# The Coign Automatic Distributed Partitioning System

Galen C. Hunt
*Microsoft Research*
*One Microsoft Way*
*Redmond, WA 98052*
*galenh@microsoft.com*

Michael L. Scott
*Department of Computer Science*
*University of Rochester*
*Rochester, NY 14627*
*scott@cs.rochester.edu*

## Abstract

*Although successive generations of middleware (such as RPC, CORBA, and DCOM) have made it easier to connect distributed programs, the process of distributed application decomposition has changed little: programmers manually divide applications into sub-programs and manually assign those sub-programs to machines. Often the techniques used to choose a distribution are ad hoc and create one-time solutions biased to a specific combination of users, machines, and networks.*

*We assert that system software, not the programmer, should manage the task of distributed decomposition. To validate our assertion we present Coign, an automatic distributed partitioning system that significantly eases the development of distributed applications.*

*Given an application (in binary form) built from distributable COM components, Coign constructs a graph model of the application's inter-component communication through scenario-based profiling. Later, Coign applies a graph-cutting algorithm to partition the application across a network and minimize execution delay due to network communication. Using Coign, even an end user (without access to source code) can transform a non-distributed application into an optimized, distributed application.*

*Coign has automatically distributed binaries from over 2 million lines of application code, including Microsoft's PhotoDraw 2000 image processor. To our knowledge, Coign is the first system to automatically partition and distribute binary applications.*

## 1. Introduction

Distributed systems have been an area of open research for more than two decades. Popular acceptance of the Internet has fueled a renewed interest in distributed systems and applications. Distributed application enable sharing of data, sharing of resources (such as memory, processor cycles, or physical devices), collaboration between users, increased reliability through redundancy, and increased security through physical isolation.

However compelling the motivations, the creation of distributed applications continues to be difficult. As a rule, the creation of a distributed application is always harder than the creation of a functionally equivalent non-distributed application. Complicating factors include protection of data integrity and security, management of disjoint address spaces, increased latency and reduced bandwidth between application components, partial system failures caused by isolated machine or network outages, and practical engineering issues such as debugging across multiple processes on distributed computers.

One of the primary challenges to create a distributed application is the need to partition and place pieces of the application. Although successive generations of middleware (such as RPC [4, 15, 34], CORBA [35, 42], and DCOM [8]) have brought the advantages of service-location transparency, dynamic object instantiation, and object-oriented programming to distributed applications, the process of distributed application decomposition has changed little: programmers manually divide applications into sub-programs and manually assign those sub-programs to machines. Often the techniques used to choose a distribution are ad hoc, creating solutions biased to a specific platform.

Given the effort required, applications are seldom re-partitioned even in drastically different network environments. Changes in underlying network, from ISDN to 100BaseT to ATM to SAN, strain static distributions as bandwidth-to-latency tradeoffs change by more than an order of magnitude. User usage patterns can also severely stress a static application distribution. Nevertheless, programmers resist repartitioning applications because doing so often requires extensive modifications to program structure and source code.

We argue that system software, not the programmer, should partition and distribute applications. Furthermore, we assert that existing applications can be partitioned and distributed automatically without ac-

cess to source code, provided the applications are built from binary components. To validate our claims, we have built a working prototype system, the Coign *Automatic Distributed Partitioning System* (ADPS).

Coign radically changes the development of distributed applications. Given an application built with components conforming to Microsoft's Component Object Model (COM), Coign profiles inter-component communication as the application is run through typical usage scenarios (a process known as *scenario-based profiling*). Based on profiled information, Coign selects a distribution of the application with minimal communication time for a particular distributed environment. Coign then modifies the application to produce the desired distribution.

Coign analyzes an application, chooses a distribution, and produces the desired distributed application all with access to only the application binary files. By solely analyzing application binaries, Coign produces distributed applications automatically without violating the primary goal of commercial component systems: building applications from reusable, binary components.

In the next two sections, we describe Coign and the implementation of the Coign runtime. In Section 4, we present experimental results demonstrating Coign's effectiveness in automatically distributing binaries from over 2 million lines of application code. In Section 5, we describe related work. Finally, in Section 6 we summarize our conclusions and discuss future work.

## 2. System Description

Coign is an automatic distributed partitioning system (ADPS) for applications built from COM components. COM is a standard for packaging, instantiating, and connecting reusable pieces of software in binary form called components. Clients talk to components through polymorphic interfaces. Abstractly, a COM interface is a collection of semantically related function entry points. Concretely, COM defines a binary standard representation of an interface as a virtual function table. All first-class communication between COM components passes through interfaces. Clients reference a component through pointers to its interfaces.

Due to a strict binary standard, COM can transparently interpose proxies, stubs, and middleware layers between communicating clients and components for true location transparency. Application code remains identical for in-process, cross-process, and cross-machine communication. The DCOM protocol, a superset of DCE RPC [18], transports messages between machines by deep copy of message arguments. By leveraging the COM binary standard, Coign can automatically distribute an application without any knowledge of the application source code.

The Coign ADPS consists of four major tools: the Coign run-time, a binary rewriter, a network profiler, and a profile analysis engine. Figure 1 contains an overview of the Coign ADPS.

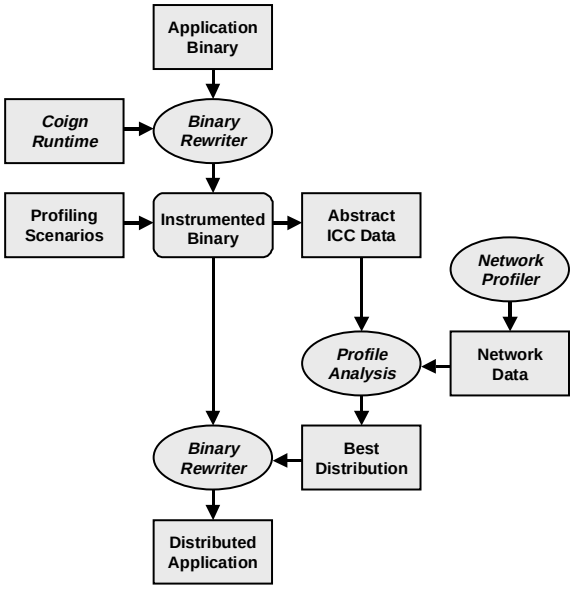



**Figure 1. The Coign ADPS.**
An application is transformed into a distributed application by inserting the Coign runtime, profiling the instrumented application, and analyzing the profiles to cut the network-based ICC graph.

Starting with the original binary files for an application, the *binary rewriter* creates a Coign-instrumented version of the application. The binary rewriter makes two modifications to the application. First, it inserts an entry into the first slot of the application's dynamic link library (DLL) import table to load the Coign runtime. Second, it adds a data segment containing configuration information at the end of application binary. The configuration information tells the Coign runtime how to profile the application and how to classify components during execution.

Because it occupies the first slot in the application's DLL import table, the Coign runtime always loads and executes before the application or any of its DLLs. At load time, the Coign runtime inserts binary instrumentation into the images of system libraries in the application's address space. The instrumentation traps all component instantiation requests in the COM library.

