An Adaptive Multi-Constraint Partitioning Algorithm for Offloading in Pervasive Systems

Shumao Ou, Kun Yang and Antonio Liotta
University of Essex, Department of Electronic Systems Engineering, UK
{smou, kunyang, aliotta}@essex.ac.uk

Abstract

Offloading is a kind of mechanism utilized in pervasive systems to leverage the severity of resource constraints of mobile devices by migrating part of the classes of a pervasive service/application to some resource-rich nearby surrogates. A pervasive service application needs to be partitioned prior to offloading. Such partitioning algorithms play a critical role in a high-performance offloading system. This paper proposes an adaptive (k+1) partitioning algorithm that partitions a given application into 1 unoffloadable partition and k offloadable partitions. Furthermore, these partitions satisfy the multiple constraints imposed by either application users or mobile device resources. Underpinning the partitioning algorithm is a dynamic multi-cost graph that models the costs of an application in terms of its component classes (including CPU cost, memory cost and communication cost), and a Heavy-Edge and Light-Vertex Matching (HELVM) algorithm to coarsen the multi-cost graph. An offloading toolkit implementing the above algorithms has been developed, upon which the evaluations are carried out. The outcomes of the evaluation have indicated a higher level of performance of our algorithm in terms of its efficiency and cost-effectiveness.

1. Introduction

Pervasive systems aim to utilize computational devices and network facilities to help users perform their traditional tasks in an easier way or even to augment a user’s perception of the world. It is expected by users that the same functions can be supported on their mobile devices (such as smart phones or PDAs) when they are on the move rather than sitting in front of a desktop PC in their offices or homes. Apparently even the most powerful PDAs today (due to the size and weight constraints) are unable to compete against their desktop siblings with regard to any type of resource, and especially battery life and network capacity. Meanwhile, in some of the working and living environments, the computing resources are often rich. For instance, in offices or cafes, some desktop PCs may be idle while the mobile device is busy. As such, it makes sense for these resource-constrained mobile devices to make use of the resources available in their vicinity to leverage its resource insufficiency. For example, a mobile device could rely on a surrogate to carry out some complex computation and to perform large amount of data exchange through the surrogate’s broadband network resource. We refer to this as an offloading mechanism. A similar idea is also referred to as surrogate computing or cyber foraging [1].

How to partition a given pervasive application is one of the most important issues for designing an adaptive, cost-effective, and efficient offloading system. Some critical issues concerning the partitioning problem include:

• **Application Component Classification:** As not all of the application components are suitable for remote execution offloading, we need to distinguish offloadable and unoffloadable components. The unoffloadable components are those that can only be executed on the mobile devices, examples of which are components that directly access local I/O devices (e.g. in Java, components with native method to access local files), components that directly access device-specific information (e.g. in Java, the system.properties contain specific information related to host system), and components directly handling the user interaction.

• **Application Component Weighing:** When choosing an application component to offload, we need to scale the weights of each application component regarding its resource utilization, such as memory, process time, and bandwidth utilizations. Normally, the weights of an application component vary for different users and in different running environments.

• **Reducing Communication Overhead** introduced by the remote communication between, for example, the mobile device and a surrogate.

• **Reducing Algorithm Complexity:** As the algorithm will be running in resource-constrained mobile devices, the algorithm itself should be light-weighted.

The problem of application partitioning is similar to that of partitioning a finite element graph into a certain number of disjoint subsets of vertices while fulfilling
some given objectives (e.g. minimizing the amount of connection between the subsets). This type of graph partitioning problem is known as NP-complete [11]. There is a lot of research work done with graph partitioning using heuristics [5-15]. Amongst these different types of algorithms, the multilevel algorithms [8, 7] have inspired us most. These types of algorithms follow the following procedure to partition a graph: it first recursively reduces the size of a graph by collapsing vertices and edges (i.e., to coarsen the graph), then partitions the coarsest graph, and finally recursively un-coarsens and refines it back to obtain a partitioned original graph. The work carried out in [8, 7, 6] demonstrated how the multiple constraints were dealt with. However, these works were motivated by static graph partitioning problems such as VLSI (Very Large Scale Integration) design rather than pervasive computing of a dynamic nature. Typically these algorithms are of high computational complexity because the real-time requirement is not that critical. Our application partitioning algorithm, which is specific to pervasive systems, is designed and implemented in a much light-weighted manner so as to make it feasible for resource-constrained mobile devices. For example, our algorithm coarsens a graph directly into a given number of partitions without going through all 3 steps as briefed above [8].

