Hermes: Latency Optimal Task Assignment for Resource-constrained Mobile Computing

Abstract—**With mobile devices increasingly able to connect to cloud servers from anywhere, resource-constrained devices can potentially perform offloading of computational tasks to either improve resource usage or improve performance. It is of interest to find optimal assignments of tasks to local and remote devices that can take into account the application-specific profile, availability of computational resources, and link connectivity, and find a balance between energy consumption costs of mobile devices and latency for delay-sensitive applications. Given an application described by a task dependency graph, we formulate an optimization problem to minimize the latency while meeting prescribed resource utilization constraints. Different from most of existing works that either rely on an integer linear programming (ILP) formulation, which is NP-hard in general and is not applicable to general task dependency graph for latency metrics, or on intuitively derived heuristics that offer no theoretical performance guarantees, we propose Hermes, a novel fully polynomial time approximation scheme (FPTAS) algorithm to solve this problem. Hermes provides a solution with latency no more than $(1 + \epsilon)$ times of the minimum while incurring complexity that is polynomial in problem size and $\frac{1}{\epsilon}$. We evaluate the performance by using real data set collected from several benchmarks, and show that Hermes improves the latency by 16% (36% for larger scale application) compared to a previously published heuristic and incurs CPU computing time by only 0.4% of overall latency.**

I. Introduction

As more embedded devices are connected, lots of resource on the network, in the form of cloud computing, become accessible. These devices, either suffering from stringent battery usage, like mobile devices, or limited processing power, like sensors, are not capable to run computation-intensive tasks locally. Taking advantage of the remote resource, more sophisticated applications, requiring heavy loads of data processing and computation [1], [2], can be realized in timely fashion and acceptable performance. Thus, computation offloading—sending computation-intensive tasks to more resourceful severs, is becoming a potential approach to save resources on local devices and to shorten the processing time [3], [4], [5].

However, implementing offloading invokes extra communication cost due to the application and profiling data that must be exchanged with remote servers. The additional communication affects both energy consumption and latency [6]. Hence, it is not trivial to figure out the **best** offloading strategy considering the balance between how much the offloading saves and how much extra cost is induced. For example, Fig. 1 shows a task graph of an arbitrary application. A task is represented by a node whose weight specifies its complexity. Each edge shows the data dependency between two tasks, and is labelled with the amount of data being communicated between them. Suppose we have three devices; we want to find the best way to assign each task on the devices to minimize the overall latency considering reasonable energy consumption. Fig. 2 shows that the optimal strategy varies as we increase the complexity of the shaded task from 8 to 12. Due to the budget constraint on energy consumption, the varying complexity on a node affects the strategy on the whole branch that contains it. More importantly, the latency increases if we use the initially optimal strategy to serve the ones with increased complexity, while there exist optimal strategies for each case that can achieve the same minimum latency. In addition to considering a single remote server, which involves in only binary decision on each task, another spectrum of offloading schemes make use of other idle and connected devices in the network [7], where the decision is made over multiple devices considering their availabilities and multiple wireless channels. From the above observations, a rigorous **optimization formulation** of the problem and the **scalability** of corresponding algorithm are the key issues that need to be addressed.

In general, we are concerned in this domain with a task assignment problem over multiple devices, subject to constraints. Furthermore, task dependency must be taken into account in formulations involving latency as a metric. The authors of Odessa [10] present a heuristic approach to task partitioning for improving latency and throughput with mobile offloading, involving iterative improvement of bottlenecks in task execution and data transmission. However, this greedy heuristic provides no performance guarantees in both latency metrics and other constraints, as we show that it can be further improved by 36% in some cases. Of all optimization formulations, integer linear programming (ILP) is the most common formulation due to its flexibility and intuitive interpretation of the optimization problem. In the well-known MAUI work, Cuervo *et al.* [8] propose an ILP formulation with latency constraint of a serial tasks. However, the ILP formulations are generally NP-hard, that is, there is no polynomial-time algorithm to solve all instances of ILP unless P $=$ NP [11]. Moreover, the overall latency is not additive over tasks for general task dependency, which is often described by a directed acyclic graph (DAG). In addition to ILP, graph partitioning is another approach. Given a task graph, the minimum cut on weighted edges specifies the minimum communication cost and cuts the tasks into two disjoint sets, one is the set of tasks that are to be executed at the remote server and the other are ones that remain at the local device. Wang *et al.* [9] provide a polynomial time algorithm to solve for the best partitioning. However, the algorithm is not applicable to latency metrics.

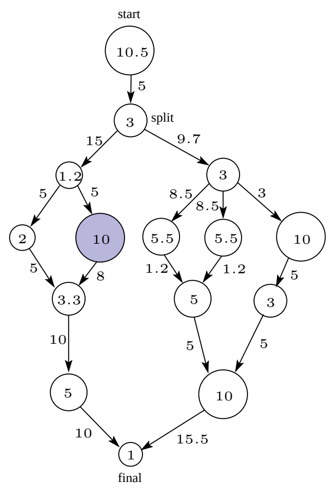

Fig. 1: A task graph of an application

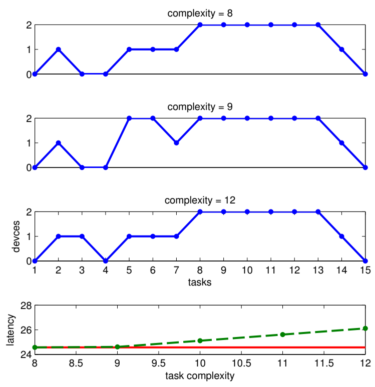

Fig. 2: The optimal strategy varies as the complexity of the shaded task in Fig. 1 increases. The latency will increase if we use sub-optimal strategy, while there exist optimal strategies for each case to achieve the same latency.

