AdaMD: Adaptive Mapping and DVFS for Energy-efficient Heterogeneous Multi-cores

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 multi-core 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 multi-threaded 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—Heterogeneous multi-cores, Multi-threaded applications, Run-time management, Energy savings.

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 multi-threaded application, and setting dynamic voltage and frequency (DVFS) operating points.

B. K. Reddy, B. M. Al-Hashimi, and G. V. Merrett are with the School of Electronics and Computer Science, University of Southampton, United Kingdom (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, United Kingdom (email: a.k.singh@essex.ac.uk).

Manuscript received Jan xx, 2019; revised xxx xx, 2019.

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 multi-cores, 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 and four LITTLE cores that can operate at 13 and 19 different frequencies, respectively) is 4080 ((4×13×4×19) + (4×13) + (4×19)). 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 paper presents AdaMD, an adaptive mapping approach coupled with DVFS for performance-constrained multi-threaded applications, executing on heterogeneous multi-cores. AdaMD selects an resource combination (number of cores and their type) that meets the application’s performance requirement while minimising energy consump-
tion. 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 paper are:

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 multi-threaded applications from PARSEC [18] and SPLASH [19] benchmarks.

The remainder of this article is organised as follows. Section II presents a motivational example for our work, while section III presents the problem definition for this work. 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 the paper.

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 email 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: i) any application(s) may finish executing, ii) an application(s) may experience performance degradation due to contention for shared resources, and iii) a new application(s) may arrive into the system. In the first case, if application 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 ii), 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, subsequent 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/limit 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 iii), 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 re-adjust 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 i, ii and iii) for adaptation and moreover, they also depend on extensive offline characterization and/or instrumentation of the chosen applications. As experimentally demonstrated in Section V, adaptation at application arrival and completion, and workload/performance variations would lead to better utilisation 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 multi-core 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:

- An existing application finishes or a new application arrives into the system
- The performance constraints of any running applications are violated
- 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 Algorithm 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 utilised 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.

#### A. Online Identification of Mapping

Proposed AdaMD approach first identifies thread-to-core mapping that minimises 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 work 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 Utilisation, 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 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 `__syscall ROI_begin` (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 inter-process 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 multi-core 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 non-scalable 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.

**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 \( \eta \) as

\[
\eta = \frac{IPC_{\text{CoreType1}}}{IPC_{\text{CoreType2}}} \tag{1}
\]

where, \( IPC_{\text{CoreType1}} \) and \( IPC_{\text{CoreType2}} \) 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 \( IPC_{\text{CoreType2}} \), by running the application on one core type and collecting the runtime parameters, and measuring its performance \( IPC_{\text{CoreType1}} \) (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 analysing the correlation between various metrics and speedup. Modern processors support monitoring of various architectural events which can be used for analysing 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], [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 characterisation data for a diverse set of applications. As part of this, we have created a diverse set of workloads, containing single and multi-threaded 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-

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>PARAMETERS USED IN THE PROPOSED APPROACH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Active Cores</td>
<td>Frequency of the Cores</td>
</tr>
<tr>
<td>L1 I-Cache Misses</td>
<td>L1 D-Cache Misses</td>
</tr>
<tr>
<td>L2 Cache Misses</td>
<td>Instructions Retired</td>
</tr>
<tr>
<td>Branch Misses</td>
<td>CPU Cycles</td>
</tr>
<tr>
<td>Per Core CPU Utilisation</td>
<td>Memory Reads Per Instruction</td>
</tr>
</tbody>
</table>

Algorithm 1 AdaMD Mapping and Adaptation

**Input:** Applications and performance constraints \( (\text{Apps}) \)

**Output:** \( V\text{Apps}, \text{mappings} \) 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 Yi ∈ Apps do
8:     if (unmapped) then
9:       Allocate a free core ‘l’ to ‘i’ and execute;
10:      Measure \( \text{MRP}_I \) 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:      \( \eta = \text{speedup\_estimate(pmcs, j);} \)
17:      Compute possible resource combinations and resource combination with minimum 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 \( \eta \) (list);
23:   while (freecores>0) do
24:     Increase the resources of app \( i \in \text{list} \) 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 \( \eta \);
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
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 multi-threaded 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 modelling. 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 $X = \{x_1, x_2, ..., x_n\}$. Given a training sample $\{y_i, X_i\}_N$ 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.

$$\hat{f}(X) = \arg\min_{f(X)} E(y, X, \psi(y, f(X)))$$

(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 shown below:

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

(3)

Here, the base learner functions $h(X; \beta)$ are simple functions of $X$ with parameters $\beta = \{\beta_1, \beta_2, ..., \beta_M\}$ and $\{\alpha_k\}_0^M$ are expansion coefficients. Owing to simplicity, decision stump (one-level decision tree) is used as a base learner in our work. 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.

3) **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 $\text{PerfApp}_i$ is the performance constraint for an application $\text{App}_i$, then the performance meeting thread-to-core mappings ($T_{\text{map}_i}$) can be defined as follows:

$$T_{\text{map}_i} = \{r \in R | \text{perf}(r) \leq \text{PerfApp}_i\}$$

(4)

Here, $\text{perf}(r)$ 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, $\text{perf}(r)$ is computed as:

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

(5)

where, $\eta = \text{IPC}_b/\text{IPC}_l$, performance on the big and LITTLE core is denoted by $\text{IPC}_b$ and $\text{IPC}_l$, respectively. Furthermore, $n_b \in N_b$, $n_l \in N_l$ and $r = n_l \cup n_b$, $\text{IPC}_o$ 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 inter-cluster communication overhead when the threads of an application run on both the big and LITTLE clusters. As shown in Equation 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).

