Coign: Efficient Instrumentation for Inter-Component Communication Analysis

Galen C. Hunt and Michael L. Scott*

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226

Abstract

Object systems, such as COM, promise to greatly simplify application development through the reuse of "black-box" components. Unfortunately, opaque components create new challenges to understanding application performance, behavior and structure.

We propose a conceptual framework, inter-component communication analysis (ICCA), *for understanding and exploring the structure of component applications. ICCA models an application as a graph with vertices representing components and edges representing communication links and instantiation relationships between components . Communication edges are labeled with the amount of communication that would cross the interface if the connected components were located in separate address spaces.*

We describe the Coign *runtime system for gathering the data necessary to create the ICCA graph. Coign is distinctive in that it creates the entire ICCA graph using only application binaries. It can in fact be used on components lacking source code.*

Quantifying inter-component communication is vital to understanding and exploiting the architecture of component applications. We demonstrate the use of ICCA and Coign to determine an optimal distribution of a component application across a network. ICCA helps programmers by providing important information about the application at exactly the level needed: the level of component composition.

Keywords: Component Software, Distributed Applications, Groupware, Component Placement, Coign, Inter-Component Communication Analysis (ICCA).

*Email: {gchunt,scott}@cs.rochester.edu. World Wide Web: http://www.cs.rochester.edu/u/gchunt/coign. The first author was supported by a Research Fellowship from Microsoft Corporation. This work was also supported in part by NSF grants CDA–9401142 and CCR–9319445.

1 Introduction

Component systems offer programmers and users great potential for simplifying development, increasing object reuse, and reducing application costs. Systems such as ActiveX/COM [19], OpenDoc [1], Java Beans [15] and others [7, 13] support creation of applications by aggregating components. Components may come from a programmer's private library, from code shared by a colleague, or from a commercial component provider. Components can be combined to create applications cheaply and with relatively little effort.

Components create new challenges to understanding application performance, behavior and structure. With often dissimilar origins, components may be written in any of a number of languages. An application may even use components for which a programmer has no access to source code. Given these obstacles, it can be very difficult for the programmer to explore and explain an application's runtime structure.

We propose a conceptual framework, *inter-component communication analysis (ICCA)*, for understanding the software architecture of component applications. ICCA depicts the runtime structure and behavior of an application as a graph. Vertices represent components. Edges represent connections between components including communication links and instantiation relationships. Edges are labeled with quantitative measurements of the communication between the connected components. Edges and vertices are also labeled with constraints, such as limits on where a component may be instantiated or which components must reside in isolated address spaces.

Using the ICCA graph, the programmer can easily see the runtime software architecture of the application. By examining vertices the programmer can determine the numbers and types of components used by the application. Instantiation edges let a programmer see parent-child creation relationships; for example, a programmer could see that a form component created a number of text field and button controls. Communication edges provide valuable

information about which components communicate with each other during execution and quantify data flow across communication interfaces.

With ICCA the programmer can make informed decisions on how to modify an application to achieve specific goals. Given the ICCA graph for a non-distributed component application, we show how the programmer can choose an optimal component distribution across a network to create client-server, groupware-enabled, or multiprocess versions of the application. ICCA can help a programmer determine which components are the best candidates for locating in a common dynamic link library (DLL) or when a DLL should be split to reduce virtual memory usage. Finally, ICCA information can help a programmer understand the impact of interface factorization in systems, such as COM, that support multiple interfaces per component.

We have developed *Coign*, an efficient instrumentation system for gathering the runtime information needed to create ICCA graphs. Implemented on Windows NT, Coign measures in-process, inter-component communication by calculating the amount of data that would be transported between two components *if* they were located in separate processes. Using a single instrumented execution of a non-distributed application, Coign can predict communication costs for a variety of distributions and networks. In addition to communication Coign captures other data useful for the ICCA graph including component instantiation and connectivity. Although tailored to the specific requirements of the Component Object Model (COM), the Coign strategy and all of the related ICCA techniques are readily adaptable to other component systems.

In the next section we describe the implementation of Coign. In Section 3 we demonstrate how information gathered by Coign for ICCA can be used to determine an optimal distribution for a client-server application. We enumerate a number of other areas where we believe ICCA will prove effective for understanding and exploiting application architecture in Section 4. Section 5 describes application overhead of the Coign runtime. We describe related work in Section 6. Finally, in Section 7 we discuss our conclusions and propose future work.

