Optimizing Dynamic Producer/Consumer Style Applications in Embedded Environments

Dong Zhou, Santosh Pande, Karsten Schwan
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{zhou,santosh,schwan}@cc.gatech.edu

Abstract

Many applications in pervasive computing environments are subject to resource constraints in terms of limited bandwidth and processing power. As such applications grow in scale and complexity, these constraints become increasingly difficult to predict at design and deployment times. Runtime adaptation is hence required for the dynamics in such constraints. However, to maintain the lightweightness of such adaptation, it is important to statically gather relevant program information to reduce the runtime overhead of dynamic adaptation. This paper presents methods that use both static program analysis and runtime profiling to support the adaptation of producer/consumer-style pervasive applications. It demonstrates these methods with a network traffic-centric cost model and a program execution time-centric cost model. A communication bandwidth critical application and a computation intensive application are used to demonstrate the significant performance improvement opportunities offered by these methods under the presence of respective resource constraints.

Keywords

Pervasive computing, producer/consumer, peer-to-peer, program analysis, runtime profiling, dynamic adaptation

I. Introduction

Many pervasive computing applications are structured for use by dynamic participants using anonymous communication paradigms like peer-to-peer or publish/subscribe systems. In addition, the environments in which they execute are comprised of heterogeneous machines and communication systems, with dynamic variations in communication and processing resources. As a result, to meet their performance goals, applications must be configured, customized, and adapted at runtime.

Ongoing research has developed a large variety of techniques to deal with dynamic application behaviors and execution environments. Techniques for dynamic resource finding[1], [2] help identify the platform elements used for application execution. At the network level, methods for concurrent use of heterogeneous network devices[3], [4] and ad-hoc networking techniques are designed to reduce power usage while maintaining ongoing communications[11]. Such techniques are enhanced by transport- or MAC-layer work that maintains[5], [6] or enhances[7] communications and their resource usage. At the same time, a plethora of techniques for reducing the usage of CPU and other platform resources have been developed by both the compiler and OS communities[8], [9], [12], [13], [10], [14].

The research presented in this paper builds on such work by presenting a middleware layer for constructing and maintaining peer-to-peer interactions in distributed pervasive systems. The JECho publish/subscribe infrastructure permits end users to dynamically establish and use logical communication channels via which they can interact anonymously[15]. The novel attribute of JECho described in this paper is its ability to dynamically adapt such interactions with respect to their use of the underlying platform's processing and communication resources. Specifically, we present novel methods for customization and adaptation of publish/subscribe style applications in pervasive computing environments. These methods use static program analysis to identify customizable code segments on the consumer side of communicating peers and to insert instrumentation code into appropriate locations in these customizable segments. The idea is to identify and move, at runtime, those portions of consumer code into the producer that helps optimizing the resource usage of both. The goals of motion of code from consumers to producers could be diverse: the motion could reduce communication needs between both (e.g., by message filtering), or could balances the use of computational resources across both communicating peers. In either of these cases, code movement occurs transparently to communicating peers, without interrupting their ongoing interactions. Adaptation involves deciding which portion of the customizable parts should reside inside the producer (residency of the code), and is based on the runtime profiling data generated by the inserted instrumentation.

This paper demonstrate JECho's novel runtime methods for code movement with sample applications drawn from the pervasive systems domain, one benefiting from the runtime adaptation of its use of network resources, the other benefiting from optimizing its use of computational power. These applications are described next.

II. Motivating Applications

Two common pervasive computing applications are remote sensing or surveillance[22], [23], [24] and ubiquitous presence[17], [18], [19], [20], [21]. The applications used in this paper contain elements of both classes of applications.

The first example is drawn from the domain of ubiquitous presence, where a large number of participants follow an ongoing action being captured by multiple devices, an example being remote viewers of an ongoing soccer match. Viewers can dynamically subscribe to any one of multiple video feeds, perhaps to focus on their favorite players or on an action of current interest to them. In this scenario, multiple 'producers' offer information of interest to a potentially large number of 'consumers' which use heterogeneous devices ranging from laptops to handhelds like iPAQs to video-capable cell phones. In addition to differing in their relative processing abilities, these devices also differ in their communication abilities and current connectivities. As a result, it is necessary for each such device to be able to customize the volume and type of data actually sent to it. Furthermore, since the data contents, connectivities, and users' interests change at runtime, such customization must be changed dynamically.

We use this application scenario to generate our experimental setup. In our experimental setup, we use a server to send compressed images to both wired and wireless devices. Compression ratios differ across video files, where video clips with high compression ratios have larger image sizes. To emulate dynamic scene changes, we randomly choose and intermittently replace video clips. A video display on a device like an iPAQ has to be able to adapt to such changes.