The instrumented binary is run through a set of profiling scenarios. Because the binary modifications are

transparent to the user (and to the application itself), the instrumented binary behaves identically to the original application. The instrumentation gathers profiling information in the background while the user controls the application. The only visible effect of profiling is a small degradation in application performance (of up to 85%). For advanced profiling, scenarios can be driven by an automated testing tool, such as Visual Test [2].

During profiling, the Coign instrumentation summarizes inter-component communication within the application. Every inter-component call is executed via a COM interface. Coign intercepts these interface calls (by instrumenting all component interfaces) and measures the amount of data communicated. The instrumentation measures the number of bytes that would be transferred from one machine to another *if* the two communicating components were distributed. It does so by invoking portions of the DCOM code, including interface proxies and stubs, within the application's address space. Coign measurement follows precisely the deep-copy semantics of DCOM. After quantifying communication (by number and size of messages), Coign compresses and summarizes the data online. Consequently, the overhead for storing communication information does not grow linearly with execution time. If desired, the application may be run through profiling scenarios for days or even weeks to more accurately track user usage patterns.

At the end of a profiling execution, Coign writes the inter-component communication profiles to a file for later analysis. In addition to information about the number and size of messages and components in the application, the profile log also contains information to classify components and to determine component location constraints. Log files from multiple profiling scenarios may be combined and summarized during later analysis. Alternatively, at the end of each profiling scenario, information from the log file may be combined into the configuration record in the application binary. The latter approach uses less storage because summary information in the configuration record accumulates communication from similar interface calls into a single entry.

The *profile analysis engine* combines component communication profiles and component location constraints to create an abstract inter-component communication (ICC) graph of the application. Location constraints can be acquired from the programmer, from analysis of component communication records, and from application binaries. For client-server distributions, the analysis engine performs static analysis on component binaries to determine which Windows APIs are called by each component. Components that access a set of known GUI or storage APIs are placed on the client or server respectively. Other components are distributed based on communication analysis.

The abstract ICC graph is combined with a network profile to create a concrete graph of potential communication time on the network. The *network profiler* creates a network profile through statistically sampling of communication time for a representative set of DCOM messages.

Coign employs the *lift-to-front minimum-cut* graph-cutting algorithm [9] to choose a distribution with minimal communication time. In the future, the concrete graph could be constructed and cut at application execution time, thus introducing the potential to produce a new distribution tailored to current network characteristics for each execution.

The lift-to-front min-cut algorithm, in our current implementation, can produce only two-machine, client-server applications. The problem of partitioning applications across three or more machines is provably NP-hard [13]. Numerous heuristic algorithms exist for multi-way graph cutting [7, 10, 12, 33, 38]. To more accurately evaluate the rest of the system, we restrict ourselves to an exact, two-way algorithm for client-server computing.

After analysis, the application's ICC graph and component classification data (to be described later) are written into the configuration record in the application binary. The configuration record is also modified to remove the profiling instrumentation. In its place, a lightweight version of the instrumentation will be loaded to realize (enforce) the distribution chosen by the graph-cutting algorithm.

## 3. Coign Runtime Description

The Coign runtime is composed of a small collection of replaceable COM components (Figure 2). The most important components are the *Coign Runtime Executive* (RTE), the *interface informer*, the *information logger*, the *instance classifier*, and the *component factory*. The RTE provides low-level services to other components in the Coign runtime. The *interface informer* walks the parameters of interface function calls and identifies the location and static type of component interfaces. The *information logger* records data necessary for post-profiling analysis. The *instance classifier* identifies component instances with similar communication profiles across multiple program executions. The *component factory* decides where component instantiation requests should be fulfilled and relocates instantiation as needed to produce a chosen

distribution. The component structure of the Coign runtime facilitates its use for a wide variety of application analysis and adaptation.

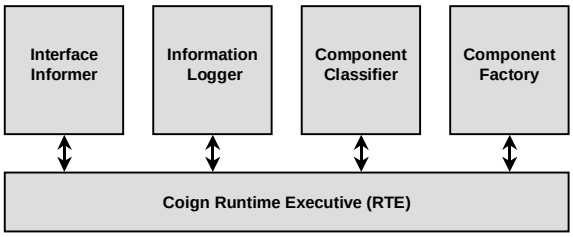



**Figure 2. Coign Runtime Architecture.**
Runtime components can be replaced to produce lightweight instrumentation, to log component activity, or to remote component instantiation.

### 3.1. Runtime Executive.

The Coign Runtime Executive (RTE) provides low-level services to other components in the Coign runtime. Services provided by the RTE include:

**Interception of component instantiation requests.** The RTE traps all component instantiation requests made by the application to the COM runtime. Instantiation requests are trapped using inline redirection (similar to the techniques pioneered by the Parasight [1] parallel debugger)[1]. The RTE invokes the instance classifier to identify the about-to-be-instantiated component. The RTE then invokes the component factory, which fulfills the instantiation request at the appropriate location based on instance classification.

**Interface wrapping.** The RTE "wraps" all COM interfaces by replacing each component interface pointer with a pointer to a Coign instrumented interface, which in turn forwards incoming calls through the original interface pointer. Once an interface is wrapped, the Coign runtime can trap all calls across the interface. An interface is wrapped using static information from the interface informer. The RTE also invokes the interface informer to process the parameters of interface function calls.

**Address space and private stack management.** The RTE tracks all binaries (.DLL and .EXE files) loaded into the application's address space. The RTE also provides distributed, thread-local stack storage for contextual information across interface calls.

**Access to configuration information stored in the application binary.** The RTE provides a set of functions for accessing information in the configuration record created by the binary rewriter. The RTE, in cooperation with the information logger, provides other Coign components with persistent storage through the configuration record.

### 3.2. Interface Informer.

The interface informer manages static interface metadata. Other Coign components use data from the interface informer to determine the static type of COM interfaces, and walk input and output parameters of interface function calls. The interface informer also aids the RTE to track the owner component for each interface [20].

The current Coign runtime contains two interface informers. The first interface informer operates during scenario-based profiling. The *profiling informer* uses format strings and interface marshaling code generated by the Microsoft IDL compiler [31] to analyze all function call parameters and precisely measure inter-component communication. Profiling currently adds up to 85% to application execution time (although in most cases the overhead is closer to 45%). Most of this overhead is directly attributable to the interface informer.

The second interface informer remains in the application after profiling to produce the distributed application. The *distribution informer* only examines function call parameters enough to identify interface pointers. Due to aggressive optimization of static interface metadata, the distribution informer imposes an overhead on execution time of less than 3%.

### 3.3. Information Logger

The information logger summarizes and records data for distributed partitioning analysis. Under direction of the RTE, Coign components pass information about application events to the information logger. Events include component instantiations, component destructions, interface instantiations, interface destructions, and interface calls. The logger is free to process the events as needed. Depending on the logger's implementation, it may ignore the events, write the events to a log file on disk, or accumulate information about the events into in-memory data structures.

The current implementation of the Coign runtime contains three separate information loggers. The *profiling logger*, summarizes data describing inter-component communication into in-memory data structures. At the end of execution, these data structures are written to disk for post-profiling analysis. The profiling logger reduces memory overhead by summarizing data for messages in common size ranges

[1]Our inline redirection and binary-rewriting tools for Windows NT are available separately [21].

