Method Partitioning - Runtime Customization of Pervasive Programs without Design-time Application Knowledge

Dong Zhou, Santosh Pande, Karsten Schwan

College of Computing

Georgia Institute of Technology

Atlanta, GA 30332

{zhou,santosh,schwan}@cc.gatech.edu

Abstract

Heterogeneity, decoupling, and dynamics in distributed, component-based applications indicate the need for dynamic program customization and adaptation. *Method Partitioning* is a dynamic unit placement based technique for customizing performance-critical message-based interactions between program components, at runtime and without the need for design-time application knowledge. The technique partitions message handling functions, and offers high customizability and low-cost adaptation of such partitioning. It consists of (a) static analysis of message handling methods to produce candidate partitioning plans for the methods, (b) cost models for evaluating the cost/benefits of different partitioning plans, (c) a *Remote Continuation* mechanism that "connects" the distributed parts of a partitioned method at runtime, and (d) Runtime Profiling and Reconfiguration Units that monitor actual costs of candidate partitioning plans and that dynamically select "best" plans from candidates. A prototypical implementation of Method Partitioning in the JECho distributed event system is applied to two distributed applications: (1) a communication-bound application running on a wireless-connected mobile platform, and (2) a compute-intensive code mapped to power- and therefore, computationally limited embedded processors. Experiments with Method Partitioning demonstrate significant performance improvements for both types of applications, derived from the fine-grain, low overhead adaptation actions applied whenever necessitated by changes in program behavior or environment characteristics.

Keywords: Mobile code, application partitioning, distributed adaptation, program analysis

1 Introduction

Heterogeneity, decoupling and dynamics are common characteristics of many distributed applications. This is particularly the case for applications that are structured with dynamic participants using communication paradigms like distributed messaging or publish/subscribe[1, 2, 3, 4]. Such applications exhibit heterogeneity in application components' behaviors and in their communication and computation resources. Components of such applications can be decoupled in space (distributed), time (asynchronous) and design (components from different sources). Such applications are also highly dynamic, as participants can come and go and in that their components run in open environments subject to frequent changes in resource availability.

Heterogeneous, dynamic resources and the decoupled design of components indicate that it is impractical to optimize such applications at design time. Further, application dynamics suggest that optimizations must be delayed beyond deployment time, as well. Runtime component customization and adaptation, therefore, are necessary ingredients of their execution frameworks.

Our work focuses on certain performance-critical elements of distributed, pervasive applications, namely, on their component interaction. Sample components and applications addressed include (1) those that implement sensor data transfers across wireless links, implying relatively lightweight processing actions like data reduction[4, 5], and (2) compute-intensive pervasive applications, where data must be processed for presentation to select clients, as in radar data processing or target identification[6]. For such applications, past work has used dynamic unit placement as one key technique for runtime adaptation[18], where a unit might be a function[36, 39, 7], a query[33, 3], an object[35, 4], or an agent[12, 9, 13, 10, 14, 16, 15, 11]. While work on adaptive systems and adaptation frameworks has addressed the algorithms and mechanisms suitable for a range of applications [17, 18, 19, 20], for the distributed, component-based systems driving new applications like cellphone-based video conferencing, ubiquitous presence, and remote sensing, several additional issues must be addressed. Focusing on the dynamic placement of units as one important customization method, the goal of our work is to achieve:

- *Application Independence and Late Binding.* While embedding adaptation into application can take advantage of application specific adaptation, it can potentially complicate application development[17]. More importantly, it usually makes assumption on runtime resource constraints, and is consequently less desirable for applications composed of independently developed components running in dynamic pervasive computing environments. For dynamic unit placement based systems, it may not be ideal to isolate, at design time, units suitable for dynamic replacement, as it is hard to anticipate all possible resource constraints without spending considerable extra effort. One of our goals thus is to delay the identification of code modules that may be moved and the association of code that performs such adaptations with the applications being adapted from design time until the time an application is deployed.

- *Fine-grain Customization.* Units in unit-based dynamic placement are usually undivisible. In comparison to agent-based systems, for instance, where an agent migrates or stays as a whole, we aim to dynamically identify the code portions to be placed or re-placed, using a compiler-based approach.
- *Low Adaptation Cost.* In mobile agent systems, migration implies the need to save, transport, and restore an agent's static and dynamic states, which could be costly. In our approach, static code analysis is coupled with dynamically used "plans" (see below) to reduce such costs.

Our approach is based on the concept of *Method Partitioning*, targeting methods that handle messages used for component interactions. Partitioning implies splitting the processing of a message into two parts, one executed inside the sender of the message, the other in the message recipient. A method can have multiple candidate *partitioning plans* that jointly define the space of available choices for dynamic customization and adaptation. Switching plans is as efficient as changing flag values. Candidate partitioning plans are generated by static program analysis, and the only programmer input required is the specification of a customization criterion (e.g., minimizing network traffic). Such criteria are represented as *cost models*, used in static analysis to produce candidate partitioning plans and used during runtime reconfiguration to evaluate which partitioning plans to use. Finally, dynamic inputs to runtime reconfiguration are provided by a Runtime Profiling Unit.

The implementation of Method Partitioning described in this paper is hosted in the JECho distributed event system[4, 5]. Since method partitioning is applied to the "handlers" operating on the events/messages being sent and received, this technique can be applied to any message-based implementation of a component-based system, ranging from research systems like Cactus[25]to commercially available systems like JMS-based business applications.

Method Partitioning is evaluated for both communication-bound and compute-bound applications. Our first application emulates media playback in wireless environments. A stationary server sends image frames to a handheld/laptop client through a wireless network. This application is communication-bound because of a mismatch in the amount of data provided by the server and the speed of the network used for its transfer. Realistic applications emulated by this scenario include the real-time display of video on handhelds[30].

The message handlers to which Method Partitioning is applied dynamically resize and/or downsample the data being sent, to customize image handling to different client needs and to dynamic changes in network capacity.

Our second application emulates mobile sensors that have the ability to process captured data prior to sending it to clients[21, 22]. This application is CPU resource-bound because of the potential complexity of such processing[6, 29]. Processing loads change dynamically either because of changes in the complexities of signals (e.g., the amounts of "interesting" vs. "uninteresting" data currently captured) or because of competition from other applications. Method partitioning is used to evenly distribute processing load across the sensor and client(s) depending on their respective current computational capabilities. The goal is to minimize the average data processing time.

