AdaMD: Adaptive Mapping and DVFS for Energy-Efficient Heterogeneous Multicores

Karunakar R. Basireddy, *Student Member, IEEE*, Amit Kumar Singh, *Member, IEEE*, Bashir M. Al-Hashimi, *Fellow, IEEE*, and Geoff V. Merrett, *Member, IEEE*

***Abstract*—Modern heterogeneous multicore systems, containing various types of cores, are increasingly dealing with concurrent execution of dynamic application workloads. Moreover, the performance constraints of each application vary, and applications enter/exit the system at any time. Existing approaches are not efficient in such dynamic scenarios, especially if applications are unknown, as they require extensive offline application analysis and do not consider the runtime execution scenarios (application arrival/completion, and workload and performance variations) for runtime management. To address this, we present AdaMD, an adaptive mapping and dynamic voltage and frequency scaling (DVFS) approach for improving energy consumption and performance. The key feature of the proposed approach is the elimination of dependency on offline profiled results while making runtime decisions. This is achieved through a performance prediction model having a maximum error of 7.9% lower than the previously reported model and a mapping approach that allocates processing cores to applications while respecting performance constraints. Furthermore, AdaMD adapts to runtime execution scenarios efficiently by monitoring the application status, and performance/workload variations to adjust the previous DVFS settings and thread-to-core mappings. The proposed approach is experimentally validated on the Odroid-XU3, with various combinations of diverse multithreaded applications from PARSEC and SPLASH benchmarks. Results show energy savings of up to 28% compared to the recently proposed approach while meeting performance constraints.**

***Index Terms*—Energy savings, heterogeneous multicores, multithreaded applications, run-time management.**

I. Introduction

MODERN mobile platforms are containing greater number of heterogeneous cores to support highly diverse and varying workloads (e.g., the Odroid-XU3 [1] and Mediatek X20 [2]). Such platforms often execute applications concurrently, which simultaneously contend for system resources and typically exhibit varying resource demands over time [3]. Each application may have different performance requirements and exhibit various workload phases during its execution [4]. To adapt to such dynamic scenarios, mobile platforms offer an increasing number of resource configurations, such as enabling and disabling cores of different types, defining the thread-to-core mapping for a multithreaded application, and setting dynamic voltage and frequency (DVFS) operating points.

The process of thread-to-core mapping and setting DVFS levels play a crucial role in exploiting the system properties such that applications can meet their, often diverse, demands on performance, and energy consumption [3]. In general, for each application, the management process first finds a thread-to-core mapping, and then core DVFS level by inspecting the workload profile while satisfying the performance requirement. This problem becomes much more complex when dynamically mapping concurrently executing applications due to contention for resources, and when the mapping is coupled with DVFS, i.e., energy-efficient allocation of processing cores and selection of DVFS settings [5], [6].

The reported approaches for solving this problem fall into three categories: 1) offline; 2) online; and 3) hybrid approaches. Several offline approaches have been proposed targeting different application domains and hardware architectures [7], [8]. These typically use computationally intensive search methods to find the optimal or near-optimal mapping for the applications that may run on the system. Conversely, online approaches [4], [9]–[11] must not be computationally intensive, as they are required to make efficient application mapping/DVFS decisions at runtime. Therefore, these techniques generally use heuristics to find a suitable platform configuration. Design time approaches usually find solutions of higher quality compared to online techniques, due to extensive design space exploration of the underlying hardware and applications. To address the drawbacks of pure offline and online approaches, various hybrid approaches [8], [12]–[17] using offline analysis to make runtime decisions based on the current state of the system are proposed.

However, a review of the prior arts (see Section VI) shows that the existing approaches, targeting heterogeneous multicores, have the following shortcomings. They use heavy application-dependent profile data and thus are not efficient in managing dynamic workloads when unknown applications with different performance constraints are executing concurrently. For example, the number of different frequency and core configurations for the Odroid-XU3 platform [1] (four big

Manuscript received January 9, 2019; revised April 23, 2019 and July 15, 2019; accepted July 26, 2019. Date of publication August 13, 2019; date of current version September 18, 2020. This work was supported in part by the Engineering and Physical Sciences Research Council under Grant EP/L000563/1, and in part by PRiME Programme under Grant EP/K034448/1 (www.prime-project.org). This article was recommended by Associate Editor H. Li. *(Corresponding author: Karunakar R. Basireddy.)*

K. R. Basireddy, B. M. Al-Hashimi, and G. V. Merrett are with the School of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, U.K. (e-mail: krb1g15@ecs.soton.ac.uk; bmah@ecs.soton.ac.uk; gvm@ecs.soton.ac.uk).

A. K. Singh is with the School of Computer Science and Electronic Engineering, University of Essex, Colchester CO43SQ, U.K. (e-mail: a.k.singh@essex.ac.uk).

Digital Object Identifier 10.1109/TCAD.2019.2935065

and four LITTLE cores that can operate at 13 and 19 different frequencies, respectively) is 4080 $((4 \times 13 \times 4 \times 19) + (4 \times 13) + (4 \times 19))$. Most importantly, all these approaches do not perform adaptations (changing the mappings and/or DVFS settings) at an application arrival/completion, and performance variations. To this end, this article presents AdaMD, an adaptive mapping approach coupled with DVFS for performance-constrained multithreaded applications, executing on heterogeneous multicores. AdaMD selects an resource combination (number of cores and their type) that meets the application's performance requirement while minimizing energy consumption. This is achieved by employing performance prediction models for resource combination enumeration and selection. Furthermore, the application workload, performance and its status (finished or newly arrived) are monitored for adaptive resource allocation and DVFS. The key contributions of this article are as follows.

1) A performance prediction model that has a maximum percentage error of 8.1%, which is 7.9% lower than the previously reported model [17].
2) An online mapping approach that allocates processing cores to application(s) based on performance constraints without using any application-dependent offline results.
3) To adapt to application arrival or completion times, and workload/performance variations, an adaptive approach that adjusts the existing thread-to-core mappings and DVFS settings during application execution is presented.
4) Experimental validation of the proposed approach on the Odroid-XU3 [1], using several multithreaded applications from PARSEC [18] and SPLASH [19] benchmarks.

The remainder of this article is organized as follows. Section II presents a motivational example for this article, while Section III presents the problem definition for this article. A detailed description of the proposed AdaMD approach is given in Section IV. The experimental setup and validation of our approach are explained in Section V. Section VI discusses the related work and highlights the difference between the proposed approach and exiting works. Finally, Section VII concludes this article.

II. Motivation

A heterogeneous computing system with two types of cores, executing multiple performance-constrained applications concurrently, is illustrated in Fig. 1. Dotted squares colored in white/black represent processing cores. For example, such scenarios could be observed when a smartphone user simultaneously runs a music player, Facebook, background e-mail service, downloading a file, etc. As shown in Fig. 1(a), the initial mapping for each application (App1, App2, and App3) is decided based on its performance constraints while considering the energy as an optimization goal. This requires finding an energy-efficient resource combination (number of cores and their type). While these applications are executing, there are primarily three runtime execution scenarios possible: 1) any application(s) may finish executing; 2) an application(s) may experience performance degradation due to contention for shared resources; and 3) a new application(s) may arrive into the system. In the first case, if application

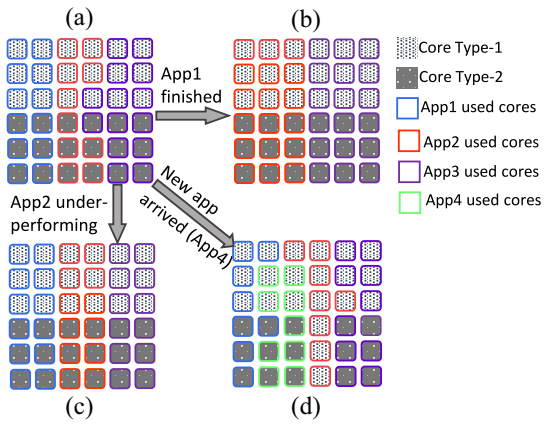

Fig. 1. Motivational example showing three possible runtime execution scenarios (b), (c), and (d) when a system, having two types of cores—Type-1 and Type-2, starts with executing three performance-constrained applications (a). Cores running the same application are encircled with a line of the same color. App1, App2, App3, and App4 represent user applications.

