Optimizing General-Purpose Software Instrumentation Middleware Performance for Distributed Real-time and Embedded Systems

Dennis C. Feiock and James H. Hill
Department of Computer and Information Science
Indiana University-Purdue University Indianapolis,
Indianapolis, IN USA
Email: dfeiock@iupui.edu, hillj@cs.iupui.edu

***Abstract*—Software instrumentation is an important aspect of software-intensive distributed real-time and embedded (DRE) systems because it enables real-time feedback of system properties, such as resource usage and component state, for performance analysis. Although it is critical not to collect too much instrumentation data to ensure minimal impact on the DRE system's existing performance properties, the design and implementation of software instrumentation middleware can impact how much instrumentation data can be collected. This can indirectly impact the DRE system's existing properties and performance analysis, and is more of a concern when using general-purpose software instrumentation middleware for DRE systems.**

This paper provides two contributions to instrumenting software-intensive DRE systems. First, it presents two techniques named the *Standard Flat-rate Envelope* and *Pay-per-use* for improving the performance of software instrumentation middleware for DRE systems. Secondly, it quantitatively evaluates performance gains realized by the two techniques in the context the *Open-source Architecture for Software Instrumentation of Systems (OASIS)*, which is open-source dynamic instrumentation middleware for DRE systems. Our results show that the Standard Flat-rate Envelope improves performance up to 57% and the Pay-per-use improves performance up to 49%.

I. Introduction

Software instrumentation [1] is the primary method for collecting data from a distributed real-time and embedded (DRE) system. With software instrumentation, it is possible to collect instrumentation data related to resource usage (*e.g.*, CPU, memory, and network usage), system state (*e.g.*, the liveliness of both hardware and software components), and system execution profiles (*e.g.*, system execution trace). Depending on the type of software instrumentation employed, instrumentation data can be collected and stored locally [1], [2], remotely (outside the DRE system's execution network) [3], [4], or a combination of both [2]. Irrespective of how instrumentation data is collected and stored, the goal of instrumenting a DRE system is to support analysis of system properties.

Traditionally, instrumentation middleware for DRE systems is custom designed and implemented for the DRE system undergoing software instrumentation [5]. This is because DRE systems have stringent performance requirements that can warrant a custom framework designed around what metrics to collect. This design approach, however, makes it *hard* for DRE system developers to incorporate new metrics late in the software lifecycle, and does not promote reuse of "business-logic" related to real-time instrumentation of DRE systems.

Because there is need for general-purpose instrumentation middleware that targets DRE systems—similar to general-purpose middleware for DRE systems based on open-standards [6]–[8] that address other concerns—the *Open-source Architecture for Software Instrumentation of Systems (OASIS)* [9] was incarnated. Like most general-purpose middleware solutions [10], [11], OASIS is highly configurable and designed to support a range of DRE system domains, middleware technologies, and programming languages. These features of OASIS, like most general-purpose middleware solutions, are met by compromising some aspects of performance through the use of abstractions [12].

Although general-purpose instrumentation middleware for DRE systems may be less optimal than custom instrumentation middleware, it is still necessary to examine different optimization techniques that can potentially improve performance. This is because it not only helps the general-purpose middleware get close to optimal, but it also describes optimization techniques that may be applicable in other application domains. Based on this desire to continue improving the performance of general-purpose middleware solutions, such as OASIS, the main contributions of this paper are as follows:

- It discusses the *Standard Flat-rate Envelope* optimization technique, which abstracts networking middleware technologies while delaying translation costs, such as performance, until the data is read;
- It discusses the *Pay-per-use* optimization technique, which reduces packaging overhead by writing an object's state directly to a predetermined packaging location; and
- It quantitatively evaluates performance gains of the Standard Flat-rate Envelope and Pay-per-use techniques in the context of OASIS.

Results from our evaluation show the Standard Flat-rate Envelope can improve performance up to 57% and the Pay-per-use can improve performance up to 49%.

Paper organization. The remainder of this paper is organized as follows: Section II provides a brief overview of

OASIS; Section III explains the Standard Flat-rate Envelope Pattern and the Pay-per-use Pattern; Section IV discusses the results of our quantitative experiments; Section V compare our work with other related work; and Section VI provides concluding remarks and lessons learned.

II. A Brief Overview of OASIS

OASIS is real-time instrumentation middleware for DRE systems that uses a metametics-driven design integrated with loosely-coupled data collection facilities. Figure 1 provides a detailed overview of OASIS's architecture.

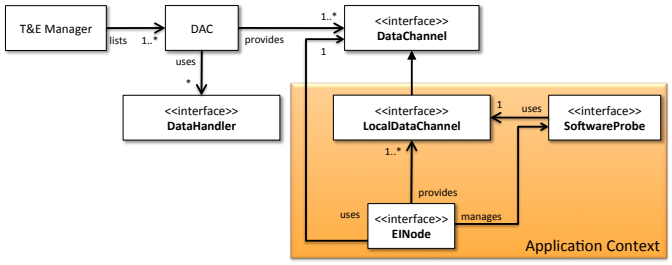

Fig. 1. An architectural diagram of OASIS that highlights the key interfaces and objects.