Our second application is drawn from the sensor processing domain. We envision a set of remote sensors (e.g., cameras mounted on robots) that provide data to a single sink (e.g., a surveillance center). Sensor data must be analyzed before it becomes useful, and timing constraints (deadlines) may be attached to these analyses based upon the rate of generation of data. Analyses are structured as processing pipelines[16], typically comprised of multiple pipeline stages potentially spread across multiple machines for increased processing bandwidth and/or latency. The importance (e.g., priority) of each pipeline-structured application depends on the nature of the data it is currently processing and therefore, varies dynamically. Moreover, since multiple pipelines share a limited pool of server resources, computational loads must be balanced continuously, both across different pipelines and within each such pipeline. An example of such load balancing is shifting the use of computational resources from a pipeline stage that consumes information to one running on a different machine that produces it.

We next explain the JECho-based implementations of these applications.

III. JECho Distributed Event System and Eager Handlers

JECho is a Java-based distributed event system. JECho offers the abstractions of *events* and *event channels*. An event is an asynchronous occurrence, and may be used both to transport data and for control. An *event endpoint* is either a *producer* that raises an event, or a *consumer* that observes an event. An event channel is a logical construct that links a certain number of event endpoints to each other. An event generated by a producer and placed onto a channel will be observed by all of the consumers attached to the channel. An *event handler* is a method resident at a consumer which is applied to each event received by that consumer.

JECho offers the novel software abstraction of *eager handler*. An eager handler is an event handler that consists of two parts, with one part executing in the consumer's space and the other part dynamically 'pushed' into each event producer's space. We term the latter *event modulator*, while the part that stays local to the consumer is termed *event demodulator*. Events first move through the modulator, then across the wire, and then through the demodulator. We call such handlers 'eager' because of the dynamic partition of the event modulator from the original handler, its movement across the wire to the event source, and its installation into the producer's address space. Due to these mechanisms, it attempts to 'eagerly' touch the producer's events before they are sent across the wire.

Since a modulator is conceptually part of a consumer's event handling code, it is natural to allow the modulator to access the states of the event consumer. Depending on the way that modulators accesses consumer states, eager handlers can be classified into different types. The type of eager handler that we cover in this paper is the *Modulator Read-Only* eager handler. A Modulator Read-Only eager handler is an eager handler whose modulator either does not share any state with rest of the consumer program, or the states shared consists of only constants (or variables that do not change after the modulator is created).

The topic of this paper is JECho's framework and methods for automatically generating eager handlers, and then using them to adapt the produces/consumer relationships existing in the applications to changes in resource needs and availabilities. Two of the criteria for eager handler generation and movement are considered in detail:

- *Generating eager handlers to reduce the volume of network data flow* is important for network bandwidth-limited applications like the one in which remote viewers follow an ongoing sports action. It is also important for applications where network traffic is costly in financial terms, as is the case of the operational information systems investigated in our ongoing work with Delta Air Lines[25], as well as for applications that must avoid excessive perturbation due to network traffic (e.g., a real-time application in an embedded platform[16]. In such cases, overall effect of minimizing network traffic not only improves communication time but also makes program behavior predictable to guarantee meeting real time deadline requirements.
- *Generating eager handlers to reduce program execution time* is significant for computationally intensive applications, where the amount of computation between the modulator (which is executed remotely) and the rest of the program must be balanced to minimize total program execution time. The sensor processing example described in Section II is

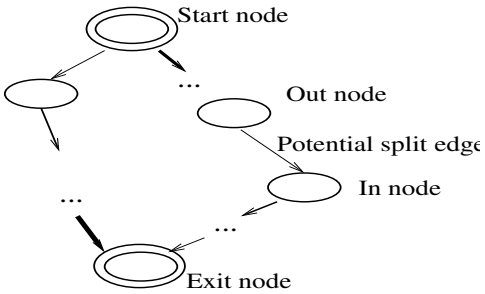
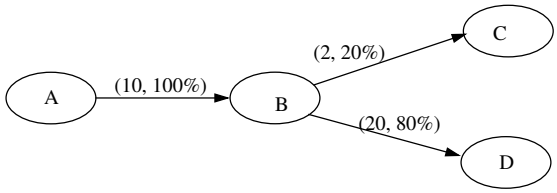


Fig. 1. Handler control flow graph and potential split edge(s).



Edge BC is a PSE because it has low cost (2), but it will never be in an actual runtime cut because edge AB is always in any actual cut (20% is percentage of times flow will go through BC instead of BD)

Fig. 2. A PSE that does not appear in any of the cuts because there is another PSE that appears in cuts dominates it

an example of such an application.

It can be easily seen that, in general, minimization of execution time and minimization of communication cost are not equivalent, prompting different approaches. We next present the methods used for automatic generation and adaptation of eager handlers with the help of static program analysis and runtime profiling to meet each of these objectives.

IV. General Framework for Generation and Adaptation

Although different eager handlers have different types and purposes, they share the same generation and adaptation framework. In a nut shell, our framework attempts to partition an event handling method into a modulator/demodulator pair for customizing the interaction between the producer and the consumer for the given objective function viz. either minimization of volume of data and traffic or for minimization of execution time. Partitioning is based on two-way cutting the control flow graph (CFG) of the method, while the weight of the edges in the CFG is determined by the cost model corresponding to the criterion used for the generation of the eager handler. Because of the application and environment dynamics, the cost of these edges can vary by time, resulting in changing optimal two-way cuts at runtime.