The multi-constraint partitioning algorithm proposed in this paper is called adaptive (k+1) as it partitions the whole component classes of an application into k unoffloadable partition that will run locally on the mobile device and k offloadable partitions for k individual surrogates respectively. The values of the metrics it employs are obtained on-the-fly instead of predefined at the application development stage. Furthermore, these partitions satisfy the multiple constraints imposed by the application users or mobile device resources. Underpinning the partitioning algorithm is a dynamic multi-cost graph that models the costs of each constituent class in terms of its CPU cost, memory cost and communication cost. The algorithm to coarsen the multi-cost graph is called Heavy-Edge and Light-Vertex Matching (HELVM).

The remainder of this paper is organized as follows. Some related work is discussed in Section 2. In Section 3, we propose a dynamic multi-cost graph, formulate the problem of application partitioning and propose our algorithm. Section 4 discusses the implementation of the algorithm and briefs an offloading system. Section 5 provides some experiment results illustrating the performance of our proposed algorithm. The conclusions are given in Section 6.

2. Related Work

Some research work on offloading mechanism for resource-constrained mobile devices has been proposed [2, 4, 18, 19, 20]. The Spectra project [2] proposes a remote execution system for mobile devices used in pervasive computing. Chen et al. proposed an offloading framework [18] that focuses on a Java-based environment and dynamically decides whether to execute locally or remotely based on the computational complexity and communication channel conditions. However, these works did not provide specific application partitioning algorithm for offloading.

Only a few offloading systems discussed application partitioning. The Coign [19] project proposed a system to use min-cut algorithm to statically partition binary applications built from Microsoft’s Component Object Model (COM) components. Li et al. [20] constructed a static cost graph and applied a partition scheme to statically divide application tasks into client and server subtasks at design time. These two pieces of work are based on static partitioning and they cannot adapt to environmental changes. Furthermore, Li’s partition scheme needs the knowledge of the source code because it needs to add some instructions in the source code while the application is being designed.

Gu et al. [4, 3] proposed an adaptive infrastructure for Java application offloading execution, which adapted a min-cut [5] heuristic algorithm to dynamically partition application at runtime. The algorithm partitions a graph into candidate partition plans according to the edge-weight, and then selects the best partition plan by using a combination metric comparison. Due to the fact that most of the resource constraints are related to vertex-weights (e.g., memory consumption) instead of edge-weights (e.g., the interaction or the level of closeness of two components), there is a possibility that Gu’s min-cut heuristic may miss some better partitioning solutions in its candidate partition plans, as to be demonstrated in Section 5. However, these solutions can be picked up by our algorithm. In addition, in [4, 3], they considered only memory constraint. In [22], the authors proposed a k-cut multi-way partition algorithm for optimal service distribution. It fits service components into k devices to fulfill QoS requirements. However, it merges a service component by the largest resource requirement, i.e., only by the vertex-weight. This algorithm can also miss some better partitioning solutions.

Our algorithm considers both edge weights and vertex weights simultaneously, aiming to relieve not only memory constraint but also CPU usage constraint and bandwidth constraint. And this consideration is managed at a cost that is as low as possible. Furthermore, our
algorithm operates on Java byte-code rather than Java source code, which naturally increases its feasibility.

3. Adaptive (k+1) Partitioning Algorithm

In this section, we begin with application cost modeling; then we formulate the partitioning problem. We discuss the adaptive (k+1) partitioning algorithm by separating it into three steps. Furthermore, we discuss the heavy-edge and light-vertex matching (HELVM) algorithm in detail, which is the core algorithm in the step 2 of the adaptive (k+1) partitioning algorithm.

3.1. Dynamic Multi-Cost Graph

An undirected graph, called dynamic multi-cost graph or DMC graph for short, $G=(V,E)$ is used to represent an application’s classes and their corresponding costs. The vertex set $V$ represents application components/classes. Each vertex $v \in V$ is annotated with $n$ weights by an $n$-tuple $(w_1, w_2, ..., w_n)$, which represents the multiple costs a vertex consumes. The edge set $E$ represents the interactions amongst components. An edge $e_{ij} \in E$ represents the frequency of invocation and data access between nodes or classes $v_i$ and $v_j$. The weight of the edge $w(e_{ij})$ is the total number of invocations and data accesses. For instance, if an application component $v_i$ invokes two different methods in component $v_j$, each and $v_j$ accesses a data structure in $v_i$ once; then the weight between these two components is $3$. We do not distinguish the direction of invocation and data access as we assume that the overhead for either direction is identical. The bandwidth required for an invocation and a data access is considered as a weight of related vertex.

There are three categories of costs in an offloading system: computational cost of application component (including memory cost, processing time cost, and so on), communication cost for application components’ interaction, and the overhead cost of offloading system itself. The last one is related to the offloading system design and will be discussed in Section 4. The other two costs are modelled in the dynamic multi-cost graph of an application, i.e., the computational cost by vertices and communication cost by both vertices (bandwidth) and edges (invocation frequency).