As shown in this figure, OASIS's architecture has the following interfaces and objects that are important to the remainder this paper:

- **Software Probe.** The software probe is the entity responsible for collecting metrics from the DRE system undergoing software instrumentation. The instrumentation data (or metrics) can be application-level (*e.g.*, the value of application variables, number of events received by a component, and a component's state) or system-level (*e.g.*, memory, CPU, and network usage stats). Lastly, software probes can be either client-driven (*i.e.*, flushed by application-level code) or autonomous agents (*i.e.*, active objects) that collect instrumentation data at a user-defined rate.
- **Embedded Instrumentation Node.** The Embedded Instrumentation (EI) Node bridges locality constrained abstractions (*i.e.*, local data channels) with networking-enabled abstractions (*i.e.*, data channels). When the EI Node receives packaged instrumentation data as a data packet, it prepends its information (*e.g.*, UUID, packet number, timestamp, and hostname) to the data packet, and passes it to a data channel (explained later).
- **Local Data Channel.** The local data channel is an abstraction that regulates control between a software probe and the EI Node. There can be different types of local data channel implementations. For example, one implementation could speculate on collected data—passing it along only if it is of interest. Another implementation could queue collected instrumentation data if there is no network connection. This design approach allows the application undergoing instrumentation to strategize how data is handled after it is collected and before sending it over the network.
- **Data Channel.** The data channel is an abstraction that sends collected instrumentation over the network to a Data Acquisition and Controller. It therefore supports inter-process communication unlike its sibling—the local data channel. There can be many different implementations of a data channel. For example, one implementation of a data channel could use CORBA [7] and another implementation can use Data Distribution Services (DDS) [13]. This design approach allows the application undergoing instrumentation to select the most appropriate networking middleware technology to send collected instrumentation data over the network.
- **Data Acquisition and Controller.** The Data Acquisition and Controller (DAC) is responsible for receiving collected instruction data from the application undergoing software instrumentation and controlling access to it. The DAC also manages *data handlers*, which are objects that act upon received instrumentation data. For example, an archive data handler stores collected metrics in a relational database; a DDS data handler transforms collected instrumentation data to DDS events and publishes it; and a WebSocket data handler that publishes data over WebSockets [14] for viewing collected data over the Web. This design approach allows OASIS to abstract away the collection facilities from data handling facilities, and places data handling facilities outside of the DRE system's execution domain.
- **Test and Execution Manager.** The Test and Execution (TnE) Manager is a naming service for DACs. It is therefore the main entry point into OASIS for external clients that want to access collected instrumentation data.

As discussed above, OASIS has many abstractions that improve both its flexibility and configurability. Abstractions, however, can reduce performance due to different levels of indirection. It is therefore necessary to understand how to improve performance without compromising flexibility and configurability provided by abstractions. With this in mind, the remainder of this paper discusses two different techniques implemented in OASIS to improve its performance without compromising its generality.

III. Optimization Techniques Implemented in OASIS to Improve Performance

This section discusses the design and implementation of two optimization techniques named Standard Flat-rate Envelope and Pay-per-use that are implemented in OASIS to improve its performance. Each technique is presented using the accepted approach for presenting software design patterns [15] because it promotes discussion of overall design, collaborators, and consequences so developers have a more in-depth understanding of the technique.

A. Standard Flat-rate Envelope

1) Intent: Abstract networking middleware technologies while delaying translation costs until the data is read.

2) Motivation: In OASIS, the DAC receives instrumentation data via different networking middleware technologies. For example, the DAC can receive instrumentation data on a TAO [11] or OpenSplice [16] endpoint. When the DAC receives instrumentation data, it then forwards it to different data handlers. Each data handler, however, will have different intentions for the instrumentation data. For example, one data handler may store the instrumentation data in a relational database, whereas another may convert the instrumentation data to a different object model.

One approach to address this problem is to have a standard object model that is used to pass collected instrumentation data between each data handler. Each networking middleware endpoint exposed by the DAC is then responsible for converting the received instrumentation data to the standard object model. Although this design is feasible, the receiving endpoint must pay an upfront cost to pass collected instrumentation data to the data handler—even if none of the content is accessed.

A better design approach is to let each networking middleware endpoint exposed by the DAC receive data using its own object model. The endpoint-specific object model is then encapsulated in a standards-based object model and passed to the data handlers. The data handlers then can access the data using a standard interface. This design approach does not require each receiving endpoint to pay an upfront cost for receiving collected instrumentation data (*i.e.*, there is a flat-rate above and beyond normal middleware cost to receive data and pass it along). Instead, cost is paid if and when a data handler accesses the collected instrumentation data.

3) Applicability: Use the Standard Flat-rate Envelope when:

- you want to delay the cost of accessing data until the data is actually being accessed.
- you need to provide a standard method for accessing data without using a standard object model upfront (*i.e.*, converting data to the standard object model).
- you want to expose multiple networking middleware technologies for receiving data, and want to provide a standard method for working with each one.

4) Structure: Figure 2 shows the structure of the Standard Flat-rate Envelope.

5) Participants: The participants in the Standard Flat-rate Envelope are as follows:

- **AbstractEnvelope.** Defines the standard interface for accessing the contents of the object model across different middleware technologies.
- **ConcreteEnvelope.** An implementation of the AbstractEnvelope participant that encapsulates a ConcreteData object.
- **ConcreteData.** The actual data received by its corresponding ConcreteEndpoint object.
- **DataHandler.** The object that is responsible for handling data encapsulated in ConcreteEnvelope objects.
- **DataHandlers.** A collection of DataHandler objects.
- **AbstractEndpoint.** The interface definition for all ConcreteEndpoint implementations.

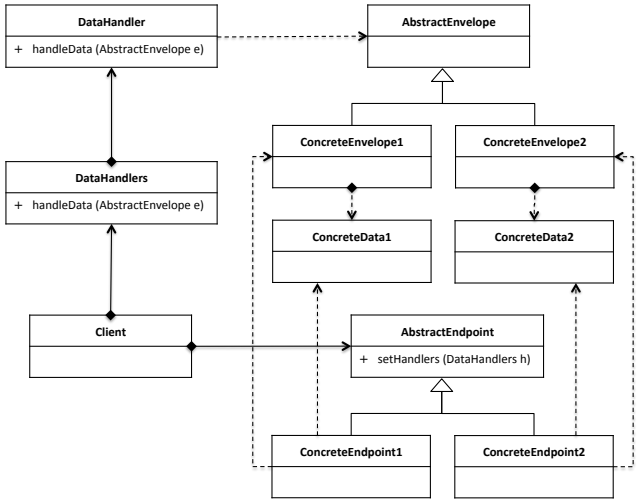

Fig. 2. Structural diagram of the Standard Flat-rate Envelope.