2 Coign: An ICCA Toolkit

Coign is a runtime instrumentation system for gathering information needed for inter-component communication analysis (ICCA). Coign gathers ICCA data by re-routing calls for the component system runtime and component member functions. Summary data is stored in a log file. Utilities in the Coign toolkit analyze the log file to create an ICCA graph and assist the programmer in understanding and exploiting the information contained within the graph.

```
interface IStreamOps : IUnknown
{
    HRESULT write([in, size_is(nSize)]
                  BYTE *pData,
                  [in] long nSize,
                  [out] long *nWrote);

    HRESULT read([out, size_is(nSize),
                  length_is(*nRead)]
                 BYTE *pData,
                 [in] long nSize,
                 [out] long *nRead);

    HRESULT seek([in] long offset);
};
```

Figure 1: IDL definition for the `IStreamOps` interface. Like all COM interfaces, `IStreamOps` inherits from `IUnknown`. The `size_is` and `length_is` attributes declare dynamic sizes for marshaling parameters.

The Coign runtime consists of a single dynamic link library, `coign.dll`. In the current implementation, `coign.dll` is manually linked into the application for analysis. In a future version of Coign, an execution loader will load the application and patch `coign.dll` into it using the same OS APIs used by debuggers.

2.1 Communication Measurement

Creating a full ICCA graph requires measurement of communication across interfaces between components. Consider the case where an array is passed between two components using a pointer. If both components are in the same address space, only the pointer moves from one component to the other. The array is not copied. If the components are located in separate address spaces, the entire array must be copied from the caller's address space to that of the callee. We quantify the communication between two components as the number of bytes that would be passed between them *if* they were located in separate address spaces.

To measure the communication that would be necessary if a message had to be passed between two address spaces, Coign uses the same information the system would use to marshal the data. Figure 1 contains the *interface definition language (IDL)* description of a sample COM interface, `IStreamOps`. Function parameters in IDL interfaces are strongly typed and labeled as either input (`[in]`), output (`[out]`) or both (`[in out]`). In addition, bounds are given for all arrays and pointer attributes are specified to aid marshaling.

For a distributed application, the IDL source files are passed through the IDL compiler to generate header files for the programmer's language of choice and RPC stubs for

marshaling and unmarshaling function parameters across address-space boundaries. It is standard practice for component providers to supply IDL definitions for the interfaces their components export.

All of the information needed for gathering ICCA data on a message is in the IDL file for the interface. Specifically, the IDL declaration of an interface contains information about the number of member functions in the interface, the number of parameters passed to each function, and complete type information about each parameter. Rather than create a new compiler to parse IDL files, we utilize information from the output files generated by the standard COM IDL compiler.

The Coign instrumentation uses information from proxies created by the IDL compiler to measure communication between components within the same address space. In essence, measurement is a side product of data marshaling with the optimization that no data needs to be copied.

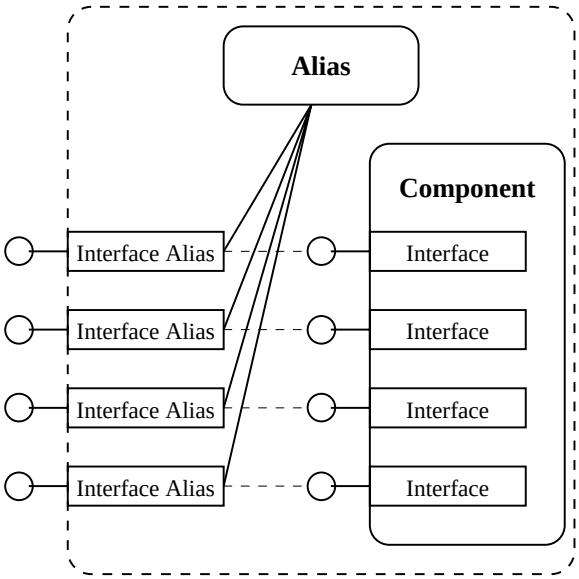

Figure 2: A component and its alias. The component alias creates an interface alias for each real interface.

2.2 Aliases

ICCA requires that all inter-component messages be instrumented. Unlike Smalltalk [11], or Objective-C [9], COM has no central message dispatcher. All messages are passed as indirect function calls through an interface's virtual function table (`vtable`). The COM runtime drops out of the way once a component has been instantiated.

While the COM calling mechanism results in fast inter-component communication, it also complicates the instrumentation required for ICCA. There is no single location where ICCA instrumentation can be inserted into an executable. Instrumentation must be inserted around *every* `vtable`.

The Coign runtime inserts instrumentation around a component by creating an *alias*. An alias is a dynamic component, created at runtime, that acts as a proxy for the real component. Any client of the real component receives an interface pointer to the alias not to the component. Any calls to alias member functions are processed by the Coign instrumentation then forwarded to the alias.

