A Guided Tour of 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

Distributed object systems such as CORBA and DCOM bring many advances to distributed computing. The distribution process itself, however, has changed little: programmers still manually divide applications into sub-programs and assign those sub-programs to machines with little automated assistance. Often the techniques used to choose a distribution are ad hoc. Due to high intellectual cost, applications are seldom repartitioned even in drastically changing network environments.

We describe Coign, an automatic distributed partitioning system (ADPS) that significantly facilitates 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 graph-cutting algorithms to partition the application across a network and minimize distribution costs. Using Coign, an end user without source code can transform a non-distributed application into an optimized, distributed application.

Through a guided tour of Coign's architecture and usage, we present an overview of its features. We describe the automatic distributed partitioning of three applications: Microsoft Picture It!, the Octarine word processor, and the Corporate Benefits Sample program. All are distributed automatically, sometimes with startling results. For example, Coign makes significant changes to the programmer-assigned distribution of the Corporate Benefits Sample.

1. Introduction

Distributed object systems such as CORBA and DCOM bring the advantages of service location transparency, dynamic program instantiation, and object-oriented programming to distributed applications. Unfortunately, the process to distribute program components has changed little: programmers still 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, one-time solutions.

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

In this paper, we present a guided tour of Coign [5], an automatic distributed partitioning system (ADPS) that promises to significantly ease the development of component-based distributed applications. Given an application built with COM components, Coign uses inexpensive scenario-based profiling on a single computer to quantify inter-component communication costs for both single-machine and multi-machine distributions. Inter-component communication is modeled as a graph in which nodes represent components and edges represent inter-component communication and location constraints. Using graph-cutting algorithms, Coign selects a distribution of the application that minimizes communication costs. At run time, Coign manipulates program execution (with negligible overhead) to produce the desired distribution.

Coign analyzes an application, chooses a distribution, and produces the desired distribution all without access to application source. As a corollary, Coign is completely language neutral; it neither knows nor cares about the source language of the components in the application. Moreover, because it operates on binaries, Coign preserves the ability to build applications from reusable, third-party components.

In the following section we describe related work. In Section 3, we illustrate how to use Coign to automatically

© 1998 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. Published in *Proceedings of the 2nd International Enterprise Distributed Object Computing Workshop (EDOC '98)*, pp. 252-262. San Diego, CA, November 1998.

distribute an application. We describe Coign's architecture in Section 4. Section 5 contains an experimental evaluation of Coign's effectiveness in distributing three applications. Finally, in Section 6 we conclude and discuss future work.

2. Related Work

2.1. ICOPS

The idea of an ADPS is not new. The Interconnected Processor System (ICOPS) [8, 14, 16] supported distributed application partitioning in the 1970's. Under the direction of Andries van Dam, 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 maximum-flow-minimum-cut (MAX-FLOW/MIN-CUT) algorithm [4] to choose distributions.

ICOPS was used to automatically distribute HUGS, a two dimensional drafting program developed at Brown University. HUGS consisted of seven modules. Three of these—consisting of 20 procedures in all—could be located on either the client or the server.

ICOPS was never intended for shrink-wrapped, commercial applications. Tied to a single language and compiler, it 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.

2.2. IDAP

Kimelman *et al.* [7] describe the Intelligent Dynamic Application Partitioning (IDAP) system, an ADPS for Smalltalk applications. IDAP 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" in the IDAP name refers to scenario-based profiling as opposed to static analysis. IDAP first generates a version of the application with an instrumented message-passing mechanism. IDAP runs the instrumented application under control of a test facility with the VisualAge system. After the 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 [3]. After choosing a distribution, VisualAge generates a new version of the application.

IDAP supports distributed partitioning only for statically allocated components. Although initially based on Smalltalk, the distributable components are large-grain components, not the fine-grained objects native to Smalltalk. Kimelman's team has tested their system on a number of real Smalltalk applications, but in each case, the application had "far fewer than 100" components [7].

The latest version of IDAP generates C++ code to connect CORBA components, but still does not support dynamic component instantiation [6]. Moreover, the use of CORBA restricts IDAP to a distribution granularity of whole processes because CORBA does not support loading multiple component servers into the same address space. The IDAP programmer must be vary 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.

Although it supports multiple languages, IDAP still requires that applications be constructed with a specific application development toolkit, the VisualAge Generator. Like ICOPS, IDAP supports automatic distributed partitioning of static application pieces only. In the case of ICOPS, the application pieces are procedures. In the case of IDAP, the pieces are CORBA components or large-grain Smalltalk objects.

2.3. Summary

Prior to Coign, no ADPS allowed distributed partitioning of binary components dynamically instantiated during the execution of the application. Dynamic component instantiation is an integral feature of modern desktop applications. One of the major contributions of our work is a set of dynamic component classification algorithms that map newly-created components to similar components identified during scenario based profiling.

