpenMP [21]
Frontend compiler	IMOP [20]
Backend compiler	gcc 4.8.2
Regression tool	Java Scientific Library [17]
OS	LUbuntu 16.04
big frequencies tested	2GHz, 1.8GHz, 1.6GHz, 1.4GHz, 1.2GHz
Configurations tested	4L4b(4 LITTLE&4 big),4L2b,4L0b,2L2b,0L4b

TABLE 11: Characteristics of NPB Benchmarks

Name	Lines	parallel	omp_for	barriers	omp_critical
BT	2653	2	70	38	0
CG	550	5	24	23	0
DC	2478	1	0	0	1
EP	169	1	1	1	1
FT	775	2	7	11	1
IS	285	2	2	2	1
LU	2658	3	36	27	1
MG	828	6	11	22	1
SP	2206	2	78	65	0

sociated performance penalty is high if there is an imbalance in the number of different kinds of cores.

In omp_for, static scheduling is most preferred because of its low overhead. In AMPs, however, static scheduling often results in load imbalance and frequent thread-migrations, which can considerably hamper performance [19]. CHOAMP employs multiple load-balancing mechanisms to reduce this inefficiency, as we discuss below.

- Using dynamic or guided scheduling: dynamic and guided scheduling mechanisms offer relatively better overall load-balancing. A naïve usage of these mechanisms incurs additional scheduling overhead, but it can be mitigated by choosing an appropriate chunk-size with the help of the predictor, to reduce the number of scheduling points.
- Using custom scheduling: Here, CHOAMP first fixes a core-to-thread mapping and then divides the iterations based on the speed-ratios of various types (such as big-to-medium and medium-to-LITTLE). Note that these ratios are not fixed, but vary based on the core frequencies chosen by the predictor. For our experiments, we use CES [9] for the custom scheduling.

6 Experimental Evaluation

We now evaluate the effectiveness of CHOAMP.

6.1 Setup

We use IMOP [20], a source-to-source transformation and analysis framework for OpenMP. The benchmark programs are pre-compiled using gcc 4.8. Precompilation expands the preprocessor directives to increase accuracy and reduce compilation time. For regression training, we use Java Scientific Library (JSL) provided by Flanagan [17]. The JSL is used offline to generate the coefficients of regression functions which are plugged into IMOP framework. The evaluation system configuration details are presented in Table 10. For our experimentation purposes, we have trained for a subset of the numerous hardware variations achievable with big.LITTLE: (i) vary the number of big cores from 0 to 4 (step size of 2), (ii) vary the number of LITTLE cores from 0 to 4 (step size of 4), and (iii) vary the frequency of big cores from 1.2 GHz to 2 GHz (step size of 0.2 GHz). For all the experiments we fix the frequency of LITTLE cores to 1.4 GHz, as found empirically to provide benefits.

We use NPB OpenMP benchmarks [7] for evaluating CHOAMP. Characteristics of NPB benchmarks relevant to this work are presented in Table 11. The number of barriers listed corresponds to the explicit barriers in the code. The number of lines of code includes the code only from the main program, and not from any libraries.

TABLE 12: O_{bl} selected for NPB benchmarks computed by linear regression.

Benchmark	Configuration chosen for a cost function		
	Execution Time	Energy	EDP
BT	4L4b(1.8)	4L0b	4L4b(1.4)
CG	0L4b(1.8)	4L0b	0L4b(1.4)
DC	4L4b(1.8)	4L0b	4L4b(1.4)
EP	4L4b(1.8)	0L4b(1.2)	4L4b(1.6)
FT	4L4b(1.8)	4L0b	4L4b(1.6)
IS	4L4b(1.8)	4L0b	4L4b(1.4)
LU	4L2b(2.0)	4L0b	0L4b(1.4)
MG	4L4b(1.8)	4L0b	4L4b(1.6)
SP	0L4b(1.8)	4L0b	0L4b(1.4)

Our experiments are carried out on a big.LITTLE system where the HMP scheduler is enabled. We use the default hardware configuration as the baseline, which is 4L4b(2) (4 LITTLE cores running at 1.4 GHz and 4 big at 2 GHz).

