Techniques for fine-grained, multi-site computation offloading

Kanad Sinha and Milind Kulkarni
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN, USA
{*sinhak, milind*}*@purdue.edu*

***Abstract*—Increasingly, mobile devices are becoming the preferred platform for computation for many users. Unfortunately, the resource limitations, in battery life, computation power and storage, restricts the richness of applications that can be run on such devices. To alleviate these concerns, a popular approach that has gained currency in recent years is *computation offloading*, where a portion of an application is run off-site, leveraging the far greater resources of the cloud.**

Prior work in this area has focused on a constrained form of the problem: a single mobile device offloading computation to a single server. However, with the increased popularity of cloud computing and storage, it is more common for the data that an application accesses to be distributed among several servers. This paper describes algorithmic approaches for performing fine-grained, multi-site offloading. This allows portions of an application to be offloaded in a data-centric manner, even if that data exists at multiple sites. Our approach is based on a novel partitioning algorithm, and a novel data representation. We demonstrate that our partitioning algorithm outperforms existing multi-site offloading algorithms, and that our data representation provides for more efficient, fine-grained offloading than prior approaches.

***Keywords*-Computation offloading; partitioning algorithms; program representations**

I. Introduction

One of the consistent trends of the 21st century is the move to mobile devices as a primary computing platform for many users: laptops have replaced desktops, netbooks have replaced laptops, tablet PCs have replaced netbooks, and, increasingly, smartphones are replacing other devices. Concurrent with that is an increased demand for rich applications on such mobile devices (with the fast growth of "app stores" for iOS devices and Android reflecting this trend). Unfortunately, the desire for rich, powerful applications on mobile devices conflicts with the reality of these devices limitations: slow processors, little storage and, most importantly, limited battery life.

One emerging approach to dealing with the resource limitations of mobile devices is to leverage *computation offloading*, where parts of an application's execution are performed on a remote server, with results communicated back to the local device [6, 8, 10, 11, 14, 15]. By choosing which modules of a computation to offload carefully, the effects of limited resources on mobile devices can be ameliorated. For example, offloading computationally intensive modules to a server with powerful processors, battery life on the mobile device can be extended. Alternately, offloading data-intensive portions of an application can allow data to be stored in the cloud, saving storage space.

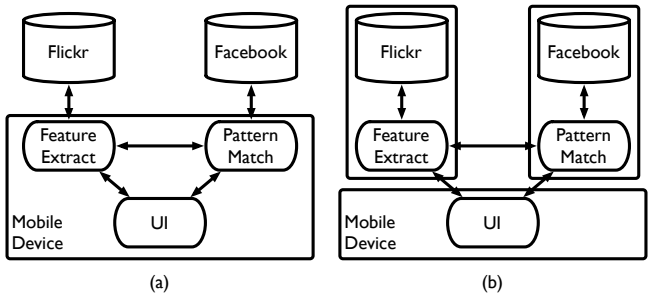


Figure 1. (a) Monolithic mobile application. (b) Mobile application after offloading computationally intensive modules to multiple sites.

Because writing cloud-enabled applications is difficult, an alternate approach is to write a monolithic application and then *automatically partition* it between the mobile device and the remote site. This paper focuses on the problem of *multisite offloading*: offloading portions of the program to multiple remote sites. This offloading is motivated by the desire to perform *data-centric* offloading, as exemplified by the following use case:

User data, especially for mobile applications, rarely resides on the local device but instead exists in the cloud, distributed across multiple sites, such as Flickr, Vimeo and Facebook. Mobile applications which reference multiple types of data therefore access multiple *sources* of data, scattered across the Internet. Consider an application which searches a user's photo collection for pictures of the Eiffel tower and then uses image-matching techniques to find photos her friends have taken of the Eiffel tower. A monolithic mobile application for this might first retrieve all the user's photos of the Eiffel tower from Flickr, extract appropriate features from the images, then retrieve her friends' photos from Facebook, using pattern matching to find photos of the

Eiffel tower. The architecture of this application is shown in Figure 1(a). This application can be extremely expensive, in terms of communication (fetching photos from Flickr and Facebook), storage (caching photos) and computationally (performing feature extraction and pattern matching).

A better approach would be to adopt a data-centric approach and *move the computation to the data*. Feature extraction would be performed on a remote site with fast access to Flickr's data stores, transmitting only the features back to the mobile application. Pattern matching would be performed on Facebook's servers, transmitting only the matching images back to the device. The resulting application architecture is shown in Figure 1(b), and demands far fewer resources of the mobile device.

Accomplishing this type of data-centric offloading requires that several key challenges be addressed:

1) Algorithms must be developed to partition a program between multiple possible execution sites. Most previous work focuses on *single-site* offloading, where an application is divided between the mobile device and a single remote server [6, 8, 10, 14]; such offloading would incur far more communication costs, as data would have to be transmitted from either Flickr or Facebook.
2) In a multi-site offloading scenario, the partitioning algorithm must account for differences in capabilities between remote sites; the feature extraction module must be assigned to Flickr, not Facebook, even though both sites may possess fast computation capabilities. Offloading a module to the wrong site can result in its executing less efficiently. Furthermore, communication between some sites might be faster than communication between others, changing the efficiency of moving data. Prior work on $k + 1$ offloading has not distinguished between the abilities of the k remote servers [11], precluding sophisticated multi-site offloading.
3) The granularity of offloading must be chosen appropriately. Most prior work performs offloading at the *class* level, making the decision to offload for all objects of a particular class *in toto*. Because photo-manipulation applications might use objects of the same class to manipulate different sources of data, we would instead like to perform offloading at the *object* level, allowing objects of the same class to be offloaded to different servers.