- **ConcreteEndpoint.** An AbstractEndpoint implementation that is the source of data using its own object model.
- **Client.** Manages both DataHandler objects and AbstractEndpoint objects.

6) Collaborations: The collaborations of the Standard Flat-rate Envelope are as follows:

- The Client initializes the ConcreteEndpoint with a reference to a DataHandlers object. This allows the ConcreteEndpoint object to send ConcreteEnvelope objects to DataHandler objects via the DataHandlers object.
- The ConcreteEndpoint object creates a ConcreteEnvelope object, and initializes the ConcreteEnvelope with received ConcreteData. The ConcreteEndpoint does not translate the ConcreteData to a standard object model.
- The ConcreteEndpoint object passes the ConcreteEnvelope object—now seen as an AbstractEnvelope object—to the DataHandlers object.
- The DataHandlers object passes the AbstactEnvelope object to each registered DataHandler object.
- The DataHandler object accesses the data using the interface defined on the AbstractEnvelope.

7) Consequences: Here are key consequences of the Standard Flat-rate Envelope:

1) *It hides the internal data representation.* The AbstractEnvelope provides the DataHandler object with an abstract interface for accessing received data. This interface lets each endpoint hide the internal representation of its object model. Moreover, it allows DataHandler objects to operate with different ConcreteEnvelope implementations uniformly.
2) *It delays the cost of accessing the data.* At some point, there must be a standard object model used to access the data—especially when dealing with complex types. Instead of converting data to the standard object model when received, it is not converted until the data is accessed. This is because the ConcreteEndpoint object

allocates a ConcreteEnvelope object that merely wraps received data. The ConcreteEnvelope is then passed to different DataHandler objects via the DataHandlers object. When the DataHandler accesses the encapsulated data it must be converted to the standard object model for that single data point.

8) Implementation: Typically, there is an AbstractEndpoint that allows the client to manage it. Also, the DataHandler defines methods for receiving objects that are of type AbstractEnvelope. The ConcreteEndpoint object is responsible for implementing the AbstractEndpoint interface, and the ConcreteEnvelope object is responsible for implementing the AbstractEnvelope interface. Aside from these implementation details, here are other implementation issues that must be considered:

1) *Cache data internally after its first access.* When the ConcreteEndpoint objects receives data, it encapsulates it in a ConcreteEnvelope. The ConcreteEnvelope does not translate the data to a standard object model. It is only when the data is accessed is it converted to a standard object model. For built-in types, such as integers and possibly strings, this conversion is trivial, if not negligible. When dealing with complex types, this conversion becomes more of a concern.
   In order to reduce performance cost of repeatedly converting between the endpoint and the standard object model, it may be beneficial to cache the converted data in the ConcreteEnvelope after its first access. This will allow faster access to the encapsulated data, and is only a one-time on-demand cost—similar to delay loading of shared libraries—but requires an additional check each time the data is accessed to ensure that it is converted. This approach therefore should be used when the cost of the conversion check is less than the cost of converting the data to the standard object model each time the data is accessed.
2) *Use free list to amortize cost of creating ConcreteEnvelope objects.* Each time a ConcreteEndpoint object receives data, it creates a ConcreteEnvelope object and passes the data the DataHandlers object. If the ConcreteEnvelope object is dynamically allocated each time, then it is recommended that the ConcreteEndpoint object allocate is corresponding ConcreteEnvelope objects from a free list. This will help reduce negative performance impact caused by the *excessive dynamic allocation* performance anti-pattern [17].
3) *Should abstractions be used for complex types in the standard data/object model?* As explained above, converting basic types in an endpoint's object model to its corresponding type in the standard object model is trivial. When dealing with complex types, *e.g.*, abstract data types, in the standard object model, it may be necessary to use abstractions to represent those types—similar to the relationship between the AbstractEnvelope and the ConcreteEnvelope. This is because each endpoint will have different object models for each contained complex type. This implementation concern therefore implies that the relationship between the AbstractEnvelope and the ConcreteEnvelope can be applied recursively throughout the standard object model to contained objects.

9) Sample Code: The following C++ code implements the Standard Flat-rate Envelope based on how it is implemented in OASIS. As shown in Listing 1, the `DataPacket` class is the AbstractEnvelope object that defines the interface implemented by all ConcreteEnvelope objects. Likewise, the `DataHandler` is the data handler object that interacts with the data encapsulated in ConcreteEnvelope objects.

```
1  class DataPacket {
2  public:
3    virtual ~DataPacket (void) { }
4    virtual char * data (void) const = 0;
5    virtual bool byte_order (void) const = 0;
6    // Other interface methods...
7
8  protected:
9    DataPacket (void) { }
10 };
11
12 class DataHandler {
13 public:
14   virtual ~DataHandler (void) { }
15   virtual void handle_data (const DataPacket & dp) { }
16
17 protected:
18   DataHandler (void) { }
19 };
```

Listing 1. Variation of the DataPacket and DataHandler implementation in the OASIS for the Standard Flat-rate Envelope.

As shown in Listing 2, the `DataHandlerSet` class is a simple wrapper class around a set collection. It also implements a method named `handle_data()` that distributes `DataPacket` objects to registered `DataHandler` objects.

```
1  class DataHandlerSet {
2  public:
3    ~DataHandlerSet (void) { }
4
5    void insert (DataHandler * handler) {
6      this->handlers_.insert (handler);
7    }
8
9    void handle_data (const DataPacket & dp) {
10     std::set <DataHandler *>::iterator
11       iter = this->handlers_.begin (),
12       iter_end = this->handlers_.end ();
13
14       for (; iter != iter_end; ++ iter)
15         (*iter)->handle_data (dp);
16   }
17
18 protected:
19   std::set <DataHandler *> handlers_.
20 };
```

Listing 2. Variation of the DataHandlerSet implementation in the OASIS for the Standard Flat-rate Envelope.