6.2 Predicted Configurations and their Performance

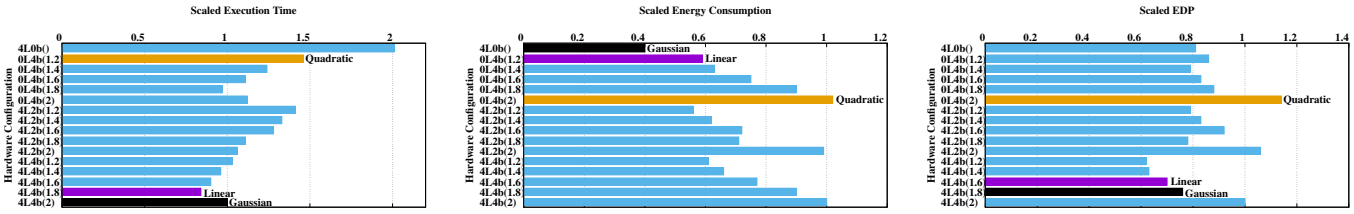
Table 12 shows the O_{bl} configurations predicted by CHOAMP when trained using the linear regression. For execution time as the cost function, all big cores are chosen in all but one benchmark (LU), which follows our expectation.

CG and SP have many barriers; this prompts CHOAMP to choose fewer cores (no LITTLE cores). While trying to optimize for energy consumption, CHOAMP chooses a core configuration which is closer to the lower end of the spectrum. Thus, when the cost function is energy, CHOAMP mostly prefers the LITTLE cores, except for EP. EP is a highly parallel benchmark with very little synchronization cost and high ILP, allowing the big cores to execute instructions much faster than the LITTLE cores, prompting CHOAMP to choose big over LITTLE for energy consumption. While training for a more complex cost function like EDP, a more balanced configuration is chosen for most of the benchmarks. The Predictor always tends to choose a hardware configuration which lies in-between the one chosen for execution time and energy in the spectrum. Most of the configurations chosen have the maximum number of cores possible to exploit maximum parallelism. For CG and SP, CHOAMP selects fewer cores because of the high synchronization present in the benchmarks. The chosen frequency of big cores is similar to that of LITTLE. This helps in load-balancing (most omp_fors in NPB use static scheduling).

Table 13 compares the execution time, energy consumption and EDP values obtained for baseline with the corresponding values for the program optimized

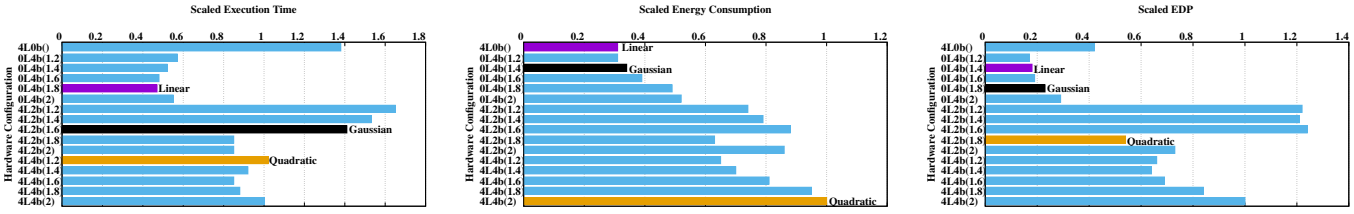
TABLE 13: Runtime cost function values of base case (4L4b(2)) compared with O_{bl} runtime cost function

Benchmark	Execution time in s		Energy in J		EDP in Js	
	baseline	O_{bl}	baseline	O_{bl}	baseline	O_{bl}
BT	295.10	258.64	1446.92	486.95	426987.66	240872.73
CG	7.29	3.46	34.74	10.67	253.29	44.54
DC	113.59	122.42	363.78	124.38	41320.86	32228.11
EP	18.52	15.62	89.61	53.00	1659.82	1159.98
FT	12.60	8.33	61.61	26.23	776.19	214.03
IS	1.83	1.73	9.49	4.12	17.36	8.65
LU	235.95	169.89	1178.02	280.41	277954.89	32355.37
MG	4.71	4.08	23.63	10.68	111.35	73.06
SP	74.36	0.18	358.73	0.22	26676.79	0.06
Mean	84.88	64.93	396.28	110.74	86195.36	34106.28