TABLE I: Comparison between existing works and Hermes

Existing Works	**MAUI** [8]	**min k-cut** [9]	**Odessa** [10]	**Hermes**
Task Dependency Graph	serial	DAG	general	subset of DAG
Objectives	energy consumption	communication cost	latency & throughput	latency
Constraints	latency	none	none	cost
Task Assignment	2 devices	multiple devices	2 devices	multiple devices
Algorithm Complexity	exponential	exponential	no guarantee	polynomial
Performance	optimal	optimal	no guarantee	near-optimal (ϵ-approximate)

Furthermore, for offloading across multiple devices, instead of making binary decision on each task, the minimum k-cut problem is NP-hard [12]. Compared with the existing formulations and algorithms, we formulate an optimization problem that aims to minimize the latency subject to a cost constraint. We propose Hermes[1], an algorithm that is a fully polynomial time approximation scheme (FPTAS). That is, the solution given by Hermes performs no more than $(1+\epsilon)$ times of the minimum objective, where ϵ is a positive number and the complexity is bounded by a polynomial in $\frac{1}{\epsilon}$ and the problem size [13]. Table I summarizes the comparison of our formulation and algorithm to the existing works. To our best knowledge, for this class of task assignment problems, Hermes applies to more sophisticated formulations than prior works and runs in polynomial time with problem size but still provides near-optimal solutions with performance guarantee. We list our main contributions as follows.

1) **A new formulation of task assignment considering both latency and resource cost:** Our formulation is practically useful for applications with a general task dependency described by a directed acyclic graph and allows for the minimization of total latency (makespan) subject to a resource cost constraint.
2) **Hermes: an FPTAS algorithm to solve the proposed formulation:** We prove that Hermes runs in $O(d_{in}NM^2\frac{l^2}{\epsilon})$ time and provides an $(1+\epsilon)$ approximation, where N is the number of tasks, M is the number of devices, d_{in} is the maximum indegree over all tasks and l is the length of the longest paths.
3) **Comparative performance evaluation:** We evaluate the performance of Hermes by using real data sets measured in several benchmarks to emulate the executions of these applications, and compare it to the previously-published Odessa scheme considered in [10]. The result shows that Hermes improves the latency by 16% (36% for larger scale application) compared to Odessa and incurs CPU computation time by only 0.4% of overall latency, which implies the latency gain of Hermes is significant enough to compensate its extra CPU load.

[1]Because of its focus on minimizing latency, Hermes is named for the Greek messenger of the gods with winged sandals, known for his speed.

II. Models and Notations

In this section, we formulate our optimization problem to solve for the optimal task assignment strategy.

A. Task Graph

An application profile can be described by a directed graph $G(\mathcal{V},\mathcal{E})$ as shown in Fig. 1, where nodes stand for tasks and directed edges stand for data dependencies. A task precedence constraint is described by an directed edge (m,n), which implies that task n relies on the result of task m. That is, task n cannot start until it gets the result of task m. The weight on each node specifies the measure of complexity of the task, while the weight on each edge shows the amount of data communication between two tasks. In addition to the application profile, there are some parameters related to the

TABLE II: Notations

Notation	Description
m_i	task complexity of task i
d_i	the amount of data generated by task i
r_j	CPU rate of device j
$G(\mathcal{V}, \mathcal{E})$	task graph with nodes (*resp.* edges) described by set $\mathcal{V}$ (*resp.* $\mathcal{E}$)
$\mathcal{C}(i)$	set of children of node i
l	the depth of task graph (the longest path)
d_{in}	the maximum indegree of task graph
δ	quantization step size
$\mathbf{x} \in [M]^N$	assignment strategy of tasks $1 \cdots N$
$T_{ex}^{(j)}(i)$	latency of executing task i on device j
$T_{tx}^{(jk)}(d)$	latency of transmitting d units of data from device j to k
$C_{ex}^{(j)}(i)$	cost of executing task i on device j
$C_{tx}^{(jk)}(d)$	cost of transmitting d units of data from device j to k
$D^{(i)}$	accumulated latency when task i finishes

graph measure in our complexity analysis. We use N to denote the number of tasks and M to denote the number of devices. For each task graph, there is an initial task (task 1) that starts the application and a final task (task N) that terminates it. A path from initial task to final task can be described by a serial of nodes, where every pair of consecutive nodes are connected by a directed edge. We use l to denote the maximum number of nodes in a path, i.e., the length of the longest path. Finally, d_{in} denotes the maximum indegree in the task graph.

B. Cost and Latency

We use the general cost and latency functions in our derivation. Let $C_{ex}^{(j)}(i)$ be the execution cost of task i on device j and $C_{tx}^{(jk)}(d)$ be the transmission cost of d units of data from device j to device k. Similarly, the latency consists of execution latency $T_{ex}^{(j)}(i)$ and the transmission latency $T_{tx}^{(jk)}(d)$. Given a task assignment strategy $\mathbf{x} \in \{1 \cdots M\}^N$, where the i^{th} component, x_i, specifies the device that task i is assigned to, the total cost can be described as follows.

$$Cost = \sum_{i=1}^{N} C_{ex}^{(x_i)}(i) + \sum_{(m,n)\in\mathcal{E}} C_{tx}^{(x_m x_n)}(d_m) \tag{1}$$

As described in the equation, the total cost is additive over nodes (tasks) and edges of the graph. On the other hand, the accumulated latency up to task i depends on its preceding tasks. Let $D^{(i)}$ be the latency when task i finishes, which can be recursively defined as

$$D^{(i)} = \max_{m\in\mathcal{C}(i)} \left\{ T_{tx}^{(x_m x_i)}(d_m) + D^{(m)} \right\} + T_{ex}^{(x_i)}(i). \tag{2}$$

We use $\mathcal{C}(i)$ to denote the set of children of node i. For example, in Fig. 3, the children of task 6 are task 4 and task 5. For each child node m, the latency is accumulating as the latency up to task m plus the latency caused by transmission the result d_m to task i. Hence, $D^{(i)}$ is determined by the slowest branch.