Experiments demonstrate significant performance improvements derived from using Method Partitioning. Results show that compared to statically optimized program configurations, Method Partitioning (1) matches the performance of such manually optimized implementations, and (2) outperforms other non-optimized manual implementations by as much as 223%. More importantly, when resources or application behavior change dynamically, experiments demonstrate that Method Partitioning provides performance improvements by 22% to 305% compared to implementations that cannot adapt to such changes.

The remainder of this paper is organized as follows. Section 2 presents the framework of Method Partitioning. For better readability, details of the static analysis part of the framework appear separately in Section 3. Section 4 uses two sample cost models, one defining cost as data size, the other focusing on program execution time. These models are then used with the two applications described in Section 5, to evaluate Method Partitioning. The paper concludes with related work and directions for future research.

2 Method Partitioning

2.1 Model and Approach

Method Partitioning is based on splitting the handling of a message into two parts, with the first part executed inside the sender of the message (we name this part *modulator*), and the other part (named *demodulator*) running inside the receiver. When a message is sent to a receiver, the message is first touched by the sender using the receiver's modulator, and any data emitted by the modulator is sent and then touched by the demodulator in the receiver. A receiver can apply handlers to messages received from

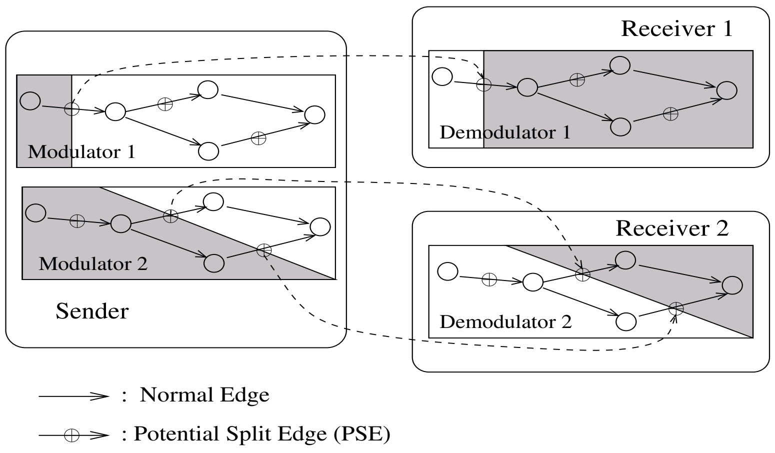

Figure 1: Illustration of Method Partitioning with one message sender and two message receivers. The shaded areas denote actual current partitioning plans.

multiple remote components, and a single method handler can be used to handle messages from multiple senders. Finally, since a sender can send messages to multiple receivers, multiple modulators (some of which may be derived from the same handling methods) may reside in a single sender.

Partitioning uses the Unit Graphs (UGs) of message handlers. A UG is similar to a Control Flow Graph except that each node is an instruction instead of a basic block. In the UG of a modulator, "along" the processing path of a message within a modulator, there are 0 to n Potential Split Edges (PSEs). A PSE is where the modulator side processing of message could end, and further processing of the message continue in the demodulator. For each PSE, there is a dedicated flag controlling whether actual splitting of the processing will happen there. When the flag is set, a special ID for the PSE is sent to the demodulator along with any relevant data. The demodulator uses this special ID to determine the location at which to continue the processing of the message. At any given time, the set of PSEs with their flags set comprise the actual partition of the handling method (see Figure 1).

An actual partitioning at a certain instance of time defines the current customization of the interaction between the message sender and receiver. Changes of the actual partitioning over time reflects adaptation to application and environment dynamics. Costs of partitioning plans are evaluated at runtime with regard to cost model, which is specified at component deployment time. The goal of Method Partitioning is for any actual partitioning having the lowest cost with respect to cost model and current environment. It has four constituents:

- *Static Analysis* identifies from all theoretically possible partitioning plans those plans that can possibly lead to actual runtime partitionings. It achieves this by statically identifying PSEs. Specifically, it uses the cost model associated with Method Partitioning to estimate the cost value of each edge in the UG of the message handling method, and it marks each edge as a PSE that can potentially have the least cost among all of the edges along any one of the valid execution paths. Static analysis also generates the modulator/demodulator pair from the handling method, which involves inserting profiling and other code along each PSE.

- *Cost Models* capture the costs of component interactions implied by method partitioning. Different sender/receiver pairs may choose different cost models. For example, a sender/receiver pair in a weakly connected environment might define the amount of data passed between them as cost, while a sender/receiver pair in a CPU-constrained environment may use processing time as cost. Stated more precisely, cost models capture the costs of edges in the UG. Different message handlers may use different cost models. An example is a model that uses the total size of variables passed along an edge as the cost of that edge. Since the costs of some edges cannot be determined at compile time, runtime profiling is required.

- *Remote Continuation* is a mechanism[24] for continuing the processing of a message in a demodulator after it has been partially performed in a modulator. This involves identifying the data the modulator must hand over to the demodulator, and it involves defining how appropriate state should be restored from this data.

- *Runtime Profiling and Reconfiguration Units* are subsystems that uses profiling code embedded in the modulator/demodulator pair to measure the runtime costs of those PSEs whose costs cannot be determined statically. Significant changes in such costs trigger a max-flow algorithm to re-select a (near) optimal partition by setting the flags of the set of PSEs of that partition.

We next discuss in more detail cost models, PSEs, and their use for dynamic partitioning.