Each cost, either on vertices or edges, is calculated dynamically, aiming to reflect the changes of resource consumptions and environment, such as the availability of surrogates. For example, the process time consumed by an initialization component in the application is changing during execution: it may cost 100% of process time in the beginning and then stays idle afterwards. Our solution to capture the change is to take snapshots to analyse the real time resource consumptions so that the partitioning algorithm can adapt to the changes.

In our algorithm, a cost tuple is implemented by a vector so it is also called weight vector. The $j^{th}$ vertex weight means the $j^{th}$ cost value in the tuple. In the multi-cost graph, cost is represented by weight and is a nonnegative integer.

Given the growing popularity in networked applications (including pervasive computing), Java is selected as the experimental programming language for pervasive applications. However, the partitioning offloading principle discussed here is equally applicable to applications developed by other object-oriented programming language. In this case, each vertex in the multi-cost graph represents a Java class. Figure 1 illustrates two examples of multi-cost graph. In Figure 1(a), each vertex has a name and a weight vector. There are three elements (i.e., weights) in the weight vector that represent memory utilization, accumulated processing time, and bandwidth usage respectively. The bandwidth usage means bandwidth used for external access, not for the interaction between classes. The integer value labeled to each edge is the edge weight representing the frequency of interaction between the two incident classes.

![Figure 1. Examples of Multi-Cost Graph](image)

To reduce the computational complexity of the algorithm, a composite-vertex-weight can be employed to replace the weight vector. This composite vertex weight is a combined evaluation of the cost of a vertex and is calculated by:

$$w^{c}_{i} = \sum_{j=1}^{n} \epsilon_i w^{i}_{j}$$  \hspace{1cm} (1)$$

where $i=1,2,...,n$, $n$ is the size of a weight vector, i.e. the number of costs in the vector. $\epsilon_i$ is the importance factor of the $i^{th}$ weight in the vector and is assigned based on the scarcity of the corresponding resource. The scarcer a resource is the higher its corresponding importance factor is. This can to some extent prevent the scarce resources to be used up quickly because the exhaustion of a certain resource might lead to the failure of initiation of a new user application. Figure 1(b) shows the vertices with composite-vertex-weights when $\epsilon_1 = \epsilon_2 = \epsilon_3 = 1$. 
3.2. Problem Formulation

Given an application’s DMC graph \( G = (V,E) \) and a nonnegative integer \( k' \), the adaptive \((k+1)\) multi-constraint partitioning algorithm is to find out one unoffloadable partition \( V^o \) and \( k \) disjoint unoffloadable partitions \( V_1^o, V_2^o, ..., V_k^o \) satisfying:

\[
\text{i) } \bigcup_{i=1}^{k} V_i^o = V \bigcup V^o \quad \text{and} \quad V_i^o \cap V_j^o = \emptyset \quad \text{for} \quad 1 \leq i, j \leq k \quad \text{and} \quad i \neq j; \\
\text{ii) the edge-cut of } V_i, V_j \text{ for } V_i, V_j \not\subseteq \{V^u, V_1^o, V_2^o, ..., V_k^o\} \text{ is minimized subject to the constraints defined by iii), i.e.,}\]

\[
C_i(j) = \sum_{(u,v) \in E(u,v)} \omega(u,v) \quad (i,j) \\
\]

is minimized subject to the constraints defined by iii), i.e.,

\[
\text{iii) } \forall i, j : \sum_{\delta_i^j} \leq (T_i^j \pm \delta_i^j) , \text{ where } \delta_i^j \text{ is the sum of } j^{th} \text{ vertex-weight in partition } V_i, \text{i.e., } \\
\delta_i^j = \sum_{uv \in \delta_i^j} \omega(u,v) \\
T_i^j \text{ and } \delta_i^j \text{ are the multiple constraints that are predefined to represent the threshold and the partitioning fluctuating factor in partition } V_i \text{ with respect to the } j^{th} \text{ vertex weight. } \\
T_i^j \text{ and } \delta_i^j \text{ define the lower and upper bound of the constraints respectively.}
\]

When a composite vertex weight is utilized in the DMC graph, equation (3) becomes

\[
\delta_i^j = \sum_{uv \in \delta_i^j} \omega(u,v) \\
\]

The correspondent composite threshold and partitioning fluctuating factor of partition \( V_i \) are \( T_i^j \) and \( \delta_i^j \) respectively to represent the above composite constraints in the rest of the paper.

3.3. The Adaptive \((k+1)\) Multi-Constraint Partitioning Algorithm

The algorithm consists of three main steps: unoffloadable vertices merging, coarse partitioning, and partition refinement. We discuss these three steps in detail as following.

1) Unoffloadable Vertices Merging: An unoffloadable vertex is a vertex that has special features making it unable to be migrated outside the mobile device and therefore located only in the unoffloadable partition. All the unoffloadable vertices need to be merged into a multinode \( uv \). The vertex weight vector of \( uv \) is the sum of all the unoffloadable vertices’ weight vectors, e.g., the \( j^{th} \) weight of \( uv \) is \( w_{ij}^u = \sum_{v \in \text{unoffloadable vertices}} w_{ij}^v \). The edges connected to \( uv \) are the unions of the edges of the unoffloadable vertices being merged.