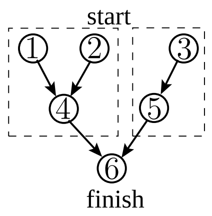

Fig. 3: A tree-structured task graph, in which the two sub-problems can be independently solved.

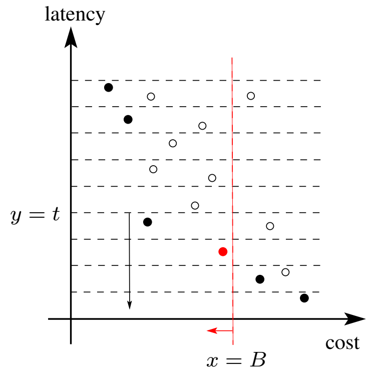

Fig. 4: The algorithm solves each sub-problem for the minimum cost within latency constraint t (the area under the horizontal line $y = t$). The filled circles are the optimums of each sub-problems. Finally, it looks for the one that has the minimum latency of all filled circles in the left plane $x \leq B$.

C. Optimization Problem

Consider an application, described by a task graph, and a resource network, described by the processing powers and link connectivity between available devices, our goal is to find a task assignment strategy $\mathbf{x}$ that minimizes the total latency and satisfies the cost constraint, that is,

$$\min_{\mathbf{x}\in[M]^N} D^{(N)}$$
$$\textbf{s.t. } Cost \leq B. \tag{3}$$

The $Cost$ and $D^{(N)}$ are defined in Eq. (1) and Eq. (2), respectively. In the following section, we propose an approximation algorithm based on dynamic programming to solve this problem and show that it runs in polynomial time in $\frac{1}{\epsilon}$ with approximation ratio $(1 + \epsilon)$.

III. Hermes: FPTAS Algorithms

We first propose the approximation scheme to solve the optimization problem for a tree-structure task graph and prove that this simplest version of the Hermes algorithm is an FPTAS. Then we solve for more general task graphs by calling the proposed algorithm for trees a polynomial number of times. Finally, we show that the Hermes algorithm also applies to the stochastic optimization problem, minimizing the expected latency subject to expected cost constraint.

Algorithm 1 Hermes FPTAS for tree-structured task graph

```
1: procedure $FPTAS_{tree}(N)$    ▷ min. cost when task $N$ finishes at devices $1, \cdots, M$ within latencies $1, \cdots, K$
2:     $q \leftarrow$ BFS $(G, N)$    ▷ run Breadth First Search of $G$ from node $N$ and store visited nodes in order in $q$
3:     for $i \leftarrow q$.end, $q$.start do    ▷ start from the last element in $q$
4:         if $i$ is a leaf then    ▷ initialize $C$ values of leaves
5:             $C[i,j,k] \leftarrow \begin{cases} C_{ex}^{(j)}(i) & \forall k \geq q_\delta\left(T_{ex}^{(j)}(i)\right) \\ \infty & \text{otherwise} \end{cases}$
6:         else
7:             for $j \leftarrow 1, M$, $k \leftarrow 1, K$ do
8:                 Calculate $C[i,j,k]$ from Eq. (5)
9: end procedure
```

A. Tree-structured Task Graph

We propose an dynamic programming method to solve the problem with tree-structured task graph. For example, in Fig. 3, the minimum latency when the task 6 finishes depends on when and where task 4 and 5 finish. Hence, prior to solving the minimum latency of task 6, we want to solve both task 4 and 5 first. We exploit the fact that the sub-trees rooted by task 4 and task 5 are independent. That is, the assignment strategy on task 1, 2 and 4 does not affect the strategy on task 3 and 5. Hence, we can solve the sub-problems respectively and combine them when considering task 6.

We define the sub-problem as follows. Let $C[i,j,t]$ denote the minimum cost when finish task i on device j within latency t. We will show that by solving all of the sub-problems for $i \in \{1, \cdots, N\}$, $j \in \{1, \cdots, M\}$ and $t \in [0, T]$ with sufficiently large T, the optimal strategy can be solved by combining the solutions of these sub-problems. Fig. 4 shows our methodology. Each circle marks the performance given by an assignment strategy, with x-component as cost and y-component as latency. Our goal is to find out the red circle, that is, the strategy that causes minimum latency and satisfies the cost constraint. Under each horizontal line $y = t$, we first identify the circle with minimum x-component, which specifies the least-cost strategy among all of strategies that causes latency at most t. These solutions are denoted by the filled circles. In the end, we look at the one in the left plane ($x \leq B$) whose latency is the minimum.

Instead of solving infinite number of sub-problems for all $t \in [0, T]$, we discretize the time domain. We define a uniform quantization function as

$$q_\delta(x) = k, \text{ if } (k-1)\delta < x \leq k\delta. \tag{4}$$

It suffices to solve all the sub-problems for $k \in \{1, \cdots, K\}$, where $K = \lceil \frac{T}{\delta} \rceil$. We will analyze how the performance is affected due to the lost of precision by doing quantization and the trade-off with complexity after we present our algorithm. Suppose we are solving the sub-problem $C[i,j,k]$, given that all of the preceding tasks have been solved, the recursive relation can be described as follows.

$$C[i,j,k] = C_{ex}^{(j)}(i) + \min_{x_m : m \in \mathcal{C}(i)} \{ \sum_{m \in \mathcal{C}(i)} C[m, x_m, k - k_m] + C_{tx}^{(x_m j)}(d_m) \},$$

$$k_m = q_\delta\big(T_{ex}^{(j)}(i) + T_{tx}^{(x_m j)}(d_m)\big).$$

That is, to find out the minimum cost within latency k at task i, we trace back to its child tasks and find out the minimum cost over all possible strategies, with the latency that excludes the execution delay of task i and data transmission delay. As the cost function is additive over tasks and the decisions on each child task is independent with each other, we can further lower down the solution space from M^z to zM, where z is the number of child tasks of task i. That is, by making decisions on each child task independently, we have

$$C[i,j,k] = C_{ex}^{(j)}(i) + \sum_{m \in \mathcal{C}(i)} \min_{x_m \in [M]} \{ C[m, x_m, k - k_m] + C_{tx}^{(x_m j)}(d_m) \}. \tag{5}$$

After solving all the sub-problems $C[i,j,k]$, we solve for the optimal strategy by performing the combining step as follows.

$$\min k \ \text{ s.t. } C[N,j,k] \leq B.$$

As the application should always terminate at the local machine, it is reasonable to fix j and solve for the minimum k. We summarize our algorithm for tree-structure task graph in Algorithm 1. Since the algorithm will be used as a basic block function for more general task graph, we neglect the combining step and simply focus on solving $C[i,j,k]$.