An edge in the CFG that could (but not must) be in any of the runtime optimal two-way cuts, or *actual cuts*, is called a *Potential Split Edge* (PSE) (Figure 1). At runtime, if a PSE is in the actual cut, then when program control reaches that PSE, the processing flow will be seamlessly transfered from the modulator side to the demodulator side where the execution continues. PSEs, from which an actual cut is selected at run time, are determined through static analysis. This approach is followed to minimize run time overhead. A lightweight *selection* approach for PSEs allows such a minimization.

The varying costs of edges in the CFG is monitored at runtime, so that up-to-date cost information is available for the runtime determination of optimal cuts.

We next discuss this general framework in detail.

A. Statically Searching for PSEs

The purpose of statically searching for PSEs is to minimize the runtime costs of monitoring edges and repartitioning the CFG, by excluding those edges that can never be in any actual cut.

A.1 Setting Stop-Nodes

The first step in finding PSEs is to generate a set called *stop-nodes*. A stop-node is a node of the CFG that should never appear in the modulator half of a partition. For example, when generating a Modulator Read-Only eager handler, we will not allow a node that modifies the state of the consumer to be included in the modulator.

The presence of stop-node(s) in the CFG implies that the only nodes that can possibly be placed into the modulator are those that are not dominated by any stop-node in the CFG.

A.2 Marking PSEs

An edge is a PSE if there exists a path that starts from the root node of the CFG, goes through this edge and ends at a stop-node, and that there is no edge with lighter cost on the path. As we can see from Figure 2, without *a priori* knowledge of entire execution path profile, it is possible to mark a PSE that does not appear in any of the actual cuts. Thus, PSE set is conservative in the sense that there may not exist program inputs which will force the selection of every member of this set at one point or another. Also, notice that there could be multiple PSEs along a path, as edges can either have identical costs, or their costs can only be determined at runtime.

The cost of each edge is directly linked with the generation criterion of the given eager handler. For example, to generate eager handlers that minimize the amount of communication between the modulator/demodulator pair, the cost model will assign a value proportional to the total sizes of the variables that are live across the edge. Cost models and their uses in generating eager handlers for different purposes are discussed in Section V.

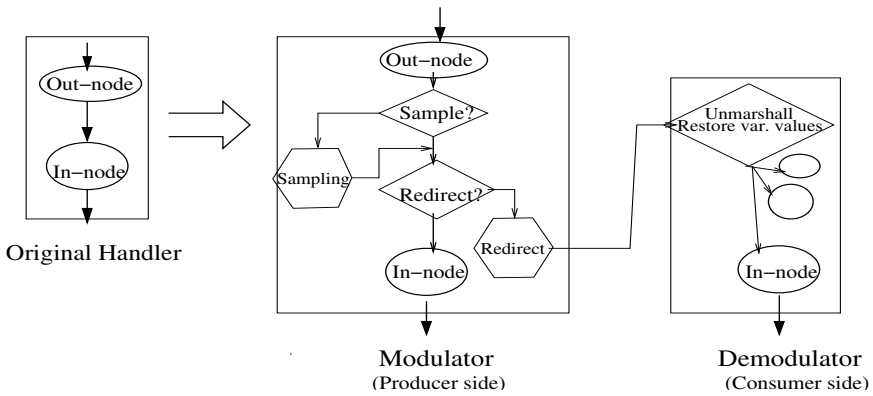


Fig. 3. Code instrumentation at a PSE.

B. *Method Remote Continuation - Jumping From Modulator to Demodulator at a PSE*

At runtime, when the control flow reaches a PSE after carrying out the operation on the 'out-node' of the PSE, a mechanism termed *Method Remote Continuation* helps it 'jump' to the demodulator side and seamlessly continue executing from there. To ensure correct execution continuation, our static analysis program inserts conditional (as PSEs are *potential* 'jump' points) redirecting code at each PSE (see Figure 3).

The redirecting code involves packing live variables for the PSE, and the identifications of the eager handler and the PSE, into a special *RedirectingEvent* and passes the event to the JECho system for delivering to the consumer side. The Modulator Read-Only eager handlers are generated so that at any point along the execution path inside the modulator, live variables only consist of:

- the *this* reference,
- the live parameters of the modulator method,
- the live local variables, and
- the live references to replicated read-only states

Among these, only live parameters and live local variables are marshaled and packed inside the RedirectingEvent.

At the receiver side, this RedirectingEvent is delivered to the corresponding demodulator. The demodulator half of the CFG is accordingly modified such that, dispatch code (a table switch statement in implementation) is inserted at the beginning of the method to jump to the right place inside the demodulator, based on the identification of the PSE. The dispatch code also restores the values of the live variables packed in the received RedirectingEvent and sets the state of the program before control is passed further for execution.

```
// sampling stub
PSESet pseset = set;
PSE apse[] = pseset.pselist;
PSE pse = apse[4];
boolean flag1 = pse.doSample;
if(flag1) {
    Object aobj[] = pse.nd_vars;
    // data needs runtime size calculation
    aobj[0] = data;
    Monitor.doProfiling(pseset, 4);
}

//redirecting stub
boolean flag6 = pse.split;
if(flag6) {
    Test_ARG_RO4 test_arg_ro4 = new Test_ARG_RO4(i);
    test_arg_ro4.key = 4; //PSE ID is 4
    decevent.setContent(test_arg_ro4);
    //Hand it to JECho transport
    enqueue(decevent);
    return;
}
```

Fig. 4. Sample code stubs for sampling and redirecting at a PSE.