4) **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 $T_{\text{map}_i} = \{3L, 4L, 1L + 1B, ...\}$, 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).

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

(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 core 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 $\text{PerfApp}_i$ is the performance constraint for an application $\text{App}_i$, then the performance meeting thread-to-core mappings ($T_{\text{map}_i}$) can be defined as follows:

$$T_{\text{map}_i} = \{r \in R | \text{perf}(r) \leq \text{PerfApp}_i\}$$

(4)

Here, $\text{perf}(r)$ 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, $\text{perf}(r)$ is computed as:

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

(5)

where, $\eta = \text{IPC}_b/\text{IPC}_l$, performance on the big and LITTLE core is denoted by $\text{IPC}_b$ and $\text{IPC}_l$, respectively. Furthermore, $n_b \in N_b$, $n_l \in N_l$ and $r = n_l \cup n_b$, $\text{IPC}_o$ 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 inter-cluster communication overhead when the threads of an application run on both the big and LITTLE clusters. As shown in Equation 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).

4) **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 $T_{\text{map}_i} = \{3L, 4L, 1L + 1B, ...\}$, 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).

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

(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
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 $\eta$ can be used 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 ($C_r^b$) is higher/close to the speedup. As big core can execute $\eta$ times faster than LITTLE core, above resource selection strategy leads to balanced workload sharing between big and LITTLE cores by executing $\eta$ times more threads on big than LITTLE. This would lead to efficient utilisation 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 $\eta$ 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 Eq. 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 Section 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 utilisation, 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 $t_h$ (Eq. 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 (line 30-34, Algorithm 1). An application performance is measured by collecting PMCs 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.
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 ($V[f[i]]$) for next epoch

3: pmcs = get_pmcs_data(i);
4: compute actual MRPI ($MRPI_a$) = $\frac{instructions\ retired}{active\ CP\ cycles}$
5: compute actual utilisation ($util_a$) = $\frac{active\ CP\ cycles}{Total\ CPU\ cycles}$
6: $MRPI_p = predict\_mrpi(mrpi_p, mrpi_u, e_m)$;
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: $V_f[i] = bin\_classify(util_p, mrpi_p)$;
11: if ($V_f[i] < f_{req}[i]$) then
12: $V_f[i] = f_{req}[i]$;
13: cpufreq_set_frequency(i, $V_f[i]$);
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 ($mrpi_u$) and utilisation ($util_a$);
19: Repeat steps 6 to 13.
20: end for

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 Eq. 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 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 utilisation 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, utilisation and application performance (Execution time) are collected at different DVFS settings available on the chosen platform. The collected utilisation and MRPI for various applications are then grouped into utilisation 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 utilisation 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 utilisation ($e_u$) is used as feedback to improve the workload prediction accuracy (lines 7 & 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], [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

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.

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 performance monitoring unit (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.
Applications: To evaluate AdaMD, applications – Blacksc-holes (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 behav-ior. 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 utilisation. 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.

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

1) HMP+ x [35]: The state-of-the-art solution for big.LITTLE multi-processing, the Heterogeneous Multi-Pro-cessing (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 set-tings 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 work-load classification-based DVFS periodically to minimize the power consumption.

B. Evaluation of Performance Predictor

The performance prediction model estimates the perfor-mance of the big core given the performance of a LITTLE core (P_{lb}) and vice versa (P_{bl}). The number of base learners (decision stumps) M in Eq. 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_{lb} and P_{bl} are given in Figures 4a and 4b 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 Figures 4a and 4b, 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. There-fore, to balance this, we built additive regression models for different numbers of decision stumps. It can be seen from Fig. 4a and 4b that the improvement in model accuracy is negligible after 900 and 1100 decision stumps for P_{lb} and P_{bl} respectively. Therefore, we have chosen these numbers for our models P_{lb} (mean absolute percentage error (MAPE) = 1.57%; maximum error (ME) = 8.1%) and P_{bl} (MAPE = 3.45%; ME = 8.5%). The maximum error of P_{lb} and P_{bl} 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_{bl} 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
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.

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).
Due 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 multi-core system is discussed. Brião et al. [10] present 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 multi-core 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 multi-threaded 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], [14] do not consider DVFS which can help to improve energy savings.

On the other hand, techniques proposed in [5], [6], [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] consider 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 non-scalable. 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], [6], however, they also require extensive offline characterisation, 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 paper addresses the above limitations by removing dependency on the application-dependent offline results, and adapting to application arrival/completion times.

VII. CONCLUSIONS

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 paper has presented an adaptive thread-to-core mapping and DVFS technique, called AdaMD, for choosing a configuration for each performance-constrained application that minimises 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.

ACKNOWLEDGEMENT

This work was supported in parts by the EPSRC Grant EP/L000563/1 and the PRiME Programme Grant EP/K034448/1 (www.prime-project.org). Experimental data used in this paper can be found at https://doi.org/10.5258/SOTON/D1041.

REFERENCES