Theorem 1. *Algorithm 1 runs in $O(d_{in} N M^2 \frac{l^2}{\epsilon})$ time and admits an $(1+\epsilon)$ approximation ratio.*

Proof. From Algorithm 1, to find $C[N,j,k]$ needs to solve NMK sub-problems, where K depends on the maximum dynamic range of the latency. Given a task i with complexity measure m_i, if it is executed at device j, the execution delay can be expressed as

$$T_{ex}^{(j)}(i) = c\frac{m_i}{r_j},$$

where r_j is the CPU rate of device j and c is a constant. The largest single stage delay is determined by the slowest device executing the most intensive task. That is,

$$T_{max} = c\frac{m_{max}}{r_{min}}.$$

Hence, the maximum latency over all assignment strategies can be bounded by lT_{max}, where l is the longest branch of the tree. If we set $\delta = \frac{\epsilon T_{max}}{l}$, then

Algorithm 2 Hermes FPTAS for serial trees

```
1: procedure $FPTAS_{path}(N)$    ▷ min. cost when task $N$ finishes at devices $1,\cdots,M$ within latencies $1,\cdots,K$
2:     for root $i_l, l \in \{1,\cdots,n\}$ do    ▷ solve the conditional sub-problem for every tree
3:         for $j \leftarrow 1, M$ do
4:             Call $FPTAS_{tree}(i_l)$ conditioning on $j$ with modification described in Eq. (8)
5:     for $l \leftarrow 2, n$ do
6:         Perform combining step in Eq. (9) to solve $C[i_l, j_l, k_l]$
7: end procedure
```

$$K = \lceil \frac{lT_{max}}{\delta} \rceil = O(\frac{l^2}{\epsilon}).$$

Let d_{in} denote the maximum indegree of the task graph. For solving each sub-problem in Eq. (5), there are at most d_{in} minimization problems over M devices. Hence, the overall complexity is

$$O(NMK \times d_{in}M) = O(d_{in}NM^2\frac{l^2}{\epsilon}).$$

Since both the depth and the maximum indegree of a tree can be bounded by a polynomial of N, Algorithm 1 runs in polynomial time of problem size and $\frac{1}{\epsilon}$.

For a given strategy $\mathbf{x}$, let $\hat{L}(\mathbf{x})$ denote the quantized latency and $L(\mathbf{x})$ denote the original one. Further, let $\tilde{\mathbf{x}}$ denote the strategy given by Algorithm 1 and $\mathbf{x}^*$ denote the optimal strategy. As $\tilde{\mathbf{x}}$ is the strategy with minimum latency solved by Algorithm 1, we have $\hat{L}(\tilde{\mathbf{x}}) \leq \hat{L}(\mathbf{x}^*)$. For a task graph with depth l, only at most l quantization procedures are taken. By the quantization defined in Eq. (4), it always over estimates by at most δ. Hence, we have

$$L(\tilde{\mathbf{x}}) \leq \delta\hat{L}(\tilde{\mathbf{x}}) \leq \delta\hat{L}(\mathbf{x}^*) \leq L(\mathbf{x}^*) + l\delta \quad (6)$$

Let $T_{min} = c\frac{m_{max}}{r_{max}}$, that is, the latency when the most intensive task is executed at the fastest device. As the most intensive task must be assigned to a device, the optimal latency, $L(\mathbf{x}^*)$, is at least T_{min}. From Eq. (6), we have

$$L(\tilde{\mathbf{x}}) \leq L(\mathbf{x}^*) + l\delta = L(\mathbf{x}^*) + \epsilon T_{max} \leq (1+\epsilon\frac{r_{max}}{r_{min}})L(\mathbf{x}^*). \quad (7)$$

For realistic resource network, the ratio of the fastest CPU rate and the slowest CPU rate is bounded by a constant c'. Let $\epsilon' = \frac{1}{c'}\epsilon$, then the overall complexity is still bounded by $O(d_{in}NM^2\frac{l^2}{\epsilon})$ and Algorithm 1 admits an $(1+\epsilon)$ approximation ratio. Hence, Algorithm 1 is an FPTAS. □

As chain is a special case of a tree, the Hermes FPTAS Algorithm 1 also applies to the task assignment problem of serial tasks. Instead of using the ILP solver to solve the formulation for serial tasks proposed previously in [8], we have therefore provided an FPTAS to solve it.

B. Serial Trees

Most applications start from an unique initial task, then split to multiple parallel tasks and finally, all the tasks are merged into one final task. Hence, the task graph is neither a chain nor a tree. In this section, we show that by calling Algorithm 1 in polynomial number of times, Hermes can solve the task graph that consists of serial of trees.