Figure 2 shows a component and its alias. The alias holds pointers to each interface instantiated by the real component. A separate interface alias exists for each real interface. Whenever the COM library or an instrumented component returns a pointer to an interface, Coign replaces the interface pointer with a pointer to the corresponding interface alias. Interface alias creation is demand driven. If the component returns a pointer to a new interface, Coign creates a new interface alias. Important functions that can trigger interface alias creation include `CoCreateInstance` and `IUnknown::QueryInterface`.

2.3 Aliased Function Calls

Each interface alias must strictly observe COM calling conventions when forwarding messages. COM follows modified C++ calling conventions for member functions. Specifically, the first parameter to any member function is the interface pointer, the implicit `this` parameter. The implicit interface pointer is followed by any explicit parameters. On the Windows NT platform, parameters are always passed on the stack from right to left. For a fixed number of arguments, the callee pops the stack.

All interface aliases within an address space use a single `vtable`. Entries in the alias `vtable` point to an array of *trampoline* functions. The trampoline functions call into the Coign runtime with a pointer to the alias being called, an the index of the trampoline function in the `vtable`, and a pointer to the current stack frame.. The alias pointer and `vtable` index are sufficient to uniquely describe the component function intended by the caller.

In the common case, where Coign can retrieve information from the IDL-generated proxy, calls on an alias pointer jump through the trampoline to the Coign runtime. The runtime records information about the input parameters and forwards the call to the "real" component function. Control returns through the runtime, where information about output parameters is recorded, to the trampoline and back to the caller. (See Figure 3.) The Coign runtime forwards the call by making a copy of the trampoline's stack frame. The runtime returns to the trampoline the size of the stack frame to be popped as needed to follow the callee-pops-stack calling convention.

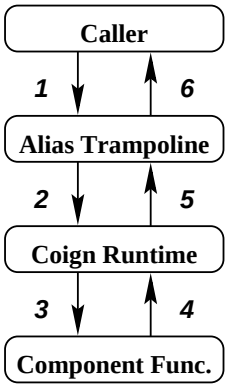

Figure 3: Calling sequence for an aliased function. Control passes from caller to trampoline function (1), through the Coign runtime (2) into the component (3) and back (4,5,6).

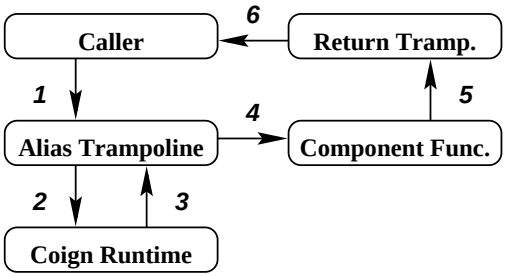

Figure 4: Calling sequence for first call to an aliased function without IDL information. Control passes from caller to trampoline function (1) then into the Coign runtime (2). The runtime returns a special value to the trampoline (3) which patches the stack. Control jumps directly into the component function (4) which returns to another trampoline (5) and then jumps back into the caller (6).

In the uncommon case, Coign has no IDL information about the interface. Shown in Figure 4, control passes from the caller through the trampoline into the Coign runtime. In this case, the call can't be forwarded through the runtime because it has no reliable way of determine the stack frame size. The runtime returns a special token to the trampoline which passes control directly into the "real" component function using a jump rather than a function call. The jump gives the component function the stack frame passed to the trampoline. However, before jumping, the trampoline modifies the return instruction pointer on the stack to point to a special *return trampoline*. Control returns from the component function directly to the return trampoline which records the size of the used stack frame by comparing the current stack pointer with the stack pointer saved before starting the call. The trampoline then jumps directly back into the caller. All future calls to the same component member function will be forwarded through the Coign runtime using the newly acquired stack frame size.

2.4 Recorded ICCA Data

The Coign runtime creates a single output file, `coign.msg` containing summary information about the component application. `Coign.msg` contains the following information (listed by event type):

Component creation (instantiation) :
Component class GUID.
Runtime identifier (RTID) of new component.
RTID of component that requesting creation.
Time of creation.

Component destruction :
Component RTID.
Time of destruction (last release).

Interface creation (instantiation) :
Interface class GUID.
RTID of new interface.
RTID of the owning component.
Time of creation.

Interface destruction :
Interface RTID.
Time of destruction (last release).

Interface member-function calls :
Interface RTID.
RTID of calling component.
RTID of called component.
Total size of incoming parameters.
Total size of outgoing parameters.

Information for member-function calls is aggregated by [caller, callee, interface] tuple. Running totals accumulate separately for incoming and outgoing parameters. Coign accumulates the number of messages and number of bytes communicated for messages of several sizes including messages up to 16 bytes, 64 bytes, 256 bytes, 1K, 4K, 16K, 64K, and messages over 64K in a form of log_4 histogram. Accumulating message totals by message size gives the programmer information about communication granularity independent of network parameters. Message totals can then be combined with parameters for a specific network to determine total networking costs.

