Scheduling with Task Duplication for Application Offloading

Arani Bhattacharya*, Ansuman Banerjee†, Pradipta De‡
*Stony Brook University and SUNY Korea, {arbhattachar@cs.stonybrook.edu}
†Indian Statistical Institute, {ansuman@isical.ac.in}
‡Georgia Southern University, { pde@georgiasouthern.edu}

***Abstract*—Computation offloading frameworks partition an application's execution between a cloud server and the mobile device to minimize its completion time on the mobile device. An important component of an offloading framework is the partitioning algorithm that decides which tasks to execute on mobile device or cloud server. The partitioning algorithm schedules tasks of a mobile application for execution either on mobile device or cloud server to minimize the application finish time. Most offloading frameworks partition parallel applications devices using an optimization solver which takes a lot of time. We show that by allowing duplicate execution of selected tasks on both the mobile device and the remote cloud server, a polynomial algorithm exists to determine a schedule that minimizes the completion time. We use simulation on both random data and traces to show the savings in both finish time and scheduling time over existing approaches. Our trace-driven simulation on benchmark applications shows that our algorithm reduces the scheduling time by 8 times compared to a standard optimization solver while guaranteeing minimum makespan.**

***Index Terms*—Mobile Cloud, Application Offloading, Code Partitioning, Mobile System, Optimization, Task Scheduling**

I. Introduction

Mobile devices are constrained by limited compute power. However, mobile applications are evolving to become more resource intensive. In this setting, offloading parts of an application to remote compute resources such as cloud servers can reduce response time of mobile applications. While execution on cloud servers is faster, offloading also requires sending of program states over wireless network which has high latency. Thus, for offloading to be useful, careful selection of tasks for execution on remote servers is essential.

Offloading frameworks model execution of a mobile application as a task graph. A task graph consists of a set of vertices representing the tasks in the application, and a set of edges representing dependencies between tasks. Each task and dependency is annotated with one or more cost representing time or energy. The offloading framework selects tasks for remote execution at application startup in order to reduce time and/or energy. Thus, the algorithm used by the offloading framework to select tasks needs to be fast and has to generate a good schedule to ensure quick startup and time and/or energy savings.

In this work, we propose utilizing scheduling using task duplication for execution on mobile device and cloud server. Existing application offloading frameworks partition the task graph into two distinct components for execution on mobile device and cloud server respectively. In this work, we show that allowing a limited number of tasks to execute on both mobile device and cloud server reduces the finish time of application, or makespan. Moreover, unlike graph partitioning, scheduling using duplication can be done in polynomial time. Thus, our technique of task duplication leads us to an algorithm that runs in polynomial time and reduces makespan compared to existing scheduling techniques.

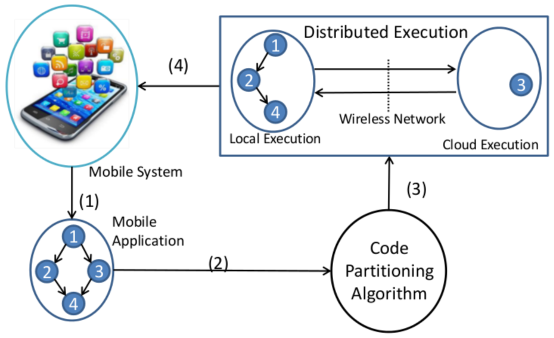
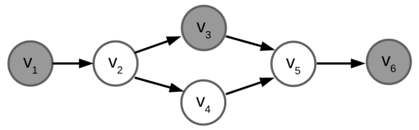


Fig. 1: Workflow of an offloading framework. Execution of a mobile application, represented as a task graph, is profiled to determine the compute workload of each task. The code partitioning algorithm uses the profile as input to schedule a task locally or on remote server.