2) \((k+1)\) Coarse Partitioning: Let the DMC graph, with all the unoffloadable vertices merged, be the original graph \( G_0 \). The task of this step is to coarsen \( G_0 \) to the coarsest graph \( G_c \), such that \( |G_c| = k+1 \) and all the vertex-weights are fulfilling the multiple constraints defined. The multinode \( uv \) from the previous step is treated as a normal vertex during coarsening. The coarsest graph \( G_c \) consists of \( k+1 \) multinodes. The multinode \( uv \) is the unoffloadable partition \( V^u \) and the others are the \( k \) offloadable partitions: \( V_1^o, V_2^o, ..., V_k^o \).

\((k+1)\) Coarse Partitioning:

1. begin
2. Define multi-constraints of partitions: \( T_i^j, \delta_i^j, T_i^j[k] \) and \( \delta_i^j[k] \);
3. \( \Psi^w = 0; \Psi^w[1] = ... = \Psi^w[k] = 0; \) // sum of vertex-weights
4. Assign the multi-node into \( \Psi^w \); \( \Psi^w \)
5. while \( |V| \times k+1 \) do { // it is not coarse enough
6. Assign \( k \) weightiest multinodes as \( \Psi^w[1], ..., \Psi^w[k] \);
7. if \( \Psi^w \neq 0; \Psi^w[1] = ... = \Psi^w[k] = 0 \) // sum of vertex-weights
8. Assign the multi-node into \( \Psi^w \); \( \Psi^w \)
9. for \( i=1 \) to \( k \) { // check the \( k \) partitions
10. if \( \Psi^w[i] \geq (T_i^c[i] - \delta_i^c[i]) \) then Mark \( \Psi^w[i] \) as matched; 
11. \{ if not hasUnmatchedVertex() then Partition Failure; \}
12. Invoke HELVM algorithm to coarsen graph;
13. Update \( \Psi^w, \Psi^w[1], ..., \Psi^w[k] \) to add new merged vertices;
14. \}
15. end

In line 7 and 9, the total weight of each multinode (i.e. partition) is checked to see if it has already reached the lower bound of the multi-cost constraint. If yes, it is marked as matched and no more vertices will be added further. If all partitions satisfy the cost constraints and there is still unmatched vertex left, the partition is a failure which means the partitions can not be found under predefined multi-constraints. In this case, either constraints need to be tightened or the application executes without being offloaded.

In line 11, the heavy-edge and light-vertex matching (HELVM) algorithm is invoked for graph coarsening, which will be discussed in Section 3.4.

3) Partition Refinement (Optional): The coarsely partitioned DMC graph as a result of Step 2 can be further refined by a lightweight refinement algorithm. The basic idea of the refinement is to try to move the vertices lying at the boundaries of the partitions. If the move of certain boundary nodes reduces the edge-cut while still keeping the related partitions satisfying the upper bound of the multiple cost constraints, the move is proceeded.

For running the algorithm in resources severely constrained mobile devices, such as a mobile phone, this
optional step can be ignored to reduce overhead. Due to
the page limit, the detailed discussions are not presented.

3.4. The HELVM Algorithm

In this sub-section, the heavy-edge and light-vertex
matching (HELVM) algorithm, which is used in step 2 (i.e.
coarse partitioning), is discussed in detail.

During graph coarsening phase, a sequence of
successively coarser graphs \( G_1, G_2, ..., G_n \) are constructed
from the original graph \( G_0 \) such that \( |V_{i,j}|<|V_i| \), i.e. the
number of vertices in successively coarser graphs is
smaller. Two main approaches have been proposed in [7]
for coarsening a graph. The first approach is to merge the
highly connected vertices into a multinode, while the second
one is to find a matching and then to collapse the
matched vertices into a multinode. These two approaches
are adapted in HELVM algorithm, which coarsens a graph
by collapsing the heavy edges.

A matching of a graph is a subset of edges with no two
edges incident upon a same vertex. The task of finding a
maximum matching is to select a maximum subset of such
edges. The coarser graph \( G_{i+1} \) is constructed from \( G_i \) by
finding a match of \( G_i \) and collapsing the matched vertices
into multinodes. The unmatched vertices are simply
copied over to \( G_{i+1} \). Since the goal is to decrease the size
of the graph \( G_i \), we are trying to find the maximum
matching of the graph \( G_i \).

