SPARTA: Runtime Task Allocation for Energy Efficient Heterogeneous Many-cores

Bryan Donyanavard*, Tiago Mück*, Santanu Sarma and Nikil Dutt
Department of Computer Science
University of California, Irvine, USA
{bdonyana,tmuck,santanus,dutt}@uci.edu

ABSTRACT

To meet the performance and energy efficiency demands of emerging complex and variable workloads, heterogeneous many-core architectures are increasingly being deployed, necessitating operating systems support for adaptive task allocation to efficiently exploit this heterogeneity in the face of unpredictable workloads. We present SPARTA, a throughput-aware runtime task allocation approach for Heterogeneous Many-core Platforms (HMPs) to achieve energy efficiency. SPARTA collects sensor data to characterize tasks at runtime and uses this information to prioritize tasks when performing allocation in order to maximize energy-efficiency (instructions-per-Joule) without sacrificing performance. Our experimental results on heterogeneous many-core architectures executing mixes of MiBench and PARSEC benchmarks demonstrate energy reductions of up to 23% when compared to state-of-the-art alternatives. SPARTA is also scalable with low overhead, enabling energy savings in large-scale architectures with up to hundreds of cores.

1. INTRODUCTION

Modern embedded platforms (e.g. mobile devices) must support highly diverse and complex workloads that typically exhibit dynamically varying resource demands, thus requiring adaptive mechanisms for energy-efficient resource management. Previous work has shown [1, 2] that memory and computational needs vary across applications. Furthermore, a workload's characteristics may dynamically change during execution. To address this trend, emerging mobile SoCs are increasingly incorporating heterogeneity in order to balance energy efficient execution with the performance demands of applications. For instance, in *ARM's big.LITTLE* architecture [3] the *big* core has both more cache capacity and computational power (i.e. wider OoO pipeline) than the *LITTLE* core. Samsung's Exynos and Mediatek's Helio x20 are examples of platforms that both incorporate big.LITTLE cores in clusters on a single SoC. Although integrating architecturally differentiated cores may yield power-performance benefits [4], these *heterogeneous many-core platforms* (HMPs) require intelligent management by the operating system, specifically for task allocation.

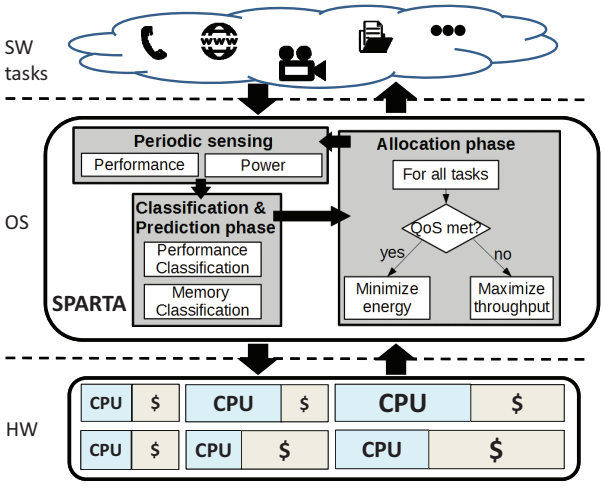


Figure 1: SPARTA role in many-core system.

However, existing operating systems are unable to fully exploit architectural heterogeneity for scalable, energy-efficient execution of dynamic workloads. Linux extensions that explicitly address this issue are limited to fixed heterogeneous platforms (e.g. ARM's GTS for big.LITTLE architectures [5]), without being adaptable to other underlying platforms. A robust solution should be adaptive enough to integrate with adverse existing and emerging HMPs. Adaptive and intelligent task allocation is necessary to meet both the system goals (e.g. maximum energy-efficiency) and the demands of diverse workloads which include both computational- and data-intensive interactive tasks concurrently, thus requiring task allocators to consider resource constraints explicitly (e.g. computational, memory).

In this paper we propose SPARTA (Figure 1), a runtime task allocator for heterogeneous many-core platforms that leverages the variability in workload memory and computational requirements in order to provide energy efficient task-to-core allocations. SPARTA's runtime adaptivity provides support for workloads that consist of numerous tasks with diverse behavior (e.g. in terms of compute and memory demands) that may enter and exit the system at any time. SPARTA defines the task allocation policy that is

∗These authors contributed equally to this work. This work was partially supported by CAPES (CSF program).

CODES/ISSS '16, October 01-07, 2016, Pittsburgh, PA, USA
 ISBN 978-1-4503-4483-8/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2968456.2968459

executed periodically by the operating system at runtime. SPARTA achieves energy efficient allocations by collecting on-chip sensor data (Section 4), using it to predict task performance/power (Section 5), and identifying opportunities to maintain a task's performance while reducing its energy consumption (Section 6). SPARTA's task allocation policies work in tandem with other run-time power management approaches such as DVFS, which further increase energy-efficiency. The main contributions of our work are:

- We propose a classification-based prediction method for predicting tasks' behavior across heterogeneous core types based on an epoch of runtime observations. Our predictor allows a comprehensive modeling of runtime performance/power by also incorporating models for Linux's ondemand DVFS governor and the completely fair scheduler (Section 5).
- We propose a task mapping heuristic that opportunistically exploits diverse workload and platform performance-power characteristics and maximizes energy-efficiency while maintaining application performance (Section 6).
- We demonstrate the efficacy of SPARTA compared to state-of-the-art alternatives for various diverse workloads executing on a real big.LITTLE-based platform, as well as simulations of 4 heterogeneous core configurations from 8 to 128 cores, showing up to 23% reduction in power without performance degradation (Section 7).

2. RELATED WORK

Several works have been proposed in the context of task mapping to heterogeneous platforms. Table 1 summarizes these works and positions SPARTA. The *in kernel switcher* (IKS) [6] is an early solution developed for ARM's big.LITTLE [3]. IKS enables only one core type to be used at a time and migrates all tasks when the overall system load reaches a certain threshold. ARM's GTS [5] improve IKS by bringing the per-core utilization awareness to the scheduler. However, they optimize for throughput only and do not consider performance variations from different memory/compute-boundness levels. Furthermore these works are limited to a specific architecture with two core types. *Koufaty* [7], *Saez* [8], and PIE [1] introduce performance models to predict workload behavior at runtime considering the different computational/memory capabilities of each core type, but do not account for power. *Annamalai et al* [9] aims at maximizing the total system energy efficiency without taking performance degradation into account. MTS [10] attempts to maximize the overall system throughput given a maximum power envelope. MTS, however, does not take into account throughput requirements of individual tasks (e.g. priority is given to tasks that provide the highest improvement in overall throughput) nor support multiple tasks mapped to the same core. Procrustes [11] applies MTS to the context of heterogeneous cores with dynamic microarchitectures (e.g. ElasticCore [12]). SmartBalance [13] provides a task mapping heuristic that optimizes the overall energy efficiency of the system. It finds near optimal mappings in terms of throughput/Watt, however it doesn't consider throughput requirements of individual tasks. *Muthukaruppan et al.* [14] proposes a price theory power management that uses application heartbeats [15] as the QoS metric and aims at minimizing power while maintaining throughput. A similar approach is proposed by *Shafik et al.* [16] and *Das et al.* [17], which both perform QoS-aware DVFS by using reinforcement-learning and control theory. In SPARTA, we similarly use throughput as the QoS metric, and consider QoS satisfied if the maximum achievable throughput (or *target throughput*) is being met (Section 4). [14, 17, 16] focus mostly on DVFS aspects ([14] focuses on clustered architectures such as ARM's big.LITTLE, while [17, 16] consider only homogeneous cores), while SPARTA focuses on the orthogonal issue of efficient task mapping to HMPs. Furthermore we use a QoS notion based on the predicted task throughput across core types, which does not require the use of external frameworks or modifications on the application.