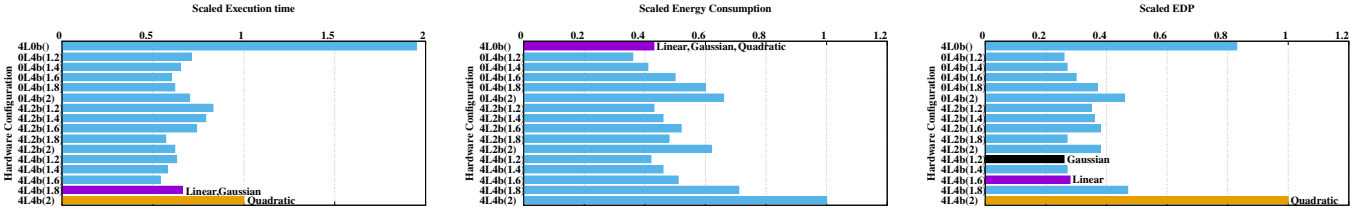
(a) Normalized execution time. (b) Normalized energy consumption. (c) Normalized EDP.

Fig. 5: Runtime characteristics of EP for different hardware configurations. Smaller is better.



(a) Normalized Execution time. (b) Normalized energy consumption. (c) Normalized EDP.

Fig. 6: Runtime characteristics of CG for different hardware configurations. Smaller is better.



(a) Normalized Execution time. (b) Normalized Energy consumption. (c) Normalized EDP.

Fig. 7: Runtime characteristics of FT for different hardware configurations. Smaller is better.

for the cost function by CHOAMP (with configurations mentioned in Table 12). For CG and SP, we witness a large improvement in the execution time as CHOAMP chooses fewer cores for executing the programs with high synchronization costs. SP, in fact, shows even better runtime for fewer threads (2 and 1) outside the trained spectrum. Since the baseline has high power consumption, we see a huge improvement in energy consumption of CHOAMP-optimized programs. A similar trend is observed when the cost function is EDP. This clearly indicates the efficacy of CHOAMP.

A closer look at some benchmarks: Now, we look into a few of these benchmarks in detail. Figures 5–7 show scaled execution time, energy consumption and EDP values obtained for EP, CG and FT for different hardware configurations we trained for. The hardware configurations are inversely sorted first in terms of the total number of cores and then in terms of the big core frequency. The first entry in the graphs represents the value corresponding to the base case (4L4b(2)) to which the rest of the values are normalized. In all the cases, CHOAMP predicts the configuration with the lowest cost function value as per the model. In the plots, we have marked the configuration selected by CHOAMP.

First, we look into EP (Figure 5), a highly parallel benchmark with very little synchronization inside a parallel