We illustrate the benefit of task duplication with an example. Fig. 2 shows a task graph, where some tasks marked in gray must execute locally, while others can be scheduled on the device or remote server. Time to execute v_1, v_3, v_6 locally is $10ms$ each, while v_2, v_4, v_5 is $20ms$ each. Assuming that the remote server is 5 times faster than the device, time to execute v_2, v_4, v_5 on cloud is $4ms$. The communication latency due to data transfer is $10ms$ for each edge. With this setting, complete local execution without offloading takes $80ms$, where v_3 and v_4 can be executed in parallel on a multi-core mobile processor. Formulating the problem as an ILP, a solver schedules v_1, v_3, v_5, and v_6 locally, and v_2 and v_4 remotely, giving a makespan of $78ms$. Now, if duplicate execution is allowed, then v_2 can be executed both locally and remotely, thereby saving the time to transfer data for the dependent tasks v_3 and v_4, where v_3 is scheduled locally and v_4 on cloud. This leads to a makespan of $70ms$, showing the benefit of task duplication.

Offloading Framework	Optimization Objective	Constraint Parameter	Application Type	Solution Technique	Type of Solution	Scheduling Time Complexity
MAUI [1]	Energy	Time	Sequential	ILP	Optimal	Exponential $[O(2^n)]$
CloneCloud [2]	Energy	Time	Concurrent	ILP	Optimal	Exponential $[O(2^n)]$
ThinkAir [3]	Energy, Time		Concurrent	Heuristic	No performance bound	Polynomial $[O(n)]$
Hermes [4]	Time	Energy	Subset of concurrent	Algorithm	Near-optimal	Polynomial $[O(n^4m^2)]$
Tango [5]	Time		Concurrent	Heuristic using duplicate execution	No performance bound	Constant
ATOM (Our Work)	Time		Concurrent	Dynamic Programming Algorithm	Optimal	Polynomial $[O(m^2n^2)]$

TABLE I: Comparison of different offloading approaches. n and m represent the number of tasks in the task graph and number of servers in the offloading system respectively.



(a) A task graph representing a mobile application. Tasks marked in gray must be executed locally on the mobile device, while the remaining tasks can be scheduled locally or on remote servers.

	v_1	v_2	v_3	v_4	v_5	v_6
Mobile device (t_j^0)	10	20	10	20	20	10
Cloud server (t_j^1)	NA	4	NA	4	4	NA

(b) Execution times of each task.

Fig. 2: A task graph along with its parameters. We assume a single cloud server, with a communication time of $10ms$ between the mobile device and cloud server for each edge.

The rest of the paper is organized as follows. Section II develops a formulation of the task scheduling problem. Section III presents the polynomial task scheduling algorithm, ATOM. Sections IV and V present the evaluation of ATOM using simulation and real-world application traces respectively. Related work is presented in Section VI. We conclude in Section VII.

II. Problem Formulation

A mobile cloud computing (MCC) system comprises of a mobile device (denoted by $\mathcal{M}_0$) and multiple cloud servers (denoted by $\mathcal{M}_k$, where $1 \leq k \leq m$). We assume that each of these machines have unbounded number of processors. Moreover, processors on each machine are homogeneous.

We represent execution of a mobile application as a directed acyclic graph (DAG) $G = (\mathbb{V}, \mathbb{E})$, where the vertex set $\mathbb{V}$ represents the set of n methods or tasks, and the edge set $\mathbb{E}$ represents the dependencies among tasks. A task v_j may be executed on one or more of the available machines $\mathcal{M}_k (0 \leq k \leq m)$. However, the first task v_1 and the last task v_n must be executed locally on mobile device $\mathcal{M}_0$. Execution of some other tasks may also be tied to the mobile device, as they may depend on some hardware such as camera, GPS, etc. Execution of v_j on $\mathcal{M}_k$ takes t_j^k time. If for a dependency (v_i, v_j), v_j is executed on a different machine $\mathcal{M}_k$ than v_i's machine $\mathcal{M}_h$, then data associated with (v_i, v_j)