App1 finishes execution, its resources can be allocated to App2 and App3, which may help them execute faster (and hence put them into a low-power mode sooner), as shown in Fig. 1(b). This may result in increased performance and lower energy consumption, because power is dissipated for a shorter duration.

For case 2), as reported by previous work [5], [20], applications go through different workload phases during their execution. For example, some workload phases could be more compute-intensive than others or vice versa. Furthermore, in case of concurrent execution, an application may experience interference from other applications due to shared resources, such as last level cache, memory, etc. All the factors above culminate into variation in an application's workload, subsequently leading to variation in application performance. Therefore, the application's performance has to be monitored periodically, and appropriate action (changing the DVFS setting or remapping) taken to avoid/minimize performance violations. Fig. 1(c) demonstrates such a case, where more resources are allocated to App2 to mitigate the performance degradation experienced during runtime. If there are no free cores available, as in our case, the cores are taken from the over-performing App3.

For case 3), considering the processing capabilities of the underlying hardware, the user may launch a new application while other applications are running. If all the processing cores have been allocated to the already running applications, the runtime management software should check if there are possibilities to readjust the current mapping and allocate resources to the newly arrived application without violating performance constraints. This is shown in Fig. 1(d), where App4 is added to the system while App1, App2, and App3 are executing. The resources of over-performing applications App1 and App3 are allocated to App4 while keeping the same number of cores for App2.

As discussed before, existing approaches do not consider the above execution scenarios (case 1), 2) and 3)) for adaptation and moreover, they also depend on extensive offline characterization and/or instrumentation of the chosen applications.

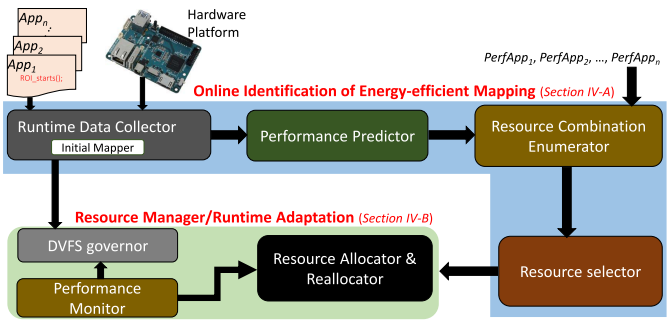

Fig. 2. Overview of the proposed AdaMD approach, showing the different steps taken.

As experimentally demonstrated in Section V, adaptation at application arrival and completion, and workload/performance variations would lead to better utilization of the system resources, and higher energy savings and performance.

III. Problem Formulation

Earlier studies have shown that the thread-to-core mapping problem alone is NP-complete [3]. Therefore, combining it with DVFS would increase the complexity of mapping problem due to the huge design space, thereby making the runtime management significantly inefficient. Similarly, if the number of cores or heterogeneity or frequency levels increases, the design space becomes too large for solving at runtime and even for offline analysis [5]. To address this, as per literature, we consider thread-to-core mapping and DVFS separately to minimize the runtime overheads. The following forms our problem definition.

Given: A set of performance-constrained applications to be executed concurrently or at different moments of time on a heterogeneous multicore platform supporting DVFS.

Find: An initial thread-to-core mapping for each application and then apply DVFS and/or adaptive remapping at runtime to minimize the energy consumption, if any of the following occur.

1) An existing application finishes or a new application arrives into the system.
2) The performance constraints of any running applications are violated.
3) The workload of an application varies during execution (e.g., from compute-intensive to memory-intensive).

Subject to: Meeting the performance requirement of each application without violating the resource constraints (number of available cores in the platform).

IV. Proposed Adaptive Mapping and DVFS Approach

This section presents a detailed discussion of the proposed AdaMD, an adaptive thread-to-core mapping and DVFS approach. An outline of the proposed approach is presented in Fig. 2 and corresponding pseudocode in Algorithms 1 and 2. The arriving performance-constrained applications are added to the queue, called `Apps`, and the *initial mapper* allocates a processing core to each application in the queue. Meanwhile, the *Runtime Data Collector* periodically gathers necessary runtime information through performance monitoring counters (PMCs) for the performance predictor, DVFS governor, and performance monitor. The *Performance Predictor* estimates the application performance, using instructions per cycle (IPC) or instructions per second (IPS), on various types of cores by using the runtime information collected on a single type of core. The estimated performance of an application on various types of cores is then utilized for enumerating the list of resource combinations (the number of cores and their type) that meet the performance constraints of the application (*Resource Combination Enumerator*). Next, the *Resource Selector* picks the resource combination that would lead to lower energy consumption. Finally, the *Resource Manager* keeps track of the variation in application performance, workload and completion/arrival time to decide on adjusting the previous mappings and DVFS settings. The following discusses each step of the proposed methodology in detail.

TABLE I
Parameters Used in the Proposed Approach

Number of Active Cores
Frequency of the Cores
L1 I-Cache Misses
L1 D-Cache Misses
L2 Cache Misses
Instructions Retired
Branch Misses
CPU Cycles
Per Core CPU Utilisation
Memory Reads Per Instruction

A. Online Identification of Mapping

Proposed AdaMD approach first identifies thread-to-core mapping that minimizes energy consumption for each performance-constrained application in a concurrent execution scenario without using offline profiled results. This process involves the following steps.

1) Runtime Data Collector: The proposed approach requires various parameters for making runtime decisions while concurrent applications are executing on the platform. These parameters are collected by the Runtime Data Collector. The list of parameters used in this article is given in Table I. Of these parameters, `CPU Cycles`, `Instructions Retired`, and `L2 Cache Misses` are periodically collected to measure memory reads per instruction (MRPI), per core CPU Utilization, and IPC or IPS for detecting the workload and/or performance variations by the DVFS governor and Performance Monitor (details are given in Section IV-B). The performance monitoring unit (PMU) of the processor is initialized to monitor the above parameters through the routine `PMU_initialize()` (line 1, Algorithm 1). Note that all the parameters are collected only when an application(s) arrives into the system, which are used by the Performance Predictor. When an application arrives, the *Initial Mapper* adds it to the application queue and allocates a free core to the application to start application execution (lines 3–9, Algorithm 1). As application execution begins with the serial section, the initial mapper tends to allocate a big core to the application. However, if an application's serial

Algorithm 1 AdaMD Mapping and Adaptation

Input: Applications and performance constraints (Apps)
Output: ∀Apps, mappings and DVFS settings

```
1: PMU_initialize()  // initialises PMCs
2: while (1) do
3:    if (NewApp) then
4:       Update the Application Queue 'Apps';
5:       NewApp = 0;
6:    end if
7:    for ∀i ∈ Apps do
8:       if (unmapped) then
9:          Allocate a free core 'l' to 'i' and execute;
10:         Measure MRPI and move onto an appropriate core (j);
11:         /*Data collection for performance model*/
12:         Wait until ROI begins;
13:         pmcs = pmcs_data_collect(j);
14:         f = cpufreq_get_freq_hardware(j);
15:         pmcs.push_back(f);
16:         η = speedup_estimate(pmcs,j);
17:         Compute resource combination meeting perf. constraint
            and having min. energy t_h (Eq. (4), (5) & (6));
18:         Allocate resources as per t_h;
19:      end if
20:   end for
21:   /*Distribute the free cores to active applications*/
22:   Sort the applications by η (list);
23:   while (freecores>0) do
24:      Increase the resources of app i ∈ list by y;
25:      freecores = freecores - y;
26:      i++;
27:   end while
28:   /*Application performance and workload adaptation*/
29:   If application workload changes call DVFS(); //Algorithm 2
30:   for i ∈ Apps do
31:      if App 'i' under-performs then
32:         Increase frequency or allocate more cores;
33:      end if
34:   end for
35:    /*Application completion detection and adaptation*/
36:   if p ∈ Apps finishes then
37:      Distribute freed resources of 'p' to under-performing apps;
38:      Allocate remaining resources to apps equally by sorting
         them based on η;
39:   end if
40:   /*if stop_governor is set, process exits*/
41:   if (stop_governor) then
42:      PMU_terminate(); //Terminates PMC collection
43:      exit(0);
44:   end if
45: end while
```

section is memory-intensive, measured by MRPI, the application is migrated to a LITTLE core as it results in a greater power efficiency [21] (line 10, Algorithm 1). Data collection starts in the region of interest (ROI) (indicating the parallel code in the application) as that is when actual computation starts and the benefit of allocating more than one processing core can be seen [18]. This is accomplished by notifying the Runtime Data Collector through the ROI_starts() routine when the ROI of an application starts, which is identified by the hook__parsec_roi_begin() [18] (lines 12–15, Algorithm 1). If an application does not support such hooks, handshaking mechanism can be used that informs runtime manager when threads are spawned [e.g., call to pthread_create()]. This can be implemented using the existing interprocess communication methods (e.g., shared memory variables, message queues, etc.).