C. Runtime Profiling and Reconfiguration

Another task of static analysis is to insert sampling code at each PSE (Figure 3). The purposes of this sampling code is to (1) calculate costs of PSEs that can only be determined at runtime, and (2) to collect statistical data on the actual execution paths inside the event handler. Some actual instrumentation code (both sampling and redirecting) at a PSE is shown in Figure 4.

The activation of such sampling code is also conditional. A flag indicating whether to execute the sampling code is controlled by a *Monitoring Unit* to avoid excessive overhead caused by runtime profiling. This is because, firstly, for some PSEs, the runtime cost and possibility of execution path going through it does not vary much. And secondly, for some PSEs, such sampling could be very expensive, and periodically performing the sampling can help reducing the overhead, at the cost of less timely statistics.

The actual measurements taken in sampling codes are eager handler generation criterion-dependent. For instance, a eager handler generated for minimizing communication costs might measure the actual amount of data sent along each PSE, and a eager handler intended for load-balancing might measure the amount of CPU time elapsed since the time of going through its previous PSE.

Measurements from sampling codes are collected by the Monitoring Units. The demodulator side Monitoring Unit periodically sends demodulator side profiling data as feedbacks to the modulator side, while the modulator side Monitoring Unit uses local and received profiling data to perform repartitioning on the PSE graph (which is a condensed CFG with only PSEs), searching for edges with the smallest costs along each path. For a PSE that is involved in more than one path, our implementation doesn't distinguish sampling data for different paths, so the partition may not be optimal for all execution paths. In other words, this approach is not based on path profiles but only upon edge profiles; gathering path profiles is much more expensive especially given the dynamic nature of the applications we are interested in.

V. Cost Models for Eager Handlers of Different Criteria

As we have stated earlier, different eager handler generation criteria correspond to different cost models in calculating the costs of PSEs. In this section, we apply two different partitioning criteria to our framework, and examine the cost models used to derive these targets. The first case concerns generating eager handlers that reduces network communication between the producer and the consumer of the events, while the second one aims at reducing the total amount of time spent for computation intensive processing of events.

A. Eager Handlers for Minimizing Network Communication

It is quite obvious that to generate eager handlers that minimizes network communication, we will use the framework to dynamically find the two-way cut with minimal cost, where cost is the amount of data sent from the modulator to its demodulator.

Recall that, at a given PSE, the *RedirectEvent* sent from modulator to the 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 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, plus the total number of duplicated *references* to these unique objects.

However, the total runtime size of a set of variables is not always determinable at static analysis time, because programs can use interfaces, superclasses and arrays whose sizes can only be known at runtime. Our implementation for searching for PSEs calculates the lower cost bounds of non-determinable edges, and some of these edges can be eliminated if their lower bounds are higher than the cost of a determinable edge in the path.

Customized object serialization is used to calculate the total size of a set of variables. This serialization performs size calculation but does not perform actual serialization for primitive arrays and for objects of size-determinable classes(classes whose instances have identical sizes). The purpose of this specialization is to speed up size calculation.

The efficiency of sampling code is critical to the performance of an automatically generated eager handler, particularly when there are a number of PSEs in the handler. The overhead for sampling is considerable, especially for complex objects, just like in the regular object serialization. This is because our size calculation code uses the Java reflection mechanism, which is costly for complex objects.

The optimization is to force objects to self-describe their sizes. That is, a method which calculates the sizes of the objects of a class can be inserted into the class at static analysis time. Figure 5 shows two classes with methods that self-describe object sizes. Notice that class AppComp can't directly call the sizeOf() method of class AppBase to get the size of fields $ab1$ and $ab2$, because AppBase is not final (so it may have subclasses which do not implement its own sizeOf() method). Section 5.4 shows very significant improvement in size calculation speed for complex objects due to the use of size self-describing methods.

Finally, for PSEs with high sampling cost, another choice is to store historical sampling data into the class file of the handler and to perform sampling infrequently (e.g., once for every 5 times that the handler is invoked at the start of the

```
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;
    }
}
```

Fig. 5. Classes with added methods that self-describe object sizes.

application, and then gradually reducing the rate to, say, once every 20 times). In addition to reducing the sampling overhead, such approach also helps short-running applications to quickly adopt the historically optimal configuration.

B. Eager Handlers for Reducing Program Execution Time

Another criterion in eager handler generation is to make generated eager handlers to be able to dynamically adapt to environmental changes to reduce the overall program execution time.