For finding maximum matching, Karypis and Kumar
proposed Random Matching (RM), Heavy Edge Matching
(HEM), Light Edge Matching (LEM) and Heavy Clique
Matching (HCM) in [7, 9]. All these heuristics only
considered the weights of edges. In the context of multi-
constraint partitioning, they are not sufficient as most of
the constraints are also related to vertex-weights. The work in [6]
considered the vertex-weights; however, its
focus is to select pairs of vertices with minimized difference for matching, so as to balance constraints in
partitions.

In offloading systems, we aim to keep highly connected
vertices in one partition to reduce communication costs
between partitions. In the HELVM algorithm, the heavy-
edge means the incident vertices are highly connected;
whereas the light-vertex implies more vertices are going to
be merged under specified constraints. The basic idea of
the algorithm is: when we select an edge for matching,
instead of only comparing the edge-weight, we also
compare the vertex-weights of the incident vertices. We
use a composite-weight to represent the weight of an edge
and its incident vertices. The composite-weight of vertex \( v \)
in relation to vertex \( u \) is:

\[
CW(u, v) = \lambda_w w(e_{uv}) + \lambda_\lambda w
\]

where \( w(e_{uv}) \) is the edge-weight, and \( w \) is the composite-
vertex-weight of \( v \) calculated by eq.(1): \( \lambda_i (i = 1, 2) \)
\( 0 \leq \lambda_1, \lambda_2 \leq 1 \) and \( \lambda_1 + \lambda_2 = 1 \) are the importance factors
of edge-weight and vertex-weight, respectively. If \( \lambda_2 = 0 \),
the algorithm becomes heavy-edge matching.

If vertex \( v \) is selected to match with \( u \) due to the
composite-weight \( CW(u, v) \) as maximum, then \( v \) is called
the tightest-and-lightest vertex in relation to \( u \). If there is
more than one vertex in relation to vertex \( u \) that has the
same maximum composite-weight, then one of them is
selected by the following approach: let \( H \) be the set of
such tightest-and-lightest vertices, the vertex \( v \in H \) is
selected if

\[
AdjW(u, v) = \sum_{(v, y) \in E(u, y) \in E} w(e_{uv})
\]

is maximized; i.e. the weight sum of the edges that
connect \( v \) to the vertices which are also adjacent to \( u \) is
maximized. That means we need to choose one not only
tightly linked with vertex \( u \) but also tightly linked with the
vertices adjacent to \( u \). If still more than one is found, the
first one or a random one in set \( H \) will be selected.

The HELVM algorithm is described as follows:

The HELVM Algorithm:

1. begin
2. Mark all vertices of vertex set \( V \) as unmatched;
3. while hasUnmatchedVertex() do |
4. \( u = \) RandomSelectUnmatchedVertex();
5. \( v = \) GetTheTightestLightestVertex(u, V);
6. if (v!=null) then |
7. Put edge \( (u,v) \) into the matching;
8. Mark \( v \) as matched vertex;
9. Mark \( u \) as matched vertex;
10. }
11. end

The \( \) GetTheTightestLightestVertex function is used to
select the tightest and lightest vertex, which is
implemented as follows:

The Tightest and Lightest Vertex Selection Function:

1. Function GetTheTightestLightestVertex(u, V)
2. input: \( u \rightarrow \) the given vertex; \( V \rightarrow \) the vertex set
3. output: the tightest and lightest vertex in relation to \( u \)
4. begin
5. \( Adj[u] = \) GetUnmatchedVerticesAdjacentTo(u);
6. CurrentTightestLightestVertex = null;
7. CurrentMaxCW = 0; // the maximum composite-weight
8. NumberOTightestLightestVertex = 0;
9. \( H=null; // \) The TightestLightestVertex set, \( H \)
10. for i = 1 to \( |Adj[u]| \) |
11. \( CW(u, Adj[i]) = \) GetCompositeWeightOf (u, Adj[i]);
12. if \( CW(u, Adj[i]) \rightarrow \) CurrentMaxCW then |
13. \( \) if \( CW(u, Adj[i]) \rightarrow \) \( \) CurrentMaxCW then |
14. \( \) CurrentMaxCW = \( CW(u, Adj[i]);
15. \( H = null; // \) Clear the TightestHeaviestVertexArray;
16. \( \) NumberOTightestLightestVertex = 1;
17. }
18. \( \) NumberOTightestLightestVertex ++; // more then one
The vertices being matched will be collapsed into
multinodes. The weight vector of the multinodes are set
equal to the sum of the weight vectors of the vertices
being merged. Meanwhile, for keeping the connectivity
information in the coarser graph, the edges of a multinode
are the union of the edges of vertices being merged.

