Scalable Optimal Greedy Scheduler for Asymmetric Multi-/Many-Core Processors

Vanchinathan Venkataramani[0000−0002−0259−6456], Anuj Pathania[0000−0002−5813−7021], and Tulika Mitra[0000−0003−4136−4188]

National University of Singapore, Singapore
{vvanchi,pathania,tulika}@comp.nus.edu.sg

Abstract. Ubiquitous asymmetric multi-core processors such as *ARM big.LITTLE* combine together cores with different power-performance characteristics on a single chip. Upcoming asymmetric many-core processors are expected to combine hundreds of cores belonging to different types. However, the accompanying task-to-core mapping schedules are the key to achieving the full potential of such processors. Run-time scheduling on asymmetric processors is a much harder problem to solve optimally than scheduling on symmetric processors with equivalent cores. We present the first-ever greedy scheduler to be proven theoretically optimal (under certain constraints) for asymmetric processors. The proposed scheduler, called *A-Greedy*, improves throughput by 26% and reduces average response time by up to 45% when compared to the default *Linux* scheduler on *ARM big.LITTLE* asymmetric multi-core.

Keywords: Multi-/Many-cores · Asymmetric Processors · Scheduling · Optimal Greedy · *ARM big.LITTLE*.

1 Introduction

Upcoming asymmetric many-core processors are expected to house together hundreds of heterogeneous cores grouped into homogeneous tiles connected using an interconnect on a single chip as shown in Figure 1 [10]. They are an evolution of current asymmetric multi-core processors such as *ARM big.LITTLE* that are now commonplace. Asymmetric many-cores are designed to execute tens of multi-threaded applications on its hundreds of cores using one-thread-per-core model in order to reduce context switching overheads [12]. An Operating System (OS) sub-routine called scheduler determines the allocation of the cores among the applications. The discrete scheduling problem of allocating multiple cores to multiple applications optimally on asymmetric processors is in general NP-Hard and is commonly solved using heuristics [21].

Authors in [9] showed that the scheduling problem can be solved optimally using dynamic programming in polynomial time for symmetric processors containing cores with equivalent performance although with high scheduling overheads. Presence of a large number of cores in many-cores puts greater emphasis

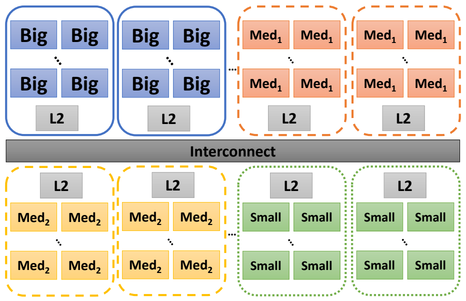

Fig. 1: An abstract block diagram of an asymmetric many-core processor containing clusters of cores with different power-performance characteristics.

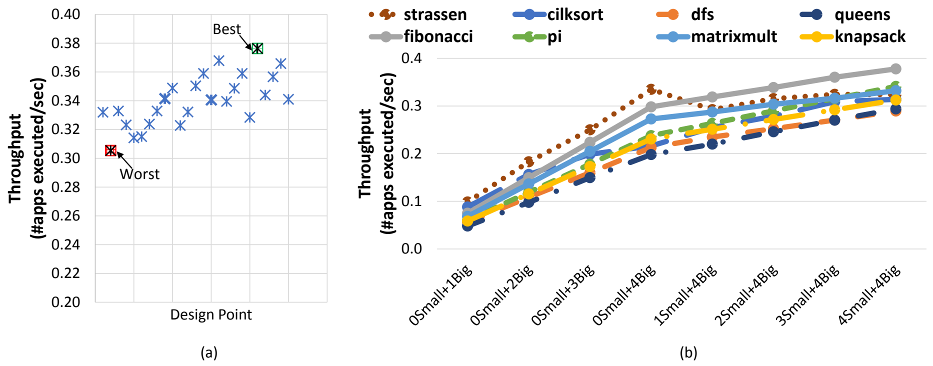

Fig. 2: (a) Throughput with two applications (*MatrixMult* and *Pi*) when executed with different asymmetric core allocations on *Kirin 960* (b) Throughput of different applications when executed in isolation with different asymmetric core allocations on *Kirin 960*.

on scheduler scalability than multi-cores especially for use at run-time [17],[23]. Authors of [18] showed that the scheduling problem for symmetric processors can also be solved optimally using a greedy scheduler for applications with concave speedups but with several magnitudes lower scheduling overheads. Thus, the greedy scheduler can be used for run-time scheduling on symmetric many-cores.

The problem of scheduling on asymmetric processors is, however, more difficult to solve optimally than equivalent scheduling on symmetric processors as the scheduler needs to determine not just the number of cores allocated to each application but also the type of each of those allocated cores [8]. Thus, the involved design space that needs to be explored is further enlarged. For example, consider the case wherein we need to schedule two applications on *Huawei Kirin 960 octa-core ARM big.LITTLE* asymmetric multi-core with a cluster of four *Big* cores and a cluster of four *Small* cores. There will be twenty five design points on *Kirin 960* (0 to 4 cores in each cluster allocated to one application and remaining cores allocated to the other application). While on an equivalent octa-core symmetric multi-core, we will have only nine design points (0 to 8 cores for one application and remaining to the other application).