Our research differs in scope from prior work because we automatically distribute an existing class of commercial applications. All of the applications in our test suite were developed by third parties with no knowledge of the Coign system.

3. A Guided Tour

To solidify the concept of an ADPS, we describe a detailed example of Coign's usage to automatically distribute an existing COM application. The application used in this example is a preliminary version of a future release of *Microsoft Picture It!* [12]. (The original, un-instrumented version of *Picture It!* application is designed to run on a

single computer—it provides no explicit support for distribution.)

3.1. Creating a Distributed Application

Starting with the original binary files for *Picture It!*, we use the `setcoign` utility to insert the Coign profiling instrumentation package, see Figure 1. `Setcoign` makes two modifications to the `pi.exe` binary file. First, it inserts an entry to load the Coign Runtime Executive (RTE) Dynamic-Link Library (DLL) into the first slot in the application's DLL import table. Second, `setcoign` adds a data segment containing configuration information to the end of `pi.exe`. The configuration information tells the Coign RTE how the application should be profiled and which of several algorithms should be used to identify components during execution.

Because it occupies the first slot in the application's DLL import table, the Coign RTE will always load and execute before the application or any of its other DLLs. It therefore has a chance to modify the application's address space before the application runs. The Coign RTE takes advantage of this opportunity to insert binary instrumentation into the image of system libraries in the application's address space. The instrumentation traps all component instantiation functions in the COM library. Before returning control to the application, the Coign RTE loads any additional Coign components (described in Section 4) as stipulated by the configuration information stored in the application.

With the Coign runtime configured for profiling, the application is ready to be run through a set of profiling scenarios; see Figure 2. Because the binary has been modified transparently to the user (and to the application itself), profiling runs behave from the user's point of view as if there were no instrumentation in place. The instrumentation gathers profiling information in the background while the user controls the application. The only visible effect of profiling is a degradation in application performance of up to 85%. For our simple example, we start *Picture It!*, load a file for preview, and exit the application. For more advanced profiling, scenarios can be driven by an automated testing tool, such as Visual Test [2].

During profiling, the Coign instrumentation maintains running summaries of the inter-component communication within the application. Coign quantifies every inter-component function call through a COM interface. The instrumentation measures the number of bytes that would have to be transferred between machines if the two communicating components were distributed. The number of bytes is calculated by invoking portions of the DCOM code, including the interface proxy and stub, within the application's address space. Coign measurement follows precisely the deep-copy semantics of DCOM. After calculating communication costs, Coign compresses and

```
D:\apps\pictureit\bin> setcoign /p pi.exe
Config:
    Logger:      Coign Profile Logger
    Informer:    Coign NDR Interface Informer
    Classifier:  Coign EP3C Classifier
Sections:  4     ___VAddr ___VSize ___VAEnd ___FAddr ___FSize R L R L
  .text              1000   10e343   10f343      400   10e400 0 0 0 0
  .rdata           110000    501c3   1601c3   10e800    50200 0 0 0 0
  .data            161000    11224   172224   15ea00     d400 0 0 0 0
  .rsrc            173000    15868   188868   16be00    15a00 0 0 0 0
  .coign           189000     6cd0   18fcd0   181800     6e00 0 0 0 0
Debug Directories:
   0. 00000000 00181800..00181910 -> 00188600..00188710
   1. 00000000 00181910..001819c0 -> 00188710..001887c0
   2. 00000000 001819c0..001819ea -> 001887c0..001887ea
Extra Data:      512 (  181a00 -   181800)
Coign Extra Data:
    {9CEEB02F-E415-11D0-98D1-006097B010E3} : 4 bytes.
```

Figure 1, Inserting Coign into the Application. `Setcoign` rewrites the pi.exe binary to insert Coign profiling instrumentation into *Picture It!*.

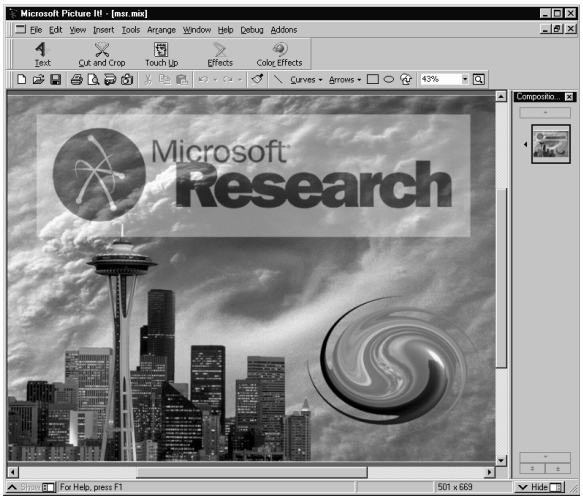

Figure 2, Executing a Profiling Scenario. With Coign instrumentation in place, the application is run through one or more profiling scenarios to measure inter-component communication. In this scenario, *Picture It!* loads and renders a composite image.

summarizes the data online to keep instrumentation storage requirements at a minimum. If desired, the application may be run through profiling scenarios for days or even weeks to more accurately track user usage patterns. In all of our tests, storage overhead for Coign never exceeded 1.5 MB.