Figure 2. An Example of HELVM

Figure 2 shows an example of how to select the tightest-
and-lightest vertex for matching. The vertices in the
circled areas by dotted lines are already matched. There
are still five unmatched vertices: c, d, e, f, and g. Suppose
vertex d is randomly selected now, one of the unmatched
adjacent vertices: c, e, f, and g, will be selected to match
with d. As shown in the figure, the composite-vertex-
weight of d: \( w^{\text{composite}}_d = 2 \). By using (5), we
calculate \( CW(d,x) \), for \( x \in \{c,e,f,g\} \), to select the
tightest-and-lightest vertex. Let \( \lambda_i = 0.5 \) (i.e. the
edge-weight and the vertex-weight are with same
importance), the composite weight of the four candidate
vertices in relation to d are calculated out:

\[
\begin{align*}
CW(d,c) &= 0.5 \times 1 + 0.5 / 2 = 0.75 \\
CW(d,e) &= 0.5 \times 2 + 0.5 / 2 = 1.25 \\
CW(d,f) &= 0.5 \times 2 + 0.5 / 1 = 1.5 \\
CW(d,g) &= 0.5 \times 2 + 0.5 / 1 = 1.5
\end{align*}
\]

We need to select the maximum one as the tightest-and-
heaviest vertex for matching. As can be seen, in relation to
d, there are two tightest-and-heaviest vertices: f and g.
Equation (6) will be used for further selection. For vertex f,
it is connected to e and g which are adjacent to d, so:

\[
\text{AdjW}(d,f) = w(e_f) + w(e_g) = 2 + 1 = 3.
\]

For vertex g, it is connected to b and f which are adjacent
to d, so:

\[
\text{AdjW}(d,g) = w(e_g) + w(e_f) = 1 + 1 = 2.
\]

Finally, the vertex f is selected as the tightest-and-heaviest
vertex and the edge \((d,f)\) will be put into the matching.

4. Implementation

We developed an offloading toolkit to implement the
\((k+1)\) partitioning algorithm, which runs on Java Virtual
Machine (JVM). The toolkit consists of three main
modules: Resource Monitor Module, Partitioning and
Offloading Module, and Remote Communication Module.
The resource monitor module monitors the system runtime
environment, which captures the memory usage, CPU
usage, and network bandwidth usage. The remote
communication module handles communication between
mobile device and surrogates. The partitioning and
offloading module implements the \((k+1)\) partitioning
algorithm, which dynamically partitions a Java application
and offloads the offloadable partitions to surrogates.

In order to offload a Java class to a surrogate, an
offloadable class needs to be instrumented. A dedicated
proxy class is generated automatically as a shadow class to
the instrumented class. The name of the proxy class will
use the original class’ name; whereas, the name of the
instrumented original class will be changed.

Figure 3. Class Invocation Procedures

Figure 3 illustrates class invocation procedures. The
arrows represent the invocation directions. 3.a shows a
normal (i.e., non-offloading) inter-class invocation: Class
App directly invokes the methods in the Class Encode
and gets the results directly from it. In 3.b, the original class
Encode is instrumented as I_Encode and a proxy class
using the original class name Encode is generated. All
invocations from Class App to Class I_Encode are via the
proxy class Encode. The instrumented class may be
offloaded to a surrogate as shown in Figure 3.c. In this
case, the invocation will be performed through remote
communication module.

Figure 4 shows an example of a class instrument. The
function OffLoadingCall in the proxy class takes the
object’s name, method’s name, and the parameters for
invoking the instrumented class.

As the source codes of applications running on mobile
devices are usually unavailable, an ideal offloading system
should be able to work on an application’s binary code.
Some approaches have proposed for byte-code
modification, such as Javassist [16] and BCA [17]. Our partitioning and offloading module is developed based on these works. Java byte-codes are directly loaded without the knowledge of source codes. For explanation convenience, we use Java source code in Figure 4.

The first step of our offloading algorithm is to construct the multi-cost graph of the application. As discussed in Section 3, the costs of each application component are changing during application executions. In general, there are two approaches to weigh the costs of application components: online-profiling and offline-profiling. The latter uses a pre-defined profile to describe the resource consumption costs of each application component. The profile needs to be defined in design time and with the knowledge of source code. The advantage of offline-profiling is -- it is simple to implement and it is necessary for pre-execution offloading, if the application is too big to load; but the drawback is it does not reflect to the dynamic cost changing during application execution. On the other hand, the online-profiling approach generates the multi-cost graph dynamically by monitoring the real time execution environment. It does not need the knowledge of source code. The demerit of online-profiling is it causes extra overhead due to its real-time and dynamic nature.

As we execute adaptive and dynamic offloading, the partitioning and offloading are based on the runtime resource snapshots. The interval of taking the snapshots can be smaller, e.g., a few million seconds for precise evaluation; however, it will increase the overhead. In our toolkit, we take the snapshots based on event-driven and complement it with a periodical time-trigger with the interval in a few seconds. If there is no significant resource usage change, it periodically takes snapshots. If significant resource utilization fluctuation occurs, the toolkit takes a snapshot immediately. Following approaches are used to obtain the cost value of edges and vertices in the multi-cost graph.