This paper describes novel approaches to tackling these challenges. We introduce a multi-way partitioning algorithm, based on *energy minimization* [3], that both allows a program to be partitioned between multiple sites and considers differences in capabilities of various sites that might affect execution efficiency. Second, we discuss a program representation that allows for partitioning to be performed at the granularity of *allocation sites* rather than classes, permitting more precise portions of the computation to be offloaded. We present results that show a) that our partitioning algorithm produces more efficient partitions than prior multi-way partitioning algorithms and, indeed, approaches optimality, and b) that performing offloading at allocation-site granularity produces more efficient partitions than partitioning at the class level.

The remainder of the paper is organized as follows: Section II describes our multi-site partitioning algorithm. Section III discusses our program representation to support fine-grained offloading, and discusses some of the challenges for generating offloading code for fine-grained offloading. Section IV evaluates the performance of our techniques for real-world Java applications, and demonstrates that they outperform prior approaches to multi-way offloading. Section V describes related work, and Section VI discusses future work and concludes.

II. Multisite offloading

The primary challenge in performing data-centric offloading is appropriately dealing with applications that manage multiple sources of data. This scenario (as exemplified by the use case in Section I) requires that an application be partitioned across multiple locations: the mobile device, which must contain at least the user-facing modules such as the user interface, and one or more servers, which can be used for computation offloading or to co-locate computation with data in order to reduce communication costs. A crucial component of successfully performing this co-location is accounting for differences between offloading sites: computation accessing a source of data should be located on the server with that data, not any arbitrary site.

Traditionally, determining which portions of a computation to offload is cast as a *graph partitioning* problem. The program is represented as a graph, with nodes representing computation modules (typically, a class in an object-oriented program), and edges between nodes representing interaction between modules (*e.g.*, invocations by one class upon another). These nodes and edges are assigned weights, according to the goal of offloading. For example, to minimize computation time, the weight of an edge can represent the communication cost (in time) between two modules, while the weight of a node can represent the computation time of a particular module on different servers. A partitioning of this graph assigns each node to a particular computation site. Edges that span two partitions represent communication that must occur in the offloaded application. Note that certain modules might have predetermined computation sites: for

example, user interface modules must reside on the local device.

The weights for nodes and edges might vary depending on the sites the various computation modules are offloaded to:

- ***Node weight:*** The cost in terms of computation and/or power consumption of running a particular portion of code on the mobile device would be different from that of running it on some cloud servers, which in turn might differ among themselves in their computational capacity or cost of their availability. This is captured by the fact that each node is assigned a different cost depending on the partition of the application graph it finally ends up in or the label it is assigned.
- ***Edge weight:*** The communication cost associated with movement of data and requisite messages in case parts of the application reside on different hosts varies depending on the hosts. For example, communication between two cloud-resident servers may be very fast, while communication between the mobile device and the cloud may be much slower.

The cost of the partitioning can be calculated by considering the weights of edges that span partitions (*i.e.*, the cost of communication) and the weights of nodes assigned to each server (*i.e.*, the cost of computation). Producing a minimum weight partitioning leads to the optimum choice of modules to offload. Table I describes a number of possible weighting strategies such that minimizing partitioning cost leads to optimal offloading. These weights can be estimated either through static analysis of the program's structure or by profiling the program with representative inputs. The next section formalizes this informal discussion, giving an integer linear programming (ILP) formulation of the multi-site offloading problem.

A. Problem formulation

Our goal is to partition a graph $G = (V, E)$, with vertices V and edges $E \in V \times V$, among a set of $k+1$ partitions, $P = \{p_0, p_1, \ldots, p_k\}$ (by convention, p_0 represents the mobile device, and $p_1 \ldots p_k$ represent the offloading sites). Each vertex $v \in V$ has an associated set of weights, $\{w_v^1, w_v^2, \ldots, w_v^k\}$ where w_v^i is the computation cost of v if it were assigned to partition p_i. Each edge $e = (v_1, v_2)$ has a set of weights, $\{w_{v_1,v_2}^{1,1}, w_{v_1,v_2}^{1,2}, \ldots, w_{v_1,v_2}^{1,k}, w_{v_1,v_2}^{2,1}, \ldots, w_{v_1,v_2}^{k,k}\}$, where $w_{v_1,v_2}^{i,j}$ is the communication cost of e if v_1 were assigned to partition p_i and v_2 to p_j.