(successive ranges grow in size exponentially). Summarization preserves network independence while significantly lowering storage requirements for communication profiles. The *event logger* creates detailed traces of all component-related events during application execution. A colleague has used logs from the event logger to drive detailed application simulations. During distributed execution, the *null logger* ignores all event log requests.

### 3.4. Instance classifier

The instance classifier identifies component instances with similar communication profiles across separate executions of an application. Automatic distributed partitioning depends on the accurate prediction of instance communication behavior. Accurate prediction is very difficult for dynamic, commercial application. The classifier groups instances with similar instantiation histories. The classifier operates on the theory that two instances created under similar circumstances will exhibit similar behavior (*i.e.* communicate equivalently with the same peers). Part of the output of the profile analysis engine is a map of instance classifications to computers in the network.

Coign currently includes seven instance classifiers although only one, the *internal-function called-by classifier*, is typically used. The best classifiers group instances of the same static type created from the same stack back-trace (call chain). Figure 3 illustrates each classifier.

The *incremental classifier* assigns each instance to a different classification based on its order of instantiation during application execution. Serving as a straw man for comparison, the incremental classifier can be expected to perform poorly on commercial, input-driven applications.

The *procedure called-by (PCB) classifier*, similar to Barrett and Zorn's classifier for lifetime prediction in memory allocators [3], groups instances with similar static type and instantiation stack back-trace. When walking the stack, the PCB classifier does not differentiate between individual instances of the same component class.

The *static-type (ST) classifier* groups instances with common component class (static type). The ST classifier cannot differentiate between instances of the same class and must therefore assign all instances to the same machine during distribution. This is a debilitating feature for all of the applications we examined.

The s*tatic-type called-by (STCB) classifier* groups instances by component class and the component classes of instances in the stack back-trace.

The *internal-function called-by (IFCB) classifier* groups instances by their component class and the set of function and instance-classification pairs in the stack back-trace.

The *entry-point called-by classifier* groups instances by their component class and the set of function and instance-classification pairs used to enter each component instance on the stack back-trace.

The depth of the stack back-trace for the PCB, STCB, IFCB, and EPCB classifiers can be tuned to evaluate tradeoffs between accuracy and overhead.

The *instantiated-by classifier* groups instances by their component class and their "parent" (the instance classification from which they were instantiated). The *instantiated-by classifier* is functionally equivalent to the IFCB classifier with a depth-1 stack back-trace.

**Program Control Flow:**

```
A::V() {  … a->W() … }
A::W() {  … b1->X() …  }
B::X() {  … b2->Y() … }
B::Y() {  … c->Z() …  }
C::Z() {  … CoCreateInstance(D) }
```

*where:*
**a** is an instance of component class **A**,
**b1** and **b2** are instances of component class **B**,
**c** is an instance of component class **C**,

**Classifier Descriptors:**

*Incremental Classifier:*
`[10]` (for 10th call to `CoCreateInstance`)

*Procedure Called-By (PCB) Classifier:*
`[C::Z, B::Y, B::X, A::W, A::V]`

*Static-Type (ST) Classifier:*
`[D]`

*Static-Type Called-By (STCB) Classifier:*
`[D, C, B, B, A]`

*Internal-Function Called-By (IFCB) Classifier:*
`[D, [c,Z], [b2,Y], [b1,X], [a,W], [a,V]]`

*Entry-Point Called-By (EPCB) Classifier:*
`[D, [c,Z], [b2,Y], [b1,X], [a,V]]`

*Instantiated-By (IB) Classifier:*
`[D, c]`

**Figure 3. Summary of Classifiers.**
Each instance classifier creates a descriptor at instantiation time to uniquely identify groups of similar component instances. Call-chain-based classifiers form a descriptor by examining the execution call stack.

### 3.5. Component Factory

The component factory produces a distributed application by manipulating instance placement. Using output from the instance classifier and the profile

analysis engine, the component factory moves each component instantiation request to the appropriate computer within the network. During distributed execution, a copy of the component factory is replicated onto each machine. The component factories act as peers. Each traps component instantiation requests on its own machine, forwards requests to other machines as appropriate, and fulfills instantiation requests destined for its machine by invoking COM to create the new component instance. The job of the component factory is straightforward because the instance classifier identifies components for remote placement and DCOM handles message transport. Coign currently contains a symbiotic pair of component factories. Used simultaneously, the first factory handles communication with peer factories on remote machines while the second factory interacts with the instance classifier and the interface informer.

## 4. Experimental Results

Our experimental environment consists of a pair of 200 MHz Pentium PCs with 32MB of RAM, running Windows NT 4.0 Service Pack 3. During distributed experiments, the PCs were connected through an isolated 10BaseT Ethernet with Intel EtherExpress Pro cards.

### 4.1. Application and Scenario Suite

For our experiments, we use a suite of three existing applications built from COM components. The applications employ between a dozen and 150 component classes and range in size from approximate 40,000 to 1.8 million lines of source code. The applications apply a broad spectrum of COM implementation idioms. We believe that these applications represent a wide class of COM applications.

**Microsoft PhotoDraw 2000** [32]**.** PhotoDraw is a consumer application for manipulating digital images. Taking input from high-resolution, color-rich sources such as scanners and digital cameras, PhotoDraw produces output such as publications, greeting cards, or collages. PhotoDraw includes tools for selecting a subset of an image, applying a set of transforms to the subset, and inserting the transformed subset into another image. PhotoDraw was a non-distributed application composed of approximately 112 COM component classes in 1.8 million lines of C++ source code.

**Octarine.** Octarine is a word-processing application developed by another group at Microsoft Research. Designed as a prototype to explore the limits of component granularity, Octarine contains approximately 150 classes of components. Octarine's components range in granularity from less than 32 bytes to several megabytes. Components in Octarine range in functionality from user-interface buttons to generic object dictionaries to sheet music editors. Octarine manipulates three major types of documents: word-processing, sheet music, and table. Fragments of any of the three document types can be combined into a single document. Octarine is composed of approximately 120,000 lines of C and 500 lines of *x86*-assembly source code.