2.2 Cost Models

Cost Models are used to determine the costs of edges, and edge costs determine the costs of partitioning plans. Such costs cannot be determined at compile time for two reasons: (1) the cost of an edge may not be statically determinable since it depends on actual runtime use, and (2) the probability of going through

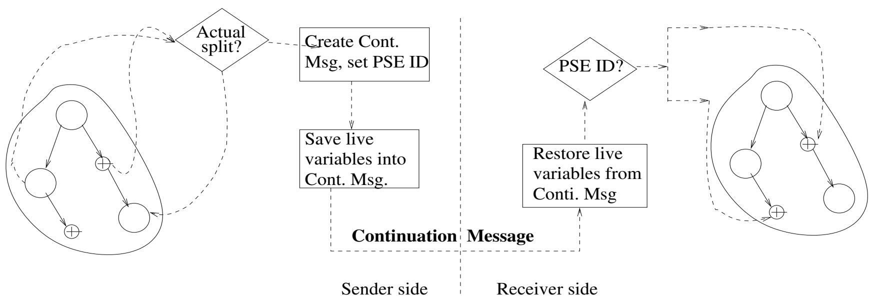
Actual split?
Create Cont. Msg, set PSE ID
Save live variables into Cont. Msg.
Continuation Message
PSE ID?
Restore live variables from Conti. Msg
Sender side
Receiver side

Figure 2: Illustration of Remote Continuation.

a PSE in the partitioning plan may not be known. The challenges in designing a cost model is to help static analysis reduce the number of PSEs, and to help the Runtime Profiling and Reconfiguration Units efficiently calculate the runtime costs of PSEs.

Section 4 uses two sample cost models to further explain the role of cost models in Method Partitioning.

2.3 Static Analysis

Static analysis identifies PSEs and generates instrumented modulator/demodulator pairs. One challenge in static analysis is to minimize the overhead caused by method partitions. This has two implications, one being to minimize the number of PSEs, especially those with costs that cannot be determined statically, the other being to minimize the interactions between the modulator and the demodulator that are caused by method partitioning. Concerning the first point, our algorithm optimizes comparative costs. For the second point, we ensure that the algorithm generates only *convex partitions*, thereby preventing data from flowing back to the modulator from the demodulator. The details of the algorithm for identifying PSEs appear in Section 3.

Another element of generating a modulator/demodulator pair is to insert instrumentation code along each PSE. This code has two parts: the profiling code and the continuation code. The profiling code is cost model specific. For example, in the cost model capturing data size, profiling code uses a size calculation tool to derive the actual amount of data passed along a PSE. In the execution time cost model, profiling code measures execution time. Profiling code can also be used to collect statistical data about actual execution paths for path-sensitive optimization.

2.4 Remote Continuation

A Remote Continuation supports the seamless continuation of a “modulated” message’s processing in a demodulator. This includes the modulator side informing the demodulator where its processing stopped and identifying and transferring all variables required by the demodulator. Variable hand-over is based on live variable analysis. It is carried out by the continuation code, including marshalling and unmarshalling these variables. More specifically, static analysis inserts continuation code along a PSE on both the modulator side and the demodulator side. At the modulator side, when the split flag of this PSE is set, the continuation code packs live variables of the PSE (which, without any path-specific optimization, is the intersection of the OUT set of the *out* node of the edge with the IN set of the *in* node) along with the unique ID for the PSE into a continuation message. It then hands this message to the runtime system for delivery. At the receiver side, this continuation message is delivered to the appropriate demodulator. Upon receiving a continuation message, the demodulator side of the continuation code restores the values of live variables, jumps to the appropriate PSE, and continues processing. Figure 2 illustrates the process of Remote Continuation.

2.5 Runtime Profiling and Reconfiguration Units

The profiling code inserted by static analysis is invoked by the Runtime Profiling Unit. The invocation of such profiling code is conditional. For each PSE, there is another dedicated flag to control whether to execute the profiling code. The purpose is to avoid excessive overhead caused by runtime profiling. For some PSEs, continuous profiling is unnecessary because their runtime costs and/or their execution probabilities do not vary much. If profiling is expensive, such costs can be reduced by periodic sampling, at the expense of having less timely statistics.

In summary, the Runtime Reconfiguration Unit collects feedback containing profiling information from both the modulator and demodulator sides. It invokes a max-flow algorithm to re-select the optimal partitioning from the graph of PSEs when profiling data changes significantly. Finally, it sends a new partitioning plan to the modulator side.

The location of the reconfiguration unit is variable. It may reside with the modulator, the demodulator, or in a third party, the latter being appropriate when repartitioning requires large amounts of computation (e.g., where there are a large number of PSEs). Concerning program monitoring[23], the exchange of such information between the modulator and demodulator sides of an interacting component is activated by application-defined triggers. For example, an application can choose to send feedback only when a certain amount of time has elapsed (rate-triggered), or when the profiling data for one of the PSEs has changed significantly (diff-triggered). The intent is for Method Partitioning to follow the "best practices" determined by our past work on the efficient adaptation of distributed programs[19, 18].

2.6 Discussion

Several key elements of the Method Partitioning approach to optimizing distributed programs are:

- *Minimal design/deployment-time knowledge.* Since modulator/demodulator pairs are automatically generated by the compiler, the only application-level knowledge required for Method Partitioning is the cost model. The model is used to select appropriate modulator/demodulator pairs at deployment time.
- *Light-weight adaptation.* The adaptations involved in Method Partitioning have small overheads. Once the modulator has been sent to the message sender, there is no need for additional code migration, and adaptations simply involve changes to a few flag values. In comparison, in Mobile Agent systems, adaptation activation always involves transporting the code and state of a mobile agent[1].
- *High Customizability.* Method Partitioning provides almost "continuous" customizability, in terms of its ability to vary the amount of processing in message senders vs. receivers.

The next two sections first provide a detailed description of the static analysis algorithm, then discuss two specific cost models, one for minimizing network traffic, the other for minimizing application execution time.

3 Algorithm for Static Analysis

Static analysis identifies PSEs and generates instrumented modulator/demodulator pairs. The algorithm for marking PSEs uses both the UG and the Data Dependency Graph (DDG). It is based on a specified cost model, which implies that the generated modulator/demodulator pair is valid only for that cost model. One challenge is to minimize the overhead caused by method partitions. This involves (1) minimizing the number of PSEs, especially those with costs that cannot be determined statically, and (2) minimizing the remote interactions between the modulator and the demodulator. The algorithm address (1) by optimizing

[1]The remote continuations performed as part of Method Partitioning should not be counted as adaptation activation costs. Their counterpart in mobile agent systems is that data sent from a mobile agent to its docking station after migration.

```
Algorithm ConvexCut
Begin
1.  MarkStopNodes (ug);
2.  foreach Edge(out, in) in the ddg do
3.      foreach path p in ug that starts from in and ends at out do
4.          Mark each edge in p with infinite cost
5.      endfor
6.  endfor
7.  PSESet = null
8.  foreach TargetPath p do
9.      PSESet = PSESet + MinCostEdgeSet (p)
10. endfor
```

Figure 3: Algorithm for generating potential convex cuts.

comparative costs. Concerning (2), the algorithm's current implementation guarantees that all generated partitions are *convex.* By convex we mean that data always flow from the modulator to the demodualtor, never the other way around.