Table 1: Summary of state-of-the-art in terms of: awarness to workload IPC variation (e.g due to varying ILP, cache missrates, etc), load/core utilization, power, and QoS (mapping attempts to meet throughput requirements); scalability of heterogeneity (# of supported core types >2); generality of task model (supports multiple tasks in the same core); implemented on a real OS/platform; and awareness to DVFS.

	Workload awareness: IPS	Load	Power	QoS	Gen. HMP	Gen. tasks	OS impl.	DVFS awr.
GTS[5]		✓		✓	✓		✓	
PIE[1]	✓	✓						
Saez[8]	✓				✓		✓	
Koufaty[7]	✓	✓		✓	✓	✓	✓	
Annam. et al[9]	✓	✓	✓					
MTS[10]	✓	✓	✓			✓		
Muth. et al.[14]		✓	✓	✓	✓		✓	✓
SmartBalance[13]	✓	✓	✓		✓	✓	✓	
SPARTA	✓	✓	✓	✓	✓	✓	✓	✓

SPARTA efficiently predicts both task performance and power at runtime, but, in contrast to previous works, SPARTA incorporates per-task QoS into its allocation decisions in order to identify power savings without sacrificing performance. Also, current solutions for performance and power prediction [1, 9, 14, 13, 11, 18] either do not support a generic task and architecture model (i.e. limit the number of tasks that can be mapped to a single core or the number of core types) or do not consider dynamic frequency variability due to ondemand DVFS. SPARTA addresses these issues with predictive DVFS models that synergistically couple task mapping, frequency scaling, and a model of the Linux CFS scheduler to support the generic Linux task model. It's worth mentioning that SPARTA is proposed in the scope of a generic Linux environment and single-ISA shared memory HMPs. Other works such as [19, 20, 21] address task partitioning in the scope of heterogeneous processing elements with multiple ISAs (e.g. CPUs vs DSPs vs FGPAs). These techniques work under specific assumptions for the runtime environment and programming models (e.g. availability of multiple implementations for the same function, application ability to reconfigure itself, no shared memory between elements, etc.), therefore are not directly comparable to SPARTA.

3. HMP ARCHITECTURE AND EXECUTION MODEL

Platform model assumptions and definitions: In this work, we consider single-ISA HMP platforms consisting of many heterogeneous cores on a single chip (Figure 1).

Throughout the paper, we define the set of cores as $C = \{c_1, c_2, \ldots, c_n\}$ and the set of core type as $D = \{d_1, d_2, \ldots, d_o\}$, such that $\gamma : C \rightarrow D$ gives the type d of a particular core c. Cores support multiple voltage/frequency (VF) pairs and DVFS. For DVFS purposes, we defined the set of clusters as $E = \{e_1, e_2, \ldots, e_q\}$, such that every core belongs to a cluster and all cores in the same cluster are set to the same VF pair. Core-level DVFS is defined by having clusters contain a single core, i.e., $|C| = |E|$. The set of frequencies supported by a core of type d is defined as F_d. Section 7 evaluates SPARTA and describes our experimental platforms in details.

Application model assumptions and definitions: We assume task are encapsulating threads similar to the Pthread model, so there is no formal or explicit dependency between tasks. Within the Linux scheduling subsystem, threads are all treated as a task entity and scheduled independently. For uniformity, in this paper the term task is used interchangeably for both single-threaded processes and for threads of the same process. Task-to-core allocations are performed periodically, and, since tasks can enter and leave the system at any time and their total execution time is unknown, we define the set of tasks to be allocated to cores $T = \{t_1, t_2, \ldots, t_m\}$ as the tasks currently active at the moment. We assume that multiple tasks mapped to the same core are scheduled according to Linux's CFS policy with the same priority. Newly created tasks are initially mapped to a random core until they can be taken into account by the SPARTA allocation approach (Section 6).

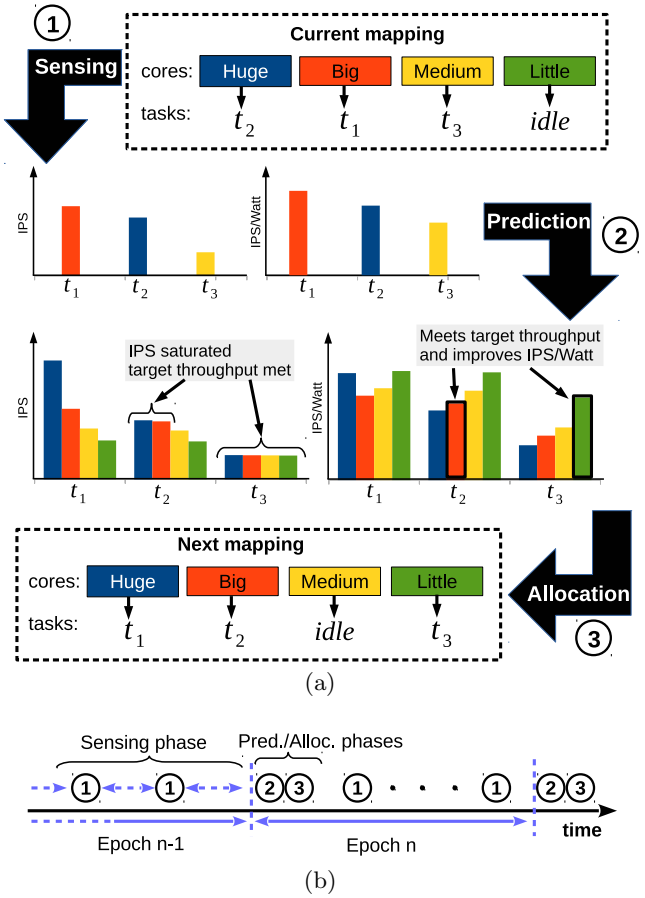
4. RUNTIME TASK ALLOCATION

SPARTA is a runtime task allocator for HMPs that consists of three phases as shown in Figure 1: *sensing*, *classification/prediction*, and *allocation*. Figure 2 gives an overview of the relationship between this three phases. As shown in Figure 2b, an *Epoch* is the time period between classify/predict and allocate phases, while the sensing phase is executed at the same rate as the Linux scheduler, so each epoch covers multiple Linux scheduling periods. Figure 2a provides and example of SPARTA's runtime task allocation. In this example, we consider three distinct tasks executing on a 4-core HMP containing one core for each heterogeneous type described by Table 3:

① During the sensing phase, hardware performance counters and power sensors are periodically sampled on each core to monitor the characteristics of the executing workload. The following counters are sampled at the same rate as the Linux scheduler (typically 10-20ms) and individually summed up for each task at the beginning of each epoch: total amount of executed instructions (I_{total}), active cycles (cy_{active}), L1 and L2 cache misses per instruction (mr_{L1I}, mr_{L1D} and mr_{L2}), and branch mispredictions per instruction (mr_{Br})[1].