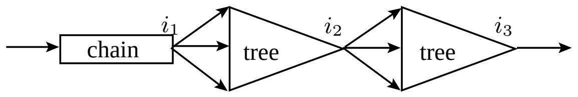

Fig. 5: A task graph of serial trees

The task graph in Fig. 5 can be decomposed into 3 trees connecting serially, where the first tree (chain) terminates in task i_1, the second tree terminates in task i_2. In order to find $C[i_3, j_3, k_3]$, we independently solve for every tree, with the condition on where the root task of the former tree ends. For example, we can solve $C[i_2, j_2, k_2|j_1]$, which is the strategy that minimizes the cost in which task i_2 ends at j_2 within delay k_2 and given task i_1 ends at j_1. Algorithm 1 can solve this sub-problem with the following modification for the leaves.

$$C[i,j,k|j_1] = \begin{cases} C_{ex}^{(j)}(i) + C_{tx}^{(j_1 j)}(d_{i_1}) & \forall k \geq q_\delta(T_{ex}^{(j)}(i) + T_{tx}^{(j_1 j)}(d_{i_1})), \\ \infty & \text{otherwise} \end{cases} \quad (8)$$

To solve $C[i_2, j_2, k_2]$, the minimum cost up to task i_2, we perform the combining step as

$$C[i_2, j_2, k_2] = \min_{j\in[M]} \min_{k_x+k_y=k_2} C[i_1, j, k_x] + C[i_2, j_2, k_y|j]. \quad (9)$$

Similarly, by combining $C[i_2, j_2, k_x]$ and $C[i_3, j_3, k_y|j_2]$ gives $C[i_3, j_3, k_3]$. Algorithm 2 summarizes the steps in solving the assignment strategy for serial trees. To solve each tree involves M calls on different conditions. Further, the number of trees n can be bounded by N. The latency of each tree is within $(1+\epsilon)$ optimal, which leads to the $(1+\epsilon)$ approximation of total latency. Hence, Algorithm 2 is also an FPTAS.

C. Parallel Chains of Trees

We take a step further to extend Hermes for more complicated task graphs that can be viewed as parallel chains of trees, as shown in Fig. 1. Our approach is to solve each chains by calling $FPTAS_{path}$ with the condition on the task where they split. For example, in Fig. 1 there are two chains that can be solved independently by conditioning on the split node. The combining procedure consists of two steps. First, solve $C[N, j, k|j_{split}]$ by Eq. (5) conditioned on the split node. Then $C[N, j, k]$ can be solved similarly by combining two serial blocks in Eq. (9). By calling $FPTAS_{path}$ at most d_{in} times, this proposed algorithm is also an FPTAS.

D. Stochastic Optimization

The dynamic resource network, where server availabilities and link qualities are changing, makes the optimal assignment strategy vary with time. For Hermes, which solves the optimal strategy based on the profiling data, it is reasonable to formulate a stochastic optimization problem of minimizing the expected latency subject to expected cost constraint. If both latency and cost metrics are additive over tasks, we can directly apply the deterministic analysis to the stochastic one by assuming that the profiling data is the 1st order expectations. However, it is not clear if we could apply our deterministic analysis for parallel computing as the latency metric is nonlinear. For example, for two random variables X and Y, $\mathbb{E}\{\max(X,Y)\} = \max(\mathbb{E}\{X\},\mathbb{E}\{Y\})$ is in general not true. In the following, we exploit the fact that the latency of a single branch is still additive over tasks and show that our deterministic analysis can be directly applied to the stochastic optimization problem.

Let $\bar{C}[i,j,k]$ be the minimum expected cost when task i finishes on device j within expected delay k. It suffices to show that the recursive relation in Eq. (5) still holds for expected values. As the cost is additive over tasks, we have

$$\bar{C}[i,j,k] = \mathbb{E}\{C_{ex}^{(j)}(i)\} + \sum_{m\in\mathcal{C}(i)} \min_{x_m\in[M]} \{\bar{C}[m,x_m,k-\bar{k}_m] + \mathbb{E}\{C_{tx}^{(x_m j)}(d_m)\}\}.$$

The $\bar{k}_m$ specifies the sum of expected data transmission delay and expected task execution delay. That is,

$$\bar{k}_m = q_\delta\left(\mathbb{E}\{T_{ex}^{(j)}(i) + T_{tx}^{(x_m j)}(d_m)\}\right).$$

Based on the fact that Hermes is tractable with respect to both the application size (N) and the network size (M), we propose an update scheme that is adaptive to dynamic resource network. The strategy is updated every period of time, which aims to minimize the expected latency in the following coherence time period. We will show how the proposed scheme adapts to the changes of network condition in Section IV.

E. More General Task Graph

The Hermes algorithm in fact can be applied to even more general graphs, albeit with weaker guarantees. In this section, we outline a general approach based on identifying the "split nodes" in a task graph — nodes in the task graph with more than one outgoing edge. From the three categories of task graph we have considered so far, each split node is only involved in the local decision of two trees. That is, in the combining stage shown in Eq. (9), there is only one variable on the node that connects two serial trees. Hence, the decision of this device can be made locally. Our general approach is to decompose the task graph into chains of trees and call the polynomial time procedure $FPTAS_{path}$ to solve each of them. If a split node connects two trees from different chains, then we cannot resolve this condition variable and have to keep it until we make the decision on the node where all of involved chains merge. We use the task graph in Fig. 1 to show an example: as the node (marked with split) splits over two chains, we have to keep it until we make decisions on the final task, where two chains merge. On the other hand, there are some nodes that split locally, which can be resolved in the $FPTAS_{path}$ procedure. A node that splits across two different chains requires $O(M)$ calls of the $FPTAS_{path}$. Hence, the overall complexity of Hermes in such graphs would be $O(M^S)$, where S is the number of "global" split nodes. If the task graph contains cycles, similar argument can be made as we classify them into local cycles and global cycles. A cycle is local if all of its nodes are contained in the same chain of trees and is global otherwise. For a local cycle, we solve the block that contains it and make conditions on the node with the edge that enters it and the node with the edge that leaves it. However, if the cycle is global, more conditions have to be made on the global split node and hence the complexity is not bounded by a polynomial.

The structure of a task graph depends on the granularity of partition. If an application is partitioned into methods, many recursive loops are involved. If an application is partitioned into tasks, which is a block of code that consists of multiple methods, the structure is simpler. As we show in the following, Hermes can tractably handle practical applications whose graph structures are similar to benchmarks in [10].