As alluded to in the previous section, Coign cannot always find IDL information for an interface. It is usually the case that IDL information is missing or incomplete only for interfaces internal to a component. Lacking IDL information, Coign cannot determine message size. Coign still manages to install sufficient instrumentation around the interface to count the number incoming calls. Coign

labels the affected ICCA graph edges with an *uncertainty* count of the number of calls that could not be fully measured.

2.5 Other Utilities

In addition to the `coign.dll` runtime, the Coign toolkit also contains a utility, `regcoi` for caching, in the system registry, information about all interfaces known to have IDL information. Another utility, `netcoi` measures available computation and network resources. Given a list of distributed machines, `netcoi` instantiates one benchmark component on each machine. The benchmakr component measures the relative computation power of the machine and gathers information about available memory from the operating system. `Netcoi` then sends multiple messages between each machine measuring average available network bandwidth and latencies for a wide range of message sizes. The network parameters are placed in the registry for use when calculating distributed communication costs.

3 Component Distribution

The shared goal of ICCA and the Coign toolkit is to make it easier for programmers not only to understand, but also to exploit the software architecture of component applications. In this section we present preliminary data demonstrating the power of ICCA to help solve a difficult software challenge: given an existing application deciding how to distribute its components across a network to create either a client-server or a groupware-enabled application.

Much effort has been invested in attempts to reduce the difficulty of developing distributed applications. Runtime systems, such as RPC [22], DCE [12], DSOM [14], CORBA [23], and DCOM [5] ease development of distributed applications by providing convenient mechanisms for transferring control and data from one machine to another.

Other systems, including Amoeba [21], , DOME [8], Globe [13], Infospheres [7] and Legion [18], attempt to reduce the development costs of distributed applications by creating completely distributed environments. Their general philosophy is that all objects, regardless of their location, are treated equal. The downside of these locality-free models is that they often fail to exploit local optimizations and place strict requirements on application structure.

The solution to creating distributed applications with ICCA is to distribute existing applications rather than create new distributed applications. A distributed client-server application normally consists of a graphical user interface (GUI) front-end and a storage back-end. The GUI resides on the user's workstation and the storage resides on a remote server. The GUI and storage may each consist of multiple components. Between GUI and storage may be a large number of processing components. The ICCA graph contains nodes for each of the components with location constraints on the GUI and storage components. The programmer's goal is often to create a distribution of the remaining components that uses minimum network resources and provides maximum performance. To a first approximation, the problem is to cut the ICCA graph so that communication costs across the cut are minimized.

Client-server and groupware applications differ mainly in their motivation for distribution and use of consistency models. The simple model for a multi-user, groupware application is similar to a client-server application with the exception that GUI components must be replicated for each user. Again the problem is to create a cut between components located on the server and components located on user workstations.

3.1 Experimental Environment

Our experimental environment consists of a suite of document-processing components. The suite contains over 180 component classes supporting approximately 200 interfaces. Components range in functionality from single buttons and labels to a word processor and a spreadsheet. The larger components, such as the word processor, aggregate many of the smaller components.

Our test application is a small shell that instantiates components from the suite based on document requirements. Code for the application and the entire component suite is over 115,000 lines of C code.

Our test equipment consists of four 486dx2 66MHz PCs connected with an isolated 10 Mb/s Ethernet network. Each machine has 32MB of RAM and at least 500MB of local disk. Unless otherwise noted, numbers reported are the average from multiple runs. The first timing for each run was discarded to limit the effects of startup VM operations.

Our experimental results were gathered using a small set of documents. The documents are categorized by their principle type of component (see Figure 5). Figure 6 lists number of components, interfaces, interface calls for each document, number of calls with uncertain parameter size, and bytes of communication across interfaces.

3.2 Experimental Procedure

To verify the applicability of ICCA to the problem of creating distributed applications, we experimented with several hand-created distributions of the word processing application. Our goal is to show that given several possible distributions, ICCA can provide the programmer with sufficient information to determine which distribution is

Document	Category	Size (Bytes)	Execution Time (ms)
text0	word proc.	8,192	2,136
text1	word proc.	1,793,024	7,173
text2	word proc.	4,955,648	20,597
text3	word proc.	10,528,768	37,811
grid0	spreadsheet	8,704	2,276
grid1	spreadsheet	895,488	20,132
grid2	spreadsheet	1,808,384	40,047
grid3	spreadsheet	6,450,176	73,606

Figure 5: Experimental Documents. Documents are categorized by their principle type of component. Execution time listed is on a 66MHz Intel 486dx2 PC.