The runtime data for ROI region is collected every 50 ms for the first 500 ms and their average values are fed into the performance predictor.

2) Performance Predictor: To allocate resources in a heterogeneous multicore system to meet the performance requirements of an application, it is essential to know how the application performs on various types of cores [21]. This can be achieved either by executing the application on all types of cores in a platform or by estimating the performance of application for different types of cores by running only on one core type. The former approach requires the migration of the application across various core types. As observed experimentally in [21], migration cost across clusters on a big.LITTLE architecture is relatively high: 2.10 ms to move a thread from a LITTLE cluster to a big cluster, and 3.75 ms to move from a big cluster to a LITTLE cluster. This overhead grows with the number of cores and types. Considering the runtime overheads and scalability, this is not an efficient approach. However, this approach would not need offline analysis as everything is measured at runtime. On the other hand, a performance prediction-based approach avoids thread migration by using the performance models built offline or online. Previous approaches have shown that learning performance models at runtime would make the approach nonscalable and has its overheads in terms of execution time and power [5], [15]. Therefore, AdaMD builds the performance models at design time through a generalized methodology, which can easily be adopted to a new platform/architecture.

3) Performance Models: Application performance is usually measured in terms of IPS or IPC, and the relative improvement in the performance is referred to as *speedup*. We define speedup η as

$$\eta = \frac{\text{IPC}_{\text{CoreType1}}}{\text{IPC}_{\text{CoreType2}}} \tag{1}$$

where, $\text{IPC}_{\text{CoreType1}}$ and $\text{IPC}_{\text{CoreType2}}$ are the IPC of the application achieved on core type-1 and core type-2, respectively. The performance model estimates the speedup, which is used for computing the application performance on a second core type ($\text{IPC}_{\text{CoreType2}}$), by running the application on one core type and collecting the runtime parameters, and measuring its performance ($\text{IPC}_{\text{CoreType1}}$) (line 16, Algorithm 1).

To build the performance models, three steps are followed. The first step is identifying the parameters/metrics that capture the most performance-limiting factors by analyzing the correlation between various metrics and speedup. Modern processors support monitoring of various architectural events which can be used for analyzing the performance, power, etc. However, not all metrics that contribute to performance can be monitored simultaneously due to the limited number of hardware PMCs provided by the platform. For example, on an Odroid-XU3/XU4, the Cortex-A15 processor allows

monitoring of seven events, including the cycle counter, at a time. Therefore, metrics that contribute more to the speedup have to be identified. Based on our analysis and the information given in [21] and [22], we have identified that cache misses (L1 I/D-Cache & L2 Cache), branch misses, CPU cycles, and instructions retired are the appropriate PMCs for estimating the speedup on our chosen platform (listed in Table I). The second step is the collection of characterization data for a diverse set of applications. As part of this, we have created a diverse set of workloads, containing single, and multithreaded applications from SPEC CPU2006 [23], LMBench [24], RoyLongbottom [25], PARSEC 3.0 [18], SPLASH [19], and MiBench [26]. The Odroid-XU3 platform has four Cortex-A7 and four Cortex-A15 cores that can operate at 19 and 13 different DVFS levels, respectively. For each application, data has been collected for every 50 ms at all available frequencies on the platform. Furthermore, in the case of multithreaded applications, the number of threads/cores are varied from one to four (number of available cores for each type). In each case, six PMCs, frequency of the big and LITTLE CPUs, execution time of the application on the big cluster and LITTLE cluster, and the number of active cores, are all used in the modeling. For consistent results, each experiment is repeated ten times, and corresponding average values are considered while create the model. To create a more general approach for deriving performance models, we explored several statistical and machine learning techniques. Using the open source WEKA workbench [27] to verify the relationship between input features/attributes and output/target variables. Of all the explored methods, we found that additive regression of decision stumps, using boosting for a regression problem, resulted in good accuracy as shown in Section V-B.

The problem of function estimation usually consists of a random *output* variable y and a set of random *input* features $X = \{x_1, x_2, \ldots, x_n\}$. Given a training sample $\{y_i, X_i\}_1^N$ of known (y, X) values, the objective is to identify a function $\hat{f}(X)$ that relates X to y, such that the expected value $(E_{y,X})$ of some specified error function $\psi\ (y, f(X))$ is minimized

$$\hat{f}(X) = \arg\min_{f(X)} E_{y,X}\psi(y, f(X)). \tag{2}$$

In general, boosting approximates $\hat{f}(X)$ by an additive expansion of the form, i.e., adding a set of base learners [28], as

$$f(X) = \sum_{k=0}^{M} \alpha_k h(X; \beta_k). \tag{3}$$

Here, the base learner functions $h(X; \beta)$ are simple functions of X with parameters $\beta = \{\beta_1, \beta_2, \ldots, \beta_M\}$ and $\{\alpha_k\}_0^M$ are expansion coefficients. Owing to simplicity, decision stump (one-level decision tree) is used as a base learner in this article. In brief, additive regression takes an initial guess for the speedup (the average speedup observed by all applications in the training set) and estimates the speedup by summing positive and negative additive-regression factors to $f_0(X)$. Each additive-regression factor is associated with an input feature the factor depends on. As the base learner is a decision stump, the input feature is associated with two regression factors, i.e., each of $\{h(X; \beta_k)\}_0^M$ produces one positive/negative additive-regression factor depending on the value of the input feature. Additive-regression factors are computed in a forward stage-wise manner to minimize the squared error of the predictions after M iterations, which decides the number of base learners. Readers can refer to [28] for more details on additive regression.

4) Resource Combination Enumerator: For each application, a set of all possible resource combinations (number of cores and their type) meeting performance constraints has to be computed to choose the one that minimizes the overall energy consumption (line 17, Algorithm 1). Let R be the set of possible resource combinations on a platform, and PerfApp_i is the performance constraint for an application App_i, then the performance meeting thread-to-core mappings (T_{map_i}) can be defined as follows:

$$T_{\text{map}_i} = \left\{ r \in R | \texttt{perf}(r) \leq \text{PerfApp}_i \right\}. \tag{4}$$

Here, $\texttt{perf}(r)$ defines the performance of an application when executed on the resource combination r. For simplicity, let us take our chosen platform, the Odroid-XU3, with two types of cores: big (B) and LITTLE (L); N_b and N_l are set of big and LITTLE cores, respectively. Then, $\texttt{perf}(r)$ is computed as

$$\texttt{perf}(r) = n_b \times \eta \times \text{IPC}_l + n_l \times \text{IPC}_l + \text{IPC}_o \tag{5}$$

where, $\eta = \text{IPC}_b/\text{IPC}_l$, performance on the big and LITTLE core is denoted by IPC_b and IPC_l, respectively. Furthermore, $n_b \in N_b$, $n_l \in N_l$, and $r = n_l \cup n_b$. IPC_o is the performance overhead incurred when an application is mapped onto cores that do not share a cache. For instance, the big and LITTLE clusters in the Odroid-XU3 do not share caches, which results in an intercluster communication overhead when the threads of an application run on both the big and LITTLE clusters. As shown in 5, for our chosen platform with eight cores, near linear speedup is expected with increase in number of cores [29]. Even if there is an error in estimation, this would anyway be compensated by performance monitor (Section IV-B2).