Listing 3 shows the base class for all `Endpoint` objects. It has one main method named `set_handlers()` that sets the collection of registered DataHandler objects. This allows the endpoint to pass received data to the DataHandler objects.

```
1  class Endpoint {
2  public:
3    virtual ~Endpoint (void) { }
4
5    void set_handlers (DataHandlerSet * handlers) {
6      this->handlers_ = handlers;
```

```
  }

protected:
  DataHandlerSet * handlers_.
};
```

Listing 3. Variation of the Endpoint implementation in the OASIS for the Standard Flat-rate Envelope.

As shown in Listing 4, integrating the TAO (www.dre.vanderbilt.edu/TAO) middleware into OASIS requires implementing both a ConcreteDataPacket object (*i.e.*, `TaoDataPacket`) and a ConcreteEndpoint (*i.e.*, `TaoDataChannel`). The `TaoDataPacket` is also an object Adapter [15] for the TAO implementation of the DataPacket. As shown in this listing, the `TaoEndpoint` implements its `DataChannel` interface (see Figure 1 in Section II). More specifically, the `handle_data()` method encapsulates the received data packet in a `TaoDataPacket` object, and passes it to the set of registered `DataHandler` objects.

```
class TaoDataPacket : public DataPacket {
public:
  TaoDataPacket (const OASIS::DataPacket & p)
    : p_ (p) { }
  virtual ~TaoDataPacket (void) { }
  virtual char * data (void) const {
    return this->p_.data ();
  }

  virtual bool byte_order (void) const {
    return this->p_.byte_order ();
  }

  // Other interface method implementations...

private:
  // CORBA version of the data packet.
  const OASIS::DataPacket & p_;
};

class TaoEndpoint :
  public Endpoint, public OASIS_POA::DataChannel {
public:
  virtual ~TaoEndpoint (void) { }

  // Implementation of handle_data method on
  // CORBA DataChannel interface.
  virtual void handle_data (const OASIS::DataPacket & p) {
    TaoDataPacket packet (p);

    if (0 != this->handlers_)
      this->handlers_->handle_data (packet);
  }

protected:
  std::set <DataHandler *> * handlers_.
};
```

Listing 4. Variation of the ConcreteDataPacket and ConcreteEndpoint implementation in the OASIS for the Standard Flat-rate Envelope.

Lastly, the `Client` (see Listing 5) bootstraps the endpoint(s) with the `DataHandlerSet`. In OASIS, the DAC dynamically loads both `DataHandler` objects and `ConcreteEndpoint` objects from shared libraries. The source code in Listing 5 is designed to show how the main abstractions are associated with each other from an interaction point-of-view. When the `Client` is activated, it begins receiving data on its endpoints, which is passed along to each `DataHandler` object using the approach discussed above.

```
class Client {
public:
  Client (void) {
    this->tao_.set_handlers_ (&this->handlers_);
  }

  // Other methods to manage the client and endpoints.

private:
  std::set <DataHandler *> handlers_;
  TaoEndpoint tao_;
};
```

Listing 5. Variation of the Client implementation in the OASIS for the Standard Flat-rate Envelope.

10) Related Software Design Patterns: The Wrapper Facade [15] pattern is similar to the Standard Flat-rate Envelope in that the it too wraps the contents of an underlying subsystem (*i.e.*, data received on different implementations). The intent of the Standard Flat-rate Envelope, however, is different from the Wrapper Facade pattern in that the Standard Flat-rate Envelope focuses on reducing and/or delaying cost, whereas the Wrapper Facade solely focuses on providing a unified interface to a set of interfaces in a subsystem.

The Abstract Factory pattern [15] is also similar to the Standard Flat-rate Envelope's structure (see Figure 2) closely resembles the Abstract Factory pattern. Besides differing intents, *i.e.*, design vs. optimization, the Standard Flat-rate Envelope has a single product, and the Client object does not have to interact with the created products.

The Envelope [18] pattern is also similar to the Standard Flat-rate Envelope. In fact, the Standard Flat-rate Envelope is inspired by the Envelope pattern. This is because the ConcreteEndpoint is merely placing received data in an envelope, and then sending it along to the next object (*i.e.*, the DataHandler). Unlike the Envelope pattern, the envelope recipient does not unwrap the envelope. Instead, each recipient accesses the envelope's contents using the abstract interface defined on the AbstractEnvelope.

B. Pay-per-use

1) Intent: Reduce packaging overhead by writing an object's state directly to a predetermined packaging location.

2) Motivation: In OASIS, software probes are responsible for packaging instrumentation data in common data representation (CDR) format before passing it to the local data channel. In some cases, not all data points collected by a software probe are updated at the same time. For example, if a software probe is application-driven and different parts of the application's execution, such as execution blocks or threads, update different parts of the software probes values, then there is chance that the current state of the software probe is flushed without updating all the values. This scenario, however, does not constitute a incomplete sample set depending on the software probe's intent.

Because of this use case, the software probe should only package the data that is set (or used). It should not have to pay the price for packaging data that has not been updated. Likewise, if all data points in the software probe are used, then the design approach for this pay-per-use goal would have

similar performance to packaging all data points with disregard for what data has been set. Finally, it is necessary to clarify this optimization technique does not mean some data points will be omitted from the packaging. Instead, this approach focus on packaging all data points by only updating the values for data points that have changed since the last data flush.

3) Applicability: Use Pay-per-use when:

- you do not want to pay the cost of packaging all the data points each time when only a few of the data points change between updates.
- you want to use the same buffer to package data each time the data is collected.

4) Structure: Figure 3 shows the structure of Pay-per-use.