IV. Evaluation of Hermes

First, we verify that indeed Hermes provides near-optimal solution with tractable complexity and performance guarantee. Then, we measure the CPU time for Hermes to solve the optimal strategy as problem size scales. Finally, we use the real data set of several benchmark profiles to evaluate the performance of Hermes and compare it with the heuristic Odessa approach proposed in [10].

A. Algorithm Performance

From our analysis in Section III, the Hermes algorithm runs in $O(d_{in}NM^2\frac{l^2}{\epsilon})$ time with approximation ratio $(1+\epsilon)$. In the following, we provide the numerical results to show the trade-off between the complexity and the accuracy. Given the task graph shown in Fig. 1, the performance of Hermes versus different values of ϵ is shown in Fig. 6. When $\epsilon = 0.4$ ($K = 98$), the performance converges to the minimum latency. Fig. 6 also shows the bound of worst case performance in dashed line. For our simulation profile, $\frac{r_{max}}{r_{min}} = 4$, the actual performance is much better than the worst case bound in Eq. 6. Fig. 7 shows how the solution converges to the optimal strategy. Although the strategy is slightly different from the optimal one, it provides the same minimum latency. In order to save the budget, we can choose the one with lower cost.

Fig. 8 shows the performance of Hermes on 200 samples of application profiles. Each sample is driven independently and uniformly from the application pool with different task complexities and data communications. The result shows that for every sample, the performance is much better than the worst case bound and converges to the optimum, that is, the approximation ratio converges to 1. On the other hand, if we

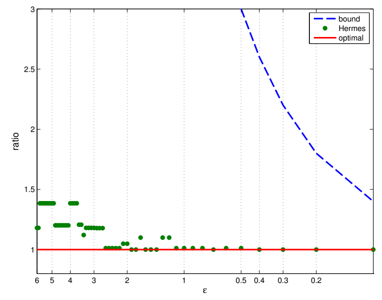

Fig. 6: The evaluation result shows that Hermes performs much better than the worst case bound. When $\epsilon = 0.4$, the objective value has converged to the minimum.

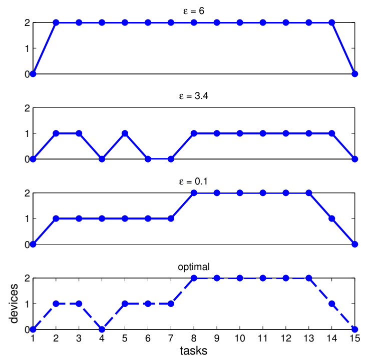

Fig. 7: The optimal strategy converges as ϵ is decreasing

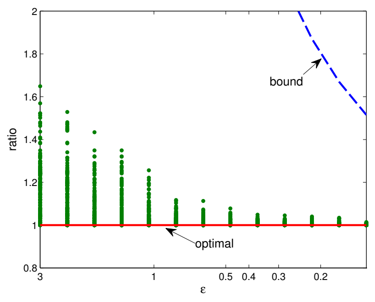

Fig. 8: The performance of Hermes over 200 samples of application profiles

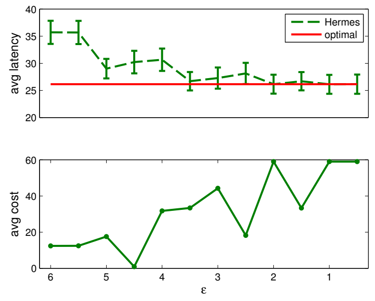

Fig. 9: The expected latency and cost over 10000 samples of resource network

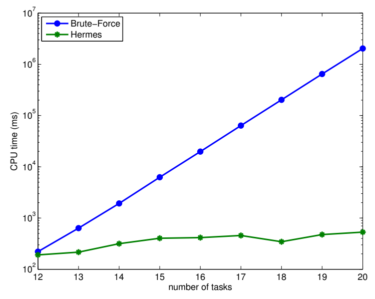

Fig. 10: The CPU time overhead for Hermes as the problem size scales ($\epsilon = 0.01$)

fix the application profile and simulate the performance of Hermes under dynamic resource network, Fig. 9 shows that the solution converges to the optimal one, which minimizes the expected latency and satisfies the expected cost constraint.

B. CPU Time Evaluation

Fig. 10 shows the CPU time for Hermes to solve for the optimal strategy as the problem size scales. We use Apple macbook Pro equipped with 2.4GHz dual-core Intel Core i5 processor and 3MB cache as our testbed and use java management package for CPU time measurement. As the number of tasks (N) increases in a serial task graph, the CPU time needed for the Brute-Force algorithm grows exponentially, while Hermes scales well and still provides the near-optimal solution ($\epsilon = 0.01$). From our complexity analysis, for serial task graph $l = N$, $d_{in} = 1$ and we fix $M = 3$, the CPU time of Hermes can be bounded by $O(N^3)$.

C. Benchmark Evaluation

In [10], Ra *et al.* present several benchmarks of perception applications for mobile device and propose a heuristic approach, called Odessa, to improve both makespan and throughput with the help of a cloud connected server. They call each edge and node in the task graph as stages and record the timestamps on every stages. To improve the performance, for each data frame, Odessa first identifies the bottleneck, evaluates each strategy with simple metrics and finally select the potentially best one to mitigate the load on the bottleneck. However, this greedy heuristic does not offer any theoretical performance guarantee, as shown in Fig. 11 Hermes can improve the performance by 36% for task graph in Fig. 1. Hence, we further choose two of benchmarks, face recognition and pose recognition, to compare the performance between Hermes and Odessa. Taking the timestamps of every stage and the corresponding statistics measured in real executions provided in [10], we emulate the executions of these benchmarks and evaluate the performance.

In dynamic resource scenarios, as Hermes' complexity is not as light as the greedy heuristic (86.87 ms in average) and its near-optimal strategy needs not be updated from frame to frame under similar resource conditions, we propose the following on-line update policy: similar to Odessa, we record the timestamps for on-line profiling. Whenever the latency difference of current frame and last frame goes beyond the threshold, we run Hermes based on current profiling to update the strategy. By doing so, Hermes always gives the near-optimal strategy for current resource scenario and enhances the performance at the cost of reasonable CPU time overhead due to resolving the strategy.