Analysis is based on the UG and the DDG. Each node in UG is a Jimple instruction instead of a basic block[27], and has a corresponding node in the DDG, and vice versa. Each node is also associated with an IN and an OUT set of live variables.

The following terms and definition are used in the algorithm description: *StartNode* is the starting node for the algorithm. It excludes instructions "before" it that are used for renaming parameters and global variables. *StopNode* is a node that has to reside at the receiver side. A node is a StopNode if the node is a return instruction, uses variable(s) that are mutable outside the event handler, or if it references native variables or invokes native methods. A *TargetPath* is a path in a UG that starts from StartNode, and ends at either the ExitNode or a StopNode, where none of the intermediate nodes are StopNodes. *Edge(out, in)* is a directed edge that goes from node *out* to node *in. INTER(e)* is the intersection of the *outset* of the *out* node, and the *inset* of the *in* node of edge *e. PSESet* is the result set of potential split edges.

The algorithm proceeds as follows. It first finds all StopNodes. It then marks with infinite costs those edges that could possibly cause reverse flows of data. The infinite values essentially prevent cuts occuring at these edges. The algorithm then looks for the PSESet from all possible TargetPaths. Not shown in the algorithm are instrumentation of the edges in these PSESets and the generation of modulator and demodulator classes. $MinCostEdgeSet(p)$ in the algorithm returns the set of edges with minimal cost among all of the edges of path p. An edge has minimal cost among a set of edges if no other edge in the set has a determinably smaller cost.

```
public void push(java.lang.Object) {
     Example r0;
     java.lang.Object  r1;
     ImageData  r2, $r3, r4;
     boolean  $z0;

1:   r0 := @this: Test;
2:   r1 := @parameter0: java.lang.Object;
3:   $z0 = r1  instanceof  ImageData;
4:   if  $z0 == 0  goto  label0;

5:   r2 = (ImageData) r1;
6:   $r3 = new ImageData;
7:   specialinvoke  $r3.<ImageData: void <init>(ImageData, int, int)>(r2, 100, 100);
8:   r4 = $r3;
9:   staticinvoke <ehag.test2.Test: void displayImage(ehag.test2.RawData)>(r4);

   label0:
10: return;
}
```

Figure 4: Jimple representation of the push method.

A simple example further explains the convex-cut algorithm. In this example, the consumer receives raw image data from the sender. The raw image data is handed to the *push()* method, which checks whether the event has the right format, then transforms the image to the size of 100x100, and finally calls a native method to display the image. The Java source file of the example can be found in Appendix A. Figure 4 has the Jimple format representation of the *push* method, which serves as the message handling method.

Assume our purpose is to minimize the amount of data sent across the network. It is obviously desirable to move the predicate at line 4 and all of the instructions preceding it to the sender's side, so that events that are not of type ImageData will be filtered out. Further, assume that images sent from the sender are of different sizes, some larger than 100x100, while others are smaller (assuming color depth remains constant). To minimize traffic, the program must perform transformations at the sender's side for large images, and at the receiver's side for smaller images. Next we demonstrate how the algorithm finds potential cuts in the *push()* method.

Figure 5 shows the combined control flow and data dependency graphs of the method, with the solid line denoting control flow and the dotted line denoting data dependency. Figure 6 adds additional information from the previous figure, by adding the INTER set for each edge in the control flow. The black node in the latter graph is the StartNode, while the grey nodes are StopNodes. Node 9 is a StopNode as it invokes a native method. Node 10 is a StopNode as well, because it is a return statement.

There are two TargetPaths in the graph: *tp1*={*2,3,4,10*}, and *tp2*={*2,3,4,5,6,7,8,9*}. For *tp1*, obviously MinCostEdgeSet(*tp1*) should return {*Edge(4,10)*}. *tp2* is more complicated: *Edge(3,4)* and *Edge(6,7)* can

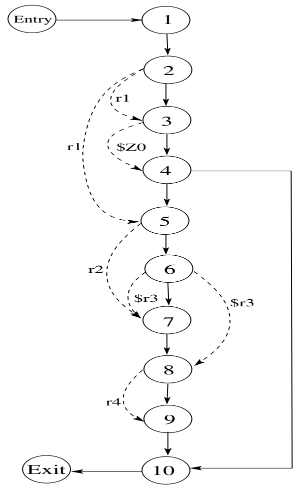

Figure 5: Unit and Data Dependency Graph of the *push()* method.

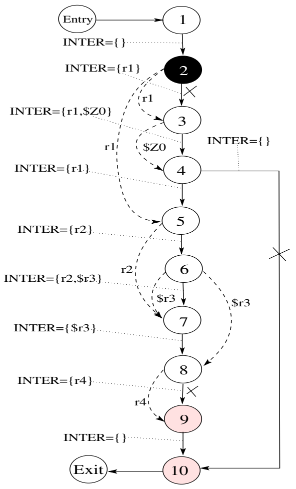

Figure 6: Left graph with INTER set for control flow edges and marked StartNode and StopNodes.

be trivially excluded as they have higher costs than at least one of their neighbors. *Edge(2,3)* and *Edge(4,5)* have identical INTER sets, so we arbitrarily remove one of them, say *Edge(4,5)*. A point-to analysis will tell that *Edge(2,3)* and *Edge(5,6)* have identical costs. So again we can remove one of them, say *Edge(5,6)*. *Edge(7,8)* and *Edge(8,9)* have a similar case. Here we assume that *Edge(7,8)* is removed, and eventually, we get the result PSESet, which is {*Edge(4,10), Edge(2,3), Edge(8,9)*}.

4 Sample Cost Models

As stated earlier, different cost models result in different modulator/demodulator pairs. In this section, we examine two sample cost models representing different customization/adaptation purposes. The first case concerns using Method Partitioning to reduce network communication between the sender and receiver of messages, while the second one aims at reducing the average amount of time spent on the computationally intensive processing of messages.

