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 completion 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.

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.
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.

![Task Graph](image)

(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.

(b) Execution times of each task.

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 \( M_0 \)) and multiple cloud servers (denoted by \( M_k \), 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 \( M_k (0 \leq k \leq m) \). However, the first task \( v_1 \) and the last task \( v_n \) must be executed locally on mobile device \( M_0 \). 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 \( M_k \) takes \( t^k_{ij} \) time. If for a dependency \( (v_i, v_j) \), \( v_j \) is executed on a different machine \( M_k \) than \( v_i \)'s machine \( M_h \), then data associated with \( (v_i, v_j) \) must be migrated to \( M_k \) before \( v_j \) can begin execution. Migrating this data takes \( r^k_{ij} \) time. However, migrating from a different processor within the same machine is assumed to take negligible time, i.e., \( r^k_{ij} = 0 \ \forall k = 0, \ldots, m, (v_i, v_j) \in \mathbb{E} \).

We assume that both execution times \( t^k_{ij} \) and migration times \( r^k_{ij} \) are obtained by prior profiling of the application.

We define makespan as the time \( T^0_n \) to finish execution of the last task \( v_n \) on \( M_0 \). We now define the execution finish time of each task \( v_j \). Let \( T^k_j \) be the execution finish time of \( v_j \) on \( M_k \). Let \( S^k_j \) denote the time when execution of \( v_j \) on \( M_k \) starts. Then, the finish time of \( v_j \) is the sum of start time \( S^k_j \) and execution time \( t^k_{ij} \):

\[
\forall v_j \in \mathbb{V}, \forall k = 0, \ldots, m, \quad T^k_j = S^k_j + t^k_{ij} \quad (1)
\]

To find the start time \( S^k_{ij} \) of \( v_j \) on \( M_k \), we note that \( v_j \) can start when all its predecessor \( v_i \)'s are available. Let \( D^k_{ij} \) denote the time when data associated with \( (v_i, v_j) \) becomes available on \( M_k \). Since each machine \( M_k \) 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:

\[
\forall v_j \in \mathbb{V}, \forall k = 0, \ldots, m, \quad S^k_{ij} = \max_{(v_i, v_j)} D^k_{ij} \quad (2)
\]

For the first task \( v_1 \), there are no predecessors. Moreover, it can be executed only on the mobile device \( M_0 \). Thus, for the
first task, we say that start time on \( M_0 \) as 0, and all other machines \( M_1, \ldots \) as \( \infty \):
\[
S^0_1 = 0, \quad \forall k = 1, \ldots, m, \quad S^k_i = \infty
\] (3)

The data arrival time of \((v_i, v_j)\) is the sum of finish time \( T^k_i \) of \( v_i \) on any \( M_k \), and migration time \( r^h_{ij} \). However, since \( v_i \) can execute on many \( M_k \)’s, and we are looking for the lowest possible data arrival time, we have:
\[
\forall (v_i, v_j) \in E, \forall k = 0, \ldots, m, \quad D^k_{ij} = \min_{h=0,...,m} (T^k_i + r^h_{ij})
\] (4)

Eqs 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 \( M_k \)’s. Let \( x^k_j \) be a decision variable denoting whether \( v_j \) is executed on \( M_k \), i.e.
\[
x^k_j = \begin{cases} 
1, & \text{if } v_j \text{ is executed on } M_k, \text{ and} \\
0, & \text{if } v_j \text{ is not executed on } M_k.
\end{cases}
\]

Then, we rewrite Eqn 1 in terms of \( x^k_j \) as:
\[
T^k_j = \begin{cases} 
S^k_j + t^k_j, & \text{if } x^k_j = 1, \\
\infty, & \text{if } x^k_j = 0.
\end{cases}
\]

We need to design an algorithm to choose values of \( x^k_j \)’s that minimizes makespan \( T_n \). 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 \( M_k \)’s. Thus, for each \( v_j \), the output of its predecessor \( v_i \) is available on each \( M_k \). Before execution of \( v_j \) on \( M_k \) begins, we need to determine which \( M_k \) 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 \( M_k \) in \( D^k_{ij} \) and store the corresponding value of \( h \) in a lookup table. When all the predecessors \( v_i \)’s have arrived at \( M_k \), execution of \( v_j \) can start. This value of time, equal to the maximum value of \( D^k_{ij} \) across all \( v_i \)’s, is stored in \( S^k_j \). The time taken to finish execution of \( v_j \), \( T^k_j \) is the sum of start time \( S^k_j \) and execution time \( t^k_j \).

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 \( M_k \) 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 \textsc{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 on Line 7, each step requires constant \( (O(1)) \) time. Thus, total number of steps to run the procedure is given by:
\[
T_1(n) = \sum_{i=1}^{n-1} d_i = O(m^2|E|).
\]

Similarly, Procedure \textsc{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 \textsc{Get-Schedule}, \( T_2(n) \) is also \( O(m|E|) \). Therefore, time complexity of our proposed algorithm is given by:
\[
T(n) = T_1(n) + T_2(n) = O(m^2|E|) + O(m|E|) = O(m^2|E|).
\]

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.
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 v2. Thus, v2 has two outgoing edges connecting v1 and v3. If we execute v2 only on M0 (mobile), then migrating (v2, v3) and then executing v3 on M1 is slower than executing only v3 on M0. Thus, v3 also executes on mobile device. If we execute v2 only on M1 (cloud server), then migrating (v2, v3) back to M0 slows down execution of v3. On the other hand, if we execute v2 on both M0 and M1, this allows execution of v3 to start much faster, and also does not require migration of (v2, v3).

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 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 v1, v2, v3, v5, v6 in one step, and v1, v2, v4, v5, v6 separately in another step. If any vj is scheduled on two different machines M_k and M_l, 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 makespan 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:

\[ \forall v_j \in V, \sum_{k=0}^{m} x_{jk} = 1 \]  

(5)
duplication increases. Thus, when more threads are spawned, the task amount of more parallelism can be utilized by utilizing more duplication.

Further increase in out-degree of tasks slightly reduces 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.

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 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
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 \( \text{prec} \), and task duplication is denoted by \( \text{dup} \). The objective is to reduce the makespan or schedule length of the last task on mobile device \( T^w_N \). Thus, this problem is denoted by \( P(2, \infty)|\text{prec, dup}|\text{makespan} \). 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**


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


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