As Hermes provides better performance in latency but larger CPU time overhead when updating, we define two metrics for comparison. Let $Latency(t)$ be the normalized latency advantage of Hermes over Odessa up to frame number t. On the other hand, let $CPU(t)$ be the normalized CPU advantage of Odessa over Hermes up to frame number t. That is,

$$Latency(t) = \frac{1}{t} \sum_{i=1}^{t} \big(L_O(i) - L_H(i) \big), \tag{10}$$

$$CPU(t) = \frac{1}{t} \Big(\sum_{i=1}^{C(t)} CPU_H(i) - \sum_{i=1}^{t} CPU_O(i) \Big), \tag{11}$$

where $L_O(i)$ and $CPU_O(i)$ are latency and update time of frame i given by Odessa, and the notations for Hermes are similar except that we use $C(t)$ to denote the number of times that Hermes updates the strategy up to frame t.

To model the dynamic resource network, the latency of each stage is driven independently and uniformly from a distribution with its mean and standard deviation provided by the statistics of the data set measured in real applications. In addition to small scale variation, the link coherence time is 20 data frames. That is, for some period, the link quality degrades significantly due to possible fading situations. Fig. 12 shows the performance of Hermes and Odessa for the face recognition application. Hermes improves the average latency of each data frame by 10% compared to Odessa and incurs CPU computing time by only 0.3% of overall latency. That is, the latency advantage provided by Hermes well-compensates its CPU time overhead. Fig. 13 shows that Hermes improves the average latency of each data frame by 16% for pose recognition application and incurs CPU computing time by 0.4% of overall latency. When the link quality is degrading, Hermes updates the strategy to reduce the data communication, while Odessa's sub-optimal strategy results in significant extra latency. Considering CPU processing speed is increasing under Moore's law but network condition does not change that fast, Hermes provides a promising approach to trade-in more CPU for less network consumption cost.

V. Related Work on Machine Scheduling

To our knowledge, no prior work has studied the polynomial time algorithm to solve the task assignment problem on multiple devices, taking both latency and costs into account. The linear integer programming is a common formulation, however, it is NP-hard to solve in general.

In this section, we compare our task assignment problem with the machine scheduling problems, which are mostly NP-hard. Theoretically, researchers have become interested in better understanding about their approximability. That is, the existence of polynomial time approximation algorithm and corresponding approximate ratio. In [14], Schuurman *et al.* study the approximation algorithms and propose ten open problems. The most related category to our problem, called the makespan minimization problem, is defined as

Definition 1. *Given a set of machines $\mathcal{M} = \{1, \cdots, m\}$ and a set of jobs $\mathcal{J} = \{1, \cdots, n\}$, and p_{ij} specifying the processing time for job j being executed at machine i, the goal is to assign each job to one of machines such that the makespan is minimized. That is,*

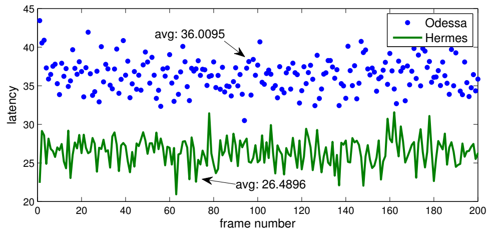

Fig. 11: Hermes can improve the performance by 36% compared to Odessa for task graph shown in Fig. 1.

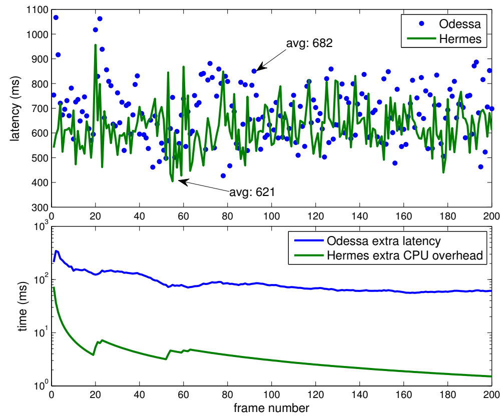

Fig. 12: Top: Hermes improves the average latency of each data frame by 10%. Bottom: the latency advantage of Hermes over Odessa ($Latency(t)$) is significant enough to compensate its CPU time overhead ($CPU(t)$) of solving the strategy.

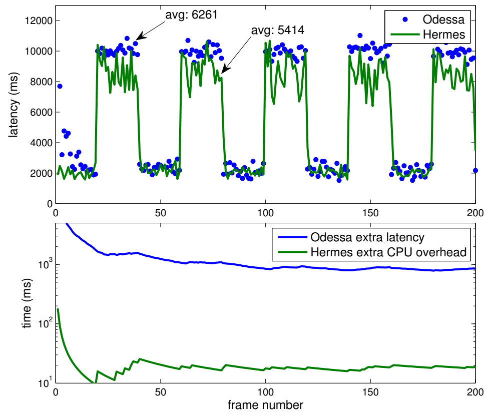

Fig. 13: Hermes improves the average latency of each data frame by 16% and well-compensates its CPU time overhead.

$$\min \max_{i \in \mathcal{M}} \sum_{j \in \mathcal{S}_i} p_{ij}$$
$$s.t. \ \cup_{i \in \mathcal{M}} \mathcal{S}_i = \mathcal{J}.$$

If the job processing time does not depend on machines, that is, $p_{ij} = p_j$ for all ij, then the problem is called the makespan minimization on *identical* machines. On the other hand, if

$p_{ij} = \frac{p_j}{s_i}$, where s_i is the speed metric of the i^{th} machine, then it is the scheduling problem on *uniform* machines. The devices in our model are uniform machines. The precedence constraints make the problem more interesting and harder to solve. In general, the task precedence can be described by a DAG, which is equivalent to our task graph. In the following, we use the standard three-field notation $P|prec|C_{max}$ to denote the problem of makespan minimization on identical machines under precedence constraints. Similarly, $Q|prec|C_{max}$ denotes the case for uniform machines.