We can formulate the multi-way partitioning problem as a 0-1 ILP problem. Our goal is to minimize the following objective function:

$$\sum_{v \in V} \sum_{i=0}^{k} w_v^i \cdot m_v^i + \sum_{v_1 \in V} \sum_{i=0}^{k} \sum_{v_2 \in V} \sum_{j=0}^{k} w_{v_1,v_2}^{i,j} \cdot e_{v_1,v_2}^{i,j} \quad (1)$$

where the output variable $m_v^i = 1$ if vertex v is assigned to p_i and 0 otherwise, and the output variable $m_{v_1,v_2}^{i,j} = 1$ if vertex v_1 is assigned to p_i and v_2 is assigned to p_j, and 0 otherwise. Equation 1 is subject to the following constraints:

$$\forall v \in V : \sum_{i=0}^{k} m_v^i = 1$$

which enforces that each vertex is assigned to exactly one partition; and

$$\begin{aligned} &\forall v_1, v_2 \in V \\ &\forall i, j \in [0..k] \\ &\quad e_{v_1,v_2}^{i,j} \leq m_{v_1}^i \\ &\quad e_{v_1,v_2}^{i,j} \leq m_{v_2}^j \\ &\quad m_{v_1}^i + m_{v_2}^j \leq 1 + e_{v_1,v_2}^{i,j} \end{aligned}$$

which collectively enforce, for each edge, $e_{v_1,v_2}^{i,j} = m_{v_1}^i \wedge m_{v_2}^j$. In other words, an edge variable is 1 iff both its endpoints have been assigned to the appropriate partitions.

Solving this ILP will result in a partitioning that assigns vertices to partitions in a manner that both reduces the total computation cost (*i.e.*, total vertex weight) and total communication cost (*i.e.*, total edge weight). Note that for most scenarios, the edge weight for an edge whose vertices are in the same partition is 0, and hence the only edge weight we need consider is that of edges that span partitions (*i.e.*, the communication costs). To handle nodes whose locations are fixed (*e.g.*, user-interface modules), the weight of assignment to any other partition can be set to ∞.

B. Shortcomings of prior approaches

Integer linear programming is NP-complete, and, more specifically, optimal multi-way graph partitioning is NP-hard, so we do not expect to be able to solve the multi-site offloading problem optimally for any but the smallest programs, which lead to the smallest graphs. Prior work has made two simplifying assumptions to make the offloading problem tractable:

Two-way partitioning: Perhaps the most common simplification is to perform only two-way partitioning. Rather than offloading computation from a mobile device to k remote sites, only one server is considered for offloading [6, 8, 10, 14, 15]. Thus, only two computation costs need be considered, the cost of computing on the mobile device and the cost of computing at the offloading site.

Goal	Node weight	Edge weight
Minimize computation time	Computation time of module on designated site.	Communication time between designated sites.
Minimize local storage needs while keeping communication low	If on local storage device, storage needs of modules. If on server, 0.	ε, to discourage communication between device and servers
Minimize battery usage	If node assigned to device, energy consumption of device while executing. If node assigned to server, energy consumption of device while idling during computation.	If communication between device and server, energy cost of communication. If communication between servers, energy consumption while idling during communication

Table I
WEIGHT ASSIGNMENTS FOR VARIOUS OFFLOADING OBJECTIVES

Further, only one edge weight need be considered: the cost of communicating between the device and the site. The advantage of this approach is that it allows a relaxation of the ILP problem into a min-cut/max-flow problem [6, 8]. Unfortunately, two-way partitioning does not encompass the full range of scenarios in which we would like to perform offloading. Notably, even a simple application such as the one described in Section I requires considering multiple offloading sites.

Homogeneous servers: Another possible simplification to the partitioning problem is to consider *homogeneous servers*; rather than assuming that every offloading site has different computational and storage capabilities, and hence requires different weights for nodes, or different communication properties, and hence requires different weights for edges connected to that server, homogenous servers assume that all offloading sites have the same behavior and capabilities. Just as in the case of two-way offloading, this assumption means that each node requires two weights: one for the mobile device, and one if it is offloaded to *any* server. Similarly, each edge only requires two weights: one weight for a connection between the mobile device and a server, and one weight for the connection between servers.

Interestingly, despite the much simpler space of node- and edge-weights, simply requiring multi-site partitioning is sufficient to make the problem NP-hard, and hence heuristics must be adopted to find suitable multi-way partitions. Ou *et al.* proposed *heavy-edge, light-vertex matching* (HELVM) as one such heuristic [11]. To our knowledge, this is the only prior work to consider multi-site offloading. Unfortunately, assuming that all servers are alike does not suffice for the types of applications we would like to consider, where there is necessarily heterogeneity across servers as different data repositories are at different locations.

In the next section, we detail our heuristic for multi-site computation offloading, which makes neither of the preceding simplifying assumptions. Despite this generality, we will argue that in the two-way partitioning scenario, our approach degenerates to min-cut/max-flow, leading to an optimal solution, and in the homogeneous server scenario our heuristic produces better partitions than HELVM.

C. Partitioning algorithm

To develop an algorithm for multi-site computation offloading that takes into account server heterogeneity, we turn to an unlikely source, the computer vision community. We present here an adaptation of *graph-cuts*-based energy minimization, first presented by Boykov *et al.* [3].