At the end of the profiling execution, Coign writes the summary log of inter-component communication to a file for later analysis; see Figure 3. In addition to information about the number and sizes of messages and components in the application, the profile log also contains information to classify components to determine component location constraints. Log files from multiple profiling executions may be combined and summarized during later analysis. Alternatively, at the end of each profiling execution, information from the log file may be inserted into the configuration record in the application executable (the pi.exe file in this example). The latter approach uses less storage because summary information in the configuration

```
D:\apps\pictureit\bin> pi.exe
[Coign Runtime Environment: 00000080 636f6900 00000000]
[Coign EP3C Classifier/9999]
[Coign NDR Interface Informer]
[Coign Profiling Logger (16 cycles)]
[CoignRTE: DLL_PROCESS_ATTACH]
[CoignRTE: DLL_THREAD_ATTACH]
[CreateFileMoniker( D:\apps\pictureit\docs\MSR.mix )]
[StgOpenStorage( D:\apps\pictureit\docs\MSR.mix )]
[CoignRTE: DLL_THREAD_DETACH]
[Elapsed time: 26400 ms]
[CoignRTE: DLL_PROCESS_DETACH]
[Inter-component communication:                                          ]
[  Messages  : ____16 ____64 ___256 __1024 __4096 __16384 __Totals ]
[  In Counts : 105240   1629    473   1599     66      45   109052 ]
[  Out Counts: 102980   4303    843    783    131      12   109052 ]
[  In Bytes  : 782022  57912  49616 815034 157619  237963  2100166 ]
[  Out Bytes : 455207 130140  95473 304592 239239   70019  1294670 ]
```

Figure 3, Logging Communication. At the conclusion of the profiling scenario, Coign logs the inter-component communication to a file for later analysis.

```
D:\apps\pictureit\bin> adpcoign pi.log
Binaries:
  pi.exe
  mso97d.dll
  mfc42d.dll
  mfco42d.dll
  oleaut32.dll
Dependencies:
          01 D:\apps\pictureit\bin\pi.exe
               piperf.dll
               oleaut32.dll
          00 D:\apps\pictureit\bin\piserv.dll
               mfc42d.dll
          00 D:\apps\pictureit\bin\mfco42d.dll
          00 C:\winnt\system32\ole32.dll
Objects:               112
Interfaces:            792
Calls:               38286
Bytes:              743534
Proc. Speed:           200MHz
```

Figure 4, Post-Profiling Analysis. `Adpcoign` analyzes the log file to create an abstract model of inter-component communication.

```
D:\apps\pictureit\bin> setcoign /f:pi.set pi.exe
Config:          pi.set
    Informer:    Coign Light Interface Informer
    Classifier:  Coign EP3C Classifier
Sections:  5     ___VAddr ___VSize ___VAEnd ___FAddr ___FSize R L R L
  .text              1000   10e343   10f343      400   10e400 0 0 0 0
  .rdata           110000    501c3   1601c3   10e800    50200 0 0 0 0
  .data            161000    11224   172224   15ea00     d400 0 0 0 0
  .rsrc            173000    15868   188868   16be00    15a00 0 0 0 0
  .coign           189000     83f8   1913f8   181800     8400 0 0 0 0
Debug Directories:
   0. 00000000 00189a00..00189b10 -> 00189c00..00189d10
   1. 00000000 00189b10..00189bc0 -> 00189d10..00189dc0
   2. 00000000 00189bc0..00189bea -> 00189dc0..00189dea
Coign Extra Data:
    {9CEEB022-E415-11D0-98D1-006097B010E3} : 4980 bytes.
    {9CEEB030-E415-11D0-98D1-006097B010E3} : 904 bytes.
    {9CEEB02F-E415-11D0-98D1-006097B010E3} : 4 bytes.
```

Figure 5, Inserting the Model into the Application. An abstract model of inter-component communication is written into the application binary for distribution.

record accumulates communication from similar interface calls into a single entry.

Invoking `adpcoign` initiates post-profiling analysis, see Figure 4. `Adpcoign` examines the system service libraries to determine any location constraints on application components. For client-server distributions, `adpcoign` recognizes components that must be placed on the client in order to access the Windows GUI libraries or that must be placed on the server in order to access persistent storage directly.