$\mathbb{V}$	Vertex set of the graph
$\mathbb{E}$	Edge set of the graph
v_j	A task in the application execution graph
v_1	First task in the application execution graph
v_n	Last task in the application execution graph
m	Number of servers in the offloading system
n	Number of tasks in the task graph
(v_i, v_j)	A dependency from the task v_i to v_j
$\mathcal{M}_0$	Mobile device
$\mathcal{M}_k$	A machine with multiple processors
t_j^k	Execution time of task v_j on machine $\mathcal{M}_k$
r_{ij}^{hk}	Time to migrate data of (v_i, v_j) from $\mathcal{M}_h$ to $\mathcal{M}_k$
x_j^k	Decision variable indicating execution of v_j on $\mathcal{M}_k$
T_j^k	Finish time of v_j on $\mathcal{M}_k$
S_j^k	Start time of v_j on $\mathcal{M}_k$
D_{ij}^k	Data arrival time of (v_i, v_j) on $\mathcal{M}_k$
T_n^0	Finish time of last task on mobile device, i.e. makespan

TABLE II: Symbols introduced in Section II

must be migrated to $\mathcal{M}_k$ before v_j can begin execution. Migrating this data takes r_{ij}^{hk} time. However, migrating from a different processor within the same machine is assumed to take negligible time, i.e. $r_{ij}^{kk} = 0 \; \forall k = 0, ...m, \forall (v_i, v_j) \in \mathbb{E}$. We assume that both execution times t_j^k and migration times r_{ij}^{hk} are obtained by prior profiling of the application.

We define makespan as the time T_n^0 to finish execution of the last task v_n on $\mathcal{M}_0$. We now define the execution finish time of each task v_j. Let T_j^k be the execution finish time of v_j on $\mathcal{M}_k$. Let S_j^k denote the time when execution of v_j on $\mathcal{M}_k$ starts. Then, the finish time of v_j is the sum of start time S_j^k and execution time t_j^k:

$$\forall v_j \in \mathbb{V}, \forall k = 0, \ldots, m, \quad T_j^k = S_j^k + t_j^k \tag{1}$$

To find the start time S_j^k of v_j on $\mathcal{M}_k$, we note that v_j can start when all its predecessor v_i's are available. Let D_{ij}^k denote the time when data associated with (v_i, v_j) becomes available on $\mathcal{M}_k$. Since each machine $\mathcal{M}_k$ has multiple processors, a task can be executed as soon as its data is available. Thus, the earliest start time is equal to the highest value of data arrival time:

$$\forall v_j \in \mathbb{V}, \forall k = 0, \ldots, m, \quad S_j^k = \max_{(v_i, v_j)} D_{ij}^k \tag{2}$$

For the first task v_1, there are no predecessors. Moreover, it can be executed only on the mobile device $\mathcal{M}_0$. Thus, for the first task, we say that start time on $\mathcal{M}_0$ as 0, and all other machines $\mathcal{M}_1, ...$ as ∞:

$$S_1^0 = 0, \quad \forall k = 1, ...m, \quad S_1^k = \infty \tag{3}$$

The data arrival time of (v_i, v_j) is the sum of finish time T_h^k of v_i on any $\mathcal{M}_h$ and migration time r_{ij}^{hk}. However, since v_i can execute on many $\mathcal{M}_h$'s, and we are looking for the lowest possible data arrival time, we have:

$$\forall (v_i, v_j) \in \mathbb{E}, \forall k = 0, \ldots, m, \quad D_{ij}^k = \min_{h=0,\ldots,m} (T_i^h + r_{ij}^{hk}) \tag{4}$$

Eqns 1 to 4 give us a recurrence relation that computes the minimum makespan. However, we note that a particular task v_j is only executed on one or more $\mathcal{M}_k$'s. Let x_j^k be a decision variable denoting whether v_j is executed on $\mathcal{M}_k$, i.e.

$$x_j^k = \begin{cases} 1, & \text{if } v_j \text{ is executed on } \mathcal{M}_k, \text{ and} \\ 0, & \text{if } v_j \text{ is not executed on } \mathcal{M}_k. \end{cases}$$

Then, we rewrite Eqn 1 in terms of x_j^k as:

$$T_j^k = \begin{cases} S_j^k + t_j^k, & \text{if } x_j^k = 1, \\ \infty, & \text{if } x_j^k = 0. \end{cases}$$

We need to design an algorithm to choose values of x_j^k's that minimizes makespan T_n^0. We utilize the recurrence relation to design a dynamic programming algorithm.

