STATIC OPTIMIZATION OF TRANSPARENTLY DISTRIBUTED NETWORK APPLICATIONS

By

SHAWN OUTMAN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2009
To Melissa
# TABLE OF CONTENTS

| LIST OF FIGURES | ........................................................................ | 8 |
| ABSTRACT | ........................................................................ | 10 |
| CHAPTER | ........................................................................ |
| 1 INTRODUCTION TO PROBLEM DOMAIN | ........................................................................ | 12 |
| Network Applications | ........................................................................ | 12 |
| Thick Client Applications | ........................................................................ | 12 |
| Thin Client Applications | ........................................................................ | 15 |
| Web Applications | ........................................................................ | 17 |
| Rich Internet Applications | ........................................................................ | 21 |
| Goals | ........................................................................ | 21 |
| Host-Independent Execution | ........................................................................ | 22 |
| 2 BACKGROUND RESEARCH | ........................................................................ | 25 |
| Other RIA Technologies | ........................................................................ | 25 |
| Improvements to AJAX | ........................................................................ | 25 |
| AJAX frameworks | ........................................................................ | 25 |
| Comet | ........................................................................ | 26 |
| Multimedia Extensions | ........................................................................ | 27 |
| Flash | ........................................................................ | 27 |
| Microsoft Silverlight | ........................................................................ | 28 |
| Scalable Vector Graphics | ........................................................................ | 29 |
| Java applets | ........................................................................ | 29 |
| JavaFX | ........................................................................ | 30 |
| Alternate Application Frameworks | ........................................................................ | 30 |
| Flex and Adobe Integrated Runtime | ........................................................................ | 30 |
| XULRunner | ........................................................................ | 31 |
| Mobile Code | ........................................................................ | 31 |
| Agent TCL | ........................................................................ | 33 |
| Tycoon | ........................................................................ | 33 |
| Telescript | ........................................................................ | 33 |
| Obliq | ........................................................................ | 34 |
| Visual Obliq | ........................................................................ | 34 |
| Limbo | ........................................................................ | 34 |
| Summary | ........................................................................ | 35 |
| 3 SYSTEM OVERVIEW | ........................................................................ | 36 |
| System Requirements | ........................................................................ | 36 |
| Usage Narrative | ........................................................................ | 37 |
Asynchronous RPC .................................................................69
Proxy Type ........................................................................71
  Futures .......................................................................72
Simplifying proxy types ...................................................74
General Dependence Analysis ............................................75

6 FUNCTIONAL TRANSFORMATION ........................................77

Introduction ....................................................................77
Side-Effect Domains ........................................................79
Dependence Analysis ........................................................81
  Side-Effect Dependencies .............................................82
  Data Dependences ......................................................82
  Control Dependences ..................................................82
  Generalized algorithm ..................................................83
Compiling .......................................................................83
  Task scheduling with side-effect dependences ...............84
  Inter-thread data dependences ....................................84
Functional Transformation ..............................................85
Mutable State ................................................................85
  Eliminating place-holder values ...................................85
  Variable re-definition ..................................................86
  Pass-by-reference .......................................................86
Conditionals ..................................................................87
Loops ..........................................................................87
Subroutine Functions ......................................................88
Related Work ..................................................................89
Conclusion .....................................................................90

7 IMPLEMENTATION OF FUNCTIONAL TRANSFORMATION IN PIP ......96

Client/Server Model as Heterogeneous Distributed Environment ....96
Domain-Specific Tasks as Function Calls ...............................97
  Arguments ...................................................................97
  Asynchronous Calls ......................................................97
Compiler Architecture ......................................................97
  Compiler Front End .......................................................98
  Function definitions .....................................................98
  Pass-by-reference .......................................................100
  Volatile identifier nodes and symbols .............................102
Compiler Middle End .......................................................103
  Scanning the AST to count site-dependent calls ..............103
  Loop outlining ............................................................104
  Constructing the function PDG ......................................106
  Binding PDG nodes to a site .........................................107
  Partitioning the PDG into site-dependent groups ..............110
  Creating functions out of groups ...................................110
Compiler Back End ......................................................................................... 111
Compiling the AST .......................................................................................... 112
Compiling the PDG .......................................................................................... 112
Assembler ......................................................................................................... 113
Conclusion ....................................................................................................... 113

8 APPLICATIONS AND PERFORMANCE BENCHMARKS .......................................... 115
Application ....................................................................................................... 115
  Collaboration through Differencing .............................................................. 115
  Collaboration through Thread Events ......................................................... 115
Performance Testing Methodology ................................................................. 116
  Simulating Latency ...................................................................................... 116
Test Parameters ............................................................................................... 117
Tests and Results ............................................................................................ 118
  Benchmark 1 - Interleaved Independent Client and Server Function Calls .... 118
    Effect of dependence analysis and pre-caching mobile code .................... 118
    Effect of number of transactions .............................................................. 118
  Benchmark 2 – Loop Execution and Asynchronous RPC ......................... 119
  Benchmark 3 – Loops with Different Guard Dependences ....................... 119
Conclusion ....................................................................................................... 120

9 CONCLUSIONS AND FUTURE WORK .................................................................. 125
Conclusions ..................................................................................................... 125
Extension Libraries ......................................................................................... 125
Future Work ..................................................................................................... 126
Performance .................................................................................................... 126
  Loop structures ........................................................................................... 126
  Inlining and SSA form ............................................................................... 127
Code Mobility .................................................................................................. 127
Resource Management .................................................................................... 128
Security ............................................................................................................ 128