**Corporate Benefits Sample** [30]**.** The Corporate Benefits Sample is an application distributed by the Microsoft Developer Network to demonstrate the use of COM to create 3-tier client-server applications. The Corporate Benefits Sample provides windows for modifying, querying, and creating graphical reports on a database of employees and their corporate human-resource benefits. The entire application contains two separate client front-ends and four alternative middle-tier servers. For our purposes, we use a client front-end consisting of approximately 5,300 lines of Visual Basic code and a middle tier server of approximately 32,000 lines of C++ source code with approximately one dozen component classes. Benefits leverages commercial components (distributed in binary form only) such as the graphing component from Microsoft Office [29].

| | Scenario | Description |
|---|---|---|
| Octarine | o_newdoc | Create text document. |
| | o_newmus | Create music document. |
| | o_newtbl | Create table document. |
| | o_oldtb0 | View 5-page table. |
| | o_oldtb3 | View 150-page table. |
| | o_oldwp0 | View 5-page text document. |
| | o_oldwp3 | View 13-page text document. |
| | o_oldwp7 | View 208-page text document. |
| | o_oldbth | View 5-page text doc. with tables. |
| | o_offtb3 | o_newdoc then o_oldtb3. |
| | o_offwp7 | o_newdoc then o_oldwp3. |
| | o_bigone | All of the above in one scenario. |
| PhotoDraw | p_newdoc | Create new image. |
| | p_newmsr | Create new composition. |
| | p_oldcur | View line drawing. |
| | p_oldmsr | View composition. |
| | p_offcur | p_newdoc then p_oldcur. |
| | p_offmsr | p_newdoc then p_oldmsr. |
| | p_bigone | All of the above in one scenario. |
| Benefits | b_vueone | View records for an employee. |
| | b_addone | Add new employee. |
| | b_delone | Delete employee. |
| | b_bigone | All of the above in one scenario. |

**Table 1. Profiling Scenarios.**
Profiling scenarios represent major usage scenarios and instantiate most component classes in each application.

Each of the applications in our test suite is dynamic and user-driven. The number and type of components instantiated in a given execution is determined by user input during execution. For example, a scenario in which a user inserts a sheet music component into an Octarine document will instantiate different components than a scenario in which the user inserts a table component into the document.

To explore the effectiveness of automatic distribution partitioning on component-based applications, our experimental suite consists of several different scenarios for each application. Scenarios range from simple to complex. The intent of the scenarios is to represent realistic usage while fully exercising the components found in the application. Table 1 describes each scenario.

### 4.2. Instance Classification

As described in Section 3.4, the instance classifier must correlate information from profiling with instantiation requests during distributed execution.

Choosing a metric to evaluate the accuracy of an instance classifier is difficult because we must evaluate how well a profile from one instance (or group of instances) correlates to another instance. In the context of automatic distributed partitioning, a profile and an instance correlate if they have similar resource usage and similar communication behavior (*i.e.* similar peers and peer-communication patterns).

To quantify communication behavior, we introduce the notion of an instance communication vector. An instance communication vector is an ordered tuple of *n* real numbers (one for each component instance in the application). Each number quantifies the communication time with another component instance (assuming that the other instance is located remotely). The communication vector can be augmented with additional dimensions representing various resources such as memory and CPU cycles. We compare the correlation between two communication vectors with the vector dot product operator. Two vectors with a dot-product correlation of one have equivalent communication behavior (i.e. they communicate equivalently with the same peers). Two vectors with a dot-product correlation of zero share no common communication behavior.

For automatic distributed partitioning, an instance classifier should identify as many unique instance classifications as possible in profiling scenarios in order to preserve distribution granularity. An instance classifier should also be reliable and stable; it should correctly identify instances with similar communication profiles and instantiation contexts.

To evaluate the instance classifiers, we ran classifiers through all of the scenarios except the `bigone` scenarios for each application to create the instance profiles. We then ran classifiers for each application through the `bigone` scenarios. The `bigone` scenarios are a synthesis of the other scenarios for the application. Because all component instances should correlate closely to prior scenarios, no new instance classifications should result from the `bigone` scenario. Table 2 lists the number of classifications identified by each classifier, the number of new classification identified in the `bigone` scenario, the average number of instances per classification, and the average correlation between instance behavior and chosen profile for the Octarine `bigone` scenario. Table 3 lists the same values for IFCB classifier with limited depth stack walks. (The called-by classifiers in Table 2 walk the complete stack.)

| Instance Classifier | Profiled Classifications | New (`bigone`) Classifications | Ave. Instances / Classification | Average Correlation |
|---|---|---|---|---|
| Incremental | 1090 | 2561 | 1.0 | 0.225 |
| Procedure Called-By | 1262 | 0 | 2.9 | 0.766 |
| Static-Type | 80 | 0 | 45.6 | 0.574 |
| Static-Type Called-By | 713 | 0 | 5.1 | 0.809 |
| Internal-Func. Called-By | 1434 | 0 | 2.6 | 0.848 |
| Entry-Pointer Called-By | 1032 | 0 | 3.5 | 0.829 |
| Instantiated-By | 590 | 0 | 6.2 | 0.809 |

**Table 2. Classifier Accuracy.**
Classifiers with a higher number of classifications recognize more unique component instances. Those with a higher average correlation are more accurate.

| *Internal-Function Called-By* Classifier Stack-Walk Depth | Profiled Classifications | Ave. Instances / Classification | Average Correlation |
|---|---|---|---|
| 1 | 590 | 6.2 | 0.809 |
| 2 | 977 | 3.7 | 0.829 |
| 3 | 1184 | 3.1 | 0.848 |
| 4 | 1383 | 2.6 | 0.848 |
| 8 | 1434 | 2.6 | 0.848 |
| 16 | 1434 | 2.6 | 0.848 |
| Complete | 1434 | 2.6 | 0.848 |

**Table 3. Accuracy as a Function of Stack Depth.**
Both classifier accuracy (average correlation) and number of classifications increase with the depth of the stack walked.

Given only the component's static type as context, the ST classifier cannot distinguish instantiations of the same component class used in radically different contexts. The "straw man" classifier, the incremental classifier, fails to correlate instances in the `bigone` scenarios with profiles from the earlier scenarios. It is strictly limited by the order of application execution and user input. Note that incremental classifier would perform well for static applications, but fails miserably for dynamic, commercial applications.

The call-chain-based instance classifiers (PCB, STCB, IFCB, EPCB, and IB) preserve more distribution granularity because they take into account contextual information when classifying an instantiation. The STCB, IFCB and EPCB classifiers are similar in accuracy. They differ however, in the number of unique component classifications they identify. As would be expected, the IFCB classifier, which uses the largest amount of contextual information, identifies the largest number of classifications.

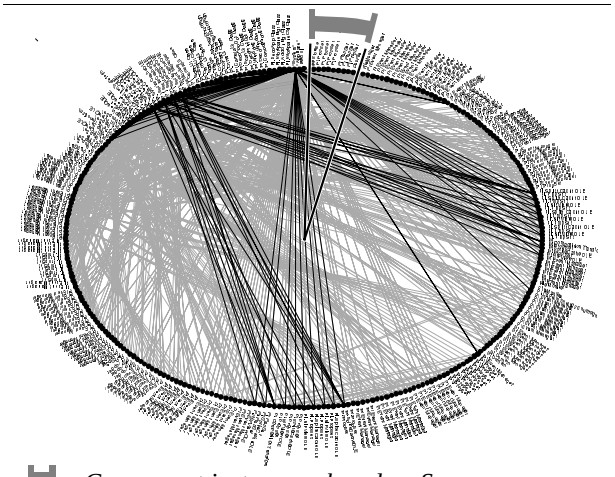
Fundamentally, our instance classifiers are limited in their accuracy by the amount of contextual information available before a component is instantiated. They cannot differentiate two instances with identical instantiation context, but vastly different communication profiles. However, experimental evidence suggests the STCB, IFCB, EPCB, and IB classifiers preserve distribution granularity and correlate profiles with sufficient accuracy to enable automatic distributed partitioning of commercial applications.