Combining location constraints and information about inter-component communication, `adpcoign` creates an abstract graph model of the application. In the current implementation, `adpcoign` combines the abstract graph model with data about the network configuration to create a concrete model of the cost of distribution on a real network. `Adpcoign` then uses a graph-cutting algorithm to choose a distribution that minimizes communication costs. In the future, the construction of the concrete model and the graph-cutting algorithm could be performed at application execution time, thus potentially producing a new distribution tailored to current network characteristics.

After analysis, the application's inter-component communication model is written into the configuration record in the application binary; see Figure 5. Any residual profiling logs are then removed from the configuration record. The configuration record is also modified to disable 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.

Aside from the inter-component communication model, perhaps the most important information written into the application configuration is data for the component classifier. The component classifier matches components created during distributed executions to components created during the profiling scenarios. The abstract model of inter-component communication contains nodes for all known components and edges representing the communication between components. To determine where a component should be located in a distributed execution, the classifier tries to match it to the most similar component in the profiling scenario. The premise of scenario-based profiling is that profiled executions closely match post-analysis executions. Therefore, if the circumstances of a component's creation are similar to those of a component in a profiling execution, then the components will most likely have similar communication patterns. Based on the chosen distribution for similar profiled components, the classifier decides where new components created during the distributed execution should be instantiated.

Figure 6 shows the distribution chosen for our profiled scenario. In this scenario, the user loads and previews an image in *Microsoft Picture It!* from a server. Each of the large black dots in Figure 6 represents a dynamic component in the profiled scenario. Lines between the components represent COM interfaces through which the connected components communicate. In the on-screen version of Figure 6, lines are colored according to the amount of communication flowing across the interface. Red lines represent interfaces with large amounts of communication (communication hot spots) and blue lines represent interfaces with minimal communication.

Solid, black lines represent interfaces that are non-remotable (i.e., pairs of components that must reside on the same machine). An interface may not be remotable

for any of the following reasons: the interface has no Interface Definition Language (IDL) description to enable parameter marshaling; one or more of the interface parameters is opaque, such as a "`void*`"; the client directly accesses the component's internal data; or the component must reside on the client or the server because it directly accesses system services.

The "pie" slice in the top half of Figure 6 contains those components that should be located on the server to minimize network traffic and thus execution time. In our example, the operating storage services, the document file component, and three "property set" components are all located on the server. Note that approximately one dozen "property set" components (of the "PI.PropSet" class) are located on the client. In order to achieve optimal performance, a component-based ADPS must be able to place components of the same class on different machines.

After the abstract distribution model is written into the binary, the application is prepared for distribution. When the application user instructs *Picture It!* to load an image from the server, the lightweight version of the Coign runtime will trap the related instantiation request and relocate it to the server. The four components chosen for distribution in Figure 6 are automatically distributed to the server. Coign distributes components to the server by starting a *surrogate* process on the server. The surrogate acts as a distributed extension of the application; described components reside in its address space. A distributed version of the Coign runtime maintains communication links between the original application process on the client and the surrogate process on the server.

Figure 7 shows the distributed version of *Picture It!*. The window in the lower right corner of Figure 7 represents the surrogate process and the components distributed on the server. Coign automatically created a distributed version of *Microsoft Picture It!* without access to the application source code or the programmer's knowledge of the application. The automatic distributed application is customized for the specific network to minimize distributed communication costs.

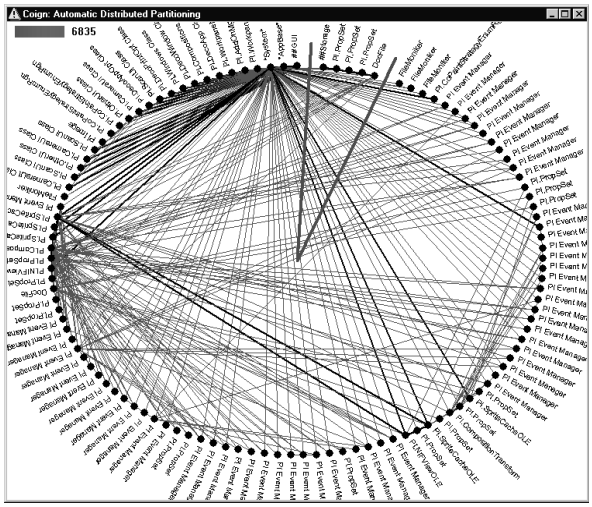

Figure 6, Choosing a Distribution. Coign cuts the graph using online network performance parameters to minimize distributed communication costs. The pie slice in the upper right contains the components selected for distribution on the server.