Motivational Example: A scheduler's efficacy can be measured using the throughput it can sustain on its underlying processor. **Throughput** is defined as the number of applications finishing per unit time. Figure 2(a) shows the throughput for all possible core allocations between the two applications (*MatrixMult* and *Pi*) on *Kirin 960*. Results show that the throughput in the best-case is 1.23x higher than the worst-case, necessitating a sophisticated scheduler.

The performance improvement with increasing core allocation for a multi-threaded application is dependent on the parallel portion of the application as governed by the Amdahl's Law [1]. Parallelization may incur overheads arising from coherence, inter-thread communication, etc. in processors. Thus, every subsequent core allocation brings in lower or equal improvement than the antecedent allocation resulting in multi-threaded applications to exhibit concave throughput increase. Fortunately, we observe that several of the multi-threaded applications that exhibit concave throughput behavior on symmetric processors also exhibit near-concave throughput behavior on asymmetric processors. Figure 2(b) shows the concavity in throughput increase for different applications when executed with different asymmetric allocations on *Kirin 960*. This concavity observation can be utilized to design an optimal greedy scheduler for asymmetric processors. In practice, the proposed scheduler can also accommodate minor non-concavity using concave approximations [22] without deviating significantly from the optimal. Though uncommon, applications that do not exhibit concave speedup exist [20] and this work is not applicable to them.

Our Novel Contributions: We make the following novel contributions within the scope of this work.

- We are the first to present a greedy scheduler that can optimally schedule multi-threaded applications with concave throughput increase on asymmetric processors. We present the proofs to theoretically support our claim.
- We implement our proposed greedy scheduler on a commercial-grade *Kirin 960* asymmetric multi-core and show its performance to be near-equivalent to an optimal ILP-based scheduler. The performance is also shown to be higher than the default *Linux* scheduler and state-of-the-art heuristic scheduler [13].

2 A-GREEDY: Greedy Scheduler for Asymmetric Multi-/Many-Cores

We describe *A-Greedy*, a greedy scheduler that exploits concave throughput behavior in multi-threaded applications to make efficient use of the underlying asymmetric processor. For brevity, we assume that the asymmetric processor contains cores of only two types - *Big* and *Small*. However, *A-Greedy* can be generalized to asymmetric processors with any number of core types.

2.1 Model

Let B and S denote the total number of cores of type *Big* and *Small*, respectively in the processor. Let T denote the total number of independent applications executing on the processor indexed using i. We assume that all applications derive higher performance on n *Big* cores than n *Small* cores, where $n \geq 1$. Furthermore, we also assume that the performance of an individual application i remains unaffected by other applications executing in parallel. Note that this assumption is needed only for designing a theoretically optimal algorithm. Our experimental evaluation includes interferences among the applications.

Let B_i and S_i be the number of cores of type *Big* and *Small* respectively that are allocated to application i. Let P_{B_i,S_i} represent the throughput (performance) of application i when simultaneously executed on B_i *Big* and S_i *Small* cores. Let THD_i denote the maximum multi-threading factor of application i, which determines the maximum number of cores, i.e., $B_i + S_i$ that can be assigned to application i. Let $P_{B_i,S_i} = 0$, if $B_i + S_i > THD_i$.

The problem of throughput maximization on asymmetric processors can then be described as:

$$\text{Maximize:} \sum_{i=1}^{T} P_{B_i,S_i}$$

subject to the following constraints:

$$\sum_{i=1}^{T} B_i \leq B \ , \sum_{i=1}^{T} S_i \leq S \text{ and } \forall i \in \{1, 2, \ldots, T\}(B_i + S_i) \leq THD_i$$

We use profiling to obtain P_{B_i,S_i} for all possible core allocations. We observe that the applications exhibit concave throughput behavior, i.e., the increase in throughput when a core is allocated is less than or equal to the increase in throughput experienced in the previous core allocation when adding cores belonging to a particular type. The concavity observed on asymmetric processors as shown in Figure 2 (b) needs to be mathematically represented. Concavity due to the allocations of additional n *Big* (or *Small*) cores while starting with having different number of *Big* (or *Small*) cores but the same number of *Small* (or *Big*) cores, i.e., $B_i \to B_i + n$ versus $B'_i \to B'_i + n$ (or $S_i \to S_i + n$ versus $S'_i \to S'_i + n$) can be described using the equations below.

$$\forall n \geq 0,\ P_{B'_i+n,S_i} - P_{B'_i,S_i} \geq P_{B_i+n,S_i} - P_{B_i,S_i} \text{ if } B'_i \leq B_i \tag{1}$$

$$\forall n \geq 0,\ P_{B_i,S'_i+n} - P_{B_i,S'_i} \geq P_{B_i,S_i+n} - P_{B_i,S_i} \text{ if } S'_i \leq S_i \tag{2}$$

We also assume that the allocation of n *Big* (or *Small*) cores while having the same number of *Big* (or *Small*) cores is independent of the number of *Small* (or *Big*) cores allocated already.

$$\forall n \geq 0,\ P_{B_i+n,S'_i} - P_{B_i,S'_i} = P_{B_i+n,S_i} - P_{B_i,S_i} \ \ \forall S'_i, S_i \in [0, S] \tag{3}$$

$$\forall n \geq 0,\ P_{B'_i,S_i+n} - P_{B'_i,S_i} = P_{B_i,S_i+n} - P_{B_i,S_i}, \ \ \forall B'_i, B_i \in [0, B] \tag{4}$$

2.2 Algorithm

The greedy algorithm proposed in *A-Greedy* scheduler consists of the following steps performed sequentially at every invocation.