5) Resource Selector: The job of *resource selector* is to minimize the energy consumption by selecting a resource combination with minimum energy from the performance meeting thread-to-core mappings $T_{\text{map}_i} = \{3L, 4L, 1L+1B, \ldots\}$, where L and B refers to big and LITTLE cores, respectively. This can be achieved by selecting a thread-to-core mapping $t_h \in T_{\text{map}_i}$ that has the highest performance per watt (PPW) (line 17, Algorithm 1)

$$t_h = \arg\max_{t \in T_{\text{map}_i}} \text{PPW}(t) \tag{6}$$

where, $\text{PPW}(t)$ is computed as the ratio between IPC achieved for the resource combination "$t \in T_{\text{map}_i}$" and its power consumption. This requires measuring the power consumption using on-chip power sensors or employing a power model when a platform does not have power sensors [30]. However, the power model would also require the collection of various PMCs data at regular intervals of time, and its PMCs may be different than the ones used by performance models [21]. This would need multiplexing PMCs, leading to runtime overheads. To address this, the estimated speedup η can be used

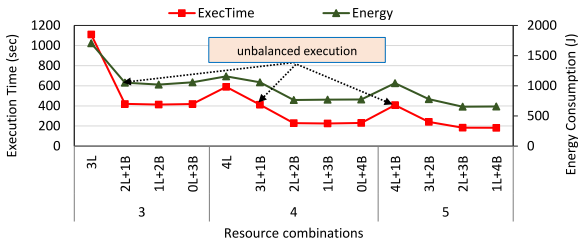

Fig. 3. Energy and execution time at different resource combinations of big (B) and LITTLE (L) for the application `Bodytrack` from PARSEC [18], executing on the Odroid-XU3.

as a proxy for identifying the energy-efficient resource combination when power sensors are not available. This is achieved by choosing a resource combination with the ratio between the minimum number of big cores to the minimum number of LITTLE cores (C_r) is higher/close to the speedup. As big core can execute η times faster than LITTLE core, above resource selection strategy leads to balanced workload sharing between big and LITTLE cores by executing η times more threads on big than LITTLE. This would lead to efficient utilization of big cores and supports the balanced execution of an application. For example, if the speedup of an application is $2\times$, then the algorithm initially tends to allocate 2-big cores and 1-LITTLE core. This is also demonstrated in Fig. 3, where unbalanced execution resulted in increased execution time and energy consumption. This figure also shows that applications with a speedup greater than one will benefit in terms of energy and performance from allocating more number of cores, as C_r reaches one or higher.

Furthermore, if η is less than 1, all LITTLE cores are allocated as the application does not benefit from executing on big cores in terms of performance/power. This makes the proposed algorithm effective for single-threaded applications as well, where it maps memory-intensive applications ($\eta \leq 1$) onto LITTLE cores, and compute-intensive ($\eta > 1$) onto big cores. Finally, the output of the resource selector is a resource combination with lower energy consumption and minimum resources that are required for meeting the performance constraints. The information about minimum resources is used by the resource manager.

B. Resource Manager/Runtime Adaptation

The *Resource Manager*, shown in Fig. 2, is responsible for adapting to application arrival/completion, performance/workload variation, and managing resources at runtime. It consists of the Resource Allocator/Reallocator, Performance Monitor, and DVFS governor. These are discussed in detail in the following sections.

1) Resource Allocator/Reallocator: The Resource Allocator manages finding free cores and allocating them to the application based on its selected resource combination (line 18, Algorithm 1). This is done by keeping track of allocated cores and free cores available in the platform. The allocated cores are maintained per application, which are used by the performance monitor for measuring application performance and for releasing the resources when the application finishes. While allocating the resources to an application, the resource allocator keeps the knowledge of cores that are leading to over-performance of an application, called extra cores. After finishing the allocation of resources to the applications in application queue (`Apps`), if there are still free resources available, these are allocated to the running applications if the energy consumption can be minimized by reducing the application execution time. The allocation of extra resources is done by first creating a sorted list of active applications in descending order of their speedup. Then, application `i` at the top of the list is selected, and its allocated cores are increased by one. This process is repeated for remaining applications in the list until no free cores are left (lines 22–27, Algorithm 1). Note that applications with $\eta < 1$ in the list are given only LITTLE cores as they do not benefit from big cores in terms of energy efficiency.

The Resource Reallocator keeps track of application completion and arrival of new applications into the system. When an application completes execution, it invokes the reallocation routine after releasing the allocated resources (lines 36–39, Algorithm 1). The reallocation routine then distributes the freed resources to the active applications. First, it measures the performance of each application (IPC or IPS) to check if any application is under-performing, i.e., measured performance is lower than the given performance constraint. If an application is under-performing, it then computes the amount of performance loss (the difference between achieved performance and given performance constraint), and then estimates the required resources using (5) to compensate it. If any resources are remaining after allocating the freed resources to under-performing applications, these resources are distributed among the applications as described in the previous paragraph. As discussed in Sections IV-B2 and IV-B3, application performance/workload adaptation is also performed to avoid performance violations as application may experience contention from other applications or workload may change over the time. This may occur at any time during application execution. Therefore, to increase the resource utilization, free cores are distributed to active applications first. Furthermore, when a new application arrives into the system, the resource reallocator tries to identify and allocate the resources as per t_h (6). This is done by checking if there are enough free resources available in the platform to satisfy the application requirements. In case free resources are not available for meeting performance constraints, the extra cores of over-performing applications are used. After doing this, if the application requirements are still not met, application execution is continued using the available resources until any running application completes and releases allocated resources.

2) Performance Monitor: Applications usually exhibit varying workload profiles (e.g., compute-intensive to memory-intensive and vice versa) during execution. When multiple applications are executing simultaneously, the workload profile of each application gets affected due to contention on shared resources [20]. As a result of this, application performance will vary over time, and may lead to the violation of performance constraints. To address this, each application's performance is periodically monitored to detect and compensate when performance constraint is violated (lines 30–34, Algorithm 1). An application performance is measured by collecting PMCs

Algorithm 2 DVFS Governor (`DVFS()`)

1: $MRPI_p = 0$, $util_p = 0$, $e_m = 0$, $e_u = 0$;
2: */*Per-core DVFS supporting platforms*/*
Input: for each core 'i', $MRPI[i]$ and $f_{req}[i]$
Output: voltage-frequency ($V\text{-}f[i]$) for next epoch
3: pmcs = `get_pmc_data(`i`)`;
4: compute actual MRPI ($MRPI_a$) = $\frac{instructions\ retired}{L2\ cache\ misses}$
5: compute actual utilisation ($util_a$) = $\frac{active\ CPU\ cycles}{TotalCPUcycles}$
6: $MRPI_p$ = `predict_mrpi(`$mrpi_p$, $mrpi_a$, e_m`)`;
7: MRPI prediction error (e_m) = $mrpi_a - mrpi_p$;
8: $util_p$ = `predict_utilisation(`$util_p$, $util_a$, e_u`)`;
9: utilisation prediction error (e_u) = $util_a - util_p$;
10: $V\text{-}f[i]$ = `bin_classify(`$util_p$, $mrpi_p$`)`;
11: **if** ($V\text{-}f[i] < f_{req}[i]$) **then**
12: $V\text{-}f[i] = f_{req}[i]$;
13: `cpufreq_set_frequency(`i, $V\text{-}f[i]$`)`;
14: **end if**
15: */*cluster-wide DVFS supporting platforms*/*
16: **for** each cluster 'j' **do**
17: Measure MRPI and utilisation of each core $i \in j$;
18: Compute the minimum MRPI ($mrpi_a$) and utilisation ($util_a$);
19: Repeat steps 6 to 13.
20: **end for**

corresponding to instructions retired and CPU cycles on all the cores that the application is currently running on. When an application's performance constraint is violated, either the operating frequency is increased, or more cores are allocated. Raising the operating frequency is given priority over assigning more cores as the latter incurs a migration overhead which is relatively large compared to the DVFS transition latency [21]. The operating frequency is increased in steps of 200 MHz until the performance constraint is satisfied and this frequency (f_{req}) is communicated to DVFS governor (discussed in the next section) to make sure it does not scale down the frequency below this value. After the above step, if any of the applications are still under-performing, as the last solution, more cores are allocated from the available free cores or extra cores of over-performing applications. This allocation is done by computing the performance loss and corresponding required cores using (5). As already explained in Section IV-B1, for applications with $\eta < 1$, LITTLE cores are preferred over big cores.

3) DVFS Governor: Applications go through different workload phases (e.g., compute-intensive, memory-intensive, etc.) and this necessitates choosing a different frequency for each workload phase to reduce the power consumption while maintaining application performance within the bounds. For example, a memory-intensive workload can be executed at a lower frequency than a compute-intensive workload with no/negligible performance loss [20]. To this end, AdaMD adopts the technique proposed in [31], modified to take f_{req} into account. Algorithm 2 presents the pseudocode of the DVFS governor.