With this information, we can define the *throughput* of a task t that executed in a core c in terms of *instructions per second* as $t.ips = \frac{c.I_{total}}{c.cy_{active}} * c.freq$. Note that $t.ips$ denotes the average throughput of task t across all scheduling periods that t executed. The average *effective throughput* of t across the entire epoch is defined as $t.ips * t.load$, where $t.load = \frac{c.cy_{active} * c.freq}{EpochLength}$ is the share of processing time used by task t during an epoch of duration $EpochLength$.

[1]contemporary heterogeneous architectures such as ARM's big.LITTLE support simultaneous sampling of these counters [3, 22]



(a)

(b)

Figure 2: Allocating cores to task using ① run-time sensing, ② classification/prediction and ③ throughput-aware allocation, during periodic epochs

② At the beginning of a new epoch, each task's performance and power is predicted for all core types using the sensed information. This phase allows SPARTA to obtain task metrics (e.g. $t.ips$, $t.load$) for cores in which the task has not executed on. Section 5 describes SPARTA's performance/power prediction in detail.

③ The final step is to determine the global task allocation for the next epoch. The first order goal is to meet the *target throughput* of the task and then maximize energy efficiency[2]. A task's target throughput is the maximum achievable throughput of the task on the fastest core type. *We consider a task's target throughput as being met if the task observes negligible effective throughput improvements when mapped to a faster core* (i.e. Little → Big). When multiple cores can achieve target throughput, the task is allocated to the most energy efficient one. For instance, in Figure 2, t_2 effective IPS is saturated in the *Big* and *Huge* types, meaning that these cores achieve t_2 target throughput. This saturation is the typical case for interactive task with *computation—IO/sleep* cycles. By providing a faster core to this task, it's IPS increases only during the computation cycles, which in turn decreases the core load, thus limiting the task's effective throughput.For tasks that we do not observe IPS saturation (e.g. t_1 in Figure 2), we assume that

[2]in terms of throughput(IPS) per Watt, which can also be thought of as instructions-per-Joule

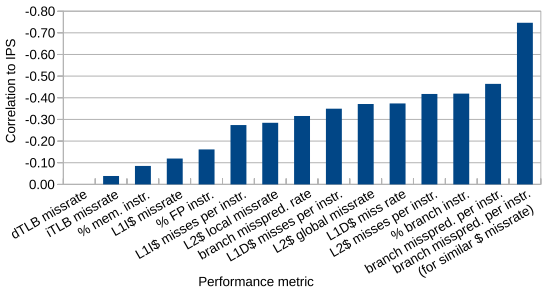


Figure 3: Correlation between various performance metrics and IPS. The rightmost bar is the correlation for workloads with similar mr_{L1D} (less then $< 10\%$ difference).

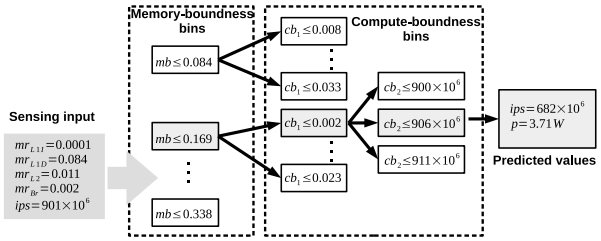


Figure 4: Example of workload classification for performance and power prediction from a *Big* core to a *Little* core (Table 3). $mb = mr_{L1I} + mr_{L1D} + mr_{L2}$ is used to bin according to memory-boundness, while $cb_1 = mr_{Br}$ and $cb_2 = ips$ are used for compute-boundness.

in the best case its target throughput can only be achieve by the fastest core (*Huge*). This is the typical case for non-interactive compute-intensive tasks with only computation cycles. Section 6 describes SPARTA's allocator in detail.

5. CLASSIFICATION AND PREDICTION

This phase predicts the task performance on all core types by using the information collected for the core on which the task is currently executing (i.e., using the workload metrics from the sensing phase). Prediction is necessary in order to avoid the overheads of periodically switching tasks between the different core types in order to obtain such information (also know as sampling [23]). In previous works, this type of prediction is most often performed using linear regression-based models [9, 18, 22, 24] due to their simplicity, while others employ a binning-based approach in which metrics sensed at runtime are used to classify workloads into categories whose performance/power are known for all core types [10].

In this work we use a binning-based approach. Our choice is motivated by the fact that linear regression typically requires parameters that have independent effects on the predicted value, which is not the case for performance prediction. Previous works have shown that workloads can be computational- or cache-capacity-bound [1, 2]. In this scenario, variations in computational-related parameters in a linear regression model have a different impact in the predicted metric when a workload is cache-capacity-bound (e.g., the impact of the number of mispredicted branches may have a negligible impact on throughput if the cache miss rate is high). In order to illustrate this scenario, we executed MiBench [25] and PARSEC [26] benchmarks on our experimental platform (refer to Section 7, Table 3 for details), and collected performance counter information. Using this information, we obtained the Pearson's correlation coefficient for metrics calculated from performance counters and the workload throughput (IPS). Figure 3 plots the average correlation across all core types described in Table 3. We observe that the cache miss and branch misprediction counters have the highest correlation with IPS. The rightmost bar in Figure 3 shows that the correlation of branch mispredictions to IPS becomes near-linear when only workloads with similar cache behavior are considered. This observation motivates us to employ a binning approach in which workloads are classified according to their memory-boundness for performance and power prediction.

Bin-based prediction: Figure 4 illustrates the basic idea of our predictor. For every core type, we define a predictor composed of different layers. The first layer classifies the workload according to memory-boundness. For every memory-boundness bin, the workload is classified according to compute-boundness metrics. The bins in the last classification layer store the predicted information for all other core types.

We used the correlations shown in Figure 3 in order to find the most relevant metrics for memory- and compute-boundness. We selected cache miss and branch misprediction counters as metrics representative of memory-boundness and compute-boundness, respectively, for the purpose of binning. As shown in Figure 4, we use two layers for compute-boundness: the branch misprediction is used as a first order metric, while the overall workload IPS is directly used in a second compute-boundness layer to cover the remaining IPS driving factors (e.g. instruction mix, TLB misses, etc).

Predictor training: Similarly to regression models, our predictor also requires offline training to define the classification bin boundaries. The first part of the training process is obtaining training samples. One training sample consists of sensing information collected after running one specific workload on all core types and frequencies. In order to generate a diverse range of training samples, we used the microbenchmarking approach described in [18]. A microbenchmark is defined as a simple function which exhibits a specific computational and/or memory behavior (e.g. high/low instruction-level parallelism, high/low cache miss-rates, etc). A total of 1536 unique workloads are obtained by combining different microbenchmarks.

Algorithm 1 describes the process for finding the classification bin boundaries. *samples* refers to the set of all training samples described previously, while *layer_metric* stands for the metric used to classify samples in a layer. When calculating the bins for the last layer, the final predicted IPS and power is set as the average IPS and power for all samples in the last bin. In order to support DVFS, a different set of bin layers is generated for all combinations of core types and frequencies.