1. Start with an empty allocation for all applications, i.e., $\forall i \in \{1, 2, \ldots, T\}$, $B_i = 0$ and $S_i = 0$.
2. Sort all applications $i \in \{1, 2, \ldots, T\}$ that has been allocated with less than THD_i cores in descending order of highest possible throughput gain by allocating one more free core of any type and add them to a queue.
3. Allocate a core of that type to application i in front of the queue from which it derives the maximum throughput.
4. Update the throughput value and reposition the entry associated with application i in the queue according to its revised throughput only if $(B_i + S_i) < THD_i$. As the queue is sorted, we can use binary search to perform fast insertions. Otherwise, pop the element from the queue.
5. Repeat Steps 3 and 4 if there are applications left in the queue and if any core is left unallocated.
6. Execute the applications with the greedy allocation.

The above greedy algorithm is straightforward to implement. Nevertheless, the algorithm results in a schedule that provides optimal allocations that maximizes total throughput on asymmetric many-cores under the throughput concavity assumption. We provide the proof for its theoretical optimality next.

2.3 Optimality Proof

Theorem 1. *The A-Greedy scheduler optimally maximizes the aggregate throughput on an asymmetric processor.*

Proof. We prove this theorem using the induction method. There are two possible scenarios that need to be inductively proven sub-optimal – unidirectional moving of cores between two applications and swapping of heterogeneous cores between two applications. Let $[\{B_1, S_1\}$, $\{B_2, S_2\}$, ..., $\{B_T, S_T\}]$ be the greedy core allocation chosen by our proposed *A-Greedy* scheduler.

Moving Cores Unidirectionally Between Applications:

Base Case: We do not start with moving one *Big* (or *Small*) core between applications as it reduces to the allocation problem in symmetric processors. Authors in [18] have proven this move to be sub-optimal. Thus, we only show detailed proof for the case where one *Big* core and one *Small* core is removed from Application x and given to Application y. Without loss of generality, the proof below will also hold for other possible combinations if (i) one *Big* and one *Small* core is removed from Application x and given one each to Application y and Application z, respectively, (ii) one *Big* and one *Small* core is removed respectively from Application x and y, and then given to Application z, (iii) one *Big* and one *Small* core is removed respectively from Application v and x, and then given to Application y and z. Due to the space constraint all the cases cannot be shown here in detail.

Let us begin with the assumption that $[\{B_1, S_1\}, ..., \{B_x - 1, S_x - 1\}, ..., \{B_y + 1, S_y + 1\}, ..., \{B_T, S_T\}]$ is instead the optimal allocation where one *Big* core and one *Small* core is removed from Application x and given to Application y. For the assumed optimal allocation to be better than the greedy core allocation, gain in throughput of Application y with allocation of more cores must outweigh the loss in throughput of Application x due to the removal of those cores and thereby the following inequality must hold.

$$P_{B_x-1,S_x-1} + P_{B_y+1,S_y+1} > P_{B_x,S_x} + P_{B_y,S_y} \tag{5}$$

As *A-Greedy* chose to allocate the additional *Big* core to Application x instead of Application y, then the following inequality must be true.

$$P_{B_x,S_x} - P_{B_x-1,S_x} \geq P_{B_y+1,S_y} - P_{B_y,S_y} \tag{6}$$

As *A-Greedy* chose to allocate the additional *Small* core to Application x instead of Application y then the following inequality must be true.

$$P_{B_x,S_x} - P_{B_x,S_x-1} \geq P_{B_y,S_y+1} - P_{B_y,S_y} \tag{7}$$

Using Equation (4), we know that the following equations are true.

$$P_{B_x-1,S_x} - P_{B_x-1,S_x-1} = P_{B_x,S_x} - P_{B_x,S_x-1} \tag{8}$$

$$P_{B_y,S_y+1} - P_{B_y,S_y} = P_{B_y+1,S_y+1} - P_{B_y+1,S_y} \tag{9}$$

Adding Equations (6) and (9) we get:

$$\begin{aligned} P_{B_x,S_x} - P_{B_x-1,S_x} + P_{B_y,S_y+1} - P_{B_y,S_y} \geq P_{B_y+1,S_y} - P_{B_y,S_y} \\ + P_{B_y+1,S_y+1} - P_{B_y+1,S_y} \end{aligned} \tag{10}$$

Using the inequality in Equation (7) in Equation (10) we get:

$$P_{B_x,S_x} - P_{B_x-1,S_x} + P_{B_x,S_x} - P_{B_x,S_x-1} \geq P_{B_y+1,S_y+1} - P_{B_y,S_y} \tag{11}$$

Using Equation (8) in Equation (11) we get:

$$P_{B_x,S_x} - P_{B_x-1,S_x} + P_{B_x-1,S_x} - P_{B_x-1,S_x-1} \geq P_{B_y+1,S_y+1} - P_{B_y,S_y} \tag{12}$$

By solving and rearranging Equation (12) we get:

$$P_{B_x,S_x} + P_{B_y,S_y} \geq P_{B_x-1,S_x-1} + P_{B_y+1,S_y+1} \tag{13}$$

Equation (13) is in contradiction to Equation (5). Hence, we prove that the allocation chosen by the greedy scheduler is optimal.