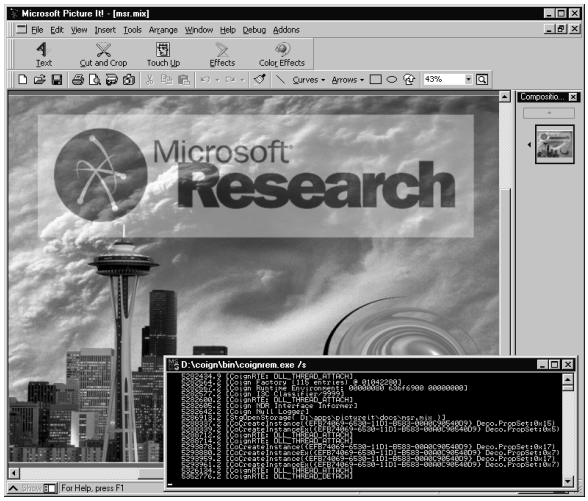

Figure 7, The Distributed Application. Component instantiation requests are relocated to produce the distributed application. The new, distributed application is functionally equivalent to the original, non-distributed application. Distribution is achieve by modifying the application binary at run time, not the application sources.

3.2. Discussion

We envision two Coign usage models 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 can fine-tune the distribution by inserting custom marshalling 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 used on-site by the application user or system administrator. The user enables application profiling through a simple GUI to the `setcoign` utility. After "training" the application to the user's usage patterns—by running the application through

representative scenarios—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, the Coign could automatically decide when usage differs significantly from the profiled scenarios and silently enable full profiling for a period 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 relative message counts from the profiling scenarios to recognize changes in application usage.

4. Architecture of the Coign ADPS

The Coign runtime is composed of a small collection of replaceable COM components; see Figure 8. The most important components are the *Coign Runtime Executive* (RTE), the *interface informer*, the *information logger*, the *component classifier*, and the *component factory*. The RTE provides low-level services to the other components in the Coign runtime. The *interface informer* identifies interfaces by their static type and provides support to walk the parameters of interface function calls. The *information logger* receives detailed information about all component-related events in the application from the RTE and the other Coign runtime components. The information logger records relevant events for post-profile analysis. The *component classifier* identifies components with similar communication patterns across multiple program executions. The *component factory* decides where component instantiation requests should be fulfilled and relocates instantiation requests as needed to produce a chosen distribution.

4.1. Runtime Executive

The Coign Runtime Executive is the first DLL loaded into the application address space. As such, the RTE runs before the application or any of its components. The RTE patches the COM library and other system services to trap component instantiation requests. The RTE reads the configuration information written into the application binary by the `setcoign` utility. Based on information in the configuration record, the RTE loads other components of the Coign runtime.

The RTE provides a number of low-level services to the other components in the Coign runtime including: interception of component instantiation requests; wrapping of component interfaces to intercept inter-component messages; management of thread-local stack storage for use by other components in the Coign runtime; and access to configuration information stored in the application binary.

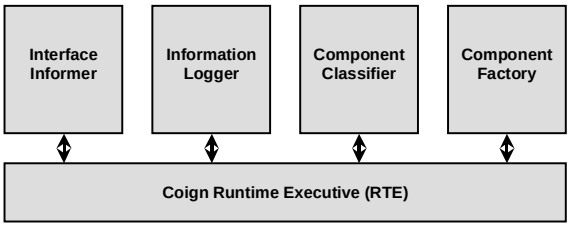

Figure 8, Coign's Component Architecture. Runtime components can be replaced to produce lightweight instrumentation, detailed inter-component communication trace logs, or remote component instantiation.

The current Coign runtime contains a single RTE. The DLLs for profiling and "regular" program execution differ in the choice of components to run on top of the RTE.

4.2. The Interface Informer

The interface informer locates and manages interface metadata. With assistance from the interface informer, other components can determine the static type of a COM interface, and walk both the input and output parameters of an interface function call. The current Coign runtime contains two interface informers. The first interface informer operates during scenario-based profiling. The profiling interface informer uses format strings generated by the MIDL compiler [13] and interface marshalling code to analyze all function call parameters and precisely measure inter-component communication. Profiling currently adds up to 85% to execution run time. Most of this overhead is incurred by the interface informer.

The second interface informer is used after profiling to produce the distributed application. The distributed informer examines function call parameters only enough to locate interface pointers. As a result of aggressive pre-execution optimization of interface metadata, the distributed informer imposes an execution overhead of less than 3% on most applications.