Prediction Error: Figure 5 shows the average error in predictions across all core types. The runtime prediction of IPS and power incurs an average error ranging from 1% to 6% across all core types in Table 3. We consider this error to be small when compared to previous works that propose performance/power prediction schemes [9, 10, 18]. Comparing to previous works, *Annamalai et al* [9] reports errors ranging from about 10% to about 16%. when using a linear regression model to predict IPC/Watt across two core types. *Pricopi et al* [22] employs a similar predictor for ARM's big.LITTLE architecture. The authors report errors

Algorithm 1 Classification bins generation

1: **for all** $d_{src} \in D$ **do**
2: **for all** $f_{src} \in F_{d_{src}}$ **do**
3: **for all** $d_{tgt} \in D$ **do**
4: **for all** $f_{tgt} \in F_{d_{tgt}}$ **do**
5: FIND_BINS($first\_layer\_metric$,$samples$,d_{src},f_{src},d_{tgt},f_{tgt})

6: **function** FIND_BINS($layer\_metric$, $samples$, d_{src},f_{src},d_{tgt},f_{tgt})
7: $bins \leftarrow$ equally sized bins such that every bin has at least one sample
8: **for all** $bin \in bins$ **do**
9: $bin\_samples \leftarrow \{s \in samples | layer\_metric(s, d_{src}, f_{src}) <= bin.max\}$
10: **if** $layer\_metric$ is the last layer **then**
11: $IPS_{d_{tgt},f_{tgt}} \leftarrow average\_ips(d_{tgt}, f_{tgt}, bin\_samples)$
12: $P_{d_{tgt},f_{tgt}} \leftarrow average\_power(d_{tgt}, f_{tgt}, bin\_samples)$
13: **else**
14: FIND_BINS($next\_layer\_metric$,$bin\_samples$,d_{src},f_{src},d_{tgt},f_{tgt})

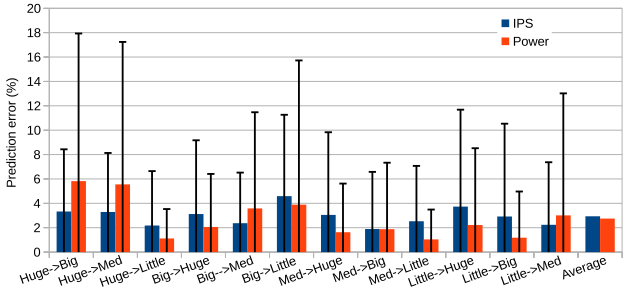


Figure 5: Average error and standard deviation of IPS and power prediction across multiple core type combinations.

of 13.4% and 16.7% for big→LITTLE and LITTLE→big performance predictions, respectively. *Liu et al* [10] uses an approach similar to ours on an architecture with four core types, with a reported error of 14% and 7% for IPS and power, respectively. *Liu et al* bins according to PARSEC benchmarks, i.e., a task whose observed performance matches one of the profiled PARSEC benchmarks on one core type is assumed to match the performance of that benchmark on all core types. Section 7.3 provides an evaluation on how prediction errors affect the outcome of task allocation.

Prediction of scheduling and DVFS-dependent metrics: As described in the previous section, the effective throughput is defined in terms of the task IPS and load. However, the task load is not directly predicted using the binning method since it is a function of the temporal activity of the other tasks to be mapped to the same core and the scheduling policy adopted by the operating system. Furthermore, dynamic load variation might trigger the OS DVFS mechanism, whose decisions may lead to further load variation. In this paper, we assume that Linux's CFS policy [27] is used to schedule tasks mapped to the same core and that Linux's ondemand DVFS governor is active. For predicting load, we initially use the CFS performance estimation model proposed by [18]. This CFS model introduces an additional *task load contribution* (TLC) metric. TLC is defined as the as maximum load a task can impose on a core, i.e., the task load when it has exclusive access to the core. The task TLC is defined on a per-task basis and can be latter used to dynamically estimate the scheduling-dependent task load given a specific combination of tasks mapped to the same core. We leverage the models proposed by [18] to estimate final task load given a fixed core frequency, the tasks' IPS when scheduled (obtained from the bin-based predictor descibed above), and the tasks' TLC). The `UpdateFreqAndLoad` function in Algorithm 2 shows how we combine these models with a DVFS prediction to obtain each task's load and effective throughput: The `predictCoreLoad` and `predictTaskLoad` take as input each task's IPS and TLC, as well as the set of all tasks mapped to the core, and implement the CFS models proposed by [18] to predict the load of each task and the total core load. The `UpdateFreqAndLoad` function then estimates the frequency that will be set by the governor in the next epoch assuming a DVFS epoch length equivalent to that of the Linux scheduler. The frequency prediction is performed based on the previous core load prediction. Since the updated frequency prediction might affect the core load, this process is repeated until the predicted frequency becomes stable or a maximum number of predictions are made (defined by $EpochLength/DVFSPeriod$). The frequency is considered stable when consecutive predictions yield the same frequency. If the maximum number of predictions are made without finding a stable frequency, the last predicted frequency is used.

Algorithm 2 DVFS-aware load prediction

function UPDATEFREQANDLOAD(core $\widehat{c}$)
$iter \leftarrow 0$
repeat
$e \leftarrow \widehat{c}$'s DVFS cluster
for all $c \in e$ **do**
▷ Core and task load given the current frequency
$c.load \leftarrow predictCoreLoad(c, T, c.freq)$
for all $t \in T, t.mapping = c$ **do**
$t.load \leftarrow predictTaskLoad(t, c, c.freq)$
$c.prev\_freq \leftarrow c.freq$
$c.freq \leftarrow$ frequency for cluster e based on DVFS governor load threshold $\forall c.load \in e$
$iter \leftarrow iter + 1$
until $iter > EpochLenght/DVFSPeriod$

6. ALLOCATION

Given a set of executing tasks and available cores, an allocation is an assignment of all tasks to a core. As described earlier, the SPARTA allocator's goal is to minimize the power consumption of a workload while maintaining its target throughput. This optimization problem is NP-hard, and is therefore infeasible to complete at runtime every epoch, even for small configurations. Therefore, we developed a heuristic solution based on *list scheduling*. The **SPARTA Allocator** presented in Algorithm 3 uses task target throughput (TT) properties to significantly reduce the complexity of finding an allocation. As described previously, a core type is considered to achieve target throughput for a task if the predicted effective throughput on the core type matches the maximum observable effective throughput for that task across all core types. Using this notion, the rationale behind the SPARTA allocator is to map each task to the most power-efficient core (in terms of IPS/Watt) that achieves the target throughput.

The first steps of Algorithm 3 are to initialize the per-task performance and power metrics described in Section 3 (lines 5–11). The `predictIPS` and `predictPower` functions implement the prediction approach described in Section 5 across all core types and frequencies. The `predictTLC` function predicts the task load contribution metric (TLC) using the Linux CFS models [18] as described in Section 5.

Next, we compute the task target throughput $t.TT$ and the number of core types that meet it (line 14). Note that we cor-

rect $t.TT$ by a constant $0 < \alpha < 1$. We currently use $\alpha = 0.95$ to account for prediction errors. Based on the throughput constraint, tasks are then separated into three lists — tasks with achievable target throughput on all core types (`full_TT_list`), tasks with achievable target throughput on a subset of core types (`partial_TT_list`), and tasks with unachievable target throughput (`no_TT_list`) (lines 15–21). All task lists are ordered for allocation by descending maximum achievable IPS/Watt for TT-satisfiable tasks, and by maximum achievable IPS for TT-unsatisfiable tasks (lines 23–25). The lists are allocated in-order to assign the tasks constrained to fewest potential core types first, i.e. first the `partial_TT_list`, followed by the `no_TT_list` and `full_TT_list` (lines 27–29).