This approach employs a binning-based approach with two classification layers (line 10). The first layer, consisting of utilization bins, classifies the compute-intensity, and the second layer classifies the memory-intensity using MRPI bins. The classification bins are computed through an offline analysis of 81 diverse workloads, including 25 from SPEC CPU2006 [23], 20 from LMBench [24], 11 from RoyLongbottom [25], 11 from PARSEC 3.0 [18], and 14 from MiBench [26]. For each application, offline profiling data consisting of MRPI, utilization and application performance (1/Execution time) are collected at different DVFS settings available on the chosen platform. The collected utilization and MRPI for various applications are then grouped into utilization bins and MRPI bins, and a corresponding voltage–frequency setting is assigned to each bin of the second classification layer. At runtime, the DVFS governor measures the MRPI and utilization and uses workload prediction to set an appropriate DVFS level (lines 3–9). To avoid violation of performance constraints, the frequency is never scaled down below f_{req} (lines 11–14). Workload prediction is based on exponential moving average filter. Prediction error during previous time epoch for MRPI (e_m) and utilization (e_u) is used as feedback to improve the workload prediction accuracy (lines 7 and 9). Furthermore, it can manage both per-core (lines 2–14), i.e., supporting fine-grained power management [32], and cluster-wide DVFS platforms (lines 15–20). For more details on binning-based DVFS approach, readers can refer to [31] and [33].

V. Experimental Results

This section presents the details of the experimental setup, covering the platform, benchmark applications, and reported approaches considered for the comparison. Furthermore, an evaluation of the performance prediction models and benefits of the AdaMD approach over the previous approaches are discussed, including associated overheads.

A. Experimental Setup

1) Platform: We use the Odroid-XU3 [1], containing the ARM big.LITTLE technology-based Samsung Exynos 5422 chip. This has four ARM Cortex-A15 (big) cores, four ARM Cortex-A7 (LITTLE) cores. The platform supports per-cluster DVFS, and all cores within a cluster can only run at the same DVFS level. The big cores have a range of frequencies between 0.2 GHz and 2.0 GHz with a 0.1 GHz step, whereas the LITTLE cores can vary their frequencies from 0.2 GHz to 1.4 GHz in steps of 0.1 GHz. The device firmware automatically adjusts the voltage for a selected frequency. The platform also contains four real-time current sensors that facilitate measurement of power consumption of each CPU cluster, GPU and memory. We used Ubuntu OS with kernel version 3.10.96. Energy consumption is computed as the product of average power consumption (dynamic and static) and application execution time. This includes both the core and memory energy consumption of all the software components, including our implementation, OS, applications, and other background processes.

2) Implementation: The proposed AdaMD approach is implemented as a *user space* application by using the `Perfmon2` [34] and `cpufrequtils` framework. `Perfmon2` enables the *user space* access to the PMU, and `cpufrequtils` helps in setting/getting the operating frequencies. Standard Linux API [`sched_setaffinity(2)`] is used to control the CPU affinity of processes, i.e., to bind the applications to specific cores. The thread-to-core mapping algorithm operates at a coarser granularity (500 ms) considering its higher

migration overhead. As the workload of application changes randomly, to capitalize on these changes for energy savings, the DVFS governor is operated at a finer granularity of 100 ms.

3) Applications: To evaluate AdaMD, applications—Blackscholes (bl), Bodytrack (bo), Swaptions (sw), Freqmine (fr), Vips (vi), Water-Spatial (wa), Raytrace (ra), fmm (fm))—from popular benchmark suites, such as PARSEC 3.0 [18] and SPLASH [19], are taken. These applications exhibit different memory behavior, data partitions, and data sharing patterns. Different execution scenarios—single application, concurrent execution of multiple applications, dynamic addition of application(s) at runtime—are also considered to mimic the real-world behavior. To ensure the deterministic execution of application and to meet its performance constraint, no two applications share the same cores. However, the threads of the same application share the allocated cores to maximize resource utilization. For each application, performance constraints are defined in terms IPC. Such performance requirements can be translated to throughput requirements for frame-based applications like audio/video applications, where throughput is expressed as a frame rate to guarantee a good user experience.

4) Comparison: To show the benefits of our approach AdaMD compared to the state-of-the-art, the selected comparison candidates from the relevant reported works are given below.

1) *HMP+x [35]:* The state-of-the-art solution for big.LITTLE multiprocessing, the heterogeneous multiprocessing (HMP) scheduler, with various default Linux power governors x [= Ondemand (O), Conservative (C), and Interactive (I)] is considered. For a fair comparison, we ran applications with different numbers of threads and chose the one meeting the performance constraint.
2) *MIM [14]:* This approach maps application threads onto only one type of core(s) based on workload memory-intensity, called a memory-intensity-based mapping (MIM). For the single-application execution scenario, a memory-intensive application is mapped onto LITTLE cores, whereas a compute-intensive one is executed on the big cores. In a multiple-application scenario, applications are sorted based on their memory-intensity, and the one with the highest memory-intensity is mapped onto LITTLE cores, and remaining applications are allocated onto the big cluster with an equal number of cores.
3) *EAM [15]:* An energy-efficient mapping is selected through an exhaustive search of voltage–frequency settings and thread-to-core mappings. For each possible thread-to-core mapping, voltage–frequency settings are varied from the lowest possible value to the highest and the one that meets performance requirement with the lowest energy consumption is chosen. We refer to this approach as energy-aware mapping (EAM).
4) *ITMD [6]:* This approach uses offline analysis of energy and performance for individual applications to decide on an energy-efficient mapping when multiple applications are run concurrently. Furthermore, it also applies workload classification-based DVFS periodically to minimize the power consumption.

B. Evaluation of Performance Predictor

The performance prediction model estimates the performance of the big core given the performance of a LITTLE core (P_{bl}) and vice versa (P_{lb}). The number of base learners (decision stumps) M in (3) impacts the model accuracy and runtime overhead. We tested our model over 148 distinct samples to evaluate the model accuracy in IPC estimation and the corresponding box plot of percentage error distribution for P_{bl} and P_{lb} are given in Fig. 4(a) and (b), respectively. As shown, the error range gets narrower with the number of decision stumps, as it would help in better predicting the speedup. Furthermore, increasing the number of decision stumps also reduces the outliers, shown as cross in Fig. 4(a) and (b), improving model stability. However, choosing more decision stumps could increase the runtime overhead, and sometimes accuracy of the prediction may not be improved after reaching a certain number of decision stumps. Therefore, to balance this, we built additive regression models for different numbers of decision stumps. It can be seen from Fig. 4(a) and (b) that the improvement in model accuracy is negligible after 900 and 1100 decision stumps for P_{bl} and P_{lb}, respectively. Therefore, we have chosen these numbers for our models P_{bl} [mean absolute percentage error (MAPE) = 1.57%; maximum error (ME) = 8.1%] and P_{lb} (MAPE = 3.45%; ME = 8.5%). The ME of P_{bl} and P_{lb} is about 7.9% and 5% lower compared to the previous model [17], respectively. The prediction accuracy of P_{lb} is 1.88% worse than P_{lb} and requires 200 extra decision stumps. This is because the LITTLE cores support accessing only four PMCs simultaneously, compared to six PMCs supported by big cores.

C. Comparison of Energy Consumption

This section presents the energy consumption results for various approaches to show the benefits of the proposed AdaMD approach. Fig. 5 shows the energy savings achieved by the AdaMD with respect to reported approaches for different single and concurrently executing applications (launched at the same time). We observed substantial energy savings compared to the reported approaches for all the application execution scenarios. For single application execution (bl, bo, sw, fr, wa, and ra), with our approach AdaMD, average energy savings of 30.7%, 25.8%, 27.3%, 37.4%, 21.8%, and 7.8% are observed compared to HMP+O, HMP+C, HMP+I, MIM, EAM, and ITMD, respectively. Furthermore, for concurrent execution of two and three applications, AdaMD shows 25.5%, 22.4%, 26.5%, 37.5%, 24.8%, and 14.2% lower energy consumption than HMP+O, HMP+C, HMP+I, MIM, EAM, and ITMD, respectively. In the single application scenario, we observed that ITMD, EAM, and AdaMD chooses a similar thread-to-core mapping, however, the energy savings observed are mainly because of the proposed DVFS technique. Unlike, ITMD and EAM, AdaMD takes the thread synchronization overhead into account while selecting a voltage–frequency setting. In concurrent execution scenarios, the energy savings are due to both DVFS and the utilization of freed resources of a finished application for active applications.