To simplify the problem, we assume that the network resources available to a producer and consumer pair is guaranteed and does not change over time, but the computational resources available to them do change, possibly as the result of competition from other applications. We model the time to send an event e as

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

where S(e) is the size for event e, 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 event. We also assume that the communication of an event can be overlapped with computation on the producer and consumer, and that the application is not communication bound, i.e.,

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

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

Further assume that the handling of an event is computationally much more expensive than the generation of it, so that it is desirable to shift part of the handling code from the consumer to the producer to speed up the program execution. Using the results described in [31], with JECho's eager handler model, the total program execution time is

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

where σ is the message size in units sent from the producer to the consumer and that

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

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

```
void Process () {

  /* code omitted */

  for (int j = 0; j<SARX.NPULSE; j++) {
    /* code omitted */
    for (i=0; i<SARX.NCSAMPLES - NIQ; i++) {
      isum = (float)0.;
      qsum = (float)0.;
      for(int k=0; k<NIQ; k++) {
        isum += in[i + k] * icoef[k];
        qsum += q[i + k] * qcoef[k];
      }
      cbufr[j][i] = isum;
      cbufi[j][i] = qsum;

      tmpx = cbufr[j][i];
      tmpy = wr[i];
      cbufr[j][i] = tmpx * tmpy - cbufi[j][i] * wi[i];
      cbufi[j][i] = tmpx * wi[i] + cbufi[j][i] * tmpy;
    }
  }

  /* code omitted */

}
```

Fig. 6. Original code segment for FIR processing.

```
void Process () {
  /* code omitted */
  int j, i;
  for (j = 0; j<SARX.NPULSE; j++) {
    for (i=0; i<SARX.NCSAMPLES - NIQ; i++) {
      isum = (float)0.;
      qsum = (float)0.;
      for(int k=0; k<NIQ; k++) {
        isum += in[i + k] * icoef[k];
        qsum += q[i + k] * qcoef[k];
      }
      cbufr[j][i] = isum;
      cbufi[j][i] = qsum;
    }
  }

  for (j = 0; j<SARX.NPULSE; j++) {
    for (i=0; i<SARX.NCSAMPLES - NIQ; i++) {
      tmpx = cbufr[j][i];
      tmpy = wr[i];
      cbufr[j][i] = tmpx * tmpy - cbufi[j][i] * wi[i];
      cbufi[j][i] = tmpx * wi[i] + cbufi[j][i] * tmpy;
    }
  }
  /* code omitted */
}
```

Fig. 7. Left code segment after loop distribution.

In real world applications, loops account for the majority of the computation time. To help reducing $max(T_{mod}(1), T_{demod}(1))$ in equation 3, or, in other words, to help adjusting load balance at a finer grain, we use loop distribution to divide large loops into multiple smaller loops, and marks the inter-loop nest data-flow edges as a PSE.

Figure 7 shows a program segment with distributed loops. The original program segment is shown in Figure 6.

The runtime profiling for each PSE measures values of $T_{mod}(1)$ (measured at the modulator side) and $T_{demod}(1)$ (demodulator side), as well as the actual data sizes passed across the network (like the previous scenario)[1]. The demodulator side sends the values of $T_{demod}(1)$ as feedbacks to the modulator side. The sending of such feedbacks can be triggered by either a significant change in the values of $T_{demod}(1)$ (Diff-Triggered), and/or at a constant rate (Rate-Triggered). The modulator side Monitoring Unit, after collecting profiling information from local and the demodulator side, calculates the costs for each PSE, as defined by $max(T_{mod}(1), T_{demod}(1))$, and reconfigures the modulator/demodulator pair at runtime.

VI. Evaluation

In this section, we evaluate our framework and our implementations of the two generation criteria and their corresponding cost models.

TABLE I

Object serialization and size calculation costs.

Class of Objects	Serialized size (in byte)	Serialization cost (in usec)	Size calculation cost (in usec)	Size calculation with self-desc methods(in usec)
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

A. Adaptation of Eager Handler Generated for Reducing Communication Traffic

Most of the overheads for such eager handlers are caused by the profiling code which calculates object sizes and does runtime repartitioning. Other overheads are due to the redirect code which involves the creation of redirecting argument objects. We conducted experiments to measure the overheads of eager handler implementation and compare it with manually optimized versions.

Table I lists the serialized sizes, the object serialization costs and the size calculation costs for objects of four classes. $Int100(w/owrapper)$ is an array of 100 ints. $Int100(w/wrapper)$ is an array of 100 ints enclosed in a wrapper class.

[1]For now, we assume that the amount of data passed without tiling is always greater than the minimal requirement of σ

AppBase and AppComp are defined as in Figure 5. Results in this table shows that the size calculation for complex objects is expensive, but this cost can be dramatically reduced by using size self-describing methods.

Table II shows the overhead of sampling and redirecting code in eager handlers. We compare the time used per frame for the eager handler version in our iPAQ video display example with that of manually optimized versions. The eager handler version executes two sampling and one redirecting code segments for each frame. Results show that the overhead caused by sampling and redirecting is very low compared with the total costs, both with and without self-describing methods.