The `allocate` function (line 31) progressively maps all of the tasks in `task_list`. For the tasks with achievable target throughput, each task is allocated in order to a core with maximum IPS/Watt that achieves target throughput (line 35). Tasks with unachievable target throughput are allocated to the core that maximizes throughput (line 38). Because each intermediate update to the mapping (line 35 and 38) can affect a task's effective throughput, `UpdateFreqAndLoad` (Algorithm 2) is called to recalculate $t.load$ and $c.freq$ every time a new possible value for $t.mapping$ is evaluated.

Algorithm 3 SPARTA Allocator

```
1: function SPARTA(tasks T, cores C, core types D)
2:   for all t ∈ T do
3:     ▷ Initialize predicted values
4:     t.TT ← 0
5:     for all d ∈ D do
6:       for all f ∈ F_d do
7:         t.ips[d][f] ← predictIPS(t, d, f)
8:         t.power[d][f] ← predictPower(t, d, f)
9:         t.tlc[d][f] ← predictTLC(t, T, C, f)
10:        if t.TT < t.ips[d][f] * t.tlc[d][f] then
11:          t.TT ← t.ips[d][f] * t.tlc[d][f]
12:    ▷ Obtain target throughput constraints
13:    t.TT ← t.TT * α
14:    t.constraint ← {d|(t.TT ≤ t.ips[d][f] * t.tlc[d][f], ∀d ∈ D, f ∈ F_d}
15:    if |t.constraint| = |D| then
16:      full_TT_list.add(t)
17:    else
18:      if |t.constraint| > 1 then
19:        partial_TT_list.add(t)
20:      else
21:        no_TT_list.add(t)
22:  ▷ Sort respective task lists according to memory-boundness
23:  sort_tasks_{constraint/IPS/Watt}(partial_TT_list)
24:  sort_tasks_{IPS}(no_TT_list)
25:  sort_tasks_{IPS/Watt}(full_TT_list)
26:  ▷ Allocate all tasks
27:  Allocate(partial_TT_list)
28:  Allocate(no_TT_list)
29:  Allocate(full_TT_list)
30:
31: function ALLOCATE(task_list)
32:  t.mapping ← ∅
33:  for all t ∈ task_list do
34:    ▷ Maximize IPS/Watt given TT constraint
35:    t.mapping ← c | γ(c) ∈ t.constraint,
                       t.ips[γ(c)][c.freq] * t.load ≥ t.TT,
                       t.ips[γ(c)][c.freq] / t.power[γ(c)][c.freq] is maximized
36:    ▷ Maximize IPS if cannot meet constraint
37:    if t.mapping = ∅ then
38:      t.mapping ← c | c ∈ C,
                       t.ips[γ(c)][c.freq] * t.load is maximized
```

SPARTA's complexity: Algorithm 3 has complexity of $|T|*|D|$, $|T|^2$, and $|T|*|C|$ for the classification, sorting and allocations phase, respectively. If we assume $|T| >> |D|$ and an average case performance of $O(|T|*log(|T|))$ for sorting, then the SPARTA's runtime is bound by $O(|T|*|C|)$.

7. EXPERIMENTAL EVALUATION

In this section we describe the implementation and evaluation of SPARTA. Our evaluation is done on two distinct platforms: 1) the ARM big.LITTLE based Exynos 5422 SoC deployed on the ODROID-XU platform[28], which contains a SoC with four big and four little cores (Table 2); and 2) a simulated platform for larger scale HMPs based on the gem5 simulator [29] integrated with McPAT [30] containing the four core types described in Table 3. The remainder of this sections is organized as follows: Section 7.1 describes our experimental framework and implementation, while Section 7.2 describes the evaluated application workloads; Section 7.3 evaluates the performance and energy efficiency of the SPARTA allocation approach with respect to state-of-the-art alternatives, as well as the accuracy of the SPARTA predictor and the overhead incurred by the entire SPARTA runtime.

7.1 Experimental framework and implementation

Linux module implementation: SPARTA is implemented as a portable kernel module that can be loaded on top of current Linux kernels (Figure 6-1). SPARTA only requires kernel support for *tracepoints* [31]. Kernel tracepoints provide means to insert hooks to call functions during specific kernel events. This mechanism is used to implement SPARTA's sensing phase.SPARTA collects performance counter information on a task-by-task basis which requires counter sampling at the granularity of tasks' context switch. A periodic kernel thread implements SPARTA's epochs and executes Algorithm 3 every $200ms$. It is worth mentioning that in our current implementation, we do not replace or disable the Vanilla Linux scheduling/task migration mechanism. We currently control the task mapping and migration process by setting each task's *core affinity mask*. We assume that the affinity masks are not changed by any other application (e.g. the *taskset* tool). Additionally, SPARTA only manages user-level tasks and tasks that are initially allowed to run on any core, and the SPARTA thread is always executed on a fixed core.

Trace Generation: We capture traces for all benchmarks on all core types of the target platform (Figure 6-2). These traces are captured by executing applications individually as a single thread on each core type at all possible operating frequencies. Traces contain periodic performance and power statistics sampled every epoch to use as input for both the predictor training (Figure 6-3) and the offline simulation framework described below.

Offline simulation framework: In order to evaluate SPARTA's scalability and adaptability to more cores and core types, we used gem5 in full system mode. However, running full system simulations in gem5 for prototyping allocation policies on platforms with 10s of cores or more proved to be impractical. Therefore, we developed an offline simulator to evaluate large-scale HMPs efficiently for a variety of core types (Figure 6-4). The simulator executes the prediction and allocation routines at every epoch using fine-grained traces. For this purpose, traces are captured using gem5 integrated with McPAT in full system mode at a granularity of 1ms epochs.

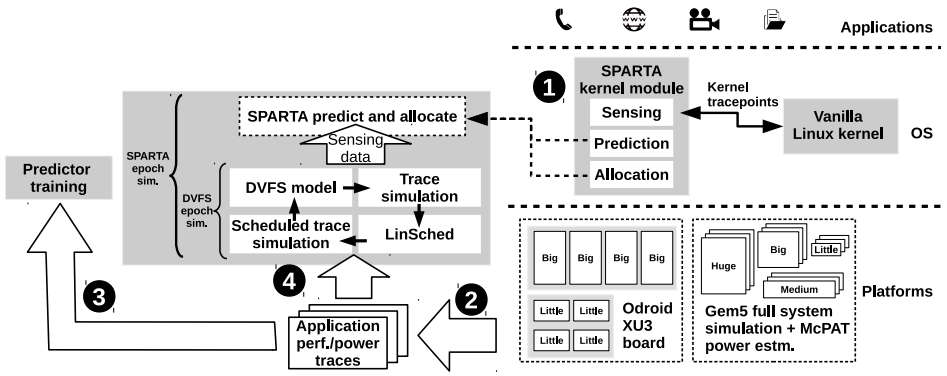


Figure 6: SPARTA framework implementation overview. (1) SPARTA phases implemented as a portable kernel module. (2) The kernel module also allows the generation of application performance/power traces that (3) are used for offline predictor training and (4) offline simulations of the prediction and allocation phases.