Doc.	Comp.	Inter.	Calls	Unc.	Comm. (KB)
text0	307	1,166	3,879	102	102
text1	309	1,164	5,166	196	29,478
text2	308	1,167	5,112	192	90,648
text3	309	1,165	5,166	203	163,770
grid0	254	981	3,775	63	115
grid1	254	984	3,775	63	115
grid2	254	987	3,775	63	115
grid3	254	983	3,775	63	115

Figure 6: Principle Coign Measurements. Listed for each document is the number of instrumented components, instrumented interfaces, interface calls, uncertainty, and bytes communicated across interfaces

optimal before requiring any coding. Our criterion for optimality is that the distribution minimize network usage and execution time.

In the future, we plan to identify algorithms for enumerating all possible distributions of an application's components and for finding the optimal distribution. We also plan to explore issues related to adapting the algorithms for dealing with other constraints on component location including binary compatibility, licensing, security and resource availability.

For our experiments we used the word-processing documents (`text0` - `text3`) described in Section 3.1. Our goal is to find a distribution of the word-processing application for a client who wants to load and view a subset of the document. The documents reside on a central server.

At runtime, the word processor uses over 300 components. The application execution, however, is dominated by four components (see Figure 7). Progressing from the server-based document file to the client display, the components are: document component, layout component, viewer component, and paint component.

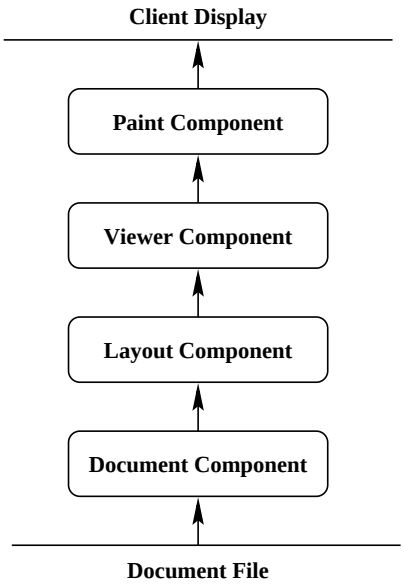

Figure 7: Simplified composition of the word-processing components.

We examine three possible distributions:

File System The first distribution retrieves files directly from the server using the CIFS [17] distributed file system. In this case, all of the components are located on the client. For this experiment, we calculate network traffic as the number of bytes retrieved from the server, namely the size of the document. While this is a conservative estimate, it is the estimate that would be calculated by ICCA if we had instrumented interfaces between components and the operating system.

Window System In the second distribution, the application is split at the interface between the display subsystem and the components. This is analogous to running the application on the server using the X Window System [24] to display documents on the client's screen. To measure network traffic in this case, we use the calculation of message traffic between the paint component and all other components in the system. Again, this is a conservative estimate, but will suffice for our needs.

Layout The final distribution splits the system roughly between the layout component and the document component. Communication at this interface is measured directly by Coign.

The first two distributions where chosen because they represent distributions normally available to programmers in workstation environments. They represent a level of

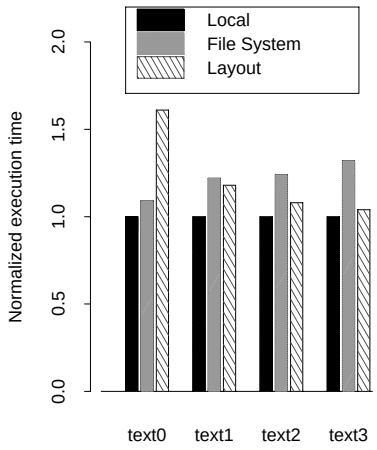

Figure 8: Execution times for application distributions normalized against time for the non-distributed (local) application.

Document	File System	Layout	Window Sys.
`text0`	1,793,024	5,068	842
`text1`	1,793,024	24,720	30,034,116
`text2`	4,955,648	24,756	92,673,476
`text3`	10,528,768	24,938	167,548,626

Figure 9: Estimated traffic (in bytes) across distributed interfaces for each distribution.

distribution most readily available to a programmer. The third distribution is entirely across interface boundaries. The program modifications to achieve this distribution were trivial. Our eventual goal is to create a future version of Coign that could load an application and create distributions similar to the third one without any source code modification.

3.3 Experimental Results and Discussion

Figure 8 shows execution times for two of the application distributions normalized against execution time for the non-distributed application. Times for the window system distribution are not shown. Measurements of communication into the paint component are sufficiently high to suggest that a window system distribution would perform very poorly. `Text3` sends just under 160MB to the paint component. While we could improve the performance of the display-system distribution with caching, the required source modifications would be non-trivial.