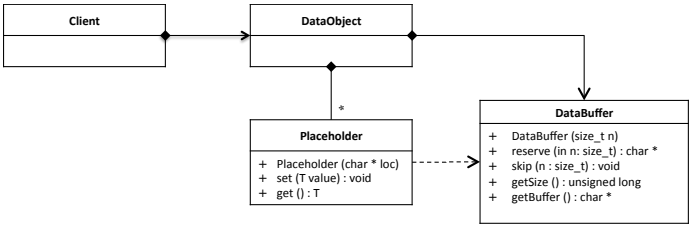

Fig. 3. Structural diagram of the Pay-per-use pattern.

5) Participants: The participants in Pay-per-use are as follows:

- **DataBuffer.** The target buffer used for packaging individual data points.
- **DataObject.** The aggregation of the data points that defines methods for setting/getting each individual data point.
- **Placeholder.** Holds an individual data point's memory location in the DataBuffer object. The Placeholder object also provides methods for setting and getting the data point's value in the DataBuffer object.
- **Client.** Use the DataObject object to set/get individual data points.

6) Collaborations: The collaborations of Pay-per-use are as follows:

- The DataObject object contains a DataBuffer object. When the DataObject object is initialized, it reserves a location for each data point in the DataBuffer object. This location is then used to initialize the data point's corresponding Placeholder object.
- The DataObject object sets/gets the values for each individual data point via its corresponding Placeholder object.
- The Placeholder object sets/gets its corresponding data point's value using the memory location reserved in the DataBuffer object.

7) Consequences: Here are key consequences of Pay-Per-use:

1) *Reserves memory ahead of time.* When the DataObject is initialized, the DataObject reserves space for each individual data point in the DataBuffer. When the individual data points are packaged into the DataBuffer object, the DataBuffer does not have to grow itself on-demand. Instead, the Placeholder object updates its reserved memory location in the DataBuffer object.
2) *No copy needed.* Because space is being reserved in the DataBuffer object and updated directly by the Placeholder object, there is no need to keep a local variable in the DataObject object that will eventually be copied into the DataBuffer object per the packaging standards.
3) *Reuses the same DataBuffer object.* In relation with the previous consequences, the contained buffer in the DataBuffer object can be reused. This is because the structure (and size) of the DataObject object does not change. This also implies that the size of the DataBuffer does not change. Once memory has been reserved in the DataBuffer object, the DataObject object can always use that memory location—similar to a local variable in a class. When dealing with variable-length data points, such as strings and arrays, this assumption does not hold. We, however, discuss a solution for handling variable-length data points in the Implementation section.
4) *Indirect access to individual data points.* The DataObject object is not able to access an individual data point's value directly. This is because the Placeholder object controls access to its corresponding data point's value via a reference to a memory location.

8) Implementation: Although implementation of Pay-per-use is usually straightforward, here are some issues to keep in mind:

1) *Generic vs. function overloaded vs. type-specific placeholder objects.* The target programming language will dictate the Placeholder object's implementation. For example, in C++ it is possible to implement the Placeholder object as a template class where the template parameter is the data point's type. This implies the Placeholder object needs a setter method that uses the parameterized type as an input parameter, and a getter method where the parameterized type is a return pattern. For programming languages that do not support templates or generics, the Placeholder object can overload the setter/getting methods for each data type supported. This implementation, however, requires the DataObject object to invoke the correct setter/getter method on the Placeholder object.
   Finally, the Placeholder object can be implemented using the Strategy pattern [15] where the Placeholder class is a degenerate class. Type-specific Placeholder classes, *e.g.*, IntegerPlaceholder, then subclasses the Placeholder class and is used directly by the DataObject object. This approach alleviates the concern of the DataObject object invoking the correct setter/getting method on the Placeholder object (as discussed above). Using type-specific Placeholder objects, however, requires manually creating a Placeholder class for each supported type.
2) *Handling fixed-length and variable-length data points.* Using Pay-per-use with fixed-length data is less complex

when compared to variable-length data. This is because with fixed-length data, the size of the DataBuffer object's underlying data buffer is known *a priori*. When using with variable-length data points, it is hard to know the size of the DataBuffer object's underlying data buffer *a priori*. It is therefore necessary to use multiple DataBuffer objects when the DataObject has variable-length data points.

Although variable-length data warrants usage of multiple DataBuffer objects, it is also possible to predict how many DataBuffer objects are needed to support both fixed- and variable-length data. For example, if the DataObject object has all fixed-length data points, then it needs 1 DataBuffer object. If the DataObject object has in the following order: fixed-length, variable-length, variable-length, fixed-length data, then it needs 4 DataBuffer objects. Based on this simple analysis, the number of DataBuffer objects needed is equal to the sum of variable-length data points and fixed-length data point blocks (*i.e.*, a group of contiguous fixed-length data points). This implementation approach, however, requires usage of input/output (I/O) vectors to send data so individual data buffers are not required to be packaged into a single contiguous data buffer before passing the packaged data along.

9) Sample Code: The following C++ code implements Pay-per-use based on how it is currently implemented in OASIS. As shown in Listing 6, the `DataBuffer` class is created with an initial size, which sets the size of the internal data buffer. The DataBuffer class also provides two methods for managing the internal data buffer. The `reserve()` method allocates space in the data buffer, and returns the location of the allocated space. The `skip()` method pads the data buffer with unused space. Finally, the `DataBuffer` class has two accessor methods.

```
class DataBuffer {
public:
  class out_of_space : public std::exception { };
  class invalid_size : public std::exception { };

  DataBuffer (size_t n)
    : buffer_ (new char [n]), curr_ (buffer_), size_ (n) { }

  ~DataBuffer (void) {
     if (0 != this->buffer_) delete [] this->buffer_;
  }

  char * reserve (size_t n) {
    if (n == 0) throw invalid_size ();
    if (this->curr_ + n > this->buffer_ + this->size_)
      throw out_of_space ();

    char * loc = this->curr_;
    this->curr_ += n;

    return loc;
  }

  void skip (size_t n) { this->reserve (n); }

  size_t size (void) const { return this->size_; }
  char * buffer (void) const { return this->buffer_; }

private:
  char * buffer_;    // The underlying buffer for rsvps
  char * curr_;      // Current location in rsvp buffer
  size_t size_;      // Size of the data buffer
};
```

Listing 6. Variation of the DataBuffer implementation in the OASIS for Pay-per-use.