- **Invocation and Data Access Number** (i.e. edge-weight):
  The number of invocation between classes is relatively easier to measure. We monitor the invocation stacks to get all method invocations. As the partitioning granularity is class, only the interactions between classes are recorded. We do not distinguish the direction of invocations. For example, there are two methods \(a\) and \(b\), which are located in two classes \(A\) and \(B\), respectively. Let \(I_{a \to b}\) be the set of method invocations from \(a\) to \(b\), the invocation number between \(A\) and \(B\) is the cardinal of \(I_{a \to b} \cap I_{b \to a} \mid a \in A \land b \in B\). The number of data accesses is obtained in the same way.

- **Memory Utilization**: The memory utilization of a class changes during execution. The resource monitoring module obtains the memory utilization of each class by monitoring and taking snapshots of the JVM heap.

- **CPU Utilization**: It is difficult to measure CPU utilization. For a given time slot, if only one class is running, the CPU utilization can go to 100%. If two classes are running and the CPU utilization is 100%, it is unreasonable to say that the classes’ CPU utilizations are 50% each or in some kind of ratio. Obviously, the percentage of CPU occupied is not a good way to represent a class’s CPU utilization. We use an objective and simple approach to tackle this issue. It is to calculate the cumulated CPU occupying times of a class during the snapshot intervals and formulate it as the CPU processing cost of this class.

- **Bandwidth Utilization**: Network bandwidth utilization comes from two aspects: data access between the mobile device and surrogates, and the mobile device exchanging data with remote hosts (e.g., downloading or uploading). Through remote data access monitoring, the bandwidth usage of each class is obtained.

### 5. Experiments and Evaluations

We conduct the experiments by using an IBM laptop X31 (PM-1.5GHz, 256M) to simulate a mobile device; and two desktop PCs (P4-2.4GHz, 1G RAM) are employed as surrogates. The laptop is equipped with IEEE 802.11b wireless network adaptor. An access point (AP) is wired connected to the desktop PCs and they are connected to the campus’ intranet by a 100Mbps switch. The laptop accesses the AP wirelessly. Both the client and surrogates are running the offloading toolkit on Sun’s J2SE JVM version 1.4.1. The operating systems are Redhat Linux 9.

#### 5.1. Experiment 1: \(\pi\) Calculator

We develop a \(\pi\) calculator for testing the offloading tool kit. It consists of two Java classes: \(PiCalculator\) and \(Pi\). The class \(Pi\) computes the value of \(\pi\), whereas the class \(PiCalculator\) handles the graphic user interface (GUI) which gets user’s input of the accuracy (i.e. how many places of decimals) of the \(\pi\); invokes the \(\pi\) calculation function in class \(Pi\); and finally shows the result.
Figure 5. π Calculator Offloading Experiment

Figure 5 shows the time, memory and CPU usage for π calculation, respectively. The y-axes represent the resource usage and the x-axes represent the accuracy (i.e., the places of decimals). The π calculator is run in three cases: mobile device only (Non-Offloading), using the offloading toolkit (Offloading), and entirely running in surrogate (Surrogate-Only). The curves in 5.a show that the response time of non-offloading is the slowest; it gets faster in offloading case due to the class Pi offloading to the surrogate; the surrogate-only case gets the quickest response. Figure 5.b and 5.c show that the CPU and memory usages in the mobile device are significantly decreased in the offloading cases. The reason of higher memory usage of the surrogate-only case is due to larger JVM heap size set for providing more memory space for offloading in the surrogate.

As can be seen in the figure, the offloading toolkit itself caused some overhead. If the places of decimal we intend to calculate is smaller than 100, the time used for offloading is longer than non-offloading. And it is the same for memory utilization and CPU utilization if the places of decimal are smaller than 80 and 50, respectively.

5.2. Experiment 2: M4Play Partitioning

As multimedia applications become popular in mobile devices, we use a Java MPEG-4 player, M4Play, to conduct application partitioning algorithm test. M4Play is a MPEG-4 video/audio player provided by IBM Toolkit for MPEG-4 [21].

We use our offloading toolkit to load the IBMToolkitForMpeg4.jar file. No source code is provided for this toolkit. The toolkit directly loads binary bytecode from the jar file. There are totally 1871 classes included in this jar file. Most of the names of classes are obfuscated to meaningless strings (as shown in Figure 6).

Since the package also included classes for audio/video generation and interaction video processing, only 374 classes are involved for the purpose of MPEG-4 video clip playing.

There are 85 system classes, including: java.io.*, java.lang.*, javax.imageio.*, java.awt.image.*, javax.sound.* and java.util.*. For the sake of showing the relationship between the system classes and the non-system classes, we keep all these system classes in the graph. There are totally 49 unoffloadable classes filtered out according to unoffloadable rules.