4.1 Cost Model for Minimizing Network Communication

This cost model defines costs as proportional to the amount of data sent from the modulator to the demodulator.

Recall that, at a given PSE, the Continuation Message sent from modulator to demodulator consists of live variables that are in the intersection of the *out-set* of the "out-node" and the *in-set* of the "in-node" of the PSE. The cost of a PSE, hence, is the total runtime size of the unique objects "reachable" from any of the variables in that intersection set, plus the total number of duplicated *references* to those unique objects. Unfortunately, the total runtime size of a set of variables is not always determinable at the time of static analysis, because programs can use interfaces, superclasses and arrays whose sizes are only known at runtime. However, we can still figure out lower cost bounds for a non-determinable edge, and if this lower bound is higher than the cost of a cost-determinable edge in a path, then we can exclude the edge with non-determinable cost from the PSE set of the path. Further, for two edges with non-determinable costs, each consisting of a determinitic partial cost and a non-deterministic partial cost, if their non-deterministic partial costs consist of identical sets of variables (Techniques such as points-to analysis can be used to identify variables with different names but identical costs) then we can eliminate one of them by comparing their deterministic partial costs. Techniques such as points-to analysis can be used to identify variables with identical costs.

Runtime profiling uses a customized object serialization algorithm to calculate the sizes of non-determinable variables. This customized serialization is fast for variables referencing primitive arrays, because it only performs size calculation but does not perform actual serialization of the arrays. For complex objects, to avoid costly reflection-based object serialization, procedures used for self-describing methods can be generated by the compiler to quickly calculate object sizes. A sample object with an attached size self-describing method appears in Appendix B. Table 1 shows the effectiveness of such size self-describing methods, especially for complex objects. The first column of the table lists four different classes used in the experiment, where *Int100(w/ wrapper)* is a wrapper class for an array of 100 ints, *Int100(w/o wrapper)* is a primitive array of 100 ints, *AppBase* is a classes several fields of primitive types, while *AppComp* is a more complex object. The 4th row of the table shows that, for *AppComp*, original size caculation cost(159μs) is close to that of serialization(189μs). But with compiler-generated, self-defined size calculation methods, this cost is dramatically reduced(1.16$mu$s).

4.2 Cost Model for Reducing Program Execution Time

To simplify the problem, we assume that the network resources available to a message sender and receiver pair are guaranteed and do not change over time, while the computational resources available to them do change, possibly as the result of competition from other applications. We model the time to send an message

Table 1: Object serialization and size calculation costs.

Class of Objects	Serialized size (in byte)	Serialization cost (in μs)	Size calculation cost (in μs)	Size calculation with self-desc methods(in μs)
Int100(w/ wrapper)	406	64	25	0.92
Int100(w/o wrapper)	402	57	2.1	n/a
AppBase	52	44	38	0.90
AppComp	216	189	159	1.16

m as:

$$T_s(m) = \alpha + \beta S(m) \tag{1}$$

where S(m) is the size for message m, in number of units, α is a constant for per message set-up time, and β is the amount of time for each unit in the message. We also assume that the communication of a message can be overlapped with computation on the sender and the receiver, and that the application is not communication bound:

$$\alpha + n\beta < n * max(T_p(1), T_c(1)) \tag{2}$$

where n is the total number of units of data to be sent from sender to the receiver in the application, $T_p(1)$ is the sender side processing time for each unit, and $T_c(1)$ is that for the receiver side.

Further assume that the handling of a message is computationally much more expensive than its generation, so that it is desirable to shift part of the handling code from the receiver to the sender to speed up program execution. Using the results described in [40], the total program execution time is:

$$T = n * max(T_{mod}(1), T_{demod}(1)) + \alpha + \sigma\beta + \sigma min(T_{mod}(1), T_{demod}(1)) \tag{3}$$

where σ is the message size in units sent from the sender to the producer, with

$$\sigma > \alpha/(max(T_{mod}(1), T_{demod}(1)) - \beta) \tag{4}$$

When computation cost is much higher than communication cost, and when n is much larger than 1, the dominant factor in equation (3) is $n * max(T_{mod}(1), T_{demod}(1))$. To simplify the implementation, we approximate the cost of each edge as $n * max(T_{mod}(1), T_{demod}(1))$. The adaptation target under such a scenario, therefore, is to balance the load between the sender and the receiver, and to choose the smallest σ that satisfies 4.

The costs in this model heavily depend on runtime profiling. Static analysis assigns an edge cost that simply depends on the differences in the edge's distances (in terms of number of instructions) from the start of a path and to the end of the path. Runtime profiling for each PSE measures the values of $T_{mod}(1)$ (measured at the modulator side) and $T_{demod}(1)$ (at demodulator side), as well as the actual data sizes passed across the network (as with the previous cost model)[2].

5 Implementation Evaluation

Our prototypical implementation of Method Partitioning uses Soot [27] for static analysis and for modulator/demodulator class generation. The JECho Distributed Event (Message) System serves as the message communication testbed [4].

The two applications mentioned in Section 1 are implemented using Method Partitioning, and compared with various versions of manual implementations. Some parts of the Method Partitioning based implementations are manually coded but can be easily replaced with compiler generated code. The first experiment uses the wireless image server/client application, where the communication resource is the performance bottleneck. The second experiment uses the sensor/processor application, which is computation bound.

5.1 Minimizing Communication Costs

In this experiment, the image stream server runs on a PII Linux laptop, while the client operates on an iPAQ 3650 handheld device. The server and the client are connected via a 802.11b wireless network. The images sent out from the server can have sizes either larger than the size of the display window on the iPAQ, or smaller than that, without the client's *a priori* knowledge.

Table 2 compares the performance of the Method Partitioning based implementation against two manually written versions, each optimized for one of the following two scenarios: one with display window size larger than the original image size (1st column in the table) , the other with one that is smaller than the original image size (2nd column). The first two experiments apply the two scenarios to all three versions of the application. There is no dynamic change of scenarios during these two experiments (first 2 rows). The third experiment (3rd row) introduces dynamic application behavior, by alternating the above two scenarios. Each scenario lasts for n frames, where n is a uniformly distributed integer ranging from 1 to 20.

[2]For simplicity, we assume that the amount of data passed is always greater than the minimal requirement of σ.