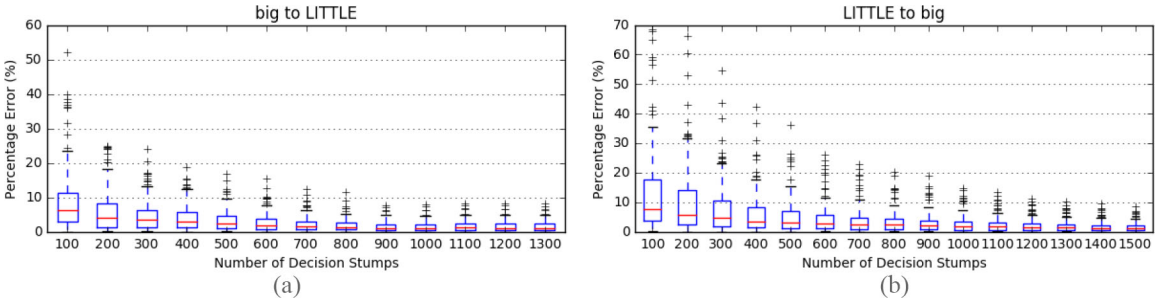

Fig. 4. Box plot of absolute percentage error in IPC prediction by our performance model for different number of decision stumps used in the additive regression, showing the median, lower quartile, upper quartile, and outliers. (a) Estimating the performance of LITTLE given the information about the big core. (b) Estimating the performance of big given the information about the LITTLE core.

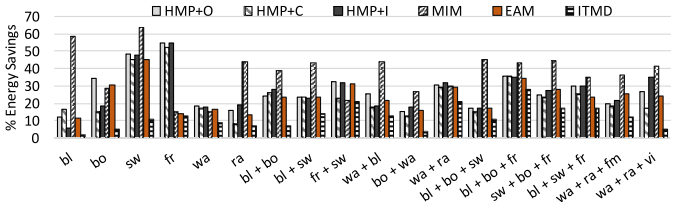

Fig. 5. Percentage improvement in energy consumption achieved by the AdaMD compared to reported approaches for single and concurrent applications.

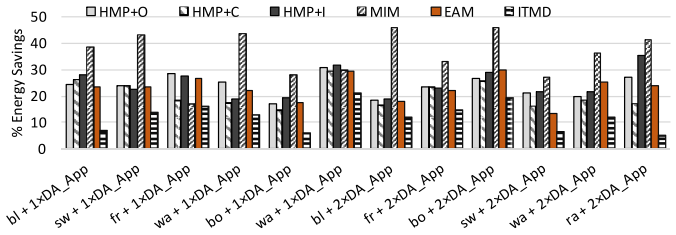

Fig. 6. Energy savings achieved by the AdaMD with respect to different approaches for one and two applications added dynamically to the system while an application is executing.

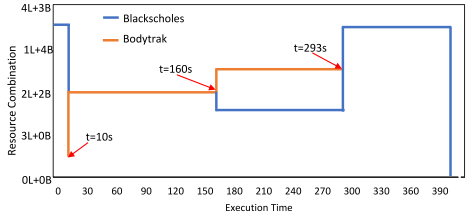

Fig. 7. Resource combination [number of big (B) and LITTLE (L) cores] allocated to `Blackscholes` and `Bodytrack` by the proposed AdaMD approach to adapt to application arrival/completion and performance variation.

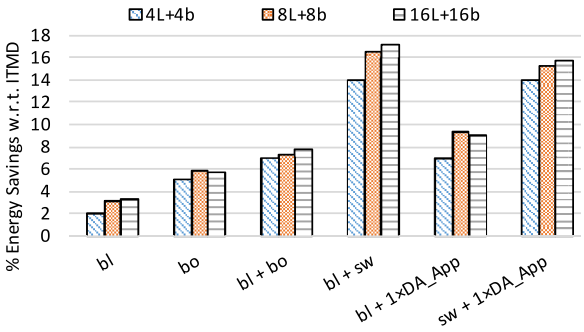

Fig. 8. Scalability of AdaMD for different core configurations of big (B) and LITTLE (L) cores: energy savings achieved by AdaMD with respect to ITMD.

Furthermore, to demonstrate the adaptiveness of AdaMD to application arrival, the following experimental evaluation is performed. The execution starts with one application and later, one (+1×`DA_Apps`) or two (+2×`DA_Apps`) applications are added at runtime. The dynamically added applications, abbreviated as DA_Apps in Fig. 6, are from those mentioned in Section V-A3. The advantages of the AdaMD with respect to other approaches in terms of energy consumption are shown in Fig. 6. On an average, AdaMD reduces the energy consumption by 23.8%, 20.6%, 24.8%, 35.8%, 12.2%, and 23.0% compared to HMP+O, HMP+C, HMP+I, MIM, EAM, and ITMD, respectively. To illustrate AdaMD's ability to adapt to different runtime scenarios, we plotted the resource combination (number of active core and their type) versus execution time for Blackscholes and Bodytrack in Fig. 7. While Blackscholes is executing with four LITTLE and three big cores (4L+3B), Bodytrack is added to the system at $t = 10$ s. Considering the performance constraints of Bodytrack, 2B+2L are allocated to Bodytrack by freeing the cores from over-performance of Blackscholes. Due to the workload variations, Bodytrack experiences performance loss at $t = 160$ s, thereby triggering the Resource Reallocator to readjust the mappings of both Blackscholes and Bodytrack. Upon Bodytrack's completion at $t = 293$ s, the freed cores are again allocated to Blackscholes as it can benefit from faster execution to lower the energy consumption.

Fig. 8 demonstrates the scalability of AdaMD, showing energy savings with respect to ITMD for two different core configurations (8L+8b, 16L+16b). The reported values in the figure have been obtained through analytical analysis of experimental results (performance and energy) collected on the Odroid-XU3 (4L+4b) and extrapolating for the considered application execution scenarios. We used linear extrapolation that takes runtime overheads associated with each application as it varies depending upon workload characteristics (e.g., frequent workload variations may incur DVFS transition latencies/thread migration overheads). As can be seen, AdaMD is able to adapt to increased design space and achieve energy savings. The increase in energy savings is mainly due to proposed

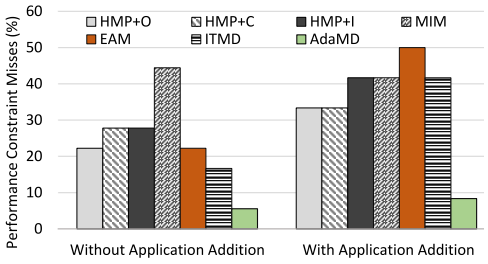

Fig. 9. Evaluation of various approaches in meeting application performance constraints.

DVFS which exploits the synchronization overheads and workload variations to lower power consumption of more number of active cores.

D. Performance

The proposed approach outperforms all reported approaches in meeting application performance constraints, as shown in Fig. 9. We evaluated the percentage of performance constraint misses for all the application scenarios presented in Fig. 5 (without application addition) and Fig. 6 (with application addition). For the without application addition case, AdaMD meets application performance constraint for 95% of the considered application scenarios, i.e., 17 out of 18 cases, shown on the horizontal axis in Fig. 5. The only scenario where the AdaMD fails to satisfy the performance requirements is `bl+sw+fr`. Even in this case, except for `sw`, the performance constraints of `bl` and `fr` are met. This is mainly because of the diverse workload profiles of the three applications and relatively higher performance requirements chosen for `sw` (approximately 2× compared to `bl` and `fr`). In case of application addition at runtime, AdaMD is able to satisfy performance constraints for 92% of the evaluated scenarios, i.e., out of 12 scenarios shown in Fig. 6, except for `sw +2×DA_Apps`, the performance constraints are met.

Compared to the recently reported approach ITMD [6], AdaMD achieves energy savings of up to 28% (for `bl+bo+fr`). Further, AdaMD satisfies performance constraints for up to 95% of the application scenarios (80% better than ITMD).