Inductive Assumption: We assume that the greedy allocation is optimal and of higher or equal performance than any allocation where n cores are removed from Application x and distributed among all the other tasks in any combination. Mathematically, we assume that the following inequality is true.

$$P_{B_x,S_x} - P_{B_x-n,S_x-n} \geq P_{B_1+\alpha_1,S_1+\beta_1} - P_{B_1,S_1} + .. + P_{B_T+\alpha_T,S_T+\beta_T} - P_{B_T,S_T} \tag{14}$$

where $\alpha_1 + ... + \alpha_T = \beta_1 + ... + \beta_T = n$

Inductive Step: Now we assume optimal allocation, which is different from the greedy allocation. In this optimal allocation, n cores are removed from Application x and distributed among all the other applications in the same combination as in the inductive assumption in Equation (14). In addition, $n + 1^{th}$ *Big* and *Small* cores are removed from Application x and are given to Application y without loss of generality. For assumed optimal allocation to be better than greedy allocation, the following inequality must hold.

$$\begin{aligned} P_{B_1+\alpha_1,S_1+\beta_1} - P_{B_1,S_1} + ... + P_{B_y+\alpha_y+1,S_y+\beta_y+1} - P_{B_y,S_y} + ...+ \\ P_{B_T+\alpha_T,S_T+\beta_T} - P_{B_T,S_T} > P_{B_x,S_x} - P_{B_x-n-1,S_x-n-1} \end{aligned} \tag{15}$$

As *A-Greedy* allocated a *Big* and *Small* core to Application x with B_{x-n-1} *Big* and S_{x-n-1} *Small* cores already allocated to it instead of Application y, by the greedy design of *A-Greedy*, the following inequality must be true.

$$P_{B_x-n,S_x-n} - P_{B_x-n-1,S_x-n-1} \geq P_{B_y+\alpha_y+1,S_y+\beta_y+1} - P_{B_y+\alpha_y,S_y+\beta_y} \tag{16}$$

By adding Equation (16) with Equation (14) we get:

$$\begin{aligned} P_{B_x,S_x} - P_{B_x-n-1,S_x-n-1} \geq P_{B_1+\alpha_1,S_1+\beta_1} - P_{B_1,S_1} \\ +... + P_{B_y+\alpha_y+1,S_y+\beta_y+1} - P_{B_y,S_y} + ... + P_{B_T+\alpha_T,S_T+\beta_T} - P_{B_T,S_T} \end{aligned} \tag{17}$$

Equation (17) is in contradiction to Equation (15) proving that the greedy allocation under *A-Greedy* is optimal instead of the assumed optimal allocation in the inductive step. Thus, unidirectional movement of cores between applications is a sub-optimal scenario.

Swapping Asymmetric Cores Between Applications:

Base Case: As in the previous scenario, we only show the detailed proof for the case where one *Small* core is removed from Application x and given to Application y and one *Big* core is removed from Application y and given to Application x. Without loss of generality, the proof below will also hold for other possible combinations. Let us begin with the assumption that $[\{B_1, S_1\}, ..., \{B_x+1, S_x-1\}, ..., \{B_y-1, S_y+1\}, ..., \{B_T, S_T\}]$ is instead the optimal allocation. For this to be true, throughput in this assumed optimal allocation has to be better than the greedy core allocation as stated in the following inequality.

$$P_{B_x+1,S_x-1} + P_{B_y-1,S_y+1} > P_{B_x,S_x} + P_{B_y,S_y} \tag{18}$$

Since *A-Greedy* adds one core at a time, let us start from an intermediate allocation $[\{B_1, S_1\}, ..., \{B_x, S_x-1\}, \{B_y-1, S_y\}, ..., \{B_T, S_T\}]$ which is the predecessor allocation to both greedy and optimal allocation. In the greedy algorithm, one *Big* core is added to Application y followed by one *Small* core to Application x from the intermediate allocation. For the assumed optimal allocation, one *Big* core is added to Application x and one *Small* core is added to Application y from the same intermediate allocation. Since *A-Greedy* chose to allocate the additional *Big* core to Application y instead of Application x, followed by additional *Small* core to Application x instead of Application y, the following inequalities must hold.

$$P_{B_y,S_y} - P_{B_y-1,S_y} \geq P_{B_x+1,S_x-1} - P_{B_x,S_x-1} \tag{19}$$

$$P_{B_x,S_x} - P_{B_x,S_x-1} \geq P_{B_y,S_y+1} - P_{B_y,S_y} \tag{20}$$

Using Equation (4), we know that the following equation is true.