Execution of the layout distribution ranges from 48% slower to 26% faster than execution using the CIFS distributed file system. For small documents, such as `text0`, the layout distribution suffers because it must wait for the document component process to start on the server. The file system distribution incurs no similar delays because the file server is always loaded. In many distribution scenarios, the process containing the server-located components will be started prior to most clients. This is particularly true for groupware applications where the server document components will be instantiated for the first user and reused for additional users. In other cases, such as clients on low bandwidth connections, high message cost will increase the performance gains of the layout distribution.

As can be seen in Figure 9, for the large documents the number of bytes transferred using the file system is much larger than the number transferred across the layout/document interface. This might lead one to predict a greater improvement in performance. However, the layout distribution suffers some performance degredation because it sends very small network packets. CIFS uses larger packets to amortize network latency delays.

Our experimental data suggests that a layout distribution will have the lowest communication costs for a simple client/server application. A groupware application would have a similar optimal distribution as multiple clients would widen the performance gap between the layout/document distribution and other distributions.

We should note that it is impossible for the Coign runtime to create automatically a groupware application. Supporting multiple concurrent users requires correct use of a consistency model for controlling access by multiple readers and writers. Correct implementation of a consistency model requires semantic information beyond the scope of ICCA. While Coign and ICCA can't create a groupware-enabled application, they can tell the programmer which interfaces are most desirable for distribution. The programmer can use this information to decide how to implement semantic consistency.

4 Other Uses of ICCA and Coign

ICCA provides programmers with information about the number and types of components instantiated at runtime. It also provides information about who created a component, what interfaces a component has exported and who has communicated with a component. Data gathered by the Coign instrumentation system contains information about the amount of communication from one component to another and the interfaces used.

We have already shown how ICCA can be used in determining how to distribute components of an application,

either for a client-server distribution or for multiple users. ICCA is also useful in answering programmer questions about the following issues:

Unknown Connections Carmichael *et al.* [6] document that even in well-designed, well-written programs there are often connections between components that are either not well understood or not known to exist by the system designers. A simple application of ICCA is to provide the designer with a mapping of the existing connections between components at runtime.

Component Space Overhead Components are normally grouped into collections in DLLs. Applications that use only a few of the components in a DLL are unnecessarily burdened. Using the ICC graph, the programmer can determine when a DLL should be split to reduce memory usage or working-set size.

Component Temporal Locality It is quite easy for the programmer to turn the Coign instrumentation on and off at desired points of execution by disabling call measurement and recording. Using time-bounded ICCA, a programmer can explore how the make-up of active components and interfaces changes during the temporal execution of an application. Programmers can tune working set size, for example, by releasing references to components that aren't being using during a memory-critical section of execution.

Custom Marshalling and Caching Candidates COM provides support for custom marshaling of interfaces. Using custom marshaling a programmer can optimize the marshaling protocol for a particular component. Custom marshaling can be used, for example, for message aggregation or local caching of information from a remote component. Again, ICCA is important because it helps a programmer know what interfaces are critical to performance. Using ICCA a programmer can also predict the benefit of reducing communication by a specified amount on a given interface before actually coding the modification.

Components pose new challenges to programmers because they are often used in large applications with dynamic behavior. ICCA helps programmers by providing important information about the application at exactly the level needed: the level of component composition.

5 Coign Runtime Overhead

A programmer gathers data needed for ICCA by running the application with Coign instrumentation. In this section we describe the overhead of the Coign runtime.

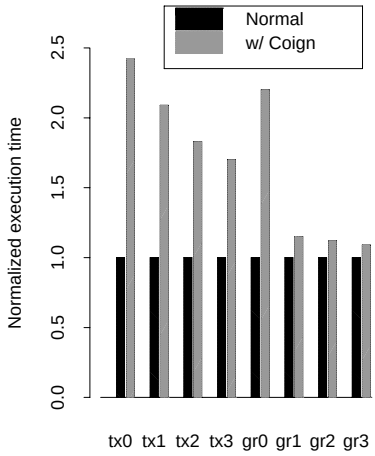

Figure 10: Normalized execution time. Execution times with Coign instrumentation are normalized against those without Coign.

Figure 10 shows the time to load, process, and display the first page of each of the documents listed in Figure 5 . Execution time with Coign instrumentation is normalized against execution time without Coign instrumentation.

The current Coign implementation requires a relatively large amount of time during startup to load information from the IDL proxies for each interface. After IDL information has been loaded, there is a marginal execution overhead for measuring and recording parameter data with each function call across an interface.