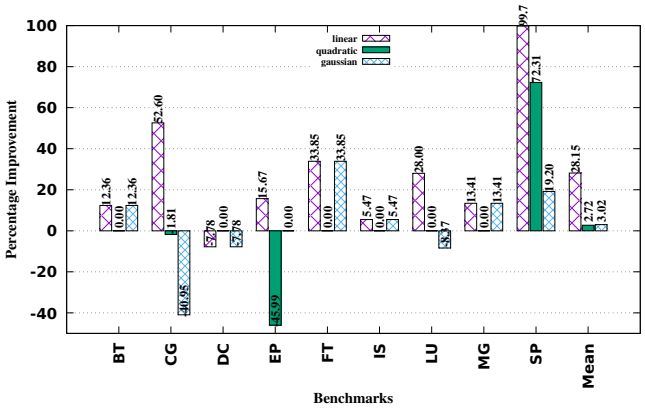


Fig. 8: Percentage improvement in execution time

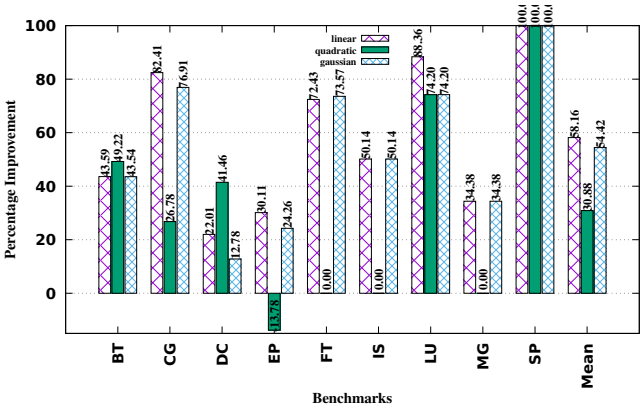


Fig. 10: Percentage improvement in EDP

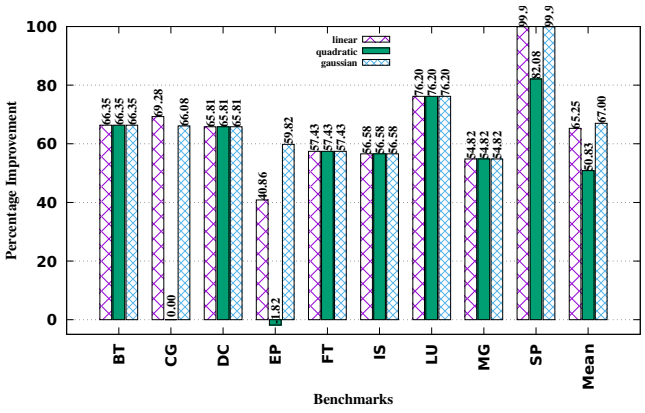


Fig. 9: Percentage improvement in Energy consumption

region. Due to its inherently parallel nature, the execution time reduces with increasing number of cores and core frequency. Linear-CHOAMP is able to accurately predict the best configuration for running EP while optimizing for execution time. High parallelism in EP helps higher hardware configurations to get better EDP values. Both linear-CHOAMP and gaussian-CHOAMP are able to predict hardware configuration close to the optimal.

As shown in Table 11, CG has a large number of barriers which increase the synchronization costs considerably (Figure 6). This restricts the ability of the program to scale well with the number of processors. As a result, CG performs better in terms of execution time and EDP for fewer cores. Linear-CHOAMP is able to predict hardware configuration that is very close to the optimal value.

In terms of program characteristics, FT (Figure 7) lies in between EP and CG. The benchmark is fairly scalable. The best configuration for execution time lies at the high end of the spectrum, while that for EDP lies in the middle. Linear-CHOAMP and gaussian-CHOAMP predict a similar trend with all O_{bl} predictions showing substantial improvement in the cost-function values.

6.3 Cost Functions

Figure 8 shows the overall percentage improvement (reduction) in execution time for CHOAMP-compiled programs over the baseline. We see an average improvement of 28% in execution time with linear-CHOAMP. DC contains very few program-specific prime features. SP and CG (for linear-CHOAMP) show high improvement in execution time for a smaller number of cores (4), due to its larger number of barrier instructions which induce high synchronization costs for more cores. Gaussian-CHOAMP predicted all LITTLE cores for LU (4L0b), resulting in a higher execution time. The improvements using quadratic-CHOAMP (2.7%) and gaussian-CHOAMP (3%) are marginal.

Figure 9 shows the overall percentage improvement (reduction) in energy consumption for CHOAMP-compiled programs over the program running in the base case. We see an average improvement of 65% in energy consumption while using linear-CHOAMP over the nine benchmarks in NPB. Both quadratic-CHOAMP (average of 50%) and gaussian-CHOAMP (67%) are also able to provide very good improvements in energy consumption values. The high level parallelism of EP prompted quadratic-CHOAMP to choose all big cores (0L4b(2)) for energy consumption. But high power consumption of big cores has resulted in negative improvement in energy consumption.