$$P_{B_y,S_y+1} - P_{B_y,S_y} = P_{B_y-1,S_y+1} - P_{B_y-1,S_y} \tag{21}$$

Replacing R.H.S in Equation (20) using Equation (21) and adding Equations (19) and (20) we get:

$$\begin{aligned} P_{B_y,S_y} - P_{B_y-1,S_y} + P_{B_x,S_x} - P_{B_x,S_x-1} \geq \\ P_{B_x+1,S_x-1} - P_{B_x,S_x-1} + P_{B_y-1,S_y+1} - P_{B_y-1,S_y} \end{aligned} \tag{22}$$

By rearranging Equation (22) we get:

$$P_{B_x,S_x} + P_{B_y,S_y} \geq P_{B_x+1,S_x-1} + P_{B_y-1,S_y+1} \tag{23}$$

Equation (23) is in contradiction to Equation (18). Hence, we prove that the allocation chosen by the greedy scheduler is optimal.

Inductive Assumption: We assume that the greedy allocation is optimal with equal or higher performance than any allocation where n *Small* cores from Application x is swapped with n *Big* cores from other applications in any combination. Mathematically, we assume that the following inequality is true.

$$P_{B_x,S_x} - P_{B_x+n,S_x-n} \geq P_{B_1-\alpha_1,S_1+\beta_1} - P_{B_1,S_1} + ... + P_{B_T-\alpha_T,S_T+\beta_T} - P_{B_T,S_T} \tag{24}$$

where $\alpha_1 + ... + \alpha_T = \beta_1 + ... + \beta_T = n$

Inductive Step: Now we assume that the optimal allocation is different from the greedy allocation. In this assumed optimal allocation, n *Small* cores from Application x are swapped with n *Big* cores from other applications in the same combination as in the inductive assumption (Equation (24)). In addition, $n+1^{th}$ *Small* core from Application x is swapped with *Big* core in Application y without loss of generality. For the assumed optimal allocation to be better than the greedy allocation, the following inequality must hold.

$$\begin{aligned} P_{B_1-\alpha_1,S_1+\beta_1} - P_{B_1,S_1} + ... + P_{B_y-\alpha_y-1,S_y+\beta_y+1} - P_{B_y,S_y} + ...+ \\ P_{B_T-\alpha_T,S_T+\beta_T} - P_{B_T,S_T} > P_{B_x,S_x} - P_{B_x+n+1,S_x-(n+1)} \end{aligned} \tag{25}$$

As *A-Greedy* allocated a *Big* core to Application y, followed by a *Small* core to Application x, by greedy design following inequality must be true.

$$P_{B_x+n,S_x-n} - P_{B_x+n+1,S_x-(n+1)} \geq P_{B_y-\alpha_y-1,S_y+\beta_y+1} - P_{B_y-\alpha_y,S_y+\beta_y} \tag{26}$$

By adding Equation (26) with Equation (24) we get:

$$\begin{aligned} P_{B_x,S_x} - P_{B_x+n+1,S_x-(n+1)} \geq P_{B_1-\alpha_1,S_1+\beta_1} - P_{B_1,S_1} + ...+ \\ P_{B_y-\alpha_y-1,S_y+\beta_y+1} - P_{B_y,S_y} + ... + P_{B_T-\alpha_T,S_T+\beta_T} - P_{B_T,S_T} \end{aligned} \tag{27}$$

Equation (27) is in contradiction to Equation (25) proving greedy allocation under *A-Greedy* is optimal instead of the assumed optimal allocation in the inductive step. Thus, swapping of asymmetric cores between applications is a sub-optimal scenario. *A-Greedy* is therefore proven to be optimal using induction.

2.4 Complexity

The *A-Greedy* scheduler requires sorting of the tasks according to their throughput, which introduces a worst-case overhead of $O(T \lg T)$. It also requires repositioning of the applications in the queue using binary search after every core allocation introducing a worst-case overhead of $O((B+S)\lg T)$. Therefore, the total worst-case computational-overhead of our *A-Greedy* scheduler is $O(\max\{B+S,T\}\lg T)$. The need for the sorted applications queue data structure introduces a worst-case space-overhead of $O(T)$ for *A-Greedy*.

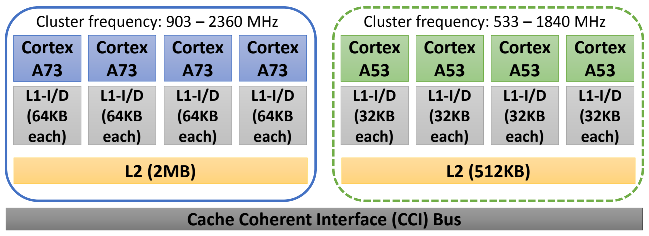

Fig. 3: *Huawei Kirin 960 octa-core ARM big.LITTLE* platform block diagram

3 Experimental Evaluations

In this section, we show the efficacy of our proposed *A-Greedy* scheduler on a commercial state-of-the-art asymmetric multi-core.