Table 2: Exynos 5422 core parameters

Parameter (Core type)	big (Cortex A15)	LITTLE (Cortex A7)
Issue width	4 (OoO)	2 (Inorder)
L1$I/$D size (KB)	32/32	32/32
L2 size (KB)[1]	2048	512
Max VF	2.0GHz/1.2V	1.4GHz/1.2V
Min VF	1.2GHz/1.0V	1.0GHz/1.1V

28nm technology node.
LQ/SQ,IQ,ROB, and reg. bank size information is not available.
[1]Per cluster shared L2 caches

Table 3: Simulated heterogeneous core parameters

Parameter (Core type)	Huge	Big	Medium	Little
Issue width	8 (OoO)	4 (OoO)	2 (OoO)	1 (Inorder)
LQ/SQ size	32/32	16/16	8/8	8/8
IQ size	64	32	16	16
ROB size	192	128	64	64
Int/float Regs	256	128	64	64
L1$I/$D size (KB)	64/64	32/32	16/16	8/8
L2 size (KB)[1]	512	256	128	64

22nm technology node.
Supported VF pairs:
2GHz/1V, 1GHz/0.7V, 500MHz/0.6V.
[1]Per core private L2 caches

The trace-based simulator works at the granularity of DVFS epochs and emulates the execution of each core in four steps: 1) The trace information is used to obtain maximum amount of processing time a task would use during the DVFS epoch and it's duty cycle; 2) this is provided to LinSched [32], which is used to obtain the exact runtime allotted by the OS scheduler to each task; 3) the execution of each task is emulated again for the allotted time; 4) The DVFS governor algorithm sets the frequency for the next epoch according to the emulated load.

7.2 Workloads

Our experimental workloads are constructed using two different approaches. Workloads are made up of benchmarks from MiBench and PARSEC benchmark suites, as well as synthetic microbenchmarks (described in Section 5).

The first set of workloads (Mixes 1, 5, 10, and 15 in Table 4) are diverse: the benchmark mixes comprise a combination of the most compute- and memory-bound benchmarks from PARSEC, as well as synthetic microbenchmarks that exercise different levels of compute and memory boundness. For each suite, we analyzed each benchmark's performance gain in terms of IPS when increasing core size, and selected the subset that benefited most for the respective resource types; Mix 1 consists of these PARSEC benchmarks. Mixes 5, 10, and 15 consist of synthetic workloads using microbenchmarks that can be either compute- or memory-bound (**CB** or **MB**) and impose a high, medium, or low average cpu utilization (**HU**, **MU**, or **LU**).

The second set of workloads (all other Mixes in Table 4) represent realistic homogeneous use-cases. Mixes 7, 14, and 11 consist of network, automotive/industrial, and consumer benchmarks from MiBench respectively. Mixes 12 and 3 each consist of a single benchmark from PARSEC representative of computer vision and data mining respectively. Mixes 4, 6, and 2 are composed based on the analysis performed by [33], which characterizes mobile use cases in terms of core utilization. We use the our microbenchmarks to define four typical mobile workloads based on the resulting utilization of big and little cores: Mix 4 is a typical load; Mix 6 is a typical load with a heavy task; Mix 2 is a heavy load. Mixes 8, 9, and 13 consist of a combination of *x264* with different frame-rate inputs in order to invoke further target throughput variation.

For all experiments, the total number of threads is half the number of cores in the configuration, which we believe is a valid use-case for heterogeneous many-cores.

7.3 Simulation results

We first evaluate all components of SPARTA on platforms composed of the core types defined in Table 3 using the offline simulator. We compare SPARTA against the following allocators: MTS, GTS, and an algorithm that maximizes energy efficiency. MTS is an implementation of the allocation algorithm with no power budget that we consider the most comparable state-of-the-art solution. GTS is ARM's big.LITTLE allocation technique that we adapted to work for more than two core types. Since GTS the only solution adopted in mainstream Linux, we will use it as a baseline. The goal of both MTS and GTS is to maximize throughput. Maximum Energy Efficiency (Max EE) is a optimal brute-force allocation for maximizing IPS/Watt. For the following experiments we use an oracle predictor – predictor with no

Table 4: Benchmark Mix composition

Mix 1	Mix 2	Mix 3	Mix 4	Mix 5	Mix 6	Mix 7	Mix 8	Mix 9	Mix 10	Mix 11	Mix 12	Mix 13	Mix 14	Mix 15
bodytrack	CB/HU	streamcluster	CB/LU	CB/MU	CB/LU	CRC32	x264 5fps	x264 2fps	CB/HU	jpeg	bodytrack	x264 15fps	basicmath	CB/HU
streamcluster	CB/MU		CB/MU	MB/MU	MB/MU	blowfish	x264 15fps	x264 5fps	CB/LU	typeset		x264 30fps	bitcount	CB/LU
	(heavy)		(typical)		(typical heavy)	patricia							qsort	MB/HU
						dijkstra							susan	MB/LU

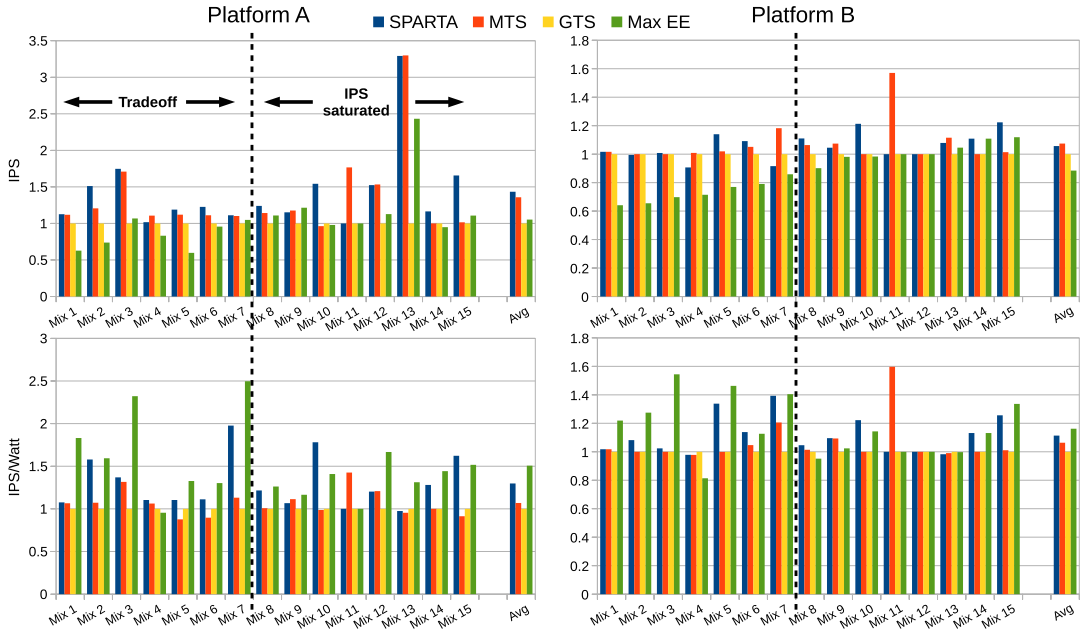


Figure 7: IPS and IPS/Watt of allocators for different benchmark mixes normalized to GTS on two different 8-core HMP configurations: Platform A with two of each core type, and Platform B with four Big and four Little cores.

error – in order to evaluate only the allocation techniques. Additionally, we use our SPARTA prediction scheme for all allocators, which means that although MTS is not DVFS-aware by design, in our experiments it benefits from our improved DVFS-aware predictor. GTS is unaffected as it is reactive and therefore does not use any prediction.