### 4.3. Distributions

Because Coign makes distribution decisions at component boundaries, it success depends on programmers to build applications with significant numbers of components. To evaluate Coign's effectiveness in automatically creating distributed applications, we ran each application in the test suite through a simple profiling scenario consisting of the simplest practical usage of the application. After profiling, Coign partitioned each application between a client and server of equal compute power on an isolated 10BaseT Ethernet network. For simplicity, we assume there is no contention for the server.

Figure 4 plots the distribution of PhotoDraw. In the profiling scenario, PhotoDraw loads a 3MB graphical composition from storage, displays the image, and exits. Of 295 components in the application, eight are placed on the server. One of the components placed on the server reads the document file. The other seven components are high-level *property sets* created directly from data in the file; with larger input sets than output sets, they are placed on the server to reduce communication.

As can be seen in Figure 4, PhotoDraw contains many significant interfaces (almost 50) that can not be distributed (shown as solid black lines). The most important non-distributable interfaces connect the *sprite cache* components (on the bottom and right) with user interface components (on the top left). Each sprite cache manages the pixels for a hierarchical subset of an image in the composition. Most of the data passed between sprite caches moves through shared memory regions. Pointers to the shared-memory regions are passed opaquely through non-distributable interfaces.



= *Component instances placed on Server.*

**Figure 4. PhotoDraw Distribution.**
Of 295 components in the application, Coign places eight on the server. Black lines represent non-distributable interfaces between components. Gray lines represent distributable interfaces.

While Coign can extract a functional distribution from PhotoDraw, most of the distribution granularity in the application is hidden by non-distributable interfaces. To enable other, potentially better distributions, either the non-distributable interfaces in PhotoDraw must be replaced with distributable IDL interfaces, or Coign must be extended to support transparent migration of shared memory regions; in essence leveraging the features of software distributed-shared memory [26].

Figure 5 shows the distribution of the Octarine word processor. In this scenario, Octarine loads and displays the first page of a 35-page, text-only document. Coign places only two components of 458 on the server. One of the components reads the document from storage; the other provides information about the properties of the text to the rest of the application.

While Figure 5 contains many non-distributable interfaces, these interfaces connect components of the GUI, and are not directly related to the document file. Unlike the other applications in our test suite, Octarine's GUI is composed of literally hundreds of components. It is highly unlikely that these GUI components would ever be located on the server. Direct document-related processing for this scenario is limited to just 24 components.

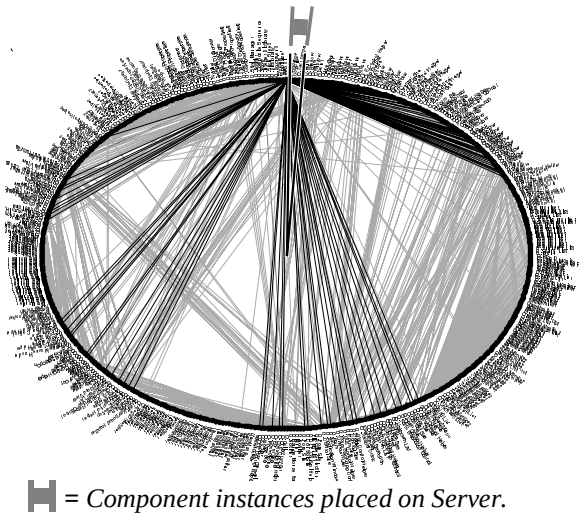



**Figure 5. Octarine Distribution.**
Of 458 components in the application, Coign places two on the server. Most of the non-distributable interfaces in Octarine connect elements of the GUI.

Figure 6 contains the distribution for the MSDN Corporate Benefits Sample. As shipped, Benefits can be distributed as either a 2-tier or a 3-tier client-server application. The 2-tier implementation places the Visual Basic front-end and the business-logic components on the client and the database, accessed through ODBC [28], on the server. The 3-tier implementation places the front-end on the client, the business-logic on the middle tier, and the database on the server. Coign cannot analyze proprietary connections between the ODBC driver and the database server. We therefore focus our analysis on the distribution of components in the front end and middle tier of the 3-tier implementation.

Coign analysis shows that application performance can be improved by moving some of the middle-tier components into the client. The distribution chosen by Coign is quite surprising. Of 196 components in the client and middle tier, Coign places 135 on the middle tier versus 187 chosen by the programmer. The new distribution reduces communication by 35%.

The intuition behind the new distribution is that many of the middle-tier components cache results for the client. Coign moves the caching components, but not the business-logic itself, from the middle-tier to the client. Although not used in this analysis, the programmer can place two kinds of explicit location constraints on components to guarantee data integrity and security requirements. Absolute constraints explicitly force an instance to a designated machine. Pair-wise constraints force the co-location of two component instances.

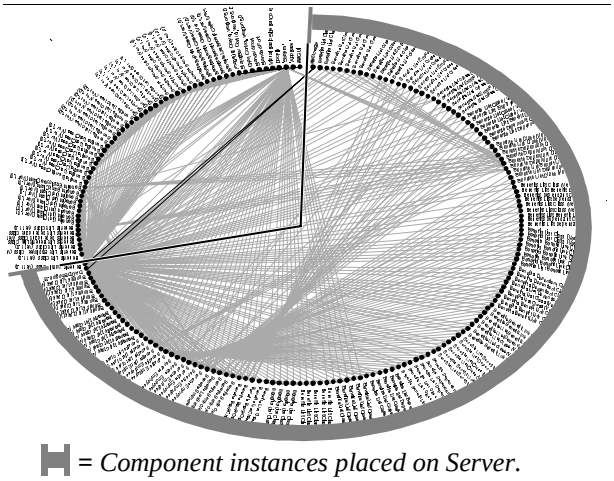



**Figure 6. Corporate Benefits Distribution.**
Of 196 components in the client and middle tier, Coign places 135 of the components on the middle tier where the programmer placed 187.

The programmer's distribution is a result of two design decisions. First, the middle tier represents a conceptually clean separation of business logic from the other pieces of the application. Second, the front-end is written in Visual Basic, an extremely popular language for rapid development of GUI applications, while the business logic is written in C++. It would be awkward for the programmer to create the distribution easily created by Coign.

The Corporate Benefits Sample demonstrates that Coign can improve the distribution of applications designed by experienced client-server programmers. In addition to direct program decomposition, Coign can also selectively enable per-interface caching (as appropriate) through COM's semi-custom marshaling mechanism.

### 4.4. Changing Scenarios and Distributions

The simple scenarios in the previous section demonstrate that Coign can automatically choose a partition and distribute an application. The Benefits example notwithstanding, one could argue that an experienced programmer with appropriate tools could partition the application at least as well manually.

Unfortunately, a programmer's best-effort manual distribution is static; it cannot readily adapt to changes in network performance or user-driven usage patterns. In the realm of changing environments, Coign has a distinct advantage as it can repartition and distribute the application arbitrarily often. In the limit, Coign can create a new distributed version of the application for each execution.