3.1 Benchmarks and Multi-threading model

In this evaluation, we utilize eight multi-threaded benchmarks: *CilkSort, DFS, Fibonacci, Knapsack, MatrixMult, Pi, Queens, Strassen*, from *LACE* benchmark suite [5]. These applications are representative of a real-world asymmetric processor workload and are implemented using a work-stealing framework in which slave threads grab work from a centralized queue managed by a master thread. We suitably modify these applications to make them *malleable* [6], i.e., it is possible for the scheduler to change the number of cores allocated to them during their execution. Malleability is shown to enable efficient utilization of the underlying processing resources [6]. Though multi-threaded benchmark suites like *Parsec* [2] are also compatible with our proposed algorithm, it is not possible for us to change the number of cores allocated to them at run-time making them non-malleable. Hence, we do not utilize them in this work.

3.2 Evaluation Setup

We use state-of-the-art *HiKey 960* embedded platform with *Huawei Kirin 960 octa-core ARM big.LITTLE* asymmetric multi-core processor (Figure 3) as a proof-of-concept testbed for our evaluation on existing silicon. Figure 3 also summarizes the system specifications of this architecture. There are two clusters: a low-performance cluster consisting of four *ARM Cortex-A53* (*Small*) cores and a high-performance cluster containing four *ARM Cortex-A73* (*Big*) cores. The clusters are connected by a cache-coherent bus interface. We ideally want to run the applications at the maximum system frequency on all eight cores as the primary objective in this work is to improve the performance. However, the system gets automatically throttled due to thermal constraints. Hence, in order to make use of all available cores without throttling affecting our evaluations, we run the small cores at 533 MHz and big cores at 903 MHz. The experimental evaluations implicitly reflect the impact of inherent shared-resource contention.

3.3 Profiling

In the *Huawei Kirin 960* SoC, there are four *Small* and four *Big* cores. Hence, the maximum value of both B_i and S_i is limited to four for each Application i. Therefore, we first run each benchmark application on all possible core configurations. Figure 2(b) shows that the throughput of the different applications is concave in almost every case. We use the profiled data and fit it to a concave curve using regression. The fitted values are then utilized in *A-Greedy* to make scheduling decisions. The comparative baselines use the original unmodified values as they do not require concave smoothing to operate.

3.4 Evaluation Systems and Metrics

A-Greedy scheduler can be utilized in both closed and open systems. We describe these systems alongside their preferred optimization metric.

Closed System. In a closed system, applications executing in the system restart execution immediately after their completion. System *throughput* is often used as the performance metric [7] in these systems.

Open System. In an open system: (i) applications enter and leave the system at any time and (ii) run-time scheduling overhead needs to be minimal. *Response time*, i.e., the time elapsed between an application's arrival and exit is a common metric optimized in open systems [7] as it captures both waiting and execution time. Thus, we use *average response time* as the performance metric for our open system evaluations. Applications arrive in the open system using a uniform distribution and permanently leave once they complete execution.

3.5 Comparative Baselines

We evaluate our proposed *A-Greedy* scheduler against three baselines:

ILP. An *ILP*-based scheduler implemented using *Gurobi*[16] that utilizes the measured throughput values as input to obtain the optimal scheduling decision. Though the *ILP* scheduler is guaranteed to present optimal results even in the absence of concavity, it cannot scale beyond dozen cores due to its exponential computational complexity and is therefore infeasible in-practice for use as a run-time scheduler in asymmetric many-cores.

Linux. Our platform's default *Linux* scheduler (4.14.0-rc7-linaro-hikey960) that is based on a load-balancing algorithm described in [4].

MTS-Like. Existing scheduler called *MTS* [13] first maximizes performance by allocating cores to the application thread that has the highest Instruction per Second (IPS). In this step, cores are considered in decreasing order of their performance. Next, it swaps cores allocated to different application threads to improve the system power-efficiency. As we focus on performance, we only utilize the first step of *MTS*. The original performance metric (IPS) used in *MTS* is changed to throughput for a fair comparison with the *A-Greedy* scheduler. *MTS* treats each application thread independently. Thus, it is unable to utilize the throughput concavity (observed when considering the allocation dependency

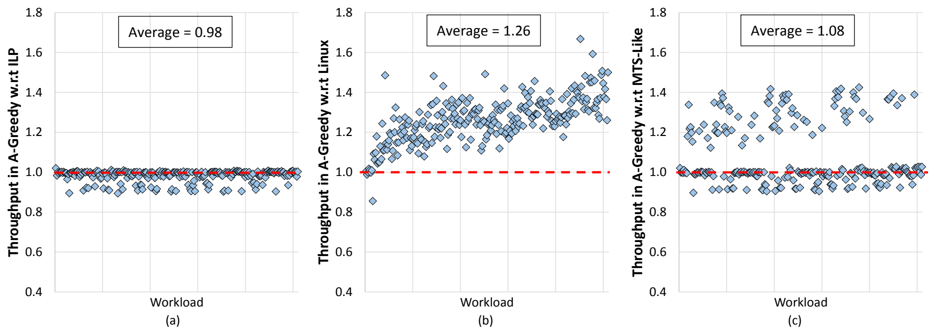

Fig. 4: Throughput obtained under proposed *A-Greedy* scheduler normalized to (a) exhaustive *ILP* scheduler (note that the slight drop in performance is due to concave approximation) (b) default *Linux* scheduler and (c) *MTS-Like* heuristic scheduler on *Kirin 960* asymmetric multi-core.

among the application threads) and thereby compute the optimal schedule. Major difference between *MTS* and *A-Greedy* is theoretical as former is proposed purely as a heuristic. Since we modify the design parameters of original *MTS* algorithm extensively to be within the purview of this work, we rename the version we implement and compare against as *MTS-Like*.