Graph-cuts is a *label-assignment* algorithm, which attempts to assign labels from a label space $\{\alpha, \beta, ...\}$ to an undirected graph $G = (V, E)$ based on two costs: (i) *assignment cost*, in which assigning a label α to a node v incurs a cost q_v^α, and (ii) a *separation cost*, in which assigning a label α to node v_1, and the distinct label β to node v_2 would incur a cost $q_{v_1,v_2}^{\alpha,\beta}$. The goal of label-assignment is to find a labeling, f, that minimizes $E(f)$, the sum of the assignment costs and the separation costs for a graph given the current labeling. As we have chosen the notation for the two costs carefully, it should be apparent how a labeling algorithm might be adapted in a straightforward manner to our multi-site partitioning problem:

- The set of partitions, $\{p_0, p_1, \ldots, p_k\}$, is mapped one-to-one to a set of labels, $\{\alpha, \beta, \ldots\}$.
- Each node weight w_v^i is mapped to to an assignment cost, q_v^α, where p_i is mapped to label α.
- Each edge weight $w_{v_1,v_2}^{i,j}$ is mapped to a separation cost, $q_{v_1,v_2}^{\alpha,\beta}$, where p_i and p_j are mapped to labels α and β, respectively.
- If the edge (v_m, v_n) does not exist in the partitioning problem, then all separation costs $q_{v_m,v_n}^{*,*}$ are 0. In other words, because these two nodes do not interact in the offloading problem, there is no cost to separating them in the labeling problem.

Given this mapping, applying an energy-minimizing heuristic to the labeling problem will also minimize the objective function of Equation 1. Upon completion of the labeling problem, if node v is assigned to label α, we assign it to partition p_i if p_i maps to α. Note that to perform

Algorithm 1 Labeling algorithm skeleton

```
Start with an arbitrary labeling, f
loop
  e_c ← E(f)
  for each label l ∈ {α, β, ...} do
    f ← move(l)
  end for
  if E(f) = e_c then
    return f
  end if
end loop
```

this mapping correctly, we must make the simplifying assumption that the edge weight between two nodes that are both assigned to the same partition is 0; this is reasonable as "communication" within a single server is orders of magnitude faster and more efficient than communication across sites.

1) Labeling algorithms: The mapping we presented above is sufficient to solve the multi-site partitioning problem given appropriate weights by simply feeding the problem to an energy minimization algorithm. In this case, we adopt the heuristics of Boykov *et al.* [3]. In lieu of fully redescribing the algorithms, we present here a capsule summary of the two heuristics we use. Both algorithms operate using the same general framework, presented in Algorithm 1. After starting with an arbitrary labeling (partitioning), the algorithm iterates over the labels, and performs a *move* on each label. The result of a move is to switch some set of nodes to that label, such that the new labeling reduces the value of the cost function, E. Eventually, the process reaches a fixpoint, and the final labeling is returned. The only difference between the two heuristics is in what $move(l)$ does. The two heuristics are:

Label swapping: The most general heuristic is *label swapping*. The function $move(l)$ in this heuristic iterates over all labels other than l, and for each such label m, attempts to change the label of any nodes labeled m to l. Note that each individual swap must reduce the cost of the labeling—no swap may increase the cost even if the move decreases the cost overall.

Label expansion: A move in *label expansion* is more complex than in label swapping. In label expansion, $move(l)$ is allowed to change the label of *any* node to l, regardless of its original label. Note that this encompasses all moves possible via label swapping, and more. This heuristic is more restrictive than label swapping, as it can only be applied when the edge weights are drawn from a *metric* space:

$$\forall \alpha, \beta, \gamma \in labels \ . \ q^{\alpha,\beta}_{v_1,v_2} < q^{\alpha,\gamma}_{v_1,v_2} + q^{\gamma,\beta}_{v_1,v_2}$$

In our context, this restriction roughly means that the cost of communication between site α and site β must be higher than communicating that data from α to γ and then to β.

Label expansion is preferred over label swapping for two reasons. First, $move(l)$ does not need to cycle through every label, reducing its computational complexity. Second, because $move(l)$ for label expansion encompasses all possible moves of label swapping, plus more, in general label expansion should be able to find more efficient labelings (although as we will see in Section IV, this is not always the case).

2) Comparisons with prior work: The $move$ functions of both the label swapping and label expansion heuristics are based on setting up and solving min-cut/max-flow problems. Crucially, in the two-label case (*i.e.*, when there is simply the mobile device and a single server), both the label swapping and label expansion heuristics simplify to solving a single min-cut/max-flow problem. Interestingly, the flow network constructed by the label expansion algorithm in this case is equivalent to the flow network constructed by prior work that uses min-cut/max-flow for the single-site problem [8, 15]. Thus, in the single-site scenario, our approach would arrive at the same offloading decisions as prior work.

We can also use our approach to deal with the homogenous server case, simply by restricting the space of weights on nodes and edges so that the weight of a node is the same regardless of which off-site server it is placed on, and the weight of each edge between the mobile device and a server is the same regardless of server. Section IV demonstrates that in this restricted case, our approach provides higher quality partitioning than HELVM.

D. Static versus dynamic partitioning

Our technique for making offloading decisions is a *static* one by nature: all offloading decisions are made at compile time using node and edge weights that are estimated through a combination of static analysis and profiling. Making offloading decisions statically allows us to use more sophisticated heuristics to obtain the partitioning (as run-time overhead is not a consideration). Most prior work that has used complex heuristics has performed static offloading [10, 15].

The primary drawback to static offloading is that the estimates of program behavior might be wrong. This can be because the profiling data used to generate node and edge weights is not representative of real inputs, or simply because program behavior is not uniform over time. Hence, other work has focused on dynamic approaches to offloading, allowing partitioning decisions to be made adaptively, adjusting to run-time conditions [8, 11].