4.3. The Information Logger

The information logger summarizes and records data for automatic distributed partitioning analysis. Under direction of the RTE, Coign runtime components pass information about a number of events to the information logger. The logger is free to process the events as it wishes. Depending on the implementation, it might ignore the event, write the event to a log file on disk, or accumulate information about the event 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-profile analysis. The *event logger* creates detailed traces of all component-related events during application execution. One of our colleagues has used traces generated by the event logger to drive detailed simulations of the execution of component-based applications. The *null logger* ignores all events. Use of the null logger insures that no extra files are generated during execution of the automatically distributed application.

4.4. Component Classifier

The component classifier identifies components with similar communication patterns across multiple executions of an application. Coign's scenario-based approach to automatic distribution depends on the premise that the communication behavior of a component during a distributed application can be predicted based on the component's similarity to another component in a profiling scenario. Because in the general case it is impossible to determine *a priori* the communication behavior of a component, the component classifier groups components with similar instantiation histories. The classifier operates on the theory that two components created under similar circumstances will display similar behavior. The output of the post-profiling graph-cutting algorithm is a mapping of component classifications to computers in the network.

The current Coign runtime includes eight component classifiers created for evaluation purposes. Experiments indicated that the most accurate results are produced (at reasonable cost) by grouping components of the same type that are created in the presence of the same stack backtrace.

4.5. Component Factory

The component factory produces the distributed application. Using output from the component classifier and the graph-cutting algorithm, 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 them to another machine 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 very straightforward since most of the difficult problems in creating a distributed application are handled either by the underlying DCOM system or by the component classifier. 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 component classifier and the interface informer.

4.6. Summary

The component structure of the Coign runtime allows it to be used for a wide range of analysis and adaptation tasks. The scenario-profiling runtime is converted to the distribution-realization runtime by changing the interface informer and information logger components. By changing just the information logger component, one of our colleagues has generated extensive traces of the activity of component-based applications. These traces were used to drive a simulator to analyze the performance of algorithms for a large distributed object system.

5. Evaluation

Because it makes distribution decisions at component boundaries, Coign depends on programmers to build applications with significant numbers of components. For our experiments, we use a suite of three applications built from COM components: *Microsoft Picture It!*, *Octarine*, and the *Corporate Benefits Sample*. We believe that these applications represent a wide class of COM applications.

Microsoft Picture It! [12] is a consumer image manipulation application. *Picture It!* includes tools to select a subset of an image, apply a set of transforms to the subset, and insert the transformed subset into another image. *Picture It!* is a non-distributed application composed of approximately 112 COM component classes in 1.8 million lines of C++ source code.

Designed as a prototype to explore the limits of component granularity, Octarine is a document processing application similar to *Microsoft Word*. Octarine contains approximately 150 classes of components ranging in granularity from less than 32 bytes to several megabytes. Octarine's components range in functionality from user-interface buttons to generic dictionaries to sheet music editors. Octarine manipulates three major types of documents: word-processing documents, sheet-music documents, and table documents. 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.

The Corporate Benefits Sample [11] is an application distributed by Microsoft Corporation to demonstrate the use of COM to create client-server applications. The Corporate Benefits Sample provides windows to modify, query, and graph a database of corporate employees and their benefits. The entire sample application contains two separate client front-ends and four distinct server back-ends. For our purposes, we use a client front-end consisting of approximately 5,300 lines of Visual Basic code and

a server back-end 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* [10].

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 at run time. 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.

5.1. Simple Distributions

For our first set of experiments, 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 computational power on an isolated 10BaseT Ethernet network. For simplicity, we assume that there is no contention for the server.

Figure 9 contains a graphical representation of the distribution of *Microsoft Picture It!*. In the profiling scenario, *Picture It!* loaded a 3 MB graphical composition from storage, displayed the image, and exited. Of 295 components in the application, eight were placed on the server. One of the components located on the server is the component that reads the document file. The other seven components are high-level *property sets* created directly from data in the file. The property sets are located on the server because their output data sets are smaller than their input data sets. The new distribution reduces overall application communication latency by 21%.

As can be seen in Figure 9, *Picture It!* contains a large number of interfaces that can not be distributed; these are represented by solid black lines. The most important non-distributable interfaces connect the *sprite cache* components (on the bottom and right) with the user interface components. Each sprite cache component manages the pixels for an image in the composition. Most of the data passed between sprite caches moves through shared memory regions. These shared-memory regions correspond to non-distributable component interfaces. While Coign can extract a functional distribution from *Picture It!*, 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 *Picture It!* must be replaced with distributed IDL interfaces, or Coign must be extended to support transparent migration of shared memory regions; in essence leveraging the features of a software distributed shared memory (DSM) system [1, 15].