III. Our proposed Algorithm

Our algorithm starts by assuming that each v_j is executed on machines $\mathcal{M}_k$'s. Thus, for each v_j, the output of its predecessor v_i is available on each $\mathcal{M}_k$. Before execution of v_j on $\mathcal{M}_k$ begins, we need to determine which $\mathcal{M}_h$ can send the data associated with (v_i, v_j) the fastest. We store the fastest time when data of (v_i, v_j) arrives at $\mathcal{M}_k$ in D_{ij}^k and store the corresponding value of h in a lookup table. When all the predecessors v_i's have arrived at $\mathcal{M}_k$, execution of v_j can start. This value of time, equal to the maximum value of D_{ij}^k across all v_i's, is stored in S_j^k. The time taken to finish execution of v_j, T_j^k is the sum of start time S_j^k and execution time t_j^k. By calculating recursively the finish times of each task, we obtain the finish time of the last task, or makespan T_0^n. Once the makespan is obtained, we use the lookup table to determine the machines M_h from each output of each predecessor v_i's was used. This lets us get the execution machines of each task. The exact algorithm is shown in detail in Algorithm 1. Table III shows the working of the algorithm on our example task graph shown in Fig. 2.

To analyze the time complexity of ATOM, we first analyze Procedure CALCULATE-MAKESPAN. We note that the loop on Line 4 runs $n-1$ times, once for each task in the DAG. The inner loop (on Line 5) runs once for each predecessor task, i.e. the number of incoming edges in the DAG. Let the number of such incoming edges to a task v_i be d_i. The loops on Lines 6 and 7 iterate a total of m^2 times. Within the innermost loop

Algorithm 1 Algorithm ATOM to compute makespan and obtain execution schedule of an application execution graph. It accepts a DAG $G = (\mathbb{V}, \mathbb{E})$ representing a mobile application as input. It returns the makespan T_n^0 and decision variable x_j^k indicating whether v_j should be executed on $\mathcal{M}_k$.

```
 1: procedure CALCULATE-MAKESPAN
 2:     $T_1^0 \leftarrow t_1^0$
 3:     $T_1^1 \leftarrow \infty$
 4:     for $j = 2$ to $n$ do
 5:         for all predecessors $v_i$ of $v_j$ do
 6:             for all $k = 0$ to $m$ do
 7:                 for all $h = 0$ to $m$ do
 8:                     if $T_i^k < T_i^h + r_{ij}^{hk}$ then
 9:                         $D_{ij}^k \leftarrow T_i^k$
10:                         $Lookup_{ij}^k \leftarrow k$
11:                     else
12:                         $D_{ij}^k \leftarrow T_i^k + r_{ij}^{hk}$
13:                         $Lookup_{ij}^k \leftarrow h$
14:                     end if
15:                 end for
16:             end for
17:             for all $k = 0$ to $m$ do
18:                 $S_j^k \leftarrow \max_{(i,j)\in\mathbb{E}}\{D_{ij}^k\}$
19:                 $T_j^k \leftarrow S_j^k + t_j^k$
20:                 if $v_j$ is tied to mobile AND $k \neq 0$ then
21:                     $T_j^k \leftarrow \infty$
22:                 end if
23:             end for
24:         end for
25:     end for
26: end procedure
27: procedure GET-SCHEDULE
28:     Set all values of $x$ to 0
29:     $x_n^0 \leftarrow 1$
30:     for $j = n$ to 2 do
31:         for all predecessors $v_i$ of $v_j$ do
32:             for all $k = 0$ to $m$ do
33:                 if $x_j^k = 1$ then
34:                     $h \leftarrow Lookup_{ij}^k$
35:                     $x_i^h \leftarrow 1$
36:                 end if
37:             end for
38:         end for
39:     end for
40:     return $x$
41: end procedure
```

on Line 7, each step requires constant ($O(1)$) time. Thus, total number of steps to run the procedure is given by:

$$\mathcal{T}_1(n) = \sum_{i=1}^{n-1} d_i = O(m^2|\mathbb{E}|).$$

Similarly, Procedure GET-SCHEDULE also has an outer loop running $n-1$ times, and an inner loop for each predecessor. Each inner loop requires m number of times. Thus, time complexity of Procedure GET-SCHEDULE, $\mathcal{T}_2(n)$ is also $O(m|\mathbb{E}|)$. Therefore, time complexity of our proposed algorithm is given by:

$$\mathcal{T}(n) = \mathcal{T}_1(n) + \mathcal{T}_2(n) = O(m^2|\mathbb{E}|) + O(m|\mathbb{E}|) = O(m^2|\mathbb{E}|).$$

Since the number of dependencies is of the order of $O(n^2)$, this gives us a time complexity of $O(m^2n^2)$, where m and n are the number of servers and number of tasks respectively.