While our approach performs static offloading, it is not

incompatible with dynamic approaches. Supporting adaptive offloading requires significant run-time monitoring capabilities to track run-time behavior that might affect offloading decisions [12]. A promising approach would be to use a static technique such as ours to derive an initial offloading plan, then identify modules that may have been incorrectly assigned and apply run-time monitoring and adaptive offloading only to those modules. This would provide many of the advantages of dynamic offloading without incurring the run-time overhead. Section III-B discusses some of these possibilities in further detail.

III. Supporting fine-grained offloading

Another challenge in data-centric offloading is to determine the appropriate granularity of offloading. Recall that we cast the offloading problem in terms of graph partitioning, where each node in the graph represents a computation module that can be offloaded. The question of granularity is one of determining what each unit of computation should be.

Prior work has considered offloading at the granularity of classes [8, 11, 15], or functions [10, 14]. Unfortunately, these units of computation are fundamentally units of *code*, which are ill-suited to data-centric offloading. Consider the use case from Section I. Both the feature extraction and pattern matching modules of computation manipulate images, and hence might use the same classes and code. Function- or class-based offloading would require that the image processing code be offloaded to just one of the two servers. Because the same piece of code can be used to operate on different pieces of data, and that data might reside at different remote sites, we would like to find a granularity of offloading that allows the same code operating on different data to be offloaded to different sites.

The most natural unit of *data* in object-oriented programs is the object, as it encompasses not only functions, but also the data they operate on. Unfortunately, using objects as the granularity of offloading presents problems. First, it is difficult to represent the program as an object graph for partitioning, as the number of objects is potentially unbounded and, moreover, indeterminable statically. Second, even given a suitably approximated object graph that is amenable to partitioning, it is unclear how to generate code to support offloading.

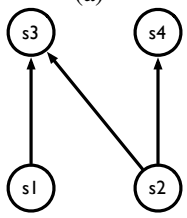
To address these issues, we propose to perform offloading at the granularity of *allocation sites*. All objects that are allocated at a particular allocation site will be treated as a single unit for offloading purposes. This approximation is a natural fit for static analyses such as pointer-analysis [1], that treats all objects that are created from a single allocation site as aliased. We introduce a program representation for partitioning called the *object interaction graph*, which allows partitioning and offloading at the allocation-site level.

```
Class A {
  void foo1(B x) {
     x.bar();
  }

  void foo2(C y) {
     y.bar
  }
}
```

```
Class Main {
  static void main() {
     A a1 = new A(); //s1
     A a2 = new A(); //s2
     B b1 = new B(); //s3
     C c1 = new C(); //s4
     a1.foo1(b1);
     a2.foo1(b1);
     a2.foo2(c1);
  }
}
```

(a)



(b)

Figure 2. (a) Code snippet. (b) OIG produced from code snippet

.

A. Object interaction graph

An object interaction graph (OIG) consists of nodes and edges between them. Each node in an OIG represents an allocation site in a program. Two nodes in an OIG have an edge between them if they interact. In other words, if an object allocated at site s_1 invokes a method on an object allocated at a site s_2, there must be an edge between s_1 and s_2. Furthermore, if an object allocated at site s_1 invokes a method that *allocates* another object, there must be an edge between s_1 and that allocation site. Figure 2(a) shows a piece of code, while Figure 2(b) shows the object interaction graph induced by that code.

1) Building the OIG: To build the OIG, we leverage several analyses. To begin with, we perform an interprocedural alias analysis. This allows each variable accessed in the program to be associated with an allocation site. We use an inclusion-based, field-sensitive, context-insensitive analysis [9], but a more precise analysis would allow for more distinction to be made between allocation sites. A call-graph analysis is also performed, allowing abstract methods to be resolved to particular concrete methods.

Each method invocation on a variable (including static methods; see below) is then analyzed. The allocation site to which the variable v is aliased, s, is the source node. The code of the invoked method is then analyzed, as well as the code of any methods called from that method on the same object. For each variable u on which a method is invoked within that code, an edge is added from s to t, the allocation

site to which u is aliased. Note that because of alias analysis imprecision, u may be aliased to multiple allocation sites; edges are added from s to all such allocation sites.

Dealing with static methods: Static methods present an interesting problem for the OIG construction method outlined above. Unlike member methods, static methods do not need to be called on a particular variable, and hence are not associated with an allocation site. We create a dummy allocation site c for each class in the program; static methods are considered to be invocations on c.

Assigning weights to the OIG: The final step in constructing the OIG is to assign weights to nodes and edges as required by the partitioning algorithm. This process can either be done statically by estimating the computation requirements for each object and the communication requirements for each method invocation, or dynamically, through the use of instrumentation and profiling. The particular weights will, of course, be dependent on the objective of partitioning, as detailed in Table I. Section IV discusses a preliminary weight-assignment approach we have implemented using a combination of static estimation and dynamic profiling.

B. Code generation

Generating code for allocation-site based offloading is part of our future work; here we briefly discuss some of the challenges and approaches to code generation.

Generating code for allocation-site based offloading presents challenges not present in class-based or function-based offloading. To generate code correctly, each method invocation needs to know whether the target of the invocation is executing at the same site as the current piece of code. In class- and function-based offloading, code generation is straightforward: a particular piece of code always executes in a particular place. For example, in class-based offloading, each class is specifically assigned to a particular execution site. Because each class knows where it is to be executed, and knows where any classes it invokes methods on will be executed, generating code is straightforward, including whether any remote invocation code need be generated.