Figure 6 shows a snapshot of the multi-cost graph of the M4Play. The black-filled circles represent the unoffloadable classes, whereas the grey filled circles represent the offloadable classes. Each class annotated with the class’ name and weight vector. For example, as an arrow pointed in the bottom-right mid of the graph, there is an offloadable class named avc which is annotated with a 3-tuple cost value <16,64,0>; they represent the accumulated CPU time (in the unit of ms), the memory usage (kb), and the bandwidth usage (kb/s), respectively. Each edge in the graph is labelled with its weight; it represents the number of interactions and data access between two classes. Let k equal to 2, our (k+1) partitioning algorithm partitions the multi-cost graph of M4Play into a unoffloadable partition which is running in the mobile device, and two partitions for running in two surrogates. Figure 6 also shows the partitioning. The multiple constraints and partitioning parameters are listed in the right-top corner. All the constraints are using composite-weights. The edge-cuts between three partitions are: C((U, O)C3 1 2 = 12(O, O)C2 1 0 = 312, C((U, O)C3 2 6 = 126, and C((U, O)C3 1 4(U, O)C1 2 6 = 210.

5.3. Experiment 3: MP4GenPlayer Partitioning

In this experiment, we compare our HELVM algorithm with Random Matching (RM), Heavy Edge Matching (HEM), Light Edge Matching (LEM), and Heavy Clique Matching (HCM) algorithms by partitioning applications in which the constituent classes have different features. For example, some classes are computational intensive, but some of them are memory or bandwidth intensive. We designed a Java program, MP4GenPlayer, which integrates the functionalities of M4Play and Avgen (an MPEG-4 audio/video generator in IBM Toolkit for MPEG-4 [21]) and it also includes some dedicated classes to download files from a remote host. The program executes the following sequences: 1) downloads a video sequence and audio sequence from a remote host; 2) generates a MPEG-4 format video clip from the downloaded sequences; 3) plays the MPEG-4 clip.

The program includes classes with different intensives. The classes involving MPEG-4 generation are computational and memory intensive. The classes for
remote file downloading are bandwidth intensive. The classes for MPEG-4 playback are computational intensive. Different matching algorithms are implemented in our offloading toolkit to partition MP4GenPlayer. For comparing with min-cut (it only partitions an application into two partitions), \( k \) is set as 1. The partition parameters are set as following: \( T^U = 1000-5000 \), \( \delta^U = 200 \), \( T^O = 25000 \), \( \delta^O = 300 \), \( \lambda_1 = \lambda_2 = 0.5 \), \( \epsilon_1 = 0.35 \), \( \epsilon_2 = 0.35 \), \( \epsilon_2 = 0.3 \).

Figure 7 shows all the edge-cuts with the change of \( T^U \). The values of RM are big and without regularity; this reflects its random-selecting feature. The edge-cuts of HEM are getting bigger when \( T^U \) becoming larger. LEM, HCM, and min-cut cannot get small edge-cuts. Due to the HELVM selecting the tightest and the lightest vertex for matching, it gets significantly lower edge-cuts under all different threshold settings.

Figure 8 shows time responses by using different algorithms for running MP4GenPlayer to download, generate, and play a 10-second audio/video clip. For non-offloading, all the tasks including clips downloading, MPEG-4 clip generating, and playing are performed in mobile device; it takes 122 seconds. By using offloading, the time of response is decreased. Excepting RM, the other five algorithms are significantly reducing the response time. HELVM gets the quickest response, 25 seconds, and it is almost keeping constant during the \( T^U \) changing.

6. Conclusions and Future Work

In this paper we proposed an adaptive partitioning algorithm, as an important part of offloading algorithms for applications operating in pervasive computing environment. Lying at the core of the so-called adaptive \((k+1)\) partitioning algorithm is the heavy-edge and light-vertex matching (HELVM) that takes into consideration...
both edges and vertices when deciding a merger. Inspired by multi-level heuristic algorithms for VLSI and specifically motivated for pervasive computing, our algorithm considers different types of costs associated with individual application classes (including CPU cycle, memory and bandwidth) and amongst these classes (mainly invocation frequency). Moreover, these costs are gathered on-the-fly by our offloading toolkit instead of being predefined at the application development stage. The fact that it works on Java byte-code instead of source code increases the confidence in its feasibility. The intensive experiments and evaluations have illustrated its feasibility, adaptability and high-performance.

Remote execution and remote communication will inevitably increase the risk of class and application failure. As such an effective and efficient fault-tolerant mechanism is highly needed for offloading systems for pervasive services and applications; also, security and trust need to be taken into account. Our future is to investigate how to adapt the traditional security and checkpoint mechanisms for the benefit of wireless applications running on resource-constrained mobile devices and how to integrate them with our adaptive \((k+1)\) partitioning algorithm. To further reduce the computational and communication complexity of the adaptive \((k+1)\) partitioning algorithm is also within the scope of our future work.

7. References