Figure 7 shows the IPS and IPS/Watt of all allocators for all benchmark mixes on two 8-core configurations normalized to the GTS baseline. In **Platform A**, the eight cores are made up of two Huge, Big, Medium, and Little cores each. **Platform B** is made up of four Big and four Little cores similarly to the Odroid platform. For benchmark Mixes 8-15 on Platform A, SPARTA is able to match the performance of the state-of-the-art technique while increasing energy efficiency in most cases. We observed an average of 26% increase in energy efficiency over GTS for Mixes 8-15 while increasing throughput by 57%. Similarly, we observed an average 19% increase in energy efficiency over MTS with 8% performance improvement for these mixes. This can be attributed to the fact that while both MTS and GTS prioritize maximizing throughput, they do not consider the utilization of each thread on different core types. By accounting for core utilization, SPARTA can find the most energy efficient thread-core mapping for threads whose performance is saturated. Mixes 8-15 mostly contain atleast one benchmark that does not completely utilize all core types. The results for these workload types show that SPARTA is able to efficiently identify opportunities to maximize energy efficiency without degradation of the target throughput.

Mixes 1-7 in Figure 7 represent the alternative scenario: when a tradeoff in performance is required to save energy. This is illustrated clearly by the Max EE allocator's poor throughput and high energy efficiency for these mixes. The manner in which SPARTA prioritizes the most energy efficient threads during allocation allows us to also beneficially exploit this scenario. As before, by recognizing when multiple core types can achieve a task's target throughput, SPARTA is able to improve energy efficiency for cases in which performance is saturated. For benchmark Mixes 1-7, SPARTA observed 27.5% and 6.5% average improvement in throughput compared to GTS and MTS respectively. SPARTA also increased energy efficiency by 33% and 27% on average over GTS and MTS respectively. These benchmark Mixes mostly consist of high performance tasks whose performance is unbound by resource allocation. For tradeoff scenarios in which the target throughput is not commonly met, SPARTA is able to achieve significant energy savings while maintaining performance.

Platform B follows a similar trend to Platform A, but with less significant deviation between SPARTA, MTS, and GTS. This is due to the platform only having two different core types. SPARTA is still able to improve performance by 5.5% and energy efficiency by 11% over GTS on average over all mixes. Compared to MTS, 5% increased energy efficiency comes at the cost of negligible performance degradation (2%).

Figure 8 compares scalability, showing the average per-

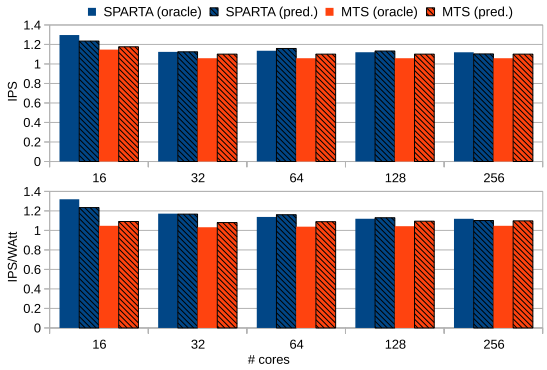


Figure 8: Scalability: average IPS and IPS/Watt for different core configurations normalized to GTS.

formance and energy efficiency of the heuristic allocators for configurations with up to 256 cores and 128 tasks. The values are averaged over all of the Mixes mentioned previously for each configuration. Each configuration also had two permutations: one with an equal distribution of the four core types (1:1:1:1 ratio), and another with a ratio of 1:3:3:1 cores of types Huge, Big, Medium, Little respectively. For configurations of 32 cores or more, SPARTA and MTS both improved throughput over GTS between 5-10%. SPARTA was able to save up to 17% and 10% energy over GTS and MTS respectively.

Figure 8 also includes evaluation of allocators using the SPARTA predictor. The predictor had minimal impact on the allocators. In some cases, predictor error resulted in improved allocations. The predictor did not affect the throughput or energy efficiency achieved by SPARTA or MTS by more than +/-5% relative to the oracle versions. The SPARTA and MTS relative relationship will always hold because they are equally a function of their predictor's quality. A more accurate predictor could be incorporated into SPARTA in the future in order to yield increased benefit over GTS.

7.4 ODROID platform results

We also evaluates SPARTA on the ODROID-XU3 platform[28]. In this platform we use Linux kernel version 3.10.9 which implements GTS. SPARTA is implemented as a separate kernel module loaded after bootup. Figure 9 shows the IPS and IPS/Watt of SPARTA for a subset of benchmark mixes normalized to the GTS baseline. Overall, SPARTA matches the throughput of GTS while improving energy efficiency by 16%. Similarly to Platform B in Figure 7, overall most mixes differ minimally or not at all between SPARTA and GTS. In some cases, like Mixes 2 and 10, SPARTA or GTS make slight tradeoffs between performance and energy efficiency. For Mix 13, SPARTA sacrifices 13% throughput for a 40% energy efficiency gain. In the few outlier cases in which the difference is not negligible and no tradeoff is made, SPARTA is able to identify opportunities to improve energy efficiency at no cost (Mixes 8 and 9) or improved performance (Mix 11). Based on these results, we can conclude that both the overhead of the SPARTA runtime as well as the predictor error is manageable on real systems.

SPARTA's overheads and scalability: We measure the run time of each SPARTA phase on the Odroid platform. The total latency of each phase is relatively low compared to the 200ms epoch length. The total time spent sensing during

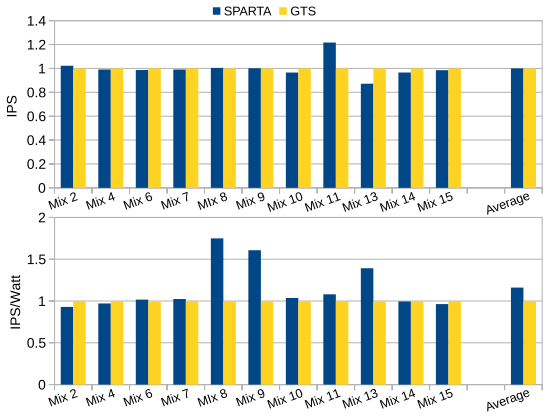


Figure 9: IPS and IPS/Watt of allocators for different benchmark mixes normalized to GTS on the Odroid platform.

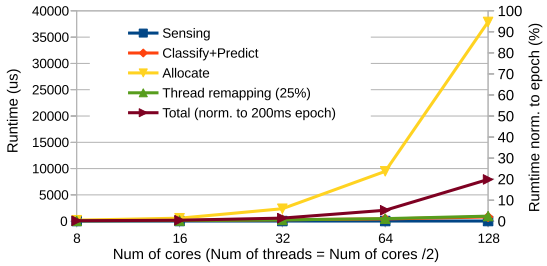


Figure 10: Runtime scalability of SPARTA phases for different number of cores using a 50ms epoch.

an epoch on all cores averages to 28μs. Across the 8-core benchmark runs described previously, the measured latency of the classification/prediction and allocation phases is 124μs and 190μs respectively (measured on big cores). Figure 10 plots how the latencies scale for larger systems and includes the thread migration overhead (assuming 25% of threads migrate every epoch and that each migration takes about 20μs1GHz[34]). The total overhead imposed by SPARTA is negligible for up to 16 cores. The classify and predict phase is the most cumbersome of the SPARTA overhead sources, but it still bellow 5% of the epoch length for up to 64 core configurations. For platforms with 100s of cores the overhead is high, but it can be mitigated by the fact that the execution of prediction and allocation phase occurs in parallel with the execution of every other task in the system. Nevertheless, this is a limitation of our current implementation of SPARTA that we plan to address in future work.