As can be seen in Figure 10, Coign slows application execution anywhere from 9% to 142%. Importantly, the proportion of overhead reduces with longer execution times as the cost of loading IDL proxies is amortized.

Figure 11 shows the absolute time overhead of the Coign runtime for each document. Overhead for the larger word-processing documents is much higher than that of the spreadsheet documents. Several of the interfaces used extensively by the word-processing components pass data proportional to the size of the document. The current Coign instrumentation scans all parameter data sequentially. Overhead increases linearly with parameter size. Spreadsheet document overhead stabilizes because the interfaces used don't pass data proportional to the document size. We believe that for most cases we can remove the need to scan sequentially all parameters. This modification would tend to stabilize overhead as the size of word-processing documents grows.

Memory overhead for Coign is approximately 3.5MB for either the word-processing or spreadsheet documents.

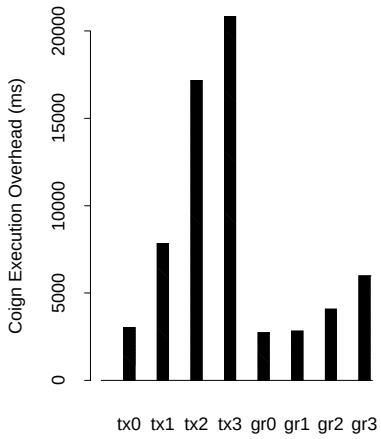

Figure 11: Absolute Coign execution overhead (in milliseconds).

For `grid0`, total overhead is 3424KB including 152KB for `coign.dll` and 2120KB for the IDL proxies. The remainder of the overhead, 1152KB, consists of aliases and running totals for gathering message statistics. Average runtime overhead per interface, not counting IDL proxies, is about 1K.

It should be noted that for all of the documents, the size of the `coign.msg` file was less than 64K. The only isolated events logged to `coign.msg` are instantiation and destruction of components and interfaces. Since instantiation and destruction tend to be relatively expensive operations, they easily hide the cost of logging the event. Information about member function calls is accumulated using the log_4 histograms described in Section 2.4. Memory costs of the histograms do not increase over time. All of the histograms are logged to `coign.msg` when the application terminates.

6 Related Work

A number of researchers have recognized the potential benefits of instrumenting message communication. Their work highlights the richness of information available from monitoring inter-object messages.

Cunningham and Beck [10] instrument message calls by modifying the Smalltalk-80 debugger. The purpose of their system is to create object call diagrams when the programmer issues `step` or `send` commands. Unlike the debugger's runtime stack display, which reflects only the current state of computation, the call diagram contains accumulated calling information.

Kleyn and Gingrich [16] describe the GraphTrace system for program visualization. They instrument the LISP runtime in STROBE to record all method invocations. Runtime information is combined with static analysis to provide several views animating the execution of the instrumented program. They use two instrumentation systems. One records message information, specifically sender, receiver and arguments; the second animates method invocations.

Böcker and Herczeg [4] instrument individual methods to uncover dynamic properties of Smalltalk programs. Their work consists of two parts, TRICK, an instrumentation system, and TRACK [2, 3], a graphical front end for manipulating the instrumentation. As with GraphTrace, the focus is on visualizing different parts of program execution. Using TRACK, the programmer can place a tracer around a particular class, object, or method. Execution enters the tracer before proceeding into the surrounded code. The user defines and attaches trace filters to the tracer. Applicable filters are called before and after the instrumented code. They suggest using filters to display message arguments and return values, check argument values for validity, and update program animations.

HookOLE [20] is a general purpose instrumentatin system for instrumenting COM interface. HookOLE redirects `vtable` function pointers to special interface wrappers. Interface wrappers call out to user-defined interface filters that are allowed to examine message parameters and return values. HookOLE is used primarily to verify compliance to interface standards using filters that check the validity and conformance of member-function parameters. HookOLE provides neither statistical information about messages nor a standard procedure for determining the sender of a message.

Our use of application internal communication measurements is distinctive. We can measure internal communication by leveraging IDL stubs to quantify the amount of data that would have been copied if the communication was not internal to the process. Whereas past researches have focused on individual messages between objects, our work focuses on the importance of total communication between objects.

7 Conclusions and Future Work

We have presented inter-component communication analysis (ICCA), a technique for understanding component programs. The principle tool of ICCA is the ICCA graph containing vertices for each component and edges describing component relationships including total communication costs.

As demonstrated in Section 3, quantifying communication between components can be extremely useful. The

component suite in our experiments includes a low-level component for painting document details in a window. In a number of our experiments, even for large documents there are only three components connected to the painting component. However, in one case almost 160MB flows across a single interface connection into the paint component. If the programmer had distributed the paint component without knowing predicted communication costs, the application would have suffered a 12-fold increase in execution time on a 10Mb/s Ethernet.