Current Task	Predecessor Task	Data Arrival Time	Start Time	Location of Predecessor	Finish Time	Data Arrival Time	Start Time of Current Task	Location of Predecessor	Finish Time
v_j	v_i	D^0_{ij}	S^0_j	$Lookup^0_{ij}$	T^0_j	D^1_{ij}	S^1_j	$Lookup^1_{ij}$	T^1_j
v_2	v_1	10	10	Mobile	30	20	20	Mobile	24
v_3	v_3	$\min(30, 22+10) = 30$	30	Mobile	40				
v_4	v_2	$\min(30, 22+10) = 30$	30	Mobile	50	$\min(30+10, 24) = 24$	24	Cloud	28
v_5	v_3	40	40	Mobile	60	$40+10 = 50$	50	Mobile	54
	v_4	$\min(50, 28+10) = 38$		Cloud		28		Cloud	
v_6	v_5	$\min(60, 52+10) = 60$	60	Mobile	70				

TABLE III: Table to minimize makespan of task graph shown in Fig. 2 used by Algorithm 1

We now explain how task duplication reduces makespan in our algorithm. First, we note that a task is duplicated only when new threads are spawned. When a new thread is spawned, one thread may be faster on the mobile device, while the other thread is faster on cloud server. In this case, executing one or more tasks preceding the spawning of the thread on both mobile device and cloud server may be faster. For example, in Fig. 2, a new thread is spawned at v_2. Thus, v_2 has two outgoing edges connecting v_3 and v_4. If we execute v_2 only on $\mathcal{M}_0$ (mobile), then migrating (v_2, v_4) and then executing v_4 on $\mathcal{M}_1$ is slower than executing only v_4 on $\mathcal{M}_0$. Thus, v_4 also executes on mobile device. If we execute v_2 only on $\mathcal{M}_1$ (cloud server), then migrating (v_2, v_3) back to $\mathcal{M}_0$ slows down execution of v_3. On the other hand, if we execute v_2 on both $\mathcal{M}_0$ and $\mathcal{M}_1$, this allows execution of v_3 to start much faster, and also does not require migration of (v_2, v_3).

Another major advantage of allowing task duplication is that it results in a polynomial algorithm. This is because allowing the same task to execute on multiple machines $\mathcal{M}_k$ allows us to divide the entire scheduling of task graphs into smaller scheduling problems. For example, in Fig. 2, it is possible to separately schedule the tasks v_1, v_2, v_3, v_5, v_6 in one step, and v_1, v_2, v_4, v_5, v_6 separately in another step. If any v_j is scheduled on two different machines $\mathcal{M}_h$ and $\mathcal{M}_k$, then it can be executed on both. Since scheduling a linear sequence of tasks is polynomial, using task duplication reduces the problem to a series of polynomial problems. Thus, the overall scheduling problem also becomes polynomial when tasks duplication is allowed.

IV. Simulation-based Evaluation

In this section, we compare ATOM with schedules generated by Integer Linear Programming (ILP), Tango [5] and local execution. We implemented the ILP (discussed in Section II), Tango and ATOM on an Intel Xeon (CPU: E5-2630) 6-core processor system in Java (openJDK 1.7) programming language. We generated call graphs of different sizes ranging from 10 to 100, with each size of call graph having 100 random samples each. We study different performance parameters like makespan, scheduling time, energy consumption and memory footprint of scheduling algorithms. We use Java ThreadMXBean interface to measure scheduling time, the energy model discussed in Section II to measure energy consumption and Java Instrumentation to measure memory footprint [6].