Experimental results show that, for a given scenario, Method Partitioning has performance close to that of the manually optimized version, but is much better than that of the non-optimized manually coded version. The important insight is that for dynamic system behavior, Method Partitioning can show substantial benefits. Note that in the third experiment, the Method Partitioning version significantly outperforms the two manually written versions.

Table 2: Effects of Runtime Adaptation with Method Partitioning (Display size = 160 * 160, values are average number of frames per second).

Implementation Versions	Small Image (80 * 80)	Large Image (200 * 200)	Mixed
Image<Display	29.79	7.53	12.98
Image>Display	12.06	12.11	12.19
Method Partitioning	29.72	12.07	17.65

Table 3: Running on heterogeneous platforms (numbers are for average message processing time in ms).

Implementation Versions	PC->Sun	Sun->PC
Consumer Version	352.10	108.92
Producer Version	143.93	139.00
Divided Version	250.19	83.59
Method Partitioning	109.34	74.67

5.2 Method Partitioning for Reducing Program Execution Time

In this experiment, we measure the performance of the Method Partitioning based implementation for reducing program execution time in the sensor data processing application. We compare it with three other implementations: a *Consumer Version* that performs all image processing inside the consumer, the *Producer Version* with all processing inside the producer, and the *Divided Version* which splits processing into two roughly equal parts that run in parallel on producer and consumer.

We experiment with these four versions on hosts with and without loads, where loads are created by running a number of perturbation threads. Perturbation threads have active and idle periods, where each period consists of multiple atomic cycles. To simulate the load changes occurring in various application

Table 4: Performance of Method Partitioning for Reducing Program Execution Time (numbers are in ms and are averages of 5 measurements).

(Producer LIndex)/ (Consumer LIndex)	Consumer Version	Producer Version	Divided Version	Method Partitioning
0/0	88.44	80.455	58.52	48.445
0/0.6	146.94	80.26	103.675	54.605
0/1.0	215.195	80.405	148.99	65.26
0.6/0.6	142.51	149.9	101.13	59.225
0.6/0	87.315	154.545	60.13	49.19
1.0/0	88.805	243.58	116.465	60.17

environments, the number of atomic cycles in a period (*PLen*), and the probability of perturbation threads being active (*AProb*) are uniformly distributed, with adjustable ranges. Active periods have a fixed load index (*LIndex*), which represents the ratio of busy cycles (when perturbation threads spin on numeric calculations) over the total number of cycles in a period. We pre-generate arrays of random numbers (for the random distribution of *PLen* and *AProb*) for each experiment setup, and use these same random numbers for all four implementations being evaluated.

Tests are performed using a SUN cluster and an Intel/Linux cluster. The SUN cluster has uni-processor Ultra-30 workstations connected via 100MB Fast Ethernet. The Intel/Linux cluster consists of dual-processor (300MHz Pentium II) Intel servers each running Redhat Linux 7.1 and also connected with Fast Ethernet. The SUN cluster and the Intel/Linux cluster are connected via a gigabit link.

Table 3 shows the performance of the four versions on heterogeneous platforms with no perturbation. The first column lists the values attained when running the producer on a SUN workstation and the consumer on an Intel server. The second column lists the values attained with the opposite configuration. The results clearly demonstrate that the Method Partitioning version outperforms (sometimes remarkably outperforms) the three other versions. For instance, when messages are sent from the PC to the Sun workstation, average processing time for the *Consumer Version* is 222% longer than Method Partitioning version, while when messages are sent from the workstation to the PC, the *Producer Version* is 86% slower than the Method partitioning version.

The other tests are carried out within the Intel/Linux cluster. We first compare the performance of the

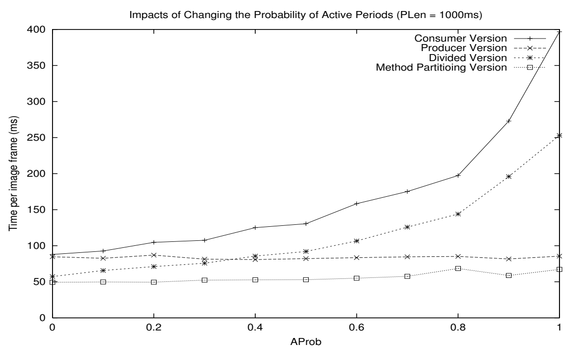

Figure 7: Impact of Consumer-side Active Period Probability Changes (Consumer side PLen=1000ms, LIndex=0.8, producer side load-free).

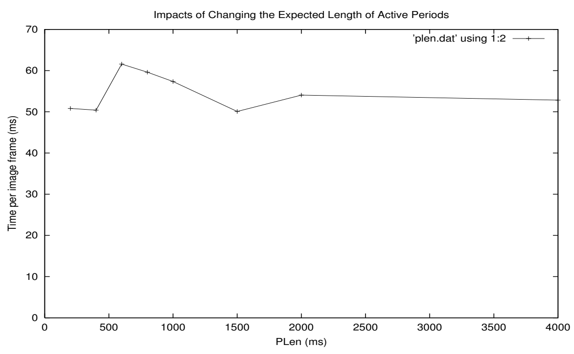

Figure 8: Impact of Consumer-side Active Period Expected Length Changes (Consumer side AProb=0.5, LIndex=0.8, producer side load-free).

four versions by varying the load indices on the producers and the consumers. The expected value of $PLen$ is set at $1000ms$, and $LIndex$ at 0.5. Table 4 depicts results for the four versions under different load distributions. Again, the Method Partitioning version clearly outperforms the other three. In fact, it outperforms the *Divided Version* even when there is no load(58.52 vs. 48.445). This is because it better balances the load by doing loop distribution.

Figure 7 depicts the performance of the four versions under varying $AProb$ at the consumer side, while the producer side remains load-free. As expected, the consumer side load change almost has no effect on the *Producer Version*, and it has very little effect on the Method Partitioning version. On the other hand, performance the other two versions severely degrades when consumer side load increases. Figure 8 shows the impact of consumer side $PLen$ on the Method Partitioning version. It shows that the Method Partitioning version is relatively stable against changes in perturbation patterns.

5.3 Discussion of Results

Our experiemnts demonstrates Method Partitioning's remarkable agility in adapting to environmental and application dynamics. Low adaptation overheads are experienced in dynamic environments, while in static environments, performance of Method Partitioning versions is close to that of manually optimized versions. This is partly because in both applications, the generated PSE graphs are relatively simple (one has 5 PSEs, the other has 21 but is almost all along the same path), resulting in negligible overheads for running the reconfiguration algorithm.