Table III shows the performance of the generated version of the application is compared against two manually written versions, each optimized for one of the following two scenarios: one with window size larger than the original image size and the other smaller than the original image size. Our first two experiments apply the two scenarios to all of the three versions of the application. There is no dynamic change of scenarios during these two experiments. In the third experiment, we alternates the two scenarios. Each scenario lasts for n frames, where n is a uniformly distributed integer ranging from 1 to 20.

Experimental results show that, for a given scenario, the automatically generated eager handler has performance close to that of the manually optimized version, but much better than the non-optimized manually coded version. In the third experiment which emulates application dynamics, the eager handler version significantly outperforms the two manually written versions.

TABLE II

Overhead of automatically generated eager handlers compared with manually optimized versions.

Implementation Versions	Time per frame in usec(overhead in %)
Manually optimized	67324
Automatically generated	67433 (%0.16)
Auto Gen with Size Self-Describing Methods	67362 (0.056%)

TABLE III

Effects of Runtime Adaptation with Automatically Generated Eager Handlers (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
Automatically Generated	29.72	12.07	17.65

TABLE IV

Performance of Eager Handlers for Reducing Program Execution Time (numbers are in msecs and are averages of 5 measurements).

(Producer LIndex)/ (Consumer LIndex)	Consumer Version	Producer Version	Divided Version	Eager Handler
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

TABLE V
RUNNING ON HETEROGENEOUS PLATFORMS.

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

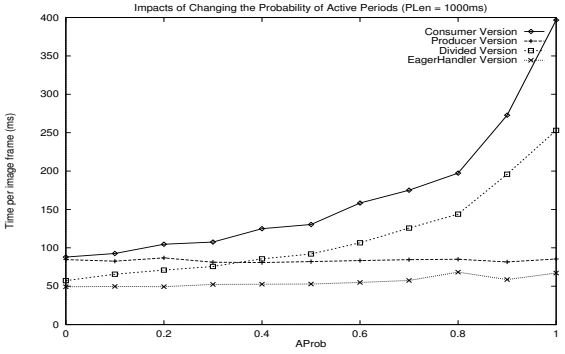


Fig. 8. Impact of Consumer-side Active Period Probability Changes (Consumer side PLen=1000ms, LIndex=0.8, producer side load-free).

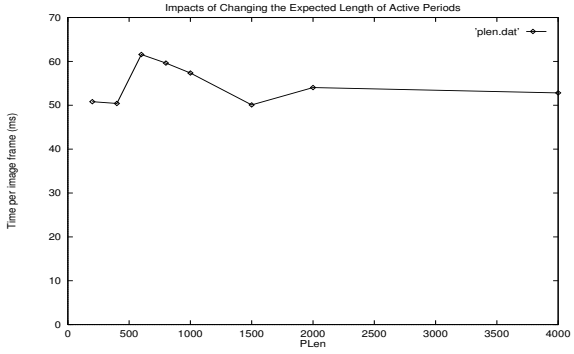


Fig. 9. Impact of Consumer-side Active Period Expected Length Changes (Consumer side AProb=0.5, LIndex=0.8, producer side load-free).

B. Adaptation of Eager Handlers Generated for Reducing Program Execution Time

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

We experiment these four versions of implementation on hosts with and without loads. Load is created by running a number of perturbation threads inside the application. Perturbation threads shared common active and idle periods. Each period consists of multiple atomic cycles. To simulate the load changes of application environments, the number of atomic cycles in a period (PLen), and the probability of perturbation threads being in an 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 spins on numeric calculation) over 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 of the four tested implementations.

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

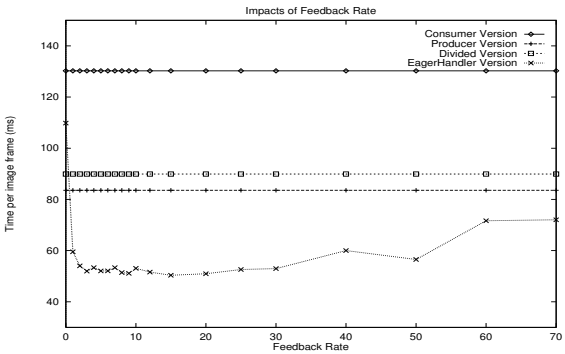


Fig. 10. Impact of Feedback Rate (Consumer side AProb=0.5, PLen=500ms, LIndex=0.8, producer side load-free).

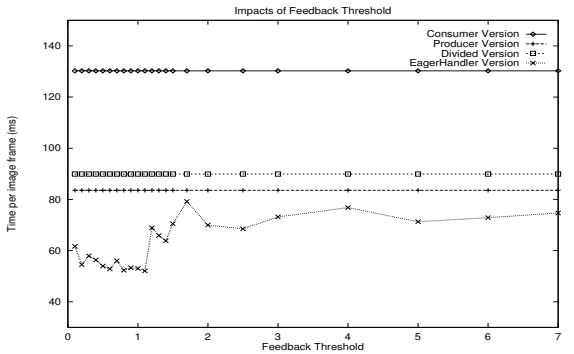


Fig. 11. Impact of Feedback Threshold (Consumer side AProb=0.5, PLen=500ms, LIndex=0.8, producer side load-free).