A. Performance Comparison

1) Makespan: We compare the makespans of different algorithms. Fig. 3(a) shows the makespan for different number of tasks in the application graph. We note that ATOM provides the smallest makespan, followed by Tango, ILP and local execution. This is because ATOM always provides the optimal makespan, and thus its maskespan must be the smallest across different methods for any given application.

2) Scheduling Time: Fig. 3(b) shows the scheduling time of ATOM and ILP for different number of tasks from 10 to 100. We omit Tango and local execution here since these techniques do not need to run any scheduling algorithm during startup. We note that for smaller applications with less than 40 tasks, an ILP is faster. For larger applications, the scheduling time of an ILP increases rapidly. This is because solving an ILP takes exponential time, whereas ATOM is a polynomial algorithm.

3) Energy Consumption: Fig. 3(c) shows the energy consumption using the four different methods. We note that an ILP consumes the least amount of energy, followed by ATOM, local execution and Tango. This is because ATOM uses task duplication to save time. However, this consumes additional energy on mobile device, since this requires both local execution and migration. Thus, Tango consumes the highest energy, since it duplicates all tasks on both mobile device and server.

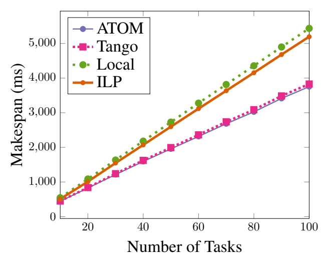
4) Memory Footprint: Fig. 3(d) shows the memory footprint of ATOM and ILP. To account for the large differences in memory footprint, we plot it on a logarithmic scale. We once again note that running ATOM consumes much smaller memory than an ILP. This is because an ILP formulation requires storing a large matrix as input. The space complexity of ATOM is linear with additional memory only being used to store the start times, finish times and execution platforms of each task.

B. Effect of Task Duplication

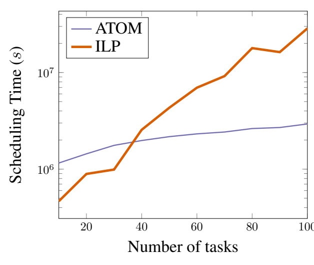
We now study the amount of task duplication performed by ATOM, and its effect on the makespan. We note that unlike general DAGs, tree-structured graphs do not require any task duplication to minimize makespan. Thus, we generate random general DAGs for these experiments.

1) Makespan: To understand the effect of task duplication on makespan, we utilize the formulation described in Section II. In the formulation described in Section II, we add an additional constraint to ensure that no task redundancy is used:

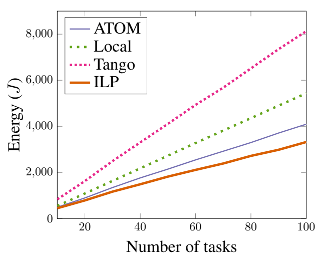
$$\forall v_j \in \mathbb{V}, \sum_{k=0}^{m} x^k_j = 1 \tag{5}$$



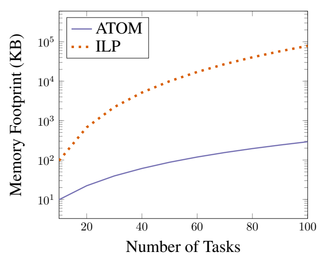
(a) Makespan



(b) Scheduling Time

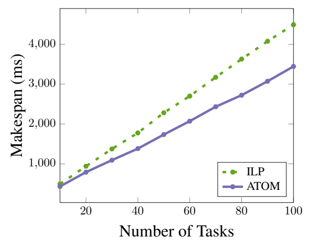


(c) Energy Consumption

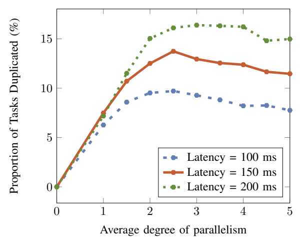


(d) Memory footprint

Fig. 3: Comparison of makespan, scheduling time, energy consumption and memory footprint of ATOM, ILP, Tango and local execution.



(a) Makespan



(b) Makespan

Fig. 4: Comparison of makespan of ATOM with an ILP formulation to obtain the effect of task duplication. The ILP formulation does not use task duplication.