E. Runtime Overheads

To compute the runtime overheads of the AdaMD, we measured the amount of time that the algorithm takes to complete various steps (A to E) explained in Section IV. Steps A (Runtime Data Collector), B (Performance Predictor), C (Resource Combination Enumerator), and D (Resource Selector) are triggered when an application arrives into the system, whereas step E (Resource Manager) operates periodically. The runtime overheads can be analytically represented as follows for each time epoch (500 ms):

$$T_o = T_{\text{AdaMap}} + \eta \times T_{\text{DVFS}} \tag{7}$$

where

$$T_{\text{AdaMap}} = T_{pmc_m} + T_{pm} + T_{th} + T_{\text{rar}} \tag{8}$$

$$T_{\text{DVFS}} = T_{pmc_{vf}} + T_{\text{metrics}} + T_{wp} + T_{\text{classify}} + T_{vfs} \tag{9}$$

where, T_{pmc_m}, T_{pm}, T_{th}, T_{rar}, $T_{pmc_{vf}}$, T_{metrics}, T_{wp}, T_{classify}, and T_{vfs} represent time taken for PMC data collection for mapping; performance prediction; identification of resource combination; resource allocation/reallocation; PMC data collection for DVFS; computation of MRPI and utilization; workload prediction; finding DVFS setting through classification bins; and DVFS transition latency, respectively. Note that performance prediction happens only when a new application is launched, therefore the overhead T_{pm} may not be present in every time epoch. Moreover, the runtime overhead T_{DVFS} is multiplied by a factor of 2.5 (η), as it operates at a finer granularity of 100 ms compared to the mapping time interval of 500 ms.

We observed an average runtime overhead of 600 μs and 1.4 ms for A to D when executed at 2 GHz and 1 GHz on a big core of Odroid-XU3, respectively. The DVFS part of step E incurs 320 μs and others parts take up to 15 μs when the overhead is measured at the maximum frequency (2 GHz). The DVFS algorithm operates at a granularity of 100 ms, so the overhead is less than 0.5%. Performance and Resource manager part of E is invoked for every 500 ms. The overhead associated with this part depends on the number of times the application misses its performance constraint and thread migrations across the cores. Here, we observed an overhead between 0.15% to 0.75%. Our results show that the total runtime overhead is very minimal and moreover, they have been included when computing energy consumption and performance.

VI. Related Work

To achieve energy savings and/or to meet performance constraints in multicore platforms, various approaches for DVFS and/or task mapping have been proposed [3]–[17], [20], [31], [36]–[41]. These works perform offline, online or hybrid (offline & online) optimization for resource management.

Approaches based on offline optimization utilize extensive design space exploration of the underlying hardware and target application(s). The techniques proposed in [7] and [40] are used for DVFS and/or task mapping. However, they consider execution of a single application at a time, and thus are not suitable for the concurrent execution of applications. The approach presented in [40] generates multiple mappings for each application offering a tradeoff between resource requirements and throughput, while Quan and Pimentel [8] proposed scenario-based online mapping approaches targeting homogeneous multicore platforms in which mappings derived from design-time DSE are stored for runtime mapping decisions. Evidently, these techniques consume more time, and cannot cope with dynamic application behavior, especially when multiple applications are run concurrently.

To adapt to dynamic application workloads, pure online optimization-based approaches, performing all processing at runtime, have also been investigated [4], [9]–[11]. In [4], an online reinforcement learning-based adaptive DVFS approach targeting frame-based applications is presented to improve energy efficiency. In [9], an online spatial mapping technique to map streaming applications onto a multicore system is discussed. Brião *et al.* [10] presented dynamic task allocation strategies based on bin-packing algorithms for soft real-time

applications. An online task allocator using the adaptive task allocation algorithm and clustering approach for minimizing the communication load is described in [11]. All of these approaches perform well for unknown applications to be executed at runtime, but lead to inefficient results as optimization decisions need to be taken quickly without offline analysis results [3].

Hybrid approaches using results of offline analysis in making online decisions have been widely proposed to improve energy efficiency/performance in homogeneous multicore platforms [8], [12]–[17]. Such approaches usually achieve better performance/energy savings compared to pure online optimizations as they take advantage of both offline and online computation. In [12], task mapping and DVFS under power constraints are discussed. Similarly, in [13], first thread-to-core mapping is obtained based on utilization, and then DVFS is applied depending upon the power budget. When considering the power-performance tradeoffs, recent research focus has shifted to heterogeneous architectures [3], [6], [14]–[17]. For multithreaded applications, most approaches tend to map an application completely onto one type of processing core(s) [14], [16], [17]. This simplifies the thread-to-core mapping problem, but cannot benefit from the power-performance tradeoffs offered by simultaneously mapping application threads onto multiple types of cores. Van Craeynest *et al.* [14] presented a performance impact estimation technique to predict which application-to-core mapping is likely to provide the best performance to map the application onto the most appropriate core type. In a similar direction, some proposals have used workload memory-intensity as an indicator to guide task mapping [38], [39]. A domain-specific hybrid task mapping is presented in [3], which relies heavily on offline DSE. However, approaches reported in [3] and [14] do not consider DVFS which can help to improve energy savings.

On the other hand, techniques proposed in [5], [6], and [15]–[17] use DVFS, but they have several shortcomings. For example, in [16], the design space is explored for a single application, which increases exponentially for concurrent execution of applications. Donyanavard *et al.* [17] considered applications with only one thread and thus use only one type of core for each application. The approach presented in [15] considers concurrent execution and mapping of application threads onto more than one type of cores. However, it requires extensive offline and/or online exploration for building regression models for performance and energy for all possible thread-to-core mappings and voltage–frequency settings, which is nonscalable. Moreover, online periodic adjustment of *V–f* setting is not explored, which is essential for adapting to workload variations and achieving better energy savings. This has been addressed in [5] and [6], however, they also require extensive offline characterization, and in particular, [5] requires application instrumentation to guide the runtime selection. Moreover, all these approaches do not perform adaptive mapping at application arrival/exit, and thus they are not efficient if a new/unknown application arrives/existing application finishes. The approach (AdaMD) presented in this article addresses the above limitations by removing dependency on the application-dependent offline results, and adapting to application arrival/completion times.

VII. Conclusion

The increasing demand for performance and energy efficiency has forced mobile systems to employ heterogeneous multiprocessor system-on-chips. These systems offer a diverse set of core and frequency configurations to runtime management systems for online tuning. This article has presented an adaptive thread-to-core mapping and DVFS technique, called AdaMD, for choosing a configuration for each performance-constrained application that minimizes energy consumption. By using runtime information while applications are executing and eliminating the need for application-dependent offline results, AdaMD is capable of managing even unknown applications efficiently. Proposed algorithm first selects a resource combination (number of cores and their type) that meets the application performance requirement using an accurate performance prediction model and resource enumerator/selector. It then monitors application performance, workload and its status (finished or newly arrived) for tuning voltage–frequency settings and adjusting thread-to-core mappings. Our experiments show an improvement of up to 28% in energy consumption compared to the most promising existing approaches. The proposed approach also outperforms previous approaches in meeting application performance constraints. Our future work includes validation with more number of cores and types having different ISA (e.g., CPU, GPU, etc.) to show the scalability and adaptability of the approach.

Acknowledgment

The experimental data used in this article can be found at https://doi.org/10.5258/SOTON/D1041.

References

[1] *Odroid-XU3*. Accessed: Oct. 1, 2018. [Online]. Available: https://www.hardkernel.com/shop/odroid-xu3

[2] *Mediatek X20*. Accessed: Oct. 1, 2018. [Online]. Available: http://www.96boards.org/product/mediatek-x20/

[3] W. Quan and A. D. Pimentel, "A hybrid task mapping algorithm for heterogeneous MPSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 1, p. 14, 2015.

[4] R. A. Shafik, A. K. Das, L. A. Maeda-Nunez, S. Yang, G. V. Merrett, and B. M. Al-Hashimi, "Learning transfer-based adaptive energy minimization in embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 6, pp. 877–890, Jun. 2016.

[5] U. Gupta, C. A. Patil, G. Bhat, P. Mishra, and U. Y. Ogras, "DyPO: Dynamic Pareto-optimal configuration selection for heterogeneous MPSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, p. 123, 2017.

[6] B. K. Reddy, A. K. Singh, D. Biswas, G. V. Merrett, and B. M. Al-Hashimi, "Inter-cluster thread-to-core mapping and DVFS on heterogeneous multi-cores," *IEEE Trans. Multi-Scale Comput. Syst.*, vol. 4, no. 3, pp. 369–382, Jul./Sep. 2018.