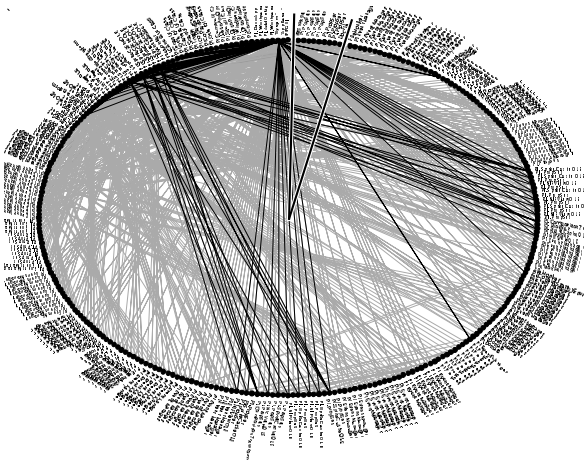

Figure 9, *Microsoft Picture It!*. Of 295 components in the application, eight are placed on the server.

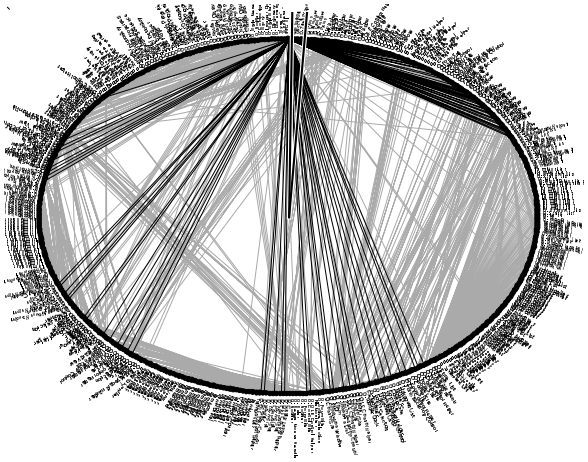

Figure 10, Distribution of Octarine. Of 458 components in the application, two are placed on the server.

Figure 10 contains a graphical representation of the distribution of the Octarine word processor. In this scenario, Octarine loaded and displayed the first page of a 35-page, text-only document. Only two components of 458 are located 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 10 contains a number of 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 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 24 components.

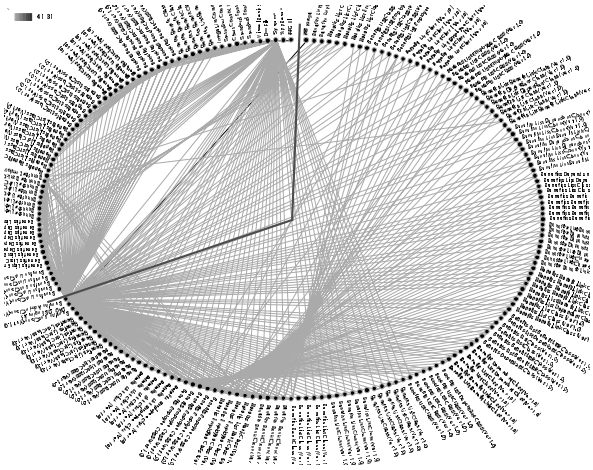

Figure 11, The Corporate Benefits Sample. Of 196 components in the client and middle tier, Coign locates 135 on the middle tier, rather than the 187 components chosen by the programmer.

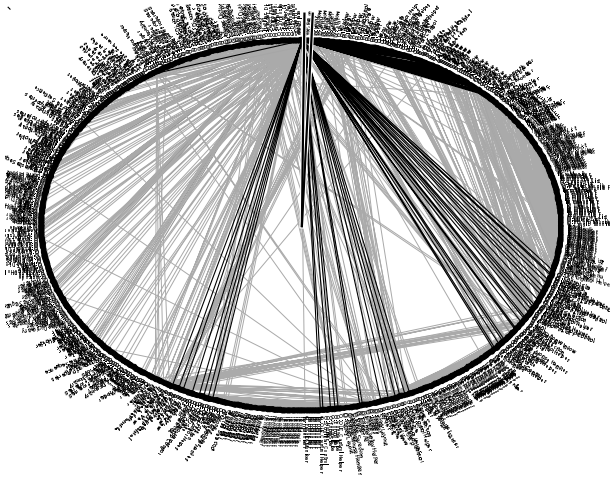

Figure 12, Octarine with a Multi-page Table. Coign locates only a single component on the server for a document containing five-page table.

Figure 11 plots the distribution for the Corporate Benefits Sample. As shipped, Benefits can be distributed as either a 2-tier or a 3-tier client-server application. In the 2-tier scenario, the Visual Basic front-end and the business-logic components are located on the client, while the database is located on the server and accessed through ODBC, a commercial database-access system [9]. In the 3-tier scenario, the Visual Basic front-end is located on the client, the business-logic components are located on the middle tier, and the database is located 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.