Experimental evaluations do not include the costs for modulator installation. Such costs include the costs of transporting the modulator to the sender and of loading the classes used by the modulator (including classes for Continuation messages). They depend on the sizes of the classes and the size of the modulator object, both of which tend to be small compared to the amount of application data passed around in both of our sample applications. The other cost is in the increment of total size of classes used for the application. For example, each additional PSE will require a new redirect argument class (around 500 to 800 bytes in our experiments), and there are increases the sizes of the modulator and demodulator classes due to instrumentation codes (about 150 bytes per PSE in experiments).

6 Related Work

Distributed application customization and adaptation has been studied extensively in the literature. Existing research can be loosely divided into two major categories: dynamic unit placement based systems and systems that make compromises concerning costs vs. qualities attained.

- *Dynamic Unit Placement Based Systems.* Coign and J-Orchestra are off-line profiling-based system that partition centralized applications into distributed components based on profiles about component resource consumption and inter-component communication [31, 32]. They operate in heterogeneous but relative stable environments and use communication as their only cost model. Method Partitioning targets different types of applications than Coign and J-Orchestra. Compared with these systems, Method Partitioning provides higher adaptability, supports multiple cost models and has inherently finer-grained customizability, but lacks application scale, rather than sender/receiver pair, optimization support. Hybrid Shipping dynamically distributes query processing loads between clients and servers of a database management system [33]. It requires *a priori* knowledge of the implementations of query operators, and its application is limited to the query processing domain. Partitionable Services achieves seamless client component customization through replicating and migrating services [34]. Partitionable Services, however, assumes that the resources available to a deployed service remain fixed over the lifetime of its deployment.

  Research on mobile agents has been abundant [9, 13, 14, 16, 15, 11]. As we discussed in the beginning, supporting the adaptation of complex applications with mobile agents requires knowledge of the application at design-time. This is also the case for Smart Messages in Cooperative Computing [35], where Smart Messages enhances mobile agents with self-routing ability, and where the system provides infrastructure to support this ability. ABACUS uses mobile objects for dynamic function placement for data-intensive cluster computing [36]. It builds on mobile agent work by providing runtime mechanisms that automate migration decisions.

- *Cost-to-Quality Compromise Based Systems.* Research in this category uses application- or data-dependent adaptation approaches. In particular, Odyssey supports application specific code components to be used for different levels of fidelity of data representation [37]. TACC of the Daedalus project uses a proxy based approach to adapt with certain cost-to-quality compromises [38]. Transformer Tunnel is a network-level approach which supports per-link adaptation by allowing functions to be attached to adaptive tunnels, with all data flowing through these tunnels being adapted similarly[39]. Research in this category is completely orthogonal to Method Partitioning.

Our own earlier work[7, 4, 8] focused on runtime systems that support data stream customization in publish/subscribe systems. Method Partitioning automates the generation of stream customization code, and provides built-in profiling and adaptation. Our work is part of the ongoing InfoSphere project, which adopts a information flow-based, rather than computation-centric approach for the "clean, reliable and timely delivery" of data from potentially large numbers of heterogeneous sources[26].

7 Conclusion, Limitations and Future Work

Method Partitioning is a novel approach for the runtime customization and adaptation of distributed applications. It requires no design-time knowledge about applications, and it uses minimal deployment-time knowledge. It supports high customizability of units for dynamic placement, and also offers low cost adaptation actuation. Customization and adaptation in Method Partitioning are implemented using static analysis of message handling methods, thereby producing a set of candidate method partitions. Cost models are used to evaluate the costs of different candidate partitions. A *Remote Continuation* mechanism "connects" the dynamically distributed parts of a partitioned method at runtime. Runtime Profiling and Reconfiguration Units assist in the dynamic selection of actual partition from candidates.

A prototypical implementation of Method Partitioning hosted in the JECho distributed event system is applied to two sample applications: one is communication bandwidth critical and the other is computation bound. Experiments demonstrate significant performance improvements attained by Method Partitioning, in part due to its high levels of customizability and its lightweight ways of adapting to application and environment dynamics.

One limitation of Method Partitioning is that it is sender/receiver pair-based, and that the modulator of the receiver can only reside in the message sender. Future work conducted in our group is integrating Third Party Derivation [28] with Method Partitioning, which allows a modulator to operate inside a "third party". In addition, we are developing methods for propagating modulators upward along a data stream, whenever this is useful for further optimization.

Our current implementation treats each method invocation inside the message handling method as an opaque instruction, rather than expanding the UG of the message handling method with a link to another UG for PSEs inside the latter CFG. In addition, partitioning currently allows only convex cuts of the UG, thus potentially excluding better partitioning plans. Our future research will address more complex, whole program based partitioning plans.

The deployment-time knowledge used for Method Partitioning is the information contained in cost models, which are used to select appropriate modulator/demodulator pairs. Our future work will try to eliminate this requirement, for specific applications and equipment infrastructures, one example being future PDA/cellphone systems. We would also like to work on extending cost models to include considerations of power consumption and security, and experiment with composite cost models.

References

[1] D. C. Schmidt and S. Vinoski. OMG Event Object Service. *SIGS*, Vol. 9, No. 2, Feb. 1997.

[2] Sun MicroSystems. Java Message Service. *http://java.sun.com/products/jms/docs.html.*

[3] M. Aguilera, R. E. Strom, D. C. Sturman, M. Astery, and T. D. Chandra. Matching Events in a Content-based Subscription System. *Principles of Distributed Computing*, 1999.

[4] D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen. JECho - Interactive High Performance Computing with Java Event Channchoels. *Proceedings of IPDPS 2001*, April 2001.

[5] D. Zhou and K. Schwan. Eager Handlers - Communication Optimization in Java-based Distributed Applications with Reconfigurable Fine-grained Code Migration. *Proceedings of the 3rd International Workshop on Java for Parallel and Distributed Computing*, April 2001.

[6] B. Zuerndorfer and G. A. Shaw. SAR Processing for RASSP Application. *Proceedings of 1st Annual RASSP Conference*, Aug., 1994.