[7] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *Proc. ACM Design Autom. Conf.*, 2008, pp. 191–196.

[8] W. Quan and A. D. Pimentel, "A scenario-based run-time task mapping algorithm for MPSoCs," in *Proc. ACM Design Autom. Conf.*, 2013, p. 131.

[9] P. K. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. M. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multiprocessor system-on-chip (MPSoC)," in *Proc. ACM Design Autom. Test Europe*, 2008, pp. 212–217.

[10] E. W. Brião, D. Barcelos, and F. R. Wagner, "Dynamic task allocation strategies in MPSoC for soft real-time applications," in *Proc. ACM Design Autom. Test Europe*, 2008, pp. 1386–1389.

[11] J. Huang, A. Raabe, C. Buckl, and A. Knoll, "A workflow for run-time adaptive task allocation on heterogeneous MPSoCs," in *Proc. ACM Design Autom. Test Europe*, 2011, pp. 1119–1134.
[12] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: Adaptive DVFS and thread packing under power caps," in *Proc. IEEE/ACM Int. Symp. Microarchitect.*, 2011, pp. 175–185.
[13] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated power-performance optimization in manycores," in *Proc. IEEE Int. Conf. Parallel Architect. Compilation Techn.*, 2013, pp. 51–61.
[14] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," *ACM SIGARCH Comput. Architect. News*, vol. 40, no. 3, pp. 213–224, 2012.
[15] A. Aalsaud, R. A. Shafik, A. Rafiev, F. Xia, S. Yang, and A. Yakovlev, "Power-aware performance adaptation of concurrent applications in heterogeneous many-core systems," in *Proc. Int. Symp. Low Power Electron. Design*, 2016, pp. 368–373.
[16] E. D. Sozzo, G. C. Durelli, E. M. G. Trainiti, A. Miele, M. D. Santambrogio, and C. Bolchini, "Workload-aware power optimization strategy for asymmetric multiprocessors," in *Proc. IEEE Design Autom. Test Europe Conf. Exhibit.*, 2016, pp. 531–534.
[17] B. Donyanavard, T. Mück, S. Sarma, and N. D. Dutt, "SPARTA: Runtime task allocation for energy efficient heterogeneous many-cores," in *Proc. ACM Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, 2016, p. 27.
[18] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Jan. 2011.
[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," *ACM SIGARCH Comput. Architect. News*, vol. 23, no. 2, pp. 24–36, 1995.
[20] B. K. Reddy, G. V. Merrett, B. M. Al-Hashimi, and A. K. Singh, "Online concurrent workload classification for multi-core energy management," in *Proc. ACM Design Autom. Test Europe Conf. Exhibit.*, Mar. 2018, pp. 621–624.
[21] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Proc. IEEE Int. Conf. Compilers Architect. Synth. Embedded Syst.*, 2013, pp. 1–10.
[22] J. C. Saez, A. Fedorova, D. Koufaty, and M. Prieto, "Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems," *ACM Trans. Comput. Syst.*, vol. 30, no. 2, p. 6, 2012.
[23] J. L. Henning, "Spec CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Architect. News*, vol. 34, no. 4, pp. 1–17, 2006.
[24] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proc. USENIX Annu. Tech. Conf.*, 1996, p. 23.
[25] *Roy Longbottom's PC Benchmark Collection*. Accessed: Oct. 10, 2018. [Online]. Available: http://www.roylongbottom.org.uk
[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. IEEE Int. Workshop Workload Characterization*, 2001, pp. 3–14.
[27] F. Eibe, M. Hall, and I. Witten, "The WEKA workbench. Online appendix FO," in *Data Mining: Practical Machine Learning Tools and Techniques*. Cambridge, MA, USA: Morgan Kaufmann, 2016.
[28] J. H. Friedman, "Stochastic gradient boosting," *Comput. Stat. Data Anal.*, vol. 38, no. 4, pp. 367–378, 2002.
[29] G. Southern and J. Renau, "Analysis of parsec workload scalability," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2016, pp. 133–142.
[30] B. K. Reddy, M. J. Walker, D. Balsamo, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Empirical CPU power modelling and estimation in the Gem5 simulator," in *Proc. IEEE Int. Symp. Power Timing Model. Optim. Simulat.*, 2017, pp. 1–8.
[31] B. K. Reddy, E. W. Wächter, B. M. Al-Hashimi, and G. V. Merrett, "Workload-aware runtime energy management for HPC systems," in *Proc. Int. Conf. High Perform. Comput. Simulat.*, 2018, p. 8.
[32] Ü. Y. Ogras, R. Marculescu, D. Marculescu, and E.-G. Jung, "Design and management of voltage-frequency island partitioned networks-on-chip," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 3, pp. 330–341, Mar. 2009.
[33] G. Liu, J. Park, and D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *Proc. IEEE Int. Conf. Comput. Design*, 2013, pp. 54–61.
[34] S. Eranian, "Perfmon2: A flexible performance monitoring interface for Linux," in *Proc. Ottawa Linux Symp.*, 2006, pp. 269–288.
[35] K. Yu, D. Han, C. Youn, S. Hwang, and J. Lee, "Power-aware task scheduling for big.LITTLE mobile processor," in *Proc. IEEE Int. SoC Design Conf.*, 2013, pp. 208–212.
[36] A. K. Singh, C. Leech, B. K. Reddy, B. M. Al-Hashimi, and G. V. Merrett, "Learning-based run-time power and energy management of multi/many-core systems: Current and future trends," *J. Low Power Electron.*, vol. 13, no. 3, pp. 310–325, 2017.
[37] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett, and B. M. Al-Hashimi, "Energy-efficient run-time mapping and thread partitioning of concurrent OpenCL applications on CPU-GPU MPSoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, p. 147, 2017.
[38] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect.*, 2010, pp. 1–12.
[39] V. Petrucci, O. Loques, D. Mossé, R. Melhem, N. A. Gazala, and S. Gobriel, "Energy-efficient thread assignment optimization for heterogeneous multicore systems," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 1, p. 15, 2015.
[40] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous MPSoC platforms," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 692–707, Nov. 2010.
[41] D. Stamoulis and D. Marculescu, "Can we guarantee performance requirements under workload and process variations?" in *Proc. ACM Int. Symp. Low Power Electron. Design*, 2016, pp. 308–313.

Karunakar R. Basireddy (S'15) received the M.Tech. degree in microelectronics and VLSI from the Indian Institute of Technology, Hyderabad, India, in 2015. He is currently pursuing the Ph.D. degree in electronic and electrical engineering with the University of Southampton, Southampton, U.K.

His current research interests include design-time and run-time optimization of performance and energy in multicore heterogeneous systems.

Amit Kumar Singh (M'09) received the B.Tech. degree in electronics engineering from the Indian Institute of Technology (Indian School of Mines) Dhanbad, Dhanbad, India, in 2006, and the Ph.D. degree from the School of Computer Engineering, Nanyang Technological University, Singapore, in 2013.

He is a Lecturer with the University of Essex, Colchester, U.K. He was with HCL Technologies, Noida, India, for a year and half until 2008. He has a Post-Doctoral Research experience for over five years at several reputed universities. He has published over 70 papers in reputed journals/conferences. His current research interests include system level design-time and run-time optimization of 2-D/3-D multicore systems for performance, energy, temperature, reliability, and security.

Dr. Singh was a recipient of several best paper awards.

Bashir M. Al-Hashimi (M'99–SM'01–F'09) is an ARM Professor of computer engineering, the Dean of the Faculty of Physical Sciences and Engineering, and the Co-Director of the ARM-ECS Research Centre, University of Southampton, Southampton, U.K. He has published over 380 technical papers. He has authored or coauthored 5 books and has graduated 35 Ph.D. students. His current research interests include methods, algorithms, and design automation tools for low-power design and test of embedded computing systems.

Geoff V. Merrett (GSM'06–M'09) received the B.Eng. and Ph.D. degrees from the University of Southampton, Southampton, U.K., in 2004 and 2009, respectively.

He is an Associate Professor with the School of Electronics and Computer Science, University of Southampton and the Head of Its Centre for IoT and pervasive systems. His current research interests include energy management of mobile and embedded systems, and has published over 175 articles in journals/conferences in the above areas.