Invocation. In closed-system, *MTS-Like* and *A-Greedy* schedulers are invoked only once as the application mix does not change over time. For open-systems, scheduling decisions in *MTS-Like* and *A-Greedy* are re-calculated when applications enter or leave the system. Note that the *Linux* scheduler is invoked at the default 10 ms period in both these systems as we make no changes to it.

3.6 Asymmetric Multi-Core Evaluations

Closed System Results. A workload is defined as the set of applications that execute in the multi-core. We restrict the number of applications from 1-8 (eight cores in the system) and generate 255 workloads by exploring all possible application combinations. Figure 4(a) reports the throughput obtained using *A-Greedy* scheduler with respect to the *ILP* scheduler in the closed system for these workloads. This figure shows that the throughput under *A-Greedy* scheduler is on average 0.98x of the throughput achievable under *ILP* scheduler. Even though both *A-Greedy* and *ILP* are theoretically optimal, the drop in performance for some workloads can be explained due to the use of regression-based convex approximations for fitting the throughput in *A-Greedy* (*ILP* utilizes real throughput values) for making scheduling decisions.

We also compare our proposed *A-Greedy* scheduler with the default *Linux* scheduler [4] that performs Completely Fair Scheduling on our asymmetric multi-core in Figure 4(b). For this experiment, we launch each application with eight threads. Current *Linux* scheduler uses priority of applications in the system to allocate time slices for executing them on different types of cores. Thus, it does not have any mechanism to use application information to take efficient decisions

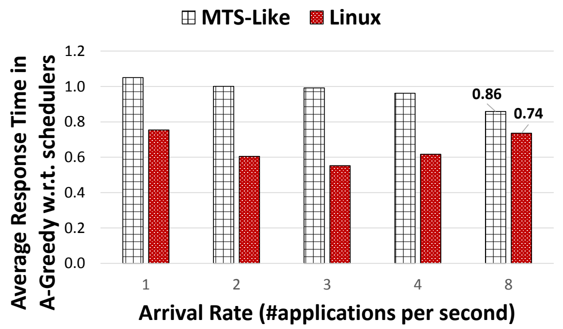

Fig. 5: Average response time under *MTS-Like* and *Linux* normalized to *A-Greedy* on *Kirin 960* asymmetric multi-core modeling an open system.

at run-time. Figure 4(b) shows that the proposed *A-Greedy* scheduler provides on average 26% higher throughput than the default *Linux* scheduler.

Finally, we compare the proposed *A-Greedy* scheduler with the *MTS-Like* heuristic scheduler on our asymmetric multi-core in Figure 4(c). On the small size asymmetric multi-core platform, *A-Greedy* scheduler results in average 8% higher throughput than *MTS-Like* scheduler.

Open System Results. We compare the performance of *A-Greedy*, *Linux* and *MTS-Like* schedulers under different open system loads in Figure 5. The number of applications arriving in the open system (per second) is varied from one to eight, i.e. low to high load. The maximum number of applications arriving in the system is fixed to eight as there are only eight cores in the system. From Figure 5, we see that the *A-Greedy* scheduler reduces the average response time by up to 14% and 45% when compared to the *MTS-Like* and *Linux* schedulers.

3.7 Scalability Analysis

We report the run-time of *ILP* and *A-Greedy* schedulers on the *Big* core for representative scheduling problems with different number of cores and workload sizes in Table 1. We fix the number of cores in a cluster to four for the scalability experiments as the hardware platform that was used to collect profiling data had only four cores per cluster. The load is varied from 12.5% to 100% many-core utilization i.e., the number of applications in the many-core ranges from ($\#cores/8$) to ($\#cores$). Though the problem-solving time of *ILP*-based scheduler is in the order of milliseconds for an 8-core asymmetric multi-core with two clusters, the problem-solving time increases exponentially for a higher number of cores (or clusters). For instance, a 64-core asymmetric many-core scheduling problem in *ILP* does not terminate even after hours. However, *A-Greedy* proposed in this work only takes 47 ms to schedule 512 applications on a 512-core processor.

4 Related Work

Scheduling for asymmetric multi-/many-cores has been an active subject of research since their inception [11]. A number of works looked into maximizing

Table 1: Problem-solving time (in ms) for *ILP* and *A-Greedy* scheduler on varisized representative scheduling problems. ILP-scheduler does not terminate (N.T.) for #cores ≥64.