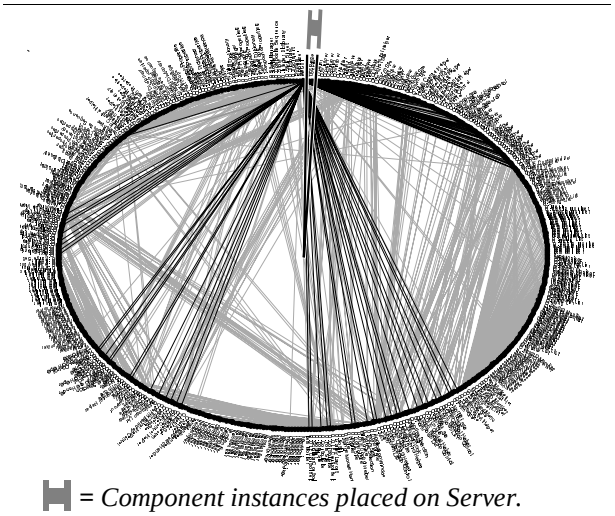



**Figure 7. Octarine with Multi-page Table.**
With a document containing a five-page table, Coign locates only a single component on the server.

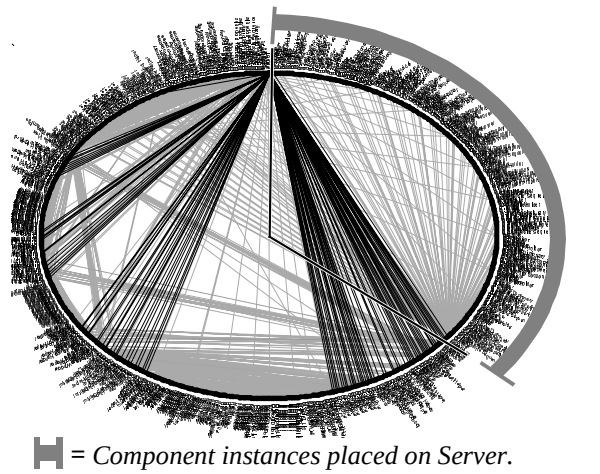



**Figure 8. Octarine with Tables and Text.**
With a five-page document containing fewer than a dozen embedded tables, Coign places 281 of 786 application components on the server.

The merits of a distribution customized to a particular usage pattern are not merely theoretical. Figure 7 plots the optimized distribution for Octarine loading a document containing a single, 5-page table. For this scenario, Coign places only a single component out of 476 on the server. The results are comparable to those of Octarine loading a document containing strictly text (Figure 5). However, if fewer than a dozen small tables are added to the 5-page text document, the optimal distribution changes radically. As can be seen in Figure 8, Coign places 281 out of 786 components on the server. The difference in distribution is due to the complex negotiations for page placement between the table components and the text components. Output from the page-placement negotiation to the rest of the application is minimal.

In a traditional distributed system, the programmer would likely optimize the application for the most common usage pattern. At best, the programmer could embed a minimal number of distribution alternatives into the application. With Coign, the programmer need not favor one distribution over another. The application can be distributed with an inter-component communication model optimized for the most common scenarios. Over the installed lifetime of the application, Coign can periodically re-profile the application and adjust the distribution accordingly. Even without updating the inter-component communication model, Coign can adjust to changes in application infrastructure, such as the relative computation power of the client and server, or network latency and bandwidth.

### 4.5. Performance of Chosen Distributions

Table 4 lists the communication time for each of the application scenarios. The default distribution is the distribution of the application as configured by the developer without Coign. For both the default and Coign-chosen distributions, data files are placed on the server. As can be seen, Coign never chooses a worse distribution than the default. In the best case, Coign reduces communication time by 99%. The Corporate Benefits Application has significant room for improvement as suggested by the change in its distribution in Section 4.3.

The results suggest that Coign is better at optimizing existing distributed applications than creating new distributed applications from desktop applications. The distribution of desktop COM-based applications is limited by the extensive use of non-remotable interfaces. For PhotoDraw in particular, Coign is severely constrained by the large number of non-remotable interfaces. It is important to note that the distributions available in Octarine and PhotoDraw are not limited by the granularity of their components, but by their interfaces. We believe that as the development of component-based applications matures, developers

will learn to create interfaces with better distribution properties, thus strengthening the benefits of Coign.

| Scenario | Comm. Time (secs.) Default | Coign | Savings |
|---|---|---|---|
| o_newdoc | 0.152 | 0.152 | 0% |
| o_newmus | 0.149 | 0.149 | 0% |
| o_newtbl | 0.006 | 0.006 | 0% |
| o_oldtb0 | 1.058 | 1.048 | 1% |
| o_oldtb3 | 15.064 | 0.042 | 99% |
| o_oldwp0 | 0.143 | 0.143 | 0% |
| o_oldwp3 | 0.696 | 0.696 | 0% |
| o_oldwp7 | 21.089 | 1.099 | 95% |
| o_oldbth | 1.734 | 0.562 | 68% |
| o_offtb3 | 15.079 | 0.037 | 99% |
| o_offwp7 | 20.878 | 1.090 | 95% |
| o_bigone | 27.497 | 22.630 | 18% |
| p_newdoc | 4.726 | 4.496 | 5% |
| p_newmsr | 17.016 | 15.014 | 12% |
| p_oldcur | 2.384 | 1.613 | 32% |
| p_oldmsr | 14.517 | 11.482 | 21% |
| p_offcur | 1.583 | 0.722 | 54% |
| p_offmsr | 14.650 | 11.497 | 22% |
| p_bigone | 33.032 | 27.084 | 18% |
| b_vueone | 1.465 | 0.954 | 35% |
| b_addone | 2.322 | 1.601 | 31% |
| b_delone | 3.414 | 2.834 | 17% |
| b_bigone | 1.754 | 1.414 | 19% |

**Table 4. Reduction in Communication Time.**
Communication time for the default distribution of the application (as shipped by the developer) and for the Coign-chosen distribution.

### 4.6. Accuracy of Prediction Models

To verify the accuracy of Coign's model of application communication time and execution time, we compare the predicted execution time for each scenario with the measured execution time (Table 5). In each case, the application is optimized for the chosen scenario before execution. Many of the scenarios had no significant difference between predicted and actual execution time; only seven had an error of 5% or greater, and none varied by more than 8%. From these measurements, we conclude that Coign's model of application communication and execution time is sufficiently accurate to warrant confidence in the distributions chosen by Coign's graph-cutting algorithm.