Intel/Linux cluster are connected with a gigabit switch.

Table V shows the performance of the four versions on heterogeneous platforms with no perturbation. In the first column are the numbers for running producer on a SUN workstation and the consumer on a Intel server. The second column shows the opposite. The results demonstrates that the eager handler version clearly outperforms (sometimes remarkably outperforms) the three other versions.

The rest of the tests are carried out inside the Intel/Linux cluster. We first compare the performances of the 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 IV lists numbers for four versions under different load distributions. Again, the eager handler version is clearly better than the other three. The reason that it outperforms the $DividedVersion$ even when there is no load is because it can better balance the load by doing loop distribution.

Figure 8 depicts the performance of the four versions under varying $AProb$ at the consumer side, while the producer side remains load-free. Figure 9 shows the impact of $PLen$ on the performance of the eager handler version on the consumer side, while the producer side remains load-free.

As we have discussed earlier, the Monitoring Unit in the modulator side depends on the feedbacks from the demodulator side Monitoring Unit for better adaptation. Feedbacks can be Rate-Triggered (a number of event have been received since last feedback sent) or Diff-Triggered (the execution time has changed considerably since last feedback sent). We used a combination of the two in our previous test. In this test, we disable one triggering mechanism to measure the other mechanism's effects on the performance. Figure 10 depicts the effects of changing the number of demodulator feedbacks in every 100 images frame with the Diff-Triggered mechanism disabled, while figure 11 shows the effects of changing difference threshold (as the difference in processing time for every five image frames) when the Rate-Triggered mechanism is disabled. Results demonstrate that for the consumer side perturbation defined as $PLen = 500ms$, $AProb = 0.5$ and $LIndex = 0.8$, a feedback rate between 2 to 30 and a difference threshold between 0.5 to 1 is ideal. Higher feedback rate will introduce higher overhead (although comparing the feedback size and the size of each image frame, such overhead is relatively small) as well as undesired "hasty" responses to "noises" in the measurements.

C. Discussion of Results

Our framework demonstrated remarkable ability in adapting to environmental and application dynamics in these experiments on the two applications with different resource constraints. It exhibited low overhead in both applications, and its performance 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 costs for the repartitioning algorithm.

Our experiments does not count in the costs for eager handler installation. Such costs consist of the cost of transporting the modulator to the producer, and the costs for loading classes used by the modulator (including redirect argument classes, and monitoring and repartitioning classes). These costs are determined by the sizes of the classes and the size of the modulator object, which are very 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 will increase the sizes of the modulator and demodulator classes because of the instrumentation codes (about 150 bytes per PSE in experiments).

VII. Related Work

Our work is related to the Active Streams project[26], which explores application-level distributed stream adaptation by dynamically deploying location-independent functional units along the event streams. However, Active Streams does not maintain meta-information needed for dynamic handler (re)partitioning, which is used in our framework for making fine-grain adjustments to stream processing that are not easily implemented in other systems.

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[27]. The relationship of JECho to the InfoSphere project mirrors its relationships to other ongoing, high profile research efforts addressing wide-area and ubiquitous computing, including the Oxygen, Gryphon and one.world projects. In particular, the Gryphon system is developing advanced messaging systems for complex distributed applications, by focusing on content-based data delivery to very large numbers of clients (e.g., hundreds of thousands)[28]. The Oxygen project has the goal of providing a "pervasive, embedded, nomadic, eternal" system for future computing environments, where computation will be as pervasive as the oxygen in the air[29]. The one.world targets for a system architecture that provides an integrated and comprehensive framework for building pervasive applications[30]. However, to our knowledge, none of these systems addresses the runtime adaptation to dynamic resource constraints in a pervasive computing environment. We have provided adaptations for both volume of data communication as well as minimization of completion time.

VIII. Conclusion and Future Work

Many applications in pervasive computing environments are subject to resource constraints in terms of limited bandwidth and processing power. As such applications grow in scale and complexity, these constraints become increasingly difficult to predict at design and deployment times. Runtime adaptation is hence required for the dynamics in such constraints. However, to maintain the lightweightness of such adaptation, it is important to statically gather relevant program information to reduce the runtime overhead of dynamic adaptation. This paper presented methods that use both static program analysis and runtime profiling to support the adaptation of producer/consumer-style pervasive applications. It demonstrated these methods with a network traffic-centric cost model and a program execution time-centric cost model. A communication bandwidth critical application and a computation intensive application were used to demonstrate the significant performance improvement opportunities offered by these methods under the presence of respective resource constraints.

Our future work will focus on other eager handler generation criteria such as to reduce the latency of event processing, or to reduce the power consumption on either the producer side or the consumer side. Investigation of better partitioning methods both in terms of adaptations as well as overheads will also be attempted. Currently our method uses edge profiles but some refinements can be done using path profiles. The key question is how to gather path profiles quickly that are necessary for adaptations? We plan to develop a demand-driven framework for this purpose. We will also investigate the benefits and overheads in eager handlers more complicated than Modulator Read-Only.

References

[1] N.B. Priyantha, A. Chakraborty, and H. Balakrishnan, *The Cricket Location-Support System*, Proc. 6th ACM MOBICOM, Boston, MA, August 2000.