Overall, the experiments have illustrated SPARTA's applicability across both contemporary HMPs as well as potential future HMPs with increased resources and resource types. The SPARTA predictor can predict behavior on a per-task basis for diverse workloads consisting of unknown tasks in conjunction with multitasking and DVFS. The allocator successfully identifies opportunities to reduce workload energy consumption by considering target throughput.

8. CONCLUSION AND FUTURE DIRECTIONS

In this work we presented SPARTA, a throughput-aware runtime task allocation approach for aggressive HMPs that employs a sense-predict-allocate approach to achieve energy efficiency. We defined a bin-based runtime prediction method

and an efficient task allocation heuristic that are compatible with DVFS, and evaluated on a real platform. Our experimental results showed energy reductions up to 23% when compared to state-of-the-art alternatives while maintaining performance on simulated platforms, and 16% compared to Linux running on a real mobile system. This holds for large configurations, saving 10% energy with no throughput degradation for up to 256 cores. Although SPARTA is a runtime task allocator, there are some opportunities to extend the adaptivity of SPARTA. As future work, we plan to move from fixed epoch lengths to adaptive epochs that adjust based on the workload variability. We also can extend the runtime predictor to update the classification bins at runtime based on the workload to potentially improve predictor accuracy. The version of heterogeneity addressed in this initial work is simplified by only assuming variation in core parameters scale uniformly, and we plan to expand this to a more comprehensive decoupled model of heterogeneity in following extensions.

9. REFERENCES

[1] K. Van Craeynest *et al.*, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *39th Annual Int. Symposium on Computer Architecture*, vol. 40, no. 3, jun 2012, pp. 213–224.

[2] W. Heirman *et al.*, "Undersubscribed threading on clustered cache architectures," in *Proc. - Int. Symposium on High-Performance Computer Architecture*, 2014, pp. 678–689.

[3] P. Greenhalgh, "big. LITTLE Processing with ARM Cortex-A15 & Cortex-A7," ARM, Tech. Rep. September 2011, 2011.

[4] R. Kumar *et al.*, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *ACM SIGARCH Computer Architecture News*, vol. 32, p. 64, mar 2004.

[5] ARM, "big. LITTLE Technology : The Future of Mobile," ARM, Tech. Rep., 2013.

[6] M. Poirier, "In Kernel Switcher: A solution to support ARM's new big.LITTLE technology," 2013. https://events.linuxfoundation.org

[7] D. Koufaty *et al.*, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. of the 5th European Conf. on Computer systems*, 2010, p. 125.

[8] J. C. Saez *et al.*, "A comprehensive scheduler for asymmetric multicore systems," in *Proc. of the 5th European Conf. on Computer systems*, 2010, p. 139.

[9] A. Annamalai *et al.*, "An opportunistic prediction-based thread scheduling to maximize throughput/watt in AMPs," in *Proc. of the 22nd Int. Conf. on Parallel architectures and compilation techniques*, sep 2013, pp. 63–72.

[10] G. Liu *et al.*, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *IEEE 31st Int. Conf. on Computer Design*, oct 2013, pp. 54–61.

[11] G. Liu *et al.*, "Procrustes: Power Constrained Performance Improvement Using Extended Maximize-then-Swap Algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 0070, pp. 1–1, 2015.

[12] M. K. Tavana *et al.*, "ElasticCore: enabling dynamic heterogeneity with joint core and voltage/frequency scaling," in *Proc. of the 52nd Annual Design Automation Conf.*, 2015, pp. 1–6.

[13] S. Sarma *et al.*, "SmartBalance: A Sensing-Driven Linux Load Balancer for Energy Efficiency of Heterogeneous MPSoCs," in *Proc. of the 52nd Annual Design Automation Conf.*, 2015, pp. 1–6.

[14] T. S. Muthukaruppan *et al.*, "Price theory based power management for heterogeneous multi-cores," in *Proc. of the 19th Int. Conf. on Architectural support for programming languages and operating systems*, 2014, pp. 161–176.

[15] H. Hoffmann *et al.*, "Application heartbeats," in *Proc. of the 7th Int. Conf. on Autonomic computing*, 2010, p. 79.

[16] R. A. Shafik *et al.*, "Learning Transfer-Based Adaptive Energy Minimization in Embedded Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, pp. 877–890, jun 2016.

[17] A. Das *et al.*, "Hardware-software interaction for run-time power optimization: A case study of embedded Linux on multicore smartphones," in *IEEE/ACM Int. Symposium on Low Power Electronics and Design*, vol. 2015-Septe, jul 2015, pp. 165–170.

[18] T. Mück *et al.*, "Run-DMC : Runtime Dynamic Heterogeneous Multicore Performance and Power Estimation for Energy Efficiency," in *Int. Conf. on Hardware/Software Codesign and System Synthesis*, 2015.

[19] J. R. Wernsing *et al.*, "The RACECAR heuristic for automatic function specialization on multi-core heterogeneous systems," in *Proc. of the 2012 Int. Conf. on Compilers, architectures and synthesis for embedded systems*, 2012, p. 81.

[20] Sheng Yang *et al.*, "Adaptive energy minimization of embedded heterogeneous systems using regression-based learning," in *25th Int. Workshop on Power and Timing Modeling, Optimization and Simulation*, sep 2015, pp. 103–110.

[21] E. M. G. Trainiti *et al.*, "A Self-Adaptive Approach to Efficiently Manage Energy and Performance in Tomorrow ' s Heterogeneous Computing Systems," in *2016 Design, Automation & Test in Europe Conf. & Exhibition*, Dresden, 2016, pp. 906–911.

[22] M. Pricopi *et al.*, "Power-performance modeling on asymmetric multi-cores," in *Int. Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, sep 2013, pp. 1–10.

[23] M. Becchi *et al.*, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proc. of the 3rd Conf. on Computing frontiers*, 2006, p. 29.

[24] K. Singh *et al.*, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Computer Architecture News*, vol. 37, p. 46, jul 2009.

[25] M. R. Guthaus *et al.*, "MiBench : A free , commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[26] C. Bienia *et al.*, "The PARSEC benchmark suite," in *Proc. of the 17th Int. Conf. on Parallel architectures and compilation techniques*, 2008, p. 72.

[27] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[28] Hardkernel, "ODROID-XU," Tech. Rep., 2016. http://www.hardkernel.com/main/main.php

[29] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, aug 2011.

[30] S. Li *et al.*, "The McPAT Framework for Multicore and Manycore Architectures," *ACM Transactions on Architecture and Code Optimization*, vol. 10, pp. 1–29, apr 2013.

[31] M. Desnoyers, "Using the Linux Kernel Tracepoints," Tech. Rep., 2015. https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

[32] J. Calandrino *et al.*, "LinSched : The Linux Scheduler Simulator," in *Proc. of the ISCA 21st Int. Conf. on Parallel and Distributed Computing and Communications Systems*, 2008, pp. 171–176.

[33] C. Gao *et al.*, "A study of mobile device utilization," in *2015 IEEE Int. Symposium on Performance Analysis of Systems and Software (ISPASS)*, no. September 2014, mar 2015, pp. 225–234.

[34] K. Yu, "big.LITTLE Switchers," in *2012 Korea Linux Forum*, 2012.