J-Orchestra [12] is one tool that can do this automatically for Java applications. Object allocation creates the object at the appropriate execution site (based on the object's class), and directs other offloading sites to create "stub" objects, which can be used to invoke methods on the newly created object via remote method invocation. Notably, the actual class code is only ever executed at a single site. A similar approach works for function-based offloading.

Unfortunately, with allocation-site based offloading, different objects of the same class can be offloaded to different locations. This means that a particular piece of code (*e.g.*, a function in a class) might be executed in up to $k+1$ different places: the mobile device, and one of k different offloading sites. Moreover, when that code needs to invoke a method m on a variable v, different versions of the code must be invoked depending on where v is located.

We can simplify this problem by requiring that all allocation sites that alias to the same variable v be offloaded as a single unit; this reduces the number of variations of the code to the number of offloading sites (since a particular variable will always be offloaded to a known site). Note, furthermore, that each allocation site is associated with a single execution site. We can thus create $k + 1$ subclasses of each class c, producing c_0 through c_k, where class c_i is generated assuming it will be executing at location i. We can then replace each allocation site for c with the appropriate subclassed version, based on the site's eventual offloading destination.

The approach outlined above does not change the semantics of the program, prior to offloading: each object of class c is now instead an object of some class c_i that has exactly the same behavior as c. However, we have now transformed the problem of offloading individual objects allocated at particular allocation sites into the problem of offloading different classes: objects of a single class will all be executed at the same site. We can therefore leverage existing class-based offloading tools like J-Orchestra to perform code generation.

J-Orchestra also allows classes to be designated as *mobile*, meaning their execution site can change during execution to better exploit locality. As discussed earlier, because our scheme produces a static partition of the application, it is possible that the offloading decision will be suboptimal at various points during execution. Moreover, the most likely sources of this inefficiency will be objects on the "boundaries" of partitions: objects that interact with objects allocated at different sites. We can thus leverage J-Orchestra's mobility capabilities to mark these boundary objects as mobile, partially ameliorating the drawbacks of static offloading. Investigating more sophisticated heuristics for dynamic modifications of offloading decisions is a topic for future research.

IV. Evaluation

We make two contributions in this paper. First, we present a novel approach to performing multi-site offloading, which allows for computation offloading to be performed when there are multiple, heterogeneous servers to use for remote execution. Second, we argue that using an allocation-site based granularity for offloading can produce better results than class-based offloading. In this section, we evaluate both

Edge base weights	10-25
Actual edge weight	$base\_wt$, for any label
Corr. HELVM edge wt.	$base\_wt$
Node base weight	5-15
Actual node weight	$base\_wt * 3$, if label corresponds to client $base\_wt$, otherwise
Corr. HELVM node wt.	$base\_wt$

Table II
PROPERTIES OF GRAPH SIMULATING EQUIVALENT SERVERS

of those contributions.

First, we show that our labeling-based partitioning scheme outperforms prior work on multi-site offloading. We show this both for randomly generated graphs, where we can compute the optimal partitioning, as well as for graphs generated from real applications. Second, we show, for real benchmarks, that using allocation sites as the granularity for offloading can lead to better offloading quality than class-based offloading.

A. Random Graphs

In this section, we compare the performance of the labeling algorithms with that of HELVM ([11]) and in case of small graphs, the optimal solution. We perform our comparisons on random graphs generated by certain schemes to reflect real-world scenarios involving a client device and two remote servers. Because solving the ILP to find the optimal solution takes time exponential in the number of nodes, it is infeasible for all but the smallest graphs.

We test the partitioning algorithms in two scenarios. One with homogeneous servers, where we differentiate between the client and a server, but not between the two servers, and one with heterogeneous servers, where different servers possess different computational capabilities, and hence present different opportunities for offloading.

First, we consider the simplest case where the computational capabilities of all offloading sites are equivalent. The goal of offloading in this situation is simply to move as much computation off the mobile device as possible while minimizing communication. The graphs were created according to the properties listed in Table II. To create a graph with N nodes, we randomly chose a weight for each node from the range given, and randomly generate an edge between pairs of nodes, with probability $1/3$. The edge weights were then selected from the given range. One of the nodes was labeled "unoffloadable," and hence had to be kept at the mobile device. We generated three random graphs for a range of different graph sizes.

Nodes	**Label Swap**	**Label Expand**	**HELVM**	**Optimal**
10	712	712	770	712
	631	631	683	631
	639	639	677	639
15	1297	1297	1395	1297
	1369	1369	1495	1369
	1254	1254	1348	1254
30	5382	5382	5598	
	5621	5621	5969	
	5289	5289	5605	
50	14168	14168	14730	
	14486	14486	14802	
	14261	14261	14549	
100	57557	57557	58799	
	57333	57333	58209	
	57933	57933	58909	

Table III
GRAPH COSTS OF PARTITIONING RANDOM GRAPHS SIMULATING EQUIVALENT SERVERS

Edge base weights	10-25
Actual edge weight	$base\_wt * 5$, if edge between client and server(1) $base\_wt * 6$, if edge between client and server(2) $base\_wt$, otherwise
Corr. HELVM edge wt.	$base\_wt$
Node base weight	5-15
Actual node weight	$base\_wt * 3$, if label corresponds to client $base\_wt * 2$, if label corresponds to server(1) $base\_wt$, if label corresponds to server(2)
Corr. HELVM node wt.	$base\_wt$

Table IV
PROPERTIES OF GRAPH SIMULATING DIFFERENT SERVERS

Table III gives the results of running our labeling-based algorithms, HELVM and optimally solving the ILP. The costs were computed by totaling the edge and node weights for the partitioning chosen by each algorithm. We see two notable trends. First, the labeling-based schemes perform very similarly and, indeed, find the optimal partitioning for small graphs. Second, they both outperform HELVM, even in a scenario where the added capabilities of our algorithms are not fully exploited.