We then compare the makespan given by the ILP with ATOM.

Fig. 4(a) shows the difference in makespan using the ILP and ATOM for different number of tasks. We note that ATOM has a lower makespan than the ILP in each case. Moreover, the difference in makespan increases with an increase in the number of tasks. Thus, for 10 tasks, ATOM has 12% lower makespan than the ILP. For 100 tasks, this increases to 25%.

This observation is explained by noting that task duplication reduces makespan. The increase in time saving with an increase in size of DAG also shows that task duplication saves more time for larger DAGs. This is because larger graphs have more scope for exploitation of parallelism, which can be better exploited with lower costs using task duplication.

2) Amount of task duplication: Fig. 4(b) shows the amount of task duplication performed by ATOM for different amounts of available parallelism. We note that when out-degree is equal to 1, the application is completely sequential. Thus, no task duplication is used. The amount of task duplication reaches a peak of around 10% when the maximum out-degree is 5. Further increase in out-degree of tasks slightly reduces the amount of task duplication.

This observation confirms that task duplication reduces makespan by reducing communication cost of parallel execution. When there is more concurrency in the application, more parallelism can be utilized by utilizing more duplication. Thus, when more threads are spawned, the task amount of duplication increases.

V. Trace-Based Evaluation

We perform trace-driven simulation on benchmark programs, and compare its performance with other algorithms. To obtain traces from any available Java program, we utilize aspect-oriented programming using AspectJ framework [7]. AspectJ allows programmers to add additional code at the call points of each method through bytecode-level modifications. We use AspectJ to obtain the traces of each method call. We also serialized arguments of each method and printed the size of arguments. This gives us the amount of data required to migrate at any particular call point. Finally, we calculated the time taken to execute each method using Java's ThreadMXBean interface [6]. We use these data to annotate the call graph. We identify the methods that require access to user input or output device (such as println method) as native.

We perform our experiments on nine selected SPEC JVM benchmarks [8]. The nine benchmarks are selected because they mirror mobile workloads. Thus, we use traces of SPEC JVM benchmarks to get results that are representative of those on real workloads.

A. Makespan

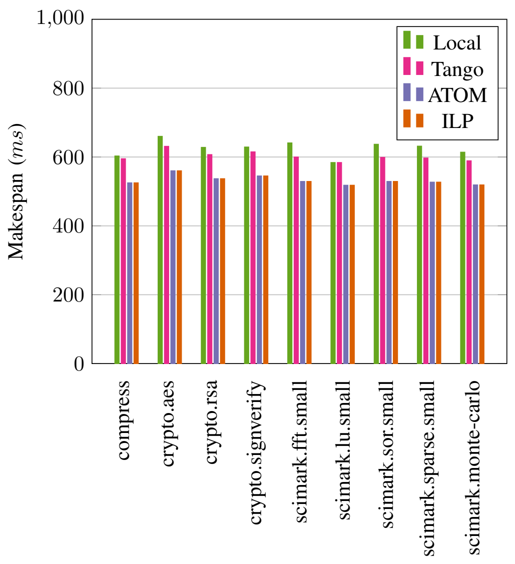
To understand the effect on execution time, we obtain the makespan or application finish time using ILP, ATOM, Tango and local execution. We use a constant bandwidth of 1 Mbps to run our traces, and a round-trip time (RTT) of $50ms$. Fig. 5(a) shows the effect of the four methods on makespan. We note that ATOM reduces the makespan by 15% compared to local execution, and 10% compared to Tango. Moreover, ATOM and ILP gives us almost the same makespan in each case.

B. Scheduling Time

We now compare the scheduling time of ILP and ATOM in Fig. 5(b). We note that the average scheduling time is less than $0.2s$ for ATOM. This is much lower than an ILP, which requires an average of over $1s$ of scheduling time. ATOM reduces the average scheduling time of applications by around 8 times.