## 5. Related Work

The idea of automatically partitioning and distributing applications is not new. The Interconnected Processor System (ICOPS) [27, 40, 41] supported distributed application partitioning in the 1970's. ICOPS pioneered the use of compiler-generated stubs for inter-process communication. ICOPS was the first system to use scenario-based profiling to gather statistics for distributed partitioning; the first system to support multiple distributions per application based on host-processor load; and the first system to use a minimum-cut algorithm [11] to choose distributions. ICOPS distributed HUGS, a co-developed, two-dimensional drafting program. HUGS consisted of seven modules. Three of these—consisting of 20 procedures in all—could be located on either the client or the server.

| Scenario | Execution Time (sec.) Predicted | Measured | Error |
|---|---|---|---|
| o_newdoc | 10.7 | 10.7 | 0% |
| o_newmus | 10.9 | 10.9 | 0% |
| o_newtbl | 9.3 | 9.3 | 0% |
| o_oldtb0 | 19.0 | 19.1 | 0% |
| o_oldtb3 | 231.1 | 231.1 | 0% |
| o_oldwp0 | 5.5 | 5.7 | -3% |
| o_oldwp3 | 7.2 | 7.3 | -2% |
| o_oldwp7 | 33.4 | 33.6 | -1% |
| o_oldbth | 33.6 | 33.6 | 0% |
| o_offtb3 | 232.7 | 232.7 | 0% |
| o_offwp7 | 67.2 | 65.6 | 2% |
| o_bigone | 416.1 | 429.7 | -3% |
| p_newdoc | 14.3 | 14.3 | 0% |
| p_newmsr | 76.8 | 72.9 | 5% |
| p_oldcur | 18.8 | 18.8 | 0% |
| p_oldmsr | 49.0 | 49.5 | -1% |
| p_offcur | 18.1 | 18.1 | 0% |
| p_offmsr | 53.8 | 54.2 | -1% |
| p_bigone | 139.6 | 136.3 | 2% |
| b_vueone | 9.4 | 8.9 | 6% |
| b_addone | 14.6 | 13.9 | 5% |
| b_delone | 8.9 | 8.4 | 7% |
| b_bigone | 5.6 | 5.2 | 8% |

**Table 5. Accuracy of Prediction Models.**
Predicted application execution time and measured application execution time for Coign distributions.

Unlike Coign, which can distributed individual component instances, ICOPS was procedure-oriented. ICOPS placed all instances of a specific class on the same machine; a serious deficiency for commercial applications. Tied to a single language and compiler, ICOPS relied on metadata generated by the compiler to facilitate transfer of data and control between computers. Modules compiled in another language (or by another compiler) could not be distributed because they did not contain appropriate metadata. ICOPS gave the application the luxury of location transparency, but still required the programmer or user to explicitly select a distribution based on machine load.

Configurable Applications for Graphics Employing Satellites (CAGES) [16, 17] allowed a programmer to develop an application for a single computer and later distribute the application across a client/server system.

Unlike ICOPS, CAGES did not support automatic distributed partitioning. Instead, the programmer provided a pre-processor with directions about where to place each program module. The programmer could change a distribution only after recompiling the application with a new placement description file. Like ICOPS, CAGES was procedure-oriented; programs could be distributed at the granularity of procedural modules in the PL/I language. The largest application distributed by CAGES consisted of 28 modules. To aid the programmer in choosing a distribution, CAGES produced a "nearness" matrix through static analysis. The "nearness" matrix quantified the communication between modules, thus hinting how "near" the modules should be placed to each other.

One important advantage of CAGES over ICOPS was its support for simultaneous computation on both the satellite and the host computers. CAGES provided the programmer with the abstraction of one dual-processor computer on top of two physically disjoint single-processor computers. The CAGES runtime provided support for RPC and asynchronous signals.

Both ICOPS and CAGES were severely constrained by their granularity of distribution: the PL/I or ALGOL-W procedural module. Neither system ever distributed an application with more than a few dozen modules. However, despite their weaknesses, each system provided some degree of support for automatic or semi-automatic distributed application partitioning.

The Intelligent Dynamic Application Partitioning (IDAP) system [22, 25], an ADPS for CORBA applications, is an add-on to IBM's VisualAge Generator. Using VisualAge Generator's visual builder, a programmer designs an application by instantiating and connecting components in a graphical environment. The builder emits code for the created application.

The "dynamic" IDAP name refers to the usage of scenario-based profiling as an alternative to static analysis. IDAP first generates a version of the application with an instrumented message-passing system. IDAP runs the instrumented application under control of a test facility with the VisualAge system. After application execution, the programmer either manually partitions the components or invokes an automatic graph-partitioning algorithm. The algorithm used is an approximation algorithm capable of multi-way cuts for two or more hosts [10]. After choosing a distribution, VisualAge generates a new version of the application. The IDAP developers have tested their system on several real applications, but in each case, the application had "far fewer than 100" components [25].

IDAP supports distributed partitioning only for statically instantiated components. IDAP requires full access to source code. Another potential restriction is the natural granularity of CORBA applications. CORBA components tend to be large-grained objects whereas COM components in the applications we examined have a much smaller granularity. Often each CORBA component must reside in a separate server process. In essence, IDAP helps the programmer decide where CORBA servers should be placed in a network, but does not facilitate program decomposition. The IDAP programmer must be very aware of distribution choices. IDAP helps the user to optimize the distribution, but does not raise the level of abstraction above the distribution mechanisms. With a full-featured ADPS, such as Coign, the programmer can focus on component development and leave distribution to the system.

### 5.1. Distributed Object Systems

Emerald [5, 6] combines a language and operating system to create an object-oriented system with first class support for distribution. Emerald objects can migrate between machines during execution; they can also be fixed to a particular machine, or be co-located under programmer control through language operators [23]. Emerald is limited to a single language and does not attempt to automatically place objects to minimize application communication.

The SOS [39], Globe [19], and Legion [14] distributed object systems provide true location-transparent objects and direct programmer control over object location. Globe and Legion each anticipate scaling to the entire Internet. However, none of these systems supports automatic program modification to minimize communication.

### 5.2. Parallel Partitioning and Scheduling

Strictly speaking, the problem of distributed partitioning is a proper subset of the general problem of parallel partitioning and scheduling. Our work differs from similar work in parallel scheduling ([24, 36-38]) in two primary respects. First, Coign accommodates applications in which components are instantiated and destroyed dynamically throughout program execution. Traditional parallel partitioning focuses on static applications. Second, because Coign operates on binary applications, it can optimize application without access to source code (a necessary feature in the domain of commercial component-based applications).

Coign does not increase the parallelism in application code, nor does it perform horizontal load-balancing between peer servers. Instead, Coign focuses on "vertical" load-balancing within the application. The question of how to minimize

communication and maximize parallelism in large dynamic, commercial applications remains open.

## 6. Conclusions and Future Work

Coign is the first ADPS to distribute binary applications and the first ADPS to partition applications with dynamically instantiated components of any kind (either binary or source). Dynamic component instantiation is an integral feature of modern desktop applications. One of the major contributions of our work is a set of dynamic instance classifiers that correlate newly instantiated components to similar instances identified during scenario-based profiling.

Evaluation of Coign shows that it minimizes distributed communication time for each of the applications and scenarios in our test suite. Surprisingly, the greatest reduction in communication time occurs in the distributed Corporate Benefits Sample where Coign places almost half of the middle-tier components on the client without violating application security. Results from Octarine demonstrate the potential for more than one distribution of an application depending on the user's predominant document type.

We envision two models for Coign to create distributed applications. In the first model, Coign is used with other profiling tools as part of the development process. Coign shows the developer how to distribute the application optimally and provides the developer with feedback about which interfaces are communication "hot spots." The programmer fine-tunes the distribution by enabling custom marshaling and caching on communication intensive interfaces. The programmer can also enable or disable specific distributions by inserting or removing location constraints on specific components and interfaces. Alternatively, the programmer can create a distributed application with minimal effort simply by running the application through profiling scenarios and writing the corresponding distribution model into the application binary without modifying application sources.