[2] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek and H. Balakrishnan, *Chord: A Peer-to-Peer Lookup Service for Internet Applications*, Proc. ACM SIGCOMM Conf., San Diego, CA, September 2001.

[3] H. Balakrishnan, *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*, PhD thesis, University of California at Berkeley.

[4] A.C. Snoeren, H. Balakrishnan, and M.F. Kaashoek, *The Migrate Approach to Internet Mobility*, Proceedings of the Student Oxygen Workshop (SOW '01), Gloucester, MA, July 2001.

[5] C. Perkins, A. Myles, and D. Johnson, *The Internet Mobile Host Protocol (IMHP)*, Proceedings of INET '94, The Annual Conference of the Internet Society, June 1994.

[6] A. Myles, and D. Skellern, *Comparing four ip based mobile host protocols*, Computer Networks and ISDN Systems, 26:349–355, 1993.

[7] L. Magalhaes, and R. Kravets, *Transport Level Mechanisms for Bandwidth Aggregation on Mobile Hosts*, Proceedings of the 9th International Conference on Network Protocols (ICNP 2001), 2001.

[8] R. Kravets, K. L. Calvert, and K. Schwan, *Power-Aware Communication for Mobile Computers*, Proceedings of the Sixth International Workshop on Mobile Multimedia Communications (MoMuc-6), 1999.

[9] L. N. Chakrapani, P. Korkmaz, V. J. Mooney III, K. V. Palem, K. Puttaswamy and W. F. Wong *The Emerging Power Crisis in Embedded Processors: What can a (poor) Compiler do?*, Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'01), pp. 176-180, November 2001.

[10] H. Aydin, R. Melhem, D. Moss, and Pedro Mejia Alvarez, *Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems*, Proceeds of RTSS'01 (Real-Time Systems Symposium), London, England, Dec 2001.

[11] E. Shih, S.-H. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan, *Physical Layer Driven Algorithm and Protocol Design for Energy-Efficient Wireless Sensor Networks*, Proceedings of MOBICOM 2001, Rome, Italy, July 2001.

[12] S. Rele, S. Pande, S. Onder, and R. Gupta, *Optimizing Static Power Dissipation by Functional Units in Superscalar Processors*, Proceeds of International Conference on Compiler Construction (CC), Grenoble, France, April 2002.

[13] A. Bhalgat, and S. Pande, *Efficient Register Allocation to Arrays in Loops for Embedded Code Generation*, Proceedings of 2nd ACM Workshop on Media Processors and DSPs, Monterey, California, December 2000.

[14] J. Flinn, and M. Satyanarayanan, *Energy-aware adaptation for mobile applications*, Proceedings of the 17th ACM Symposium on Operating Systems Principles.

[15] D. Zhou, K. Schwan, G. Eisenhauer, and Y. Chen, *JECho - Interactive High Performance Computing with Java Event Channels*, Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS 2001), April 2001.

[16] 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), San Francisco, USA, Dec. 1997.

[17] M. Weiser, *The Computer for the Twenty-First Century*, Scientific American, pp. 94-10, September 1991.

[18] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper, *Implementing a Sentient Computing System*, IEEE Computer, Vol. 34, No. 8, August 2001 pp 50-56.

[19] C. Kidd, et al. *The Aware Home: A Living Laboratory for Ubiquitous Computing Research.*

[20] S. Narayanaswamy, S. Seshan, E. Amir, E. Brewer, R. Brodersen, F. Burghardt, A. Burstein, Y. Chang; A. Fox, J. Gilbert, R. Han, R. Katz, A. Long, D. Messerschmitt and J. Rabaey, *A Low-Power, Lightweight Unit to Provide Ubiquitous Information Access: Application and Network Support for InfoPad*, IEEE Personal Communications, no. 2, April 1996, pp 4-17.

[21] T. Truman, T Pering, R. Doering, and R. Brodersen, *The InfoPad Multimedia Terminal: A Portable Device For Wireless Information Access*, IEEE Transactions on Computers, vol. 47, no. 10, October 1998, pp. 1073-1087.

[22] J. M. Kahn, R. H. Katz, and K. S. J. Pister, *Next century challenges: mobile networking for "Smart Dust"*, MobiCom 1999, pp. 271-278.

[23] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan, *Building Efficient Wireless Sensor Networks with Low-Level Naming*, Proceedings of the 18th ACM Symposium on Operating Systems Principles.

[24] N. Bulusu, J. Heidemann, and D. Estrin, *Adaptive Beacon Placement*, ICDCS 2001.

[25] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu and D.k Amin *Operational Information Systems - An Example from the Airline Industry*, First Workshop on Industrial Experiences with Systems Software (WIESS), 2000.

[26] The Active Streams Project, http://www.cc.gatech.edu/systems/projects/AStreams/

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

[28] Gryphon, http://www.research.ibm.com/gryphon/.

[29] MIT Project Oxygen, http://oxygen.lcs.mit.edu/

[30] one.world, http://one.cs.washington.edu/
[31] 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.