Listing 7 shows how the `Placeholder` class is currently implemented in OASIS. As shown in the listing, the Placeholder class is a template class that is initialized with a memory location. The location is then cast to the concrete parameter type. Finally, the `Placeholder` class has setter/getter methods (as overloaded operators) for updating/reading the actual value. It is worth noting that the `Placeholder` class in Listing 7 only works for fixed-length types, such as integers and real numbers.

```
template <typename T>
class Placeholder {
public:
  Placeholder (char * loc)
    : loc_ (reinterpret_cast <T *> (loc) { }

  ~Placeholder (void) { }

  void operator = (const T & value) { *this->loc_ = value; }
  operator T (void) const { return *this->loc_; }

private:
  T * loc_;       // Location of the actual value.
};
```

Listing 7. Variation of the Placeholder implementation in the OASIS for Pay-per-use.

Finally, Listing 8 illustrates an implementation of the `DataObject` class named `MemoryProbe` for collecting memory usage stats. As shown in this listing, the data buffer is initialized to the total size of all the data points. Each data point then reserves space in the `DataBuffer` class. Lastly, the Client object updates the data points using methods exposed on the data object. When the data is ready to be sent to the DAC, the software probe flushes its contents.

```
class MemoryProbe {
public:
  MemoryProbe (void)
    : data_buf_ (32), mem_used_ (data_buf_.reserve (8)),
      mem_total_ (data_buf_.reserve (8)),
      virtual_used_ (data_buf_.reserve (8)),
      virtual_total_ (data_buf_.reserve (8)) { }

  void memory_used (unsigned long long val) {
    this->mem_used_ = val;
  }

  unsigned long long memory_used (void) const {
    return this->mem_used_;
  }

  void memory_total (unsigned long long val) {
    this->memory_total_ = val;
  }

  unsigned long long memory_total (void) const {
    return this->memory_total_;
  }

  void virtual_used (unsigned long long val) {
    this->virtual_used_ = val;
  }

  unsigned long long virtual_used (void) const {
    return this->virtual_used_;
  }
```

```
33    void virtual_total (unsigned long long val) {
34      this->virtual_total_ = val;
35    }
36
37    unsigned long long virtual_total (void) const {
38      return this->virtual_total_;
39    }
40
41    void flush (void) {
42      // flushes the buffer contents
43    }
44
45  private:
46    DataBuffer data_buf_;
47    Placeholder <unsigned long long> mem_used_;
48    Placeholder <unsigned long long> mem_total_;
49    Placeholder <unsigned long long> virtual_used_;
50    Placeholder <unsigned long long> virtual_total_;
51  };
```

Listing 8. Variation of a data object implemented in the OASIS for Pay-per-use.

10) Related Software Design Patterns: The Wrapper Facade pattern is closely related to this pattern in the both the DataObject object and Placeholder object are defining a high-level interface that make it easier to set/get values from an underlying data buffer (*i.e.*, the subsystem).

IV. Evaluating Performance Gains from Standard Flat-rate Envelope and Pay-per-use

This section discusses experimental results for the Standard Flat-rate Envelope and the Pay-per-use optimization techniques discussed in Section III. All experiments were conducted in the System Integration (SI) Lab (www.emulab.cs.iupui.edu) at Indiana University-Purdue University Indianapolis, which is powered by Emulab [19] software. Each node in the SI Lab is a Dell PowerEdge R415, AMD Opteron 4130 processor with 8GB of memory executing 32-bit Fedora Core 15 (32 bit).

A. Standard Flat-rate Envelope Performance Experiments

1) Experiment Design and Setup: For this experiment the DAC, EI Node and sample application with embedded software probes were executed on a single host using localhost for network communication. This was done so we can remove networking overhead from the measurements. Each test execution loaded a single OASIS probe and flushed its state in a loop as fast as possible. The sample application monitored the number of flushes that occurred over the test duration to calculate the average throughput. One test included executing the sample application for 2, 10 and 60 seconds for five iterations. The results for all the iterations were then averaged. Lastly, software probe with data models (or payloads) of size 0, 8, 16, 32, 64, 128 and 256 bytes were loaded by the sample application.

For this experiment, we used the following OASIS configurations:

- **No data handler.** This configuration uses the EI node, software probe and DAC, but the DAC does not load any data handlers. The purpose of this configuration is to highlight networking middleware performance without including the overhead of interacting with received instrumentation data, which decreases performance.
- **TAO handler.** This configuration uses the EI node, software probe and DAC loading the TAO publisher data handler. The TAO publisher data handler allows TAO clients to receive data in real-time. This test therefore will include additional performance overhead because the TAO data handler must examine the received instrumentation data before publishing it to the appropriate TAO clients.

2) Experiment Results: Prior to implementing the Standard Flat-rate Envelope in OASIS, a baseline performance test was completed. Figure 4 shows a comparison of throughput between the *no data handler* and *TAO handler* OASIS configurations.

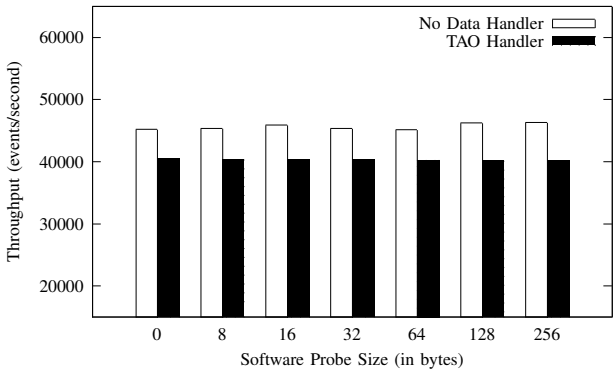

Fig. 4. Baseline throughput results comparing the No Data Handler and TAO Handler configurations.