Load \ **#cores**	**ILP**				**A-Greedy**						
	8	**16**	**32**	**≥ 64**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
12.5%	16	266	26,190	N.T.	0.03	0.03	0.09	0.23	0.60	2	6
25%	30	506	52,086	N.T.	0.02	0.05	0.13	0.34	1	3	12
50%	45	823	110,849	N.T.	0.03	0.06	0.21	0.54	2	6	24
100%	56	1,603	209,830	N.T.	0.03	0.09	0.27	0.91	3	12	47

performance under power-constrains for asymmetric processors [3], [13], [19], [24]. For instance, authors in [13] first map application tasks to achieve high throughput but swap tasks in the second phase to meet the power constraints. However, these works do not determine the number of cores (spanning multiple asymmetric core types) allocated to multi-threaded applications at run-time and do not perform evaluations on real-world platforms. Authors of [14], [15] proposed controller/economic theory based mechanisms on *ARM big.LITTLE* asymmetric multi-core platforms for meeting the quality of service requirements under power-constraints. However, threads belonging to same/different applications are considered independently. Additionally, it is very hard to provide any guarantees on scheduling decisions taken by these mechanisms. To the best of our knowledge, none of the aforementioned work has proposed a low-overhead scheduling algorithm that can be proven to obtain an optimal schedule (under certain constraints) with a proof-of-concept implemented and evaluated on a real-world hardware platform.

5 Conclusion

In this work, we present *A-Greedy*, the first-ever scheduler to be proven theoretically optimal (under certain constraints) for asymmetric multi-/many-core processors. Experimental evaluation on *Kirin 960* asymmetric multi-core shows that the throughput in *A-Greedy* is on average 0.98x the throughput of an optimal ILP-based scheduler but with minimal scheduling overheads. *A-Greedy* provides 26% higher throughput and up to 45% lower average response time than the default Linux Scheduler on *Kirin 960*. *A-Greedy* also provides up to 8% higher throughput and up to 14% lower average response time than state-of-the-art *MTS-Like* heuristic scheduler. Scalability analysis show that *A-Greedy* is fast enough to perform run-time scheduling for large-size asymmetric many-cores.

Acknowledgment

This work was supported by the National Research Foundation, Prime Minister's Office, Singapore under its Industry-IHL Partnership Grant NRF2015-IIP003. We thank the anonymous reviewers for their valuable feedback and insights.

References

1. Amdahl, G.M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: AFIPS (1967)
2. Bienia, C., et al.: Parsec 2.0: A New Benchmark Suite for Chip-Multiprocessors. In: PMBS (2009)
3. Cong, J., et al.: Energy-Efficient Scheduling on Heterogeneous Multi-Core Architectures. In: ISLPED (2012)
4. Corrêa, M., et al.: Operating System Multilevel Load Balancing. In: SAC (2006)
5. van Dijk, T., et al.: Lace: Non-Bocking Split Deque for Work-Stealing. In: Euro-Par (2014)
6. Feitelson, D.G., et al.: Toward Convergence in Job Schedulers for Parallel Supercomputers. In: JSSPP (1996)
7. Feitelson, D.G., et al.: Metrics and Benchmarking for Parallel Job Scheduling. In: JSSPP (1998)
8. Goens, A., et al.: Analysis of Process Traces for Mapping Dynamic KPN Applications to MPSoCs. In: IESS (2015)
9. Gulati, D.P., et al.: Multitasking Workload Scheduling on Flexible-Core Chip Multiprocessors. In: PACT (2008)
10. Henkel, J., et al.: Invasive Manycore Architectures. In: ASP-DAC (2012)
11. Kumar, R., et al.: Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In: MICRO (2003)
12. Libutti, S., et al.: Co-Scheduling Tasks on Multi-Core Heterogeneous Systems: An Energy-Aware Perspective. COMPUT DIGIT TEC (2016)
13. Liu, G., et al.: Dynamic Thread Mapping for High-Performance, Power-Efficient Heterogeneous Many-Core Systems. In: ICCD (2013)
14. Muthukaruppan, T.S., et al.: Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era. In: DAC (2013)
15. Muthukaruppan, T.S., et al.: Price Theory Based Power Management for Heterogeneous Multi-cores. In: ASPLOS (2014)
16. Optimization, G.: Gurobi optimizer 8.0. Gurobi: http://www.gurobi.com (2018)
17. Pathania, A.: Scalable Task Schedulers for Many-Core Architectures. Ph.D. thesis, Karlsruhe Institute of Technology, Germany (2018)
18. Pathania, A., et al.: Optimal Greedy Algorithm for Many-Core Scheduling. TCAD (2017)
19. PD, S.M., et al.: 3D Many-Core Microprocessor Power Management by Space-Time Multiplexing based Demand-Supply Matching. TC (2015)
20. Ristov, S., et al.: Superlinear Speedup in HPC Systems: Why and When? In: FedCSIS (2016)
21. Singh, A.K., et al.: Mapping on Multi/Many-core Systems:Survey of Current and Emerging Trends. In: DAC (2013)
22. Stuber, M.D., et al.: Convex and Concave Relaxations of Implicit Functions. Optimization Methods and Software (2015)
23. Venkataramani, V., et al.: Scalable Dynamic Task Scheduling on Adaptive Many-Core. In: MCSoC (2018)
24. Winter, J.A., et al.: Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architectures. In: PACT (2010)