We next studied a more "realistic" scenario, where the client device and servers each had different computational resources and communication properties. The topologies of

Nodes	Label Swap	Label Expand	HELVM	Optimal
10	827	827	1935	827
	737	737	1548	737
	890	890	1909	890
15	1564	1564	3446	1564
	1696	1696	3547	1696
	1633	1633	3866	1633
30	5762	5762	14532	
	5730	5730	14614	
	6081	6081	15832	
50	14517	14517	38358	
	15099	15099	39907	
	15222	15222	41685	
100	59241	59241	165143	
	60351	60351	167935	
	58847	58847	161205	

Table V
GRAPH COSTS OF PARTITIONING RANDOM GRAPHS SIMULATING NON-EQUIVALENT SERVERS

the input graphs, and the base weights for the nodes and edges were the same. However, these base weights were then scaled depending on the execution site in question. In keeping with the limited resources of the mobile device, edges incident on the mobile device and nodes assigned to the mobile device had higher cost. Edge weights and node weights also differed between the servers, but to a lesser degree. Note that server 1 has more efficient communication, but slower computation, than server 2. Because HELVM cannot distinguish between computation sites, the algorithm was run using the base weights in all cases. The weight ranges and scaling factors are given in Table IV.

The comparison results for the case of heterogeneous servers are tabulated in Table V. Note that because HELVM uses a simplified set of weights, the HELVM column is calculated using the partitioning obtained by HELVM but the actual weights. We see, once again, that our algorithms are consistently able to find better partitioning schemes than HELVM and, for small graphs where finding the optimal solution is tractable, find schemes as good as the optimal solution. Furthermore, because our algorithms are able to exploit the differences between servers, their advantage over HELVM is much larger in this scenario than in the homogeneous one.

B. Benchmarks

We next evaluated our partitioning algorithms on two real-world benchmarks drawn from the DaCapo benchmark suite [2], *lusearch* and *h2*. To do this, we used the Soot analysis framework [13] to perform the static analysis necessary to build the object interaction graph, as well as to instrument and profile the benchmarks in order to assign weights to the graph.

1) Analysis and Instrumentation: We formed the object interaction graph by using Soot's built-in points-to analysis (Spark) and call-graph analysis, as well as a new analysis pass which computes which objects call methods on other objects. The OIG was constructed using the techniques described in Section III-A.

To assign weights to the graphs generated from the benchmarks, we used a combination of static and dynamic analysis. First, we used Soot to estimate the "static weight" of each method: an estimate of the number of cycles it would take to execute the method (excluding method calls, but taking into account loops)[1].

Second, instrumentation was inserted into each class to count the number of times that methods were invoked and that particular interaction edges in the OIG were exercised. When running the instrumented program, the number of times a method was invoked was multiplied by the static weight, and the total weight accrued to the object on which the method was invoked. In this way, weights could be calculated for each node in the OIG, based on how much computation was performed in a particular set of objects. Each time a method was invoked, the weight of the corresponding edge in the OIG was incremented, accounting for arguments and return values (as passing that information requires additional communication).

2) Results for object-based graphs: We use benchmark applications *lusearch* and *h2* from the DaCapo Benchmark suite to test our algorithms. The partitioning is done for 3 labels: one corresponding to client and two corresponding to similar servers. The properties of the graph are mentioned in following table. Note that we used the same scaling factors as in the random graph experiments.

Edge/Node base weights	Obtained from object-interaction graph
Actual edge weight	$base\_wt * 5$, if client label involved $base\_wt$, otherwise
Corr. HELVM edge wt.	$base\_wt$
Actual node weight	$base\_wt * 3$, if label corresponds to client $base\_wt$, otherwise
Corr. HELVM node wt.	$base\_wt$

Table VI
PROPERTIES OF GRAPH

Table VII lists the partitioning costs obtained after partitioning the OIG obtained from *lusearch* and *h2*. Once again,

[1]This information could be calculated dynamically, but for ease of profiling, a static estimate was used

the HELVM cost is calculated using the actual weights. As was the case with the random graphs, our labeling algorithms obtain a significantly better partitioning than does HELVM.

Benchmark	Label Swap	Label Expand	HELVM
lusearch	32,169,126	32,169,126	104,952,956
h2	870,579,048	870,579,048	2,846,995,711

Table VII
RESULTS FOR PARTITIONING OIG FOR *lusearch* AND *h2*

C. Object-based vs. Class-based offloading

To study the differences in object-based and class-based partitioning, we take the same object-interaction graph obtained for benchmarks *lusearch* and *h2*, and convert them to class-based graphs by merging together all nodes belonging to the same classes. This will allow us to determine whether we can obtain better partitions using allocation site-based partitioning or class based partitioning. Note that while the partitioning algorithms were run on the condensed, class-based graph, the quality of partitioning was evaluated using the original, allocation site-based OIG. Table VIII gives the partitioning costs for the class graphs. We note that the cost of *lusearch* is the same using both graphs; this is because there is only one allocation site of each object type in this benchmark. In contrast, using the OIG with *h2* produces a notably better partitioning than the class-based graph. This confirms our hypothesis that the finer-grained partitioning enabled by the OIG can lead to more efficient offloading.