[7] G. Eisenhauer, F. Bustamente, and K. Schwan. A Middleware Toolkit for Client-Initiated Service Specialization. *Proceedings of the PODC Middleware Symposium*, July, 2000.

[8] F. E. Bustamante, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active Streams: An approach to adaptive distributed systems. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May, 2001.

[9] R. S. Gray. Agent Tcl: A transportable agent system. *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, 1995.

[10] M. Straber, J. Baumann and F. Hohl. Mole-A Java Based Mobile Agent System. *Proceedings of the ECOOP '96 Workshop on Mobile Object Systems*, 1996.

[11] D. B. Lange and D. T. Chang. IBM Aglets Workbench. White paper, IBM Research, Sep. 1996.

[12] C. Mascolo, G. P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility. *Proceedings of the Seventh European Software Engineering Conference*, September 1999.

[13] D. Johansen, R. van Renesse, and F. B. Schneider. Operating system support for mobile agents. *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[14] A. L. Murphy, and G. P. Picco. Reliable Communication for Highly Mobile Agents. *Proceedings of the 1st International Symposium on Agent Systems and Applications*, Oct., 1999.

[15] U. Kubach and K. Rothermel. Estimating the Benefit of Location-Awareness for Mobile Data Management Mechanisms. *Proceedings of the International Conference on Pervasive Computing 2002.*

[16] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh. Mobile Agent Programming in Ajanta. *Proceedings of ICDCS '99*, 1999.

[17] R. Vanegas, J. A. Zinky, J. P. Loyall. D. A. Karr, R. E. Schantz, D. E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, September 1998.

[18] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, Dec., 1997.

[19] D. Rosu, K. Schwan, and S. Yalamanchili. FARA - A Framework for Adaptive Resource Allocation in Complex Real-Time Systems. *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Jun., 1998.

[20] ARMADA: A Real-time Middleware Architecture for Distributed Applications. *http://www.eecs.umich.edu/RTCL/arpa-project.*

[21] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng. Cooperative Mobile Robotics: Antecedents and Directions. *Proceedings of IEEE/TSJ International Conference on In- telligent Robots and Systems*, 1995.

[22] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun. Collaborative Multi-Robot Exploration. *Proceedings of the International Conference on Robotics and Automation*, 2000.

[23] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. *Concurrency: Practice and Experience*, August 1998.

[24] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[25] The Cactus Project, *http://www.cs.arizona.edu/cactus/.*

[26] The InfoSphere Project, *http://www.cc.gatech.edu/projects/infosphere/.*

[27] Soot: a Java Optimization Framework. *http://www.sable.mcgill.ca/soot/.*

[28] D. Zhou, Y. Chen, G. Eisenhauer, and K. Schwan. Active Brokers and Their Runtime Deployment in the ECho/JECho Distributed Event Systems, *Proceeds of the Third Annual International Workshop on Active Middleware Services*, 2001.

[29] A. Schoedl, K. Schwan, and I. Essa. Adaptive Parallelization of Model-based Head Tracking. *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, July 1999.

[30] , PacketVideo, *http://www.packetvideo.com/.*

[31] G. C. Hunt and M. L. Scott. The Coign Automatic Distributed Partitioning System. *Proceedings of the 3rd Symposium on Operating System Design and Implemetation (OSDI)*, Feb. 1999.

[32] E. Tilevich and Y. Smaragdakis. J-Orchestra - Automatic Java Application Partitioning. *Proceedings of ECOOP 2002*, June, 2002.

[33] M.J. Franklin, B.T. Jonsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. *Proc. ACM SIGMOD Conf.*, pp. 149-160, 1996.

[34] A.-A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogenous Environments. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, July 2002.

[35] C. Borcea, D. Iyer, P. Kang, A. Saxena and L. Iftode. Cooperative Computing for Distributed Embedded Systems . *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS 2002.

[36] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing, *Prooceedings of USENIX Annual Technical Conference*, June 2000.

[37] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K.R.Walker. Agile application-aware adaptation for mobility. *Proceedings of the 16th ACM Symposium on Operating System Principles.*

[38] A. Fox, S.D. Gribble, Y. Chawathe, and E.A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *IEEE Personal Communications*, 5(4):10-19, August 1998.

[39] P. Sudame and B. R. Badrinath. Transformer tunnels: A framework for providing route-specific adaptations. *Proc. of the USENIX Technical Conf.*, 1998.

[40] S. Kim, S. S. Pande, D. P. Agrawal, and J. Mayney. A message segmentation technique to minimize task completion time. *Proceedings of the Fifth International Parallel Processing Symposium*, 1991.

[41] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-to Analysis. *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, January 1997.

Appendix A: Java source file for the example used in section 3.

```
public class Test {
    static native void displayImage (ImageData data);

    public void push (Object event) {
        if (event instanceof ImageData) {
            ImageData rd = (ImageData)event;
            rd = new ImageData (rd, 100, 100);
            displayImage (rd);
        }
    }
}

public class ImageData {
    public int width;
    public byte buff[];

    public ImageData (ImageData template, int w, int h) {
        buff = new byte[w * h];
        width = w;
        for (int i = 0; i < h; i++)
            for (int j = 0; j < w; j++)
                buff[i * w + j] = template.buff[template.width * i + j];
    }
}
```

Appendix B: Classes with added methods that self-describe object sizes.

```
package jecho.bench.base;
public class AppBase implements SelfSizedObject {
    int a = 0, b =2;
    long c = 1202;
    String d = "rrr";

    public int sizeOf () {
        return 16 + ObjectSize.STRING_HEADER_SIZE + d.length ();
    }
}

package jecho.bench.base;
public class AppComp implements SelfSizedObject {
    public String s1, s2;
    public AppBase ab1, ab2;
    public int[] ia;
    public float[] fa;

    public AppComp () {
        s1 = "aa";
        ab1 = new AppBase ();
        ia = new int[20];
        fa = new float[10];
        s2 = "This is a string!";
    }

    public int sizeOf () {
        return s1.length() + s2.length() + 2 * ObjectSize.STRING_HEADER_SIZE
          + JECho.getSize(ab1) + JECho.getSize (ab2) + 2 * ObjectSize.OBJECT_HEADER_SIZE
          + ia.length * 4 + fa.length * 4;
    }
}
```