In the original distribution, chosen by a programmer, 187 of 196 components are located in the middle tier. The programmer's distribution is a result of two design choices. 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, a popular language for rapid development of GUI applications, while the business logic is written in C++.

Coign analysis shows that application performance can be improved by moving some of the business-logic components from the middle tier into the client. The distribution chosen by Coign is quite surprising. Of 196 components in the client and middle tier, Coign would locate just 135 on the middle tier. The new distribution reduces communication latency by 35%. Note that this distribution does not violate any data security restriction in the original distribution. The new distribution would be extremely awkward for the programmer to create the distribution chosen by Coign because it runs counter to the programmer's logical separation of the application. The Corporate Benefits Sample demonstrates that Coign can improve the distribution of applications designed by experienced client-server programmers.

5.2 Changing Scenarios and Distributions

The results 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 a changing environment, 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.

The merits of a distribution customized to a particular usage pattern are not merely theoretical. Figure 12 plots the optimized distribution for Octarine loading a document containing a single, 5-page table. For this scenario, Octarine places only a single component out of 480 in the application on the server. The results are comparable to those of Octarine loading a document containing strictly text, see Figure 10. 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 13, Coign places over 281 components on the server out of 786 created in the scenario. 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.

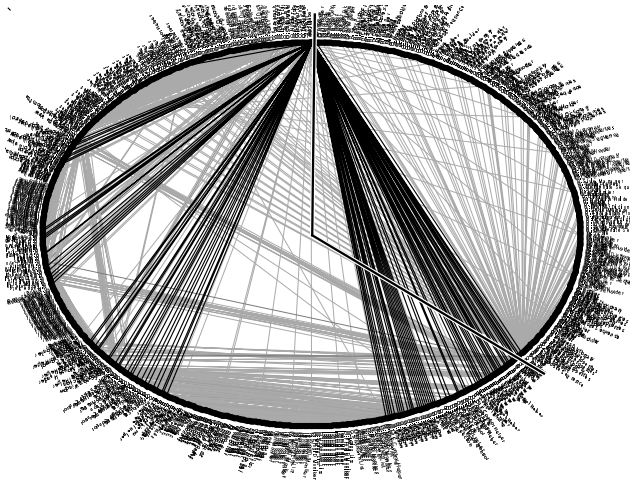

Figure 13, Tables Interspersed Between Text. With a five-page document containing fewer than a dozen embedded tables, Coign places 281 of 786 application components on the server.

In a traditional distributed system, the programmer would most likely be forced to 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 changes in the relative computation power of the client or server (through an extension to the graph-cutting algorithm), or changes in network latency and bandwidth.

6. Conclusions and Future Work

We have described the Coign ADPS. Coign is the first ADPS to automatically distribute binary applications built from components. Coign has successfully distributed three commercial-grade applications: Microsoft Picture It!, the Octarine word processor, and the Corporate Benefits Sample from the Microsoft Developer Network (MSDN). The largest of these programs is 1.8 million lines of source code. In terms of complexity, several of the Octarine scenarios we have tested instantiate over 3,800 components. Coign has even proven effective in re-optimizing manually distributed applications.

Coign is part of the Millennium project at Microsoft Research. The project aims to develop distributed systems that provide a new level of abstraction for application programmers and users, managing machines and network connections for the programmer in the same way that operating systems today manage pages of memory and disk sectors. Beyond the capabilities of Coign, Millennium will provide automatic migration of live components. We are attempting to create distributed systems that are self-tuning and self-configuring; automatically adapting to changes in hardware resources and application workload.

Creating a higher level of abstraction does not come without cost in performance. However, Millennium hopes to gain efficiency by spontaneously optimizing applications for the current distributed environment and eliminating unnecessary boundaries between the operating system and programming language runtimes.

Bibliography

[1] Amza, Cristiana, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. In *Computer*, 29(2):18-28, February 1996.

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

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

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

[5] Hunt, Galen C. Automatic Distributed Partitioning of Component Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, July 1998.

[6] International Business Machine, Inc. *VisualAge Generator*. Version 3.0, Raleigh. NC, September 1997.

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

[8] Michel, Janet and Andries van Dam. Experience with Distributed Processing on a Host/Satellite Graphics System. In *Proceedings of the ACM SIGGRAPH*, pp. 190-195. Philadelphia, PA, July 1976.

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

[10] Microsoft Corporation. *Microsoft Office 97.* , Redmond, WA, February 1997.

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

[12] Microsoft Corporation. *Picture It!* Version 2.0, Redmond, WA, September 1997.

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

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

[15] Stets, Robert, Sandhya Dwarkadas, Nikolaos Hardavellas, Galen Hunt, Leonidas Kontothanassis, Srivinasan Parthasarathy, and Michael Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pp. 170-183. Saint Malo, France, October 1997.

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