There are some positive and negative results. We only list the most related results to our problem and direct the readers to more relevant literature [14], [15]. The negative results disprove the existence of the polynomial time approximation algorithm under the assumption P $\neq$ NP. For example, a strongly NP-hard problem remains NP-hard even if the numbers in its input are unary encoded. Garey *et al.* [16] show that if P $\neq$ NP, a strongly NP-hard problem cannot have an FPTAS. For simpler problem where the precedence constraints are chains, Du *et al.* [17] show that $Q|chain|C_{max}$ is strongly NP-hard. On the positive side, Graham *et al.* [18] design the list scheduling algorithm for $Q|prec|C_{max}$ and show that it is a $(2 - \frac{1}{m})$ approximation algorithm. There also have been some improved results recently [19], [20].

We further compare our task assignment problem with a specific category of machine scheduling problems, $Q|prec|C_{max}$. From complexity respective point of view, the number of quantization levels, $K = \frac{lT_{max}}{\delta}$, can be encoded in polynomially many unary bits, hence, the overall complexity $O(d_{in}NM^2K)$ implies that our problem is not strongly NP-hard. On the other hand, our task graph is a subset of general DAG, in which the application starts from an initial task and terminates at a single task. Moreover, we do not consider the maximum number of available processors on each device. That is, the number of tasks being executing at the same time must not exceed the number of processors on the device. If our offloading strategy leads to this situation, some of tasks must have to be queued, which results in longer latency. Considering the modern mobile devices have up to 8 cores of processing cores [21] and is applicable to multi-threading computing, and the observation that the task graphs are in general more chain-structured with narrow width, we argue that Hermes is tractable and applicable to real-world applications.

VI. Conclusions

We have formulated a task assignment problem for computational offloading and provided a FPTAS algorithm, Hermes, to solve for the optimal strategy that makes the balance between latency improvement and energy consumption of mobile devices. Compared with previous formulations and algorithms, to our best knowledge, Hermes is the first polynomial time algorithm to address the latency-resource tradeoff problem with provable performance guarantee. Moreover, Hermes is applicable to more sophisticated formulations on the latency metrics considering more general task dependency constraints as well as multi-device scenarios. The CPU time measurement shows that Hermes scales well with problem size. We have further emulated the application execution by using the real data set measured in several mobile benchmarks, and shown that our proposed on-line update policy, integrating with Hermes, is adaptive to dynamic network change. Furthermore, the strategy suggested by Hermes performs much better than greedy heuristic so that the CPU overhead of Hermes is well compensated. Existing works have been using pipelining techniques to improve both makespan and system throughput. It would further improve the performance if we can extend Hermes to also make decisions on pipelining strategies.

References

[1] M. Kolsch *et al.*, *Vision based hand gesture interfaces for wearable computing and virtual environments*. University of California, Santa Barbara, 2004.

[2] E. Miluzzo, T. Wang, and A. T. Campbell, "Eyephone: activating mobile phones with your eyes," in *ACM SIGCOMM*. ACM, 2010, pp. 15–20.

[3] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[4] D. Shivarudrappa, M. Chen, and S. Bharadwaj, "Cofa: Automatic and dynamic code offload for android," *University of Colorado, Boulder*.

[5] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *ACM Computer systems*. ACM, 2011, pp. 301–314.

[6] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *IEEE INFOCOM*. IEEE, 2013, pp. 1285–1293.

[7] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura, "Serendipity: enabling remote computing among intermittently connected mobile devices," in *ACM MobiHoc*. ACM, 2012, pp. 145–154.

[8] 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 *ACM MobiSys*. ACM, 2010, pp. 49–62.

[9] C. Wang and Z. Li, "Parametric analysis for adaptive computation offloading," *ACM SIGPLAN*, vol. 39, no. 6, pp. 119–130, 2004.

[10] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *ACM MobiSys*. ACM, 2011, pp. 43–56.

[11] G. L. Nemhauser and L. A. Wolsey, *Integer and combinatorial optimization*. Wiley New York, 1988, vol. 18.

[12] O. Goldschmidt and D. S. Hochbaum, "A polynomial algorithm for the k-cut problem for fixed k," *Mathematics of operations research*, vol. 19, no. 1, pp. 24–37, 1994.

[13] G. Ausiello, *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer, 1999.

[14] P. Schuurman and G. J. Woeginger, "Polynomial time approximation algorithms for machine scheduling: Ten open problems," *Journal of Scheduling*, vol. 2, no. 5, pp. 203–213, 1999.

[15] K. Jansen and R. Solis-Oba, "Approximation algorithms for scheduling jobs with chain precedence constraints," in *Parallel Processing and Applied Mathematics*. Springer, 2004, pp. 105–112.

[16] M. R. Garey and D. S. Johnson, "“strong”np-completeness results: Motivation, examples, and implications," *Journal of the ACM (JACM)*, vol. 25, no. 3, pp. 499–508, 1978.

[17] J. Du, J. Y. Leung, and G. H. Young, "Scheduling chain-structured tasks to minimize makespan and mean flow time," *Information and Computation*, vol. 92, no. 2, pp. 219–236, 1991.

[18] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[19] M. Kunde, "Nonpreemptive lp-scheduling on homogeneous multiprocessor systems," *SIAM Journal on Computing*, vol. 10, no. 1, pp. 151–173, 1981.

[20] D. Gangal and A. Ranade, "Precedence constrained scheduling in optimal," *Journal of Computer and System Sciences*, vol. 74, no. 7, pp. 1139–1146, 2008.

[21] S. Knight, "Mediatek's new octa-core processor will be powered by the new arm cortex-a17," February 2014, online; accessed 3-March-2014.