We have described the implementation of Coign, a toolkit for gathering runtime data and creating the ICCA graph. Among Coign's strengths are low instrumentation overhead, full support for dynamic applications and the ability to instrument components for which the programmer lacks source code.

We are currently exploring a number of areas for future work including:

Distributed Runtime Our primary test domain deals with client-server and groupware distributed applications. While our preliminary research has dealt with the issue of how to take a single-machine application and distribute it, it is also important to be able to tune the performance of an existing distributed application. We plan to add full distributed support through a distributed Coign runtime.

ICCA Visualization The current ICCA graph generated by Coign is quite primitive. We plan to create a GUI interface to the runtime to aid the programmer in analysis and visualization of the graph. An important part of such a GUI interface would let programmers ask and receive immediate answers to "what-if" questions about possible application distributions.

Automatic Distribution Our eventual goal is to take an unmodified component application and distribute it without modifying any source across a network. There are two ways of distributing the application. One is to distribute the application components according to a one-time distribution suggested and approved by the programmer. The alternative is to determine the distribution dynamically every time the program is run. The second alternative would allow the program in effect to customize its distribution to the currently available resources. We consider this a promising direction for further research.

References

[1] Apple Computer, Inc. The OpenDOC Software Development Kit. Cupertino, CA, November 1995.

[2] H.-D. Böcker and J. Herczeg. Browsing Through Program Execution. In *Proceedings of INTERACT '90, The Third Internation Conference on Human-Computer Interaction*, pages 991– 997, Cambridge, UK, August 1990.

[3] H.-D. Böcker and J. Herczeg. TRACK- A Trace Construction Kit. In *CHI-90, Human Factors in Computer Systems Conference Proceedings*, pages 415–422, Seattle, WA, April 1990.

[4] H.-D. Böcker and J. Herczeg. What Tracers Are Made of. In *OOPSLA/ECOOP '90 Conference Proceedings*, pages 89–99, Ottawa, Ontario, Canada, October 1990.

[5] N. Brown and C. Kindel. *Distributed Component Object Model Protocol – DCOM/1.0*. Microsoft Corporation, Redmond, WA, November 1996.

[6] I. Carmichael, V. Tzerpos, and R. C. Holt. Design Maintenance: Unexpected Architectural Interactions. In *Proceedings of the IEEE International Conference on Software Maintenance*, Opio, France, October 1995.

[7] K. M. Chandy, A. Chelian, B. Dimitrov, H. Le, J. Mandelson, M. Richardson, A. Rifkin, P. A. G. Sivilotti, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[8] J. N. Cotrim Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Distributed Computing Environment. In *Proceedings of the Tenth International Parallel Processing Symposium*, Honolulu, HI, April 1996.

[9] B. J. Cox. Objective C: Programming Smalltalk-80 Methods in C Language. In *Proceedings of the 1893 USENIX Software Tools Users Group Summer Conference*, page 236, Toronto, Canada, July 1983. USENIX.

[10] W. Cunningham and K. Beck. A Diagram for Object-Oriented Programs. In *OOPSLA '86 Conference Proceedings*, pages 361–367, Portland, OR, September – October 1986.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[12] D. Hartman. Unclogging Distributed Computing. *IEEE Spectrum*, 29(5):36–39, May 1992.

[13] P. Homburg, M. van Steen, and A. S. Tanenbaum. An Architecture for a Scaleable Wide Area Distributed System. In *Proc. Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[14] International Business Machines. *SOMobjects Developer Toolkit Version 2.1*. IBM, White Plains, NY, 1995.

[15] JavaSoft. JavaBeans, Version 1.0 API Specification. Mountain View, CA, October 1996.

[16] M. F. Kleyn and P. C. Gingrich. GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views. In *OOPSLA '88 Conference Proceedings*, pages 191–205, San Diego, CA, September 1988.

[17] P. Leach and D. Perry. CIFS: A Common Internet File System. In *Microsoft Interactive Developer*, November 1996.

[18] M. Lewis and A. Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996.

[19] Microsoft Corporation and Digital Equipment Corporation. The Component Object Model Specification. Redmond, WA, October 1995.

[20] Microsoft Corporation. HookOLE Architecture, Alpha Release. Redmond, WA, July 1996.

[21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. vanStaveren. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[22] B. J. Nelson. Remote Procedure Call. Ph. D. dissertation, Technical Report CMU-CS-81-119, Carnegie-Mellon University, May 1981.

[23] Object Management Group, Inc. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Framingham, MA, July 1996.

[24] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.