Figure 10 shows the overall percentage improvement (reduction) in EDP values for CHOAMP-compiled programs over the program running in the base case. We see an average improvement of 58% in EDP for linear-CHOAMP over the nine benchmarks in NPB. For SP, the EDP value obtained for CHOAMP-compiled program is negligible compared to the base case, and hence reported as 100% improvement (corrected to two decimal places). Quadratic-CHOAMP predicted the base configuration for MG for optimizing EDP, resulting in zero improvement. Quadratic-CHOAMP suggested all big (0L4b(2)) for EP. High power consumption of big cores leads to a higher EDP than the base case. However, linear-CHOAMP provides considerable benefits.

The current model can be strengthened by incorporating other variations of the regression kernel. For instance, we studied regression using non-overlapping rolling windows. Our preliminary evaluation did not provide significant improvements in terms of the performance of the predicted configuration. However, a detailed study of such variants would be an interesting future work.

6.4 CHOAMP and CES

In this subsection, we analyze the compatibility of CHOAMP with a dynamic scheduler named CES (Compiler Enhanced

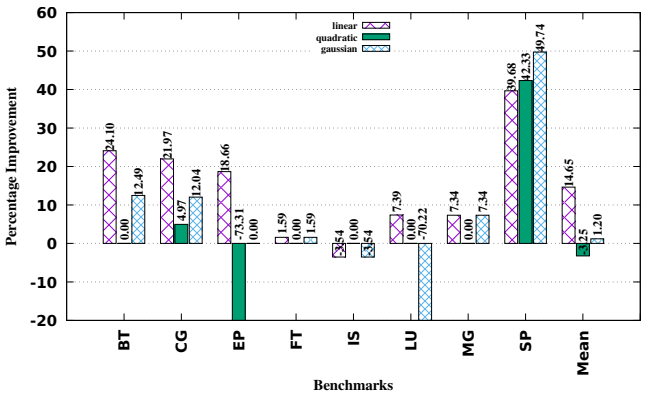


Fig. 11: Percentage execution time gain while executing various NPB Benchmarks (CES + CHOAMP). The base case is CES compiled program for 4L4b(2.0) configuration.

Scheduler) [9]. Figure 11 shows the average percentage improvement in execution time of (CHOAMP + CES)-compiled code over CES-compiled code. If CHOAMP outputs the program for a homogeneous hardware configuration (for example CG and SP while using linear-CHOAMP), the CES compiler is left ineffective and will not change the code. We see an average of 14% improvement in execution time while using linear-CHOAMP when compared to CES compiled code. CES transforms the omp_fors to create separate worklist for each thread and balances the workload in the worklist based on the core computing power. This results in lower waiting time at the barriers. As a result, we observe lower gains in CHOAMP execution time with SP and CG while comparing with CES compiled code. For EP with quadratic-CHOAMP (0L4b(2)) and for LU with gaussian-CHOAMP (4L0b(2)), the execution times increase considerably.

Overall, we illustrate that CHOAMP offers considerable benefits over the baseline, and allows a user to seamlessly optimize parallel programs for AMP.

7 Related Work

Our scheme relies completely on compile time analysis, is tested on real hardware, and does not require any extra hardware support. A new hardware configuration can be easily integrated into the existing model. We simply need to train the Predictor to learn about the new hardware configurations. Here, we discuss relevant related works on selecting optimal hardware configuration and scheduling in heterogeneous multicore processors.

Majority of the work on scheduling in AMP relies on dynamic analysis as precise information of runtime parameters is available [22], [23], [24], [25], [26]. There have been a few works that rely on compile-time profilers for making scheduling decisions [27], [28]. Compared to the dynamic schedulers (which operate at execution time), static schedulers scale better with the number of cores as well as program complexity [27]. With dynamic scheduling, the schedulers are often required to be fast and lightweight.

Some of the dynamic scheduling techniques proposed for normalizing the execution time of threads [26], [27], [28] can be easily integrated into CHOAMP in place of HMP and CES. Chen and John [28] proposed a dynamic scheduling technique to map the inherent program characteristics and their corresponding resource demands to the characteristics of different core types in AMPs. The proposed method combines the program characteristics and available core configurations into a unified solution space using weighted Euclidean distance. We also use Euclidean distance to combine the effect of individual prime factors to estimate the overall cost of the program.