V. RELATED WORK

Two-way partitioning for a weighted graph modeled as a max-flow min-cut problem has been a very well-studied topic. The *maximum flow problem* is to find a feasible flow through a single-source, single-sink flow-network that is maximum. The most notable algorithms for solving this problem are the augmenting path method of Ford and Fulkerson [5] and the push-relabel method of Goldberg and Tarjan [7]. *n-way* partitioning, where $n > 2$, is however NP-hard.

Some amount of research has been done on partitioning applications for the purpose of offloading to remote computation centers, but most of them have focused on two-way partitions. Li et al. proposed statically partitioning an application into client tasks and server tasks on the basis of a cost graph [10]. They consider each invocation site as a task, so that the partitioning is at the level of function calls, which one could argue to be too limiting for object-oriented programs. A general branch and bound algorithm is used to find the optimal partition, which by itself is computationally quite intensive. Some heuristics can however be applied so that a slightly more inaccurate solution can be obtained in a practically short period of time.

Benchmark	Label Swap	Label Expand
lusearch	32,169,126	32,169,126
h2	1,112,748,351	1,112,748,351

Table VIII
CLASS GRAPH PARTITIONING COSTS

Wang and Franz adopt a class-based, static partition scheme to split an application between a client and a single server [15]. Like our work, their partitioning approach is based on building a program graph and assigning weights to nodes and edges based on their contributions to some objective function that should be minimized. Because their approach focuses on two-way partitioning, they can find an optimal solution using min-cuts. Our approach is more general than their work both by supporting additional offloading sites and by supporting finer-granularity offloading.

Gu et al. [8] developed a framework for partitioning an application and adaptively offloading them. Their partitioning algorithm is based on the standard min-cut algorithm, and the partitioning itself is done at the level of classes. Additionally, in [8], they only attempt to enable execution of memory-intensive applications through offloading, ignoring other resources such as CPU utilization.

Ou et al. proposed a *(k+1)*-way application partitioning and offloading scheme in [11], wherein the application could be split into potentially more than two partitions - one unoffloadable part for the client and *k* offloadable partitions to be run on remote sites. They have also proposed a light-weight, n-way graph partitioning algorithm, *Heavy Edge Light Vertex Matching Algorithm*, which heuristically splits the application graph, so as to satisfy some pre-defined constraints. Their scheme, however, splits the application at the level of classes. Moreover, our algorithm is better modeled to reality as it can handle the presence of different label weights for nodes - *HELVM* treats all servers as equivalent, possibly differing only in their constraints-set. As has been shown, our algorithms are able to generate better partitions than HELVM.

VI. FUTURE WORK AND CONCLUSIONS

The primary avenues for future work concern the object-interaction graph. First, we intend to build a compiler and run-time system that can generate code and perform partitioning in a manner consistent with the scheme described in Section III-B. Second, we aim to refine the object graph to allow more precise offloading. Many data structures are composed of smaller objects that are allocated within larger

methods. Our current scheme would force all such objects to be offloaded to the same site, even though they might be part of different data structures. This can be mitigated by considering objects that are fully enclosed by other objects to be part of the enclosing object rather than requiring a separate offloading decision for them; this notion of enclosure is similar to that defined by *ownership types* [4]. Deciding the right scope for this enclosure is a subject for future work.

In conclusion, we argued that efficient computation offloading requires a notion of *data-centric* offloading, where computation is co-located with data. We presented two schemes for supporting data-centric offloading: multi-site partitioning, which allows an offloading approach to support multiple differentiated offloading sites, and allocation-site based partitioning, which allows finer, object- and data-centric offloading decisions to be made. Our evaluation demonstrates that both of these approaches can lead to more efficient partitioning and offloading than prior work.

REFERENCES

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[3] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23:1222–1239, November 2001.

[4] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, 1998.

[5] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[6] N. Geoffray, G. Thomas, and B. Folliot. *Transparent and Dynamic Code Offloading for Java Applications*, volume 4276, pages 1790–1806. Springer, 2006.

[7] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[8] X. Gu, A. Messer, I. Greenberg, D. Milojicic, and K. Nahrstedt. Adaptive offloading for pervasive computing. *IEEE Pervasive Computing*, 3:66–73, July 2004.

[9] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[10] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '01, pages 238–246, New York, NY, USA, 2001. ACM.

[11] S. Ou, K. Yang, and A. 'Liotta. An adaptive multi-constraint partitioning algorithm for offloading in pervasive systems. *Pervasive Computing and Communications, IEEE International Conference on*, 0:116–125, 2006.

[12] E. Tilevich and Y. Smaragdakis. J-orchestra: Enhancing java programs with distribution capabilities. *ACM Trans. Softw. Eng. Methodol.*, 19:1:1–1:40, August 2009.

[13] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[14] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 119–130, New York, NY, USA, 2004. ACM.

[15] L. Wang and M. Franz. Automatic partitioning of object-oriented programs for resource-constrained mobile devices with multiple distribution objectives. In *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 369–376, Washington, DC, USA, 2008. IEEE Computer Society.