In the second usage model, Coign is applied onsite by the application user or system administrator. The user enables application profiling through a simple GUI to Coign. After "training" the application to the user's usage patterns—by running the application through representative tasks with profiling—the GUI triggers post-profiling analysis and writes the distribution model into the application. In essence, the user has created a customized version of the distributed application without any knowledge of the underlying details.

In the future, Coign could automatically decide when usage differs significantly from profiled scenarios and silently enable profiling to re-optimize the distribution. The Coign runtime already contains sufficient infrastructure to allow "fully automatic" distribution optimization. The lightweight version of the runtime, which relocates component instantiation requests to produce the chosen distribution, could count messages between components with only slight additional overhead. Run time message counts could be compared with related message counts from the profiling scenarios to recognize changes in application usage.

## References


[1] Aral, Ziya, Illya Gertner, and Greg Schaffer. Efficient Debugging Primitives for Multiprocessors. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87-95. Boston, MA, April 1989.

[2] Arnold, Thomas R., II,. *Software Testing with Visual Test 4.0*. IDG Books Worldwide, Foster City, CA, 1996.

[3] Barrett, David A. and Benjamin G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 187-196. Albuquerque, NM, June 1993. ACM.

[4] Birrell, A. D. and B. J. Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems*, 2(1):39-59, 1984.

[5] Black, A., N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 78-86. Portland, OR, October 1986.

[6] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, 1987.

[7] Bokhari, Shahid. Partitioning Problems in Parallel, Pipelined, and Distributed Computing. *IEEE Transactions on Computers*, 37(1):48-57, 1988.

[8] Brown, Nat and Charlie Kindel. *Distributed Component Object Model Protocol -- DCOM/1.0*. Microsoft Corporation, Redmond, WA, 1996.

[9] Cormen, Thomas H., Charles E. Leiserson, and Rondald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[10] Dahlhaus, E., D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23(4):864-894, 1994.

[11] Ford, Lester R., Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[12] Gary, Naveen, Vijay V. Vazirani, and Mihalis Yannakakis. Multiway Cuts in Directed and Node Weighted Graphs. *Proceedings of the 21st International Colloquim on Automata, Languages, and Programming (ICALP)*, pp. 487-498. Jerusalem, Isreal, July 1994. Springer-Verlag.

[13] Goldberg, Andrew V., Éva Tardos, and Robert E. Tarjan. Network Flow Algorithms. Computer Science Department, Stanford University, Technical Report STAN-CS-89-1252, 1989.

[14] Grimshaw, Andrew S., William A. Wulf, and the Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1), 1997.

[15] Hamilton, K. G. A Remote Procedure Call System. Ph. D. Dissertation, Computer Laboratory TR 70. University of Cambridge, Cambridge, UK, 1984.

[16] Hamlin, Griffith, Jr. Configurable Applications for Satellite Graphics. *Proceedings of the Third Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '76)*, pp. 196-203. Philadelphia, PA, July 1976. ACM.

[17] Hamlin, Griffith, Jr. and James D. Foley. Configurable Applications for Graphics Employing Satellites (CAGES). *Proceedings of the Second Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '75)*, pp. 9-19. Bowling Green, Ohio, June 1975. ACM.

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

[19] Homburg, Philip, Martin van Steen, and Andrew S. Tanenbaum. An Architecture for a Scalable Wide Area Distributed System. *Proceedings of the Seventh ACM SIGOPS European Workshop*. Connemara, Ireland, September 1996.

[20] Hunt, Galen. Automatic Distributed Partitioning of Component-Based Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, 1998.

[21] Hunt, Galen. Detours: Binary Interception of Win32 Functions. Microsoft Research, Redmond, WA, MSR-TR-98-33, July 1998.

[22] IBM Corporation. *VisualAge Generator*. Version 3.0, Raleigh. NC, 1997.

[23] Jul, Eric, Henry Levy, Norman Hutchinson, and Andew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109-133, 1988.

[24] Kennedy, Ken and Ajay Sethi. A Communication Placement Framework for Unified Dependency and Data-Flow Analysis. *Proceedings of the Third International Conference on High Performance Computing*. India, December 1996.

[25] Kimelman, Doug, Tova Roth, Hayden Lindsey, and Sandy Thomas. A Tool for Partitioning Distributed Object Applications Based on Communication Dynamics and Visual Feedback. *Proceedings of the Advanced Technology Workshop, Third USENIX Conference on Object-Oriented Technologies and Systems*. Portland, OR, June 1997.

[26] Li, Kai and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *Transactions on Computer Systems*, 7(4):321-359, 1989.

[27] Michel, Janet and Andries van Dam. Experience with Distributed Processing on a Host/Satellite Graphics System. *Proceedings of the Third Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '76)*, pp. 190-195. Philadelphia, PA, July 1976.

[28] Microsoft Corporation. *Microsoft Open Database Connectivity Software Development Kit.* Version 2.0. Microsoft Press, 1994.

[29] Microsoft Corporation. *Microsoft Office 97*. Version 6.0, Redmond, WA, 1997.

[30] Microsoft Corporation. Overview of the Corporate Benefits System. *Microsoft Developer Network*, 1997.

[31] Microsoft Corporation. *MIDL Programmer's Guide and Reference*. Windows Platform SDK, Redmond, WA, 1998.

[32] Microsoft Corporation. *PhotoDraw 2000*. Version 1.0, Redmond, WA, 1998.

[33] Naor, Joseph and Leonid Zosin. A 2-Approximation Algorithm for the Directed Multiway Cut Problem. *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pp. 548-553, 1997.

[34] Nelson, B. J. Remote Procedure Call. Ph.D. Dissertation, Department of Computer Science. Carnegie-Mellon University, 1981.

[35] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. vol. Revision 2.0, Framingham, MA, 1995.

[36] Ousterhout, J. K. Techniques for Concurrent Systems. *Proceedings of the Third International Conference on Distributed Computing Systems*, pp. 22-30. Miami/Ft. Lauderdale, FL, October 1982. IEEE.

[37] Polychronopolous, C. D. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Boston, MA, 1988.

[38] Sarkar, Vivek. Partitioning and Scheduling for Execution on Multiprocessors. Ph.D. Dissertation, Department of Computer Science. Stanford University, 1987.

[39] Shapiro, Marc. Prototyping a Distributed Object-Oriented Operating System on Unix. *Proceedings of the Workshop on Experiences with Distributed and Multiprocessor Systems*, pp. 311-331. Fort Lauderdale, FL, October 1989. USENIX.

[40] Stabler, George M. A System for Interconnected Processing. Ph.D. Dissertation, Department of Applied Mathematics. Brown University, Providence, RI, 1974.

[41] van Dam, Andries, George M. Stabler, and Richard J. Harrington. Intelligent Satellites for Interactive Graphics. *Proceedings of the IEEE*, 62(4):483-492, 1974.

[42] Vinoski, Steve. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), 1997.