As shown in this figure, when the DAC receives the software probe's instrumentation data using a networking middleware technology, the data is converted to a standard object model and then passed to any loaded data handlers. For the *No Data Handler* test, the cost of conversion is incurred without needing the instrumentation data. The performance decrease observed in Figure 4 shows that the relative implementation cost of the TAO publisher data handler is 11%.

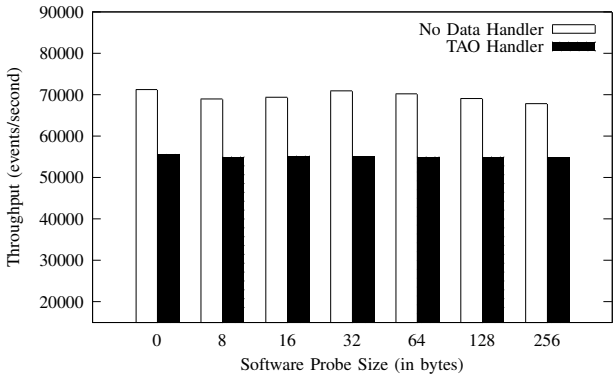

Fig. 5. Standard Flat-rate Envelope pattern throughput results comparing the No Data Handler and TAO Handler configurations.

After implementing the Standard Flat-rate Envelope in OASIS, the performance tests were repeated. The results for

this test are shown in Figure 5. As shown in this figure, performance decreased by 28% when the TAO data handler was loaded by the DAC.

As discussed in Section III-A, the Standard Flat-rate Envelope seeks to abstract networking middleware technologies while delaying translation costs until the data is read. These results show that the relative performance cost of the TAO publisher handler increased from 11% to 28% between these two implementations. The cost of the DAC receiving the instrumentation data from the networking middleware technology and sending it a data handler therefore has been significantly reduced. This is especially important in OASIS because the DAC is part of its core whereas data handlers are extensions designed and implemented by third-party developers. The Standard Flat-rate Envelope therefore helps the developer who design and implemented the data handler better understand its performance overhead.

Overall, delaying conversion of the instrumentation data in the *No Data Handler* configuration yielded a performance increase between 46.44% and 57.55%. This performance increase was reduced to between 36.31% and 37.58% for the *TAO Handler* configuration. This pattern therefore moves the cost of processing instrumentation data to the data handlers, which can be optimized accordingly.

B. Pay-per-use Performance Experiments

1) Experiment Design and Setup: For this experiment, the EI node and sample application with embedded software probes were executed on a single system. Each test execution loaded a 256 byte OASIS software probe that consisted of four 8-byte values. We then implemented a loop that set a different number of data points in the software probe, and flushed the software probe's state as fast as possible. The sample application monitored the number of flushes that occurred over the test duration calculated the average throughput. One test included executing the sample application for 2, 10, and 60 seconds for five iterations. The results for all the iterations were then averaged. Lastly, we used the an OASIS configuration where no DAC is executed to highlight the software probe's packaging performance without including network overhead.

2) Experiment Results: Prior to implementing Pay-per-use in OASIS, a baseline performance test was executed. After implementing Pay-per-use in OASIS, the performance test was repeated. Figure 6 compares the throughput of these two implementations using the *No DAC* configuration.

As shown in this figure, Pay-per-use addresses two major packaging concerns: excessive resizing the data buffer, and packaging every data point regardless of value change. The baseline implementation preallocates the header in the stream, but not the data points. As a result, the CDR data buffer must grow for each data point during the packaging process. After sending the packaged data points to the EI Node, the CDR data buffer must be reset accordingly. This growing and shrinking of the CDR data buffer results in a high performance cost.

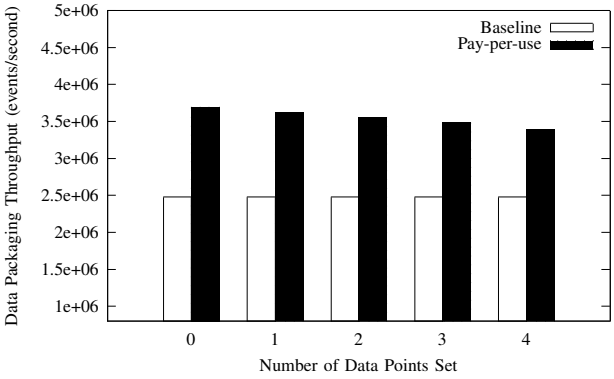

Fig. 6. Pay-Per-use throughput results using the No DAC configuration.

Implementing Pay-per-use addresses this concern by preallocating a fixed-sized CDR data buffer. Because the 256-byte software probe does not have variable-length values, initializing the CDR data buffer to a fixed-size drastically reduces packaging overhead. Figure 6 displays the impact of this change using the *No DAC* configuration that set all four data points and resulted in a 36.58% performance increase.

By supporting fixed-sized CDR data buffers, the cost of packaging the data point occurs when the value is set using the DataObject object. Because the CDR data buffer is also reused between flushes, the cost of packaging unset data points is avoided and the previous value is used. This is a drastic improvement when compared to the baseline. As shown in Figure 6, throughput remains constant across all baseline tests. The Pay-per-use pattern, however, looses between 1.84% and 2.61% of it's throughput per data point set. These results therefore support the pattern's name and consequences.

C. Importance of Results to DRE Systems

DRE systems typically have limited resources and strict quality-of-service (QoS) requirements. It is therefore important that different concerns of a DRE system, such as software instrumentation, have optimal performance. As shown in the results above, the optimization techniques have improved the performance of the OASIS middleware. This is due in part to Standard Flat-rate Envelope and Pay-per-use embodying principles of implementing fast networked devices [20].