VI. Related Work

To systematically study the varieties of job scheduling problems, they are classified based on machine architecture (α), task model (β) and optimization objective (γ) [9]. This



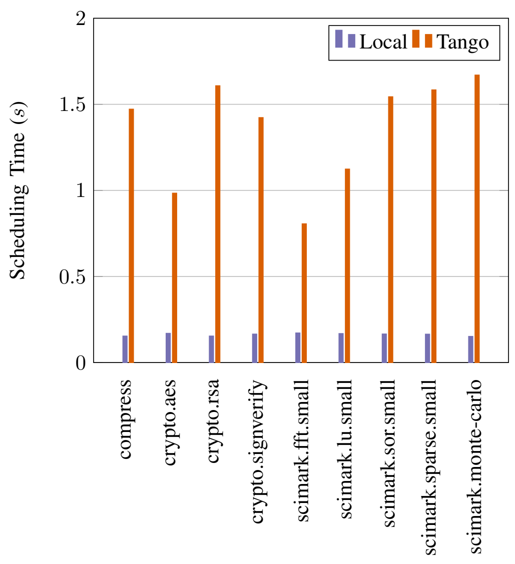


Fig. 5: Comparison of energy consumption of SPEC benchmarks using ILP, ATOM, Tango and local execution.

classification scheme is referred to as $\alpha|\beta|\gamma$ model, based on the three parameters of classification. We explain related scheduling algorithms in the context of this scheme.

In our machine architecture, communication costs differ based on the execution platform of each processor. This is known as a cluster machine model, and is denoted by $P(a, b)$. Here a denotes the number of clusters, and b represents the number of processors in each cluster. Thus, in our case, $a = 2$ and $b = \infty$. Precedence constraints between tasks are denoted by $prec$, and task duplication is denoted by dup. The objective is to reduce the makespan or schedule length of the last task on mobile device T_N^m. Thus, this problem is denoted by $P(2, \infty)|prec, dup|$makespan. Most previous studies have proposed scheduling algorithms for machine models that are either completely homogeneous or heterogeneous.

Existing Mobile Cloud Computing frameworks fall into two categories based on their scheduling techniques. MAUI [1] and CloneCloud [2] utilize an Integer-Linear Programming (ILP) solver to optimally schedule tasks in exponential time. The alternative approach, used by ThinkAir [3], utilizes heuristic to schedule tasks. This has a low time complexity, but does not guarantee minimization of time or energy. Hermes [4] presents an approximation scheme to minimize makespan within a given energy budget. Tango [5] uses duplicate execution of all possible tasks on mobile device and server to speed up applications. Our algorithm ATOM combines the advantages of Tango and ILP by guaranteeing minimum makespan while having low time complexity.

VII. Conclusion

Mobile devices continue to be limited by their compute power. In this setting, offloading parts of the application to resource rich remote servers can enable wide class of applications. Typically offloading algorithms were designed as optimization problems solved as Integer Linear Programs, or using heuristics, thereby lacking performance guarantees, and may scale poorly. We show that allowing duplicate execution of a few selected tasks leads to a polynomial time scheduling algorithm that minimizes the total completion time of an application. Our algorithm ATOM (Algorithm for Time Optimization on Mobiles) determines a schedule to execute tasks of a concurrent application with duplication such that makespan is minimized. Our simulation and trace-driven experiments show that ATOM significantly reduces makespan and energy consumption while executing in polynomial time.

Acknowledgement

This research was funded by the MSIP, Korea, under the ”ICT Consilience Creative Program” (IITP-2015-R0346-15-1007) supervised by the IITP (Institute for Information & Communication Technology Promotion).

References

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, “Maui: making smartphones last longer with code offload,” in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “Clonecloud: elastic execution between mobile device and cloud,” in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[3] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, “Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading,” in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 945–953.

[4] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai, “Hermes: Latency optimal task assignment for resource-constrained mobile computing,” in *Proceedings of INFOCOM*. IEEE, 2015.

[5] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao, “Accelerating mobile applications through flip-flop replication,” in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 137–150.

[6] “Threadmxbean (java se 7),” http://docs.oracle.com/javase/7/docs/api/.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “Getting started with aspectj,” *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, 2001.

[8] “Specjvm2008,” https://www.spec.org/jvm2008/.

[9] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, “Optimization and approximation in deterministic sequencing and scheduling: a survey,” *Annals of discrete mathematics*, vol. 5, pp. 287–326, 1979.