Former research in this area has focused on scheduling in AMPs to improve execution time [22], [24], energy consumption [29], [30], [31], or a function of both [32], [33]. These works propose novel techniques to exploit the hardware heterogeneity for optimal performance and/or power consumption. However, most of the proposals are tied to optimizing one of the runtime features and are inflexible towards the user-requirements. CHOAMP optimizes a runtime cost-function provided by the user.

A few works have looked into dynamic frequency scaling along with thread scheduling [32], [34] for optimal performance measures. Annamalai et al. [32] proposed a scheduler which dynamically chooses between swapping two threads and changing the frequency to optimize for throughput per Watt.

Multiple works have relied on regression for estimating execution time and power [3] with a good amount of accuracy. A few of the dynamic algorithms designed for AMPs which use similar basic concepts are discussed below. Bias scheduling [24] presents a dynamic scheduling technique where a thread is given a big or LITTLE core bias based on the speedup ratio. PIE (Performance Impact Estimation) [22] shows that taking the ratio of ILP (Instruction Level Parallelism) and MLP (Memory Level Parallelism) provides a good estimate of the relative performance of the big and the LITTLE cores.

Taylor et al. [35] have proposed to use SVMs to map OpenCL kernels to heterogeneous multicores. The proposed method is implemented in LLVM and trained for multiple CPU-GPU configurations. The CPU configurations are all homogeneous in nature which might not provide the best hardware configuration as demonstrated in Figures 5–7. Memti and Pllana [25] have proposed to use simulated annealing to reduce the solution space exploration to find the optimal system configuration in a heterogeneous system.

8 Conclusions

In this work, we propose a probabilistic model CHOAMP, with a micro-benchmark trained Predictor which predicts the best asymmetric hardware configuration for a given cost function. This eliminates the manual effort involved in the trial-and-error method for tuning for an optimal configuration. CHOAMP is flexible to different user needs as the same program can be transformed for optimizing execution time, energy consumption, and EDP. Experimental results with NAS Parallel Benchmarks show 28% improvement in execution time, 65% improvement in energy consumption and 58% improvement in EDP with Linear-CHOAMP over the baseline.

In future, we plan to add support for more cost functions such as execution time given an energy or power budget, energy consumption given a limit on execution time, as a factor of the base case values. Also, we would like to introduce more hardware configurations permitted by AMPs such as variable cache sizes and memory bandwidths. We would also like to consider the inter-dependencies of the prime factors which would require generating training sets for multiple prime features and using more sophisticated learning models.

ACKNOWLEDGMENT

Thanks to the reviewers and Anchu, Dennis, Indu, Raghesh for lending their time for constant review, suggestions and comments on this work. We would also like to thank Aman and Krishna Nandivada for their IMOP framework, and Dr. Flanagan for his Java Scientific Library.

REFERENCES

[1] J. Brian, "Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration." in *DAC*, P. Groeneveld, D. Sciuto, and S. Hassoun, Eds., 2012, pp. 1143–1146.

[2] NVIDIA, "NVIDIA Tegra K1 A New Era in Mobile Computing," NVIDIA, Tech. Rep., 2014.

[3] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro*, vol. 30, no. 4, pp. 23–24, 2010.

[4] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *CASES '13*. IEEE Press, 2013, pp. 15:1–15:10.

[5] J. M. Kim, S. K. Seo, and S. W. Chung, "Looking into heterogeneity: when simple is faster," in *2nd IWPMP*, 2014.

[6] X. Liang, M. Nguyen, and H. Che, "Wimpy or brawny cores: A throughput perspective," *JPDC*, vol. 73, no. 10, pp. 1351–1361, 2013.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks-Summary and Preliminary Results," in *SC*. New York, NY, USA: ACM, 1991, pp. 158–165.

[8] H. C. H. Chung, M. Kang, "Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM big.LITTLE Technology," 2013. [Online]. Available: http://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf

[9] V. S. Jyothi Krishna and S. Balachandran, "Compiler Enhanced Scheduling for OpenMP for Heterogeneous Multiprocessors," in *HiPEAC EEHCO '16*, 2016, pp. 692:1–692:6.