For example, the Standard Flat-rate Envelope embodies the principle of shifting computation in time by moving computation to the data handlers so endpoints on the DAC are not bogged down with transforming data and focus on receiving as much data as possible—especially for multi-threaded endpoints. Likewise, Pay-per-use embodies this same principle and embodies the principle of avoiding obvious waste. This is because there is obvious waste experienced when the software probe stores data internally, and then copies it when the data is being packaged for collection.

By implementing both techniques in OASIS, we believe that OASIS has improved is applicability to DRE systems.

V. Related Work

Ganglia [3] is a scalable distributed monitoring system for high-performance computing systems, such as clusters and Grids. Similar to OASIS, Ganglia allows it end-users to write their own probes, which are called *Gmond metric modules*. Unlike OASIS, Ganglia uses callbacks to collect each single data point in a Gmond metric modules. For example, if a Gmond metric module has six data points, then it will receive six callbacks when data needs to be collected. In OASIS, packaging is a single call. OASIS also allows the developer to select what middleware technology to use when sending data from a node to the DAC. Similar functionality in Ganglia is handled using multi-casts. Lastly, we believe that Ganglia can leverage the Pay-per-use pattern discussed in this paper to possibly reduce packaging overhead.

Nagios [4] is an open-source real-time monitoring system that is used heavily by IT from industry. End-users extend Nagios by crafting a wrapper script that executes a binary and collects instrumentation data. The script is then integrated with Nagios Core, which manages the time intervals for collecting instrumentation data. OASIS differs from Nagios in that Nagios uses a polling model to collect instrumentation data; and OASIS uses a push model to send instrumentation data to the DAC and a push/pull model to send instrumentation data to external clients. Moreover, OASIS focuses on optimizing each phase of the instrumentation process (*i.e.*, collection, packaging, and data handling). In Nagios, it is the responsibility of the end-user to ensure the executable managed by the plug-in has little impact on the system undergoing software instrumentation. It is believed that Nagios (and plug-ins) can leverage some the patterns discussed in this paper to possibly reduce performance overhead.

VI. Concluding Remarks

It is critical that real-time instrumentation middleware have minimal performance overhead as possible. This enables collection of more instrumentation data over a period of time, and ensures minimal impact of the software system under instrumentation. In this paper, we discussed two optimizations implemented in OASIS named the Standard Flat-rate Envelope and Pay-per-use. As shown in the experimental results, both optimizations improve overall performance of OASIS, and reduce OASIS' instrumentation overhead. Based on our experience with these two optimization techniques, the following is a list of lessons learned:

- **Programming language may limit applicability.** Some programming languages make it hard to implement either of the patterns presented in this paper. For example, it may be *hard*, not impossible, to implement the Pay-per-use pattern in Java and C# because the programming languages do not support directly accessing memory locations, which is needed to reserve space in a buffer, and manage the reserved space. We will therefore research how the Pay-per-use pattern can be implemented in such programming languages.
- **Pay-per-use is easier to apply using code generation.** This is because the data object will be defined using a definition language. A compiler will then parse the data object's definition to generate its corresponding source code. During the source code generation phase, the compiler can preprocess the data object's definition and determine the best method for integrating the Pay-per-use pattern into the generated source code.

OASIS is freely available from http://oasis.cs.iupui.edu.

References

[1] Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic Instrumentation of Production Systems. In: 6th Symposium on Operating Systems Design and Implementation. (2004) 15–28

[2] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. SIGPLAN Notes **40** (June 2005) 190–200

[3] Massie, M.L., Chun, B.N., Culler, D.E.: The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing **30**(7) (2004) 817–840

[4] Barth, W.: Nagios: System and Network Monitoring. No Starch Pr (2008)

[5] Shankaran, N., Schmidt, D.C., Chen, Y., Koutsoukous, X., Lu, C.: The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems. In: Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007), Santorini Island, Greece (May 2007)

[6] Object Management Group: Deployment and Configuration Adopted Submission. Object Management Group. OMG Document mars/03-05-08 edn. (July 2003)

[7] Object Management Group: The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces. OMG Document formal/2008-01-04 edn. (January 2008)

[8] Object Management Group: DDS for Lightweight CCM (DDS4CCM). ptc/2009-10-25 edn. (February 2009)

[9] Hill, J.H., Sutherland, H., Staudinger, P., Silveria, T., Schmidt, D.C., Slaby, J.M., Visnevski, N.: OASIS: An Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems. International Journal of Computer Systems Science and Engineering, Special Issue: Real-time Systems (April 2011)

[10] Schmidt, D.C.: The ADAPTIVE Communication Environment (ACE). www.cs.wustl.edu/∼schmidt/ACE.html (1997)

[11] Schmidt, D.C., Natarajan, B., Gokhale, A., Wang, N., Gill, C.: TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. IEEE Distributed Systems Online **3**(2) (February 2002)

[12] Arvind Krishna: Optimization Techniques for Enhancing Middleware Quality of Service for Software Product Line Architectures. PhD thesis, Vanderbilt University, Nashville, TN 37235 (November 2005)

[13] Object Management Group: Data Distribution Service for Real-time Systems Specification. 1.2 edn. (January 2007)

[14] Lubbers, P., Greco, F.: HTML5 Websockets: A Quantum Leap in Scalability for the Web (2011)

[15] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)

[16] Prism Technologies: OpenSplice Data Distribution Service. www.prismtechnologies.com/ (2006)

[17] Smith, C., Williams, L.: Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software. Addison-Wesley Professional, Boston, MA, USA (September 2001)

[18] Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc. (2003)

[19] Ricci, R., Alfred, C., Lepreau, J.: A Solver for the Network Testbed Mapping Problem. SIGCOMM Computer Communications Review **33**(2) (April 2003) 30–44

[20] Varghese, G.: Network Algorithmics. Chapman & Hall/CRC (2010)