[10] R P Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[11] "Odroid-XU3," http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127.

[12] "Big.LITTLE Switchers, Evaluation on Exynos.bL Processor," 2012. [Online]. Available: http://events.linuxfoundation.org/images/stories/pdf/klf2012_yu.pdf

[13] "CFS Scheduler," https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, linux, Tech. Rep.

[14] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "EigenBench: A simple exploration tool for orthogonal TM characteristics," *IISWC'10*, 2010.

[15] V. S. Jyothi Krishna, "OpenMP EigenBench," https://bitbucket.org/jkrishnavs/openmp-eigenbench, 2016.

[16] J. Burkardt, "C Examples of Parallel Programming with OpenMP," 2010. [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/openmp/openmp.html

[17] M. T. Flanagan, "Michael Thomas Flanagan's Java Scientific Library," 2008. [Online]. Available: http://www.ee.ucl.ac.uk/~mflanaga/java

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC '09*. IEEE Computer Society, 2009, pp. 44–54.

[19] A. Gutierrez, R. G. Dreslinski, and T. Mudge, "Evaluating private vs. shared last-level caches for energy efficiency in asymmetric multi-cores," in *SAMOS '14*. IEEE, 2014.

[20] A. Nougrahiya and K. Nandivada, "IIT Madras OpenMP(IMOP) Framework," http://www.cse.iitm.ac.in/~amannoug/imop.

[21] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[22] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)," in *39th ISCA*, ser. ISCA '12, 2012, pp. 213–224.

[23] O. Khan and S. Kundu, "A self-adaptive scheduler for asymmetric multi-cores," in *GLSVLSI '10*. ACM, 2010, pp. 397–400.

[24] D. Koufaty, D. Reddy, and S. Hahn, "Bias Scheduling in Heterogeneous Multi-core Architectures," in *5th EuroSys '10*. ACM, 2010, pp. 125–138.

[25] S. Memeti and S. Pllana, "Combinatorial optimization of work distribution on heterogeneous systems," *CoRR*, vol. abs/1606.05134, 2016.

[26] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-isa heterogeneous multi-cores," in *22nd PACT*. IEEE Press, 2013, pp. 177–188.

[27] 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, Apr. 2009.

[28] J. Chen and L. K. John, "Efficient program scheduling for heterogeneous multi-core processors," in *DAC '09*. ACM, 2009, pp. 927–930.

[29] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, pp. 64–, Mar. 2004.

[30] R. Nishtala, D. Mossé, and V. Petrucci, "Energy-aware thread co-location in heterogeneous multicore processors," in *EMSOFT '13*. IEEE Press, 2013, pp. 21:1–21:9.

[31] L. Sawalha and R. D. Barnes, "Energy-efficient phase-aware scheduling for heterogeneous multicore processors," in *2012 IEEE Green Technologies Conference*, April 2012, pp. 1–6.

[32] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu, "An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps," in *22nd PACT*, Sept 2013, pp. 63–72.

[33] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, and O. Khan, "Performance per watt benefits of dynamic core morphing in asymmetric multicores," in *20th PACT*, Oct 2011, pp. 121–130.

[34] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *DAC '13*. ACM, 2013, pp. 174:1–174:9.

[35] B. Taylor, V. S. Marco, and Z. Wang, "Adaptive optimization for opencl programs on embedded heterogeneous systems," in *LCTES 2017*. ACM, 2017, pp. 11–20.

Jyothi Krishna V. S. Jyothi Krishna is currently pursuing PhD in the CSE department at IIT Madras. His research interests are Compilers and Heterogeneous Multicore processors.

Shankar Balachandran. Shankar works at Intel Labs, Bangalore. Formerly, he was an Associate Professor in the CSE department at IIT Madras. He received his PhD from the University of Texas at Dallas. His research interests are in Computer Architecture, VLSI Design Automation, and Parallel Algorithms.

Rupesh Nasre. Rupesh is an Assistant Professor in the CSE department at IIT Madras. He completed PhD from IISc Bangalore and Post-Doctoral Fellowship from the University of Texas at Austin. His research focus is in Compilers and Parallelization.