APPENDIX
A LANGUAGE GRAMMAR .................................................................................. 133
B EDITOR APPLICATION WITH COLLABORATION BY Differencing ............... 136
C EDITOR APPLICATION WITH COLLABORATION BY THREAD EVENTS .......... 141
LIST OF REFERENCES ....................................................................................... 146
BIOGRAPHICAL SKETCH .................................................................................. 150
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Thin client versus thick client.</td>
<td>13</td>
</tr>
<tr>
<td>3-1</td>
<td>Architecture overview.</td>
<td>37</td>
</tr>
<tr>
<td>3-2</td>
<td>A nested series of transactions composes a dialog.</td>
<td>39</td>
</tr>
<tr>
<td>3-3</td>
<td>A session conceptually encapsulates the entire distributed machine state.</td>
<td>41</td>
</tr>
<tr>
<td>3-4</td>
<td>A Session-ID string refers to a session that encapsulates threads referred to by session-unique Thread-ID.</td>
<td>42</td>
</tr>
<tr>
<td>3-5</td>
<td>Host B refers to Host A’s resources by handle number from Host A’s resource table.</td>
<td>44</td>
</tr>
<tr>
<td>3-6</td>
<td>Events example.</td>
<td>47</td>
</tr>
<tr>
<td>3-7</td>
<td>Extensions are only loaded on the binding site. RPCs are automatically dispatched for remote-bound extensions.</td>
<td>51</td>
</tr>
<tr>
<td>3-8</td>
<td>Registered capabilities are determined by the intersection of developer requested and user granted security descriptors.</td>
<td>53</td>
</tr>
<tr>
<td>3-9</td>
<td>Flow of execution to determine unavailable functionality.</td>
<td>54</td>
</tr>
<tr>
<td>4-1</td>
<td>Offline failure versus partial failure.</td>
<td>58</td>
</tr>
<tr>
<td>5-1</td>
<td>A future spawns a concurrent thread to evaluate the given expression.</td>
<td>72</td>
</tr>
<tr>
<td>5-2</td>
<td>Transactions to evaluate nested RPCs.</td>
<td>73</td>
</tr>
<tr>
<td>6-1</td>
<td>Sample imperative program.</td>
<td>91</td>
</tr>
<tr>
<td>6-2</td>
<td>API functions designated to distinct side-effect domains.</td>
<td>91</td>
</tr>
<tr>
<td>6-3</td>
<td>Multi-threaded version of sample program.</td>
<td>91</td>
</tr>
<tr>
<td>6-4</td>
<td>Dependence graph of statements in sample program (Figure 6-1).</td>
<td>92</td>
</tr>
<tr>
<td>6-5</td>
<td>Dependence graph of AST nodes in sample program.</td>
<td>93</td>
</tr>
<tr>
<td>6-6</td>
<td>Program with if statement and its dependence graph.</td>
<td>94</td>
</tr>
<tr>
<td>6-7</td>
<td>Multi-threaded version of Figure 6-6.</td>
<td>94</td>
</tr>
<tr>
<td>6-8</td>
<td>Program (Figure 6-1) with subroutines.</td>
<td>95</td>
</tr>
</tbody>
</table>
MySQL extension function definitions used by the compiler are located in the comments of the VM’s header file for that extension..........................114

Flags used to mark identifiers found during loop outlining.................................114

Benchmark program 1..........................................................................................121

Effect of dependence analysis and pre-caching of mobile code from benchmark 1 ......121

Performance impact of latency as the number of synchronous requests increase. ..........122

Benchmark program 2..........................................................................................122

Execution times for benchmark 2 with and without asynchronous requests for different numbers of loop iterations..............................................................................123

Closer examination of benchmark 2 with asynchronous requests reveals slightly non-linear behavior .................................................................................................123

Benchmark program 3..........................................................................................124

Execution times of benchmark 3 with and without dependence analysis loop transformation..............................................................................................................124
Abstract of Dissertation Presented to the Graduate School of the University of Florida in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

STATIC OPTIMIZATION OF TRANSPARENTLY DISTRIBUTED NETWORK APPLICATIONS

By

Shawn Outman

August 2009

Chair: Douglas Dankel II
Major: Computer Engineering

With increasing pervasiveness of the Internet, users are becoming accustomed to accessing their applications and data from anywhere. Vendors providing software-as-a-service enjoy diminished piracy and more consistent revenue. However, we are approaching the practical limits of what can be done within the stateless web browser architecture. A new architecture is needed which will allow Internet applications to be written as single cohesive applications, where the physical location of code and data are not a major factor in the design of the application. This will allow Internet applications to be developed rapidly and to simplify maintenance. The details of client/server implementation would no longer interfere with the application architecture. We introduce the Pip system which implements such a transparent system for creating Internet applications.

This work details the limitations of the current Internet application architectures and technologies, and examines other recent technologies which attempt to address some of these limitations. We also look at past distributed application systems, both academic and commercial, to see how innovations might be applied to the domain of Internet applications. We introduce the Pip system and explain specific features that address our goals, and investigate methods to overcome potential pitfalls in implementing transparent locality. We specifically address the
issue of performance of distributed applications and how optimal performance can be achieved while involving the developer as little as possible, to the end that we maintain the local application development model. Finally, we present details of the current implementation, performance tests verifying the effectiveness of our performance optimizations, and directions for future work with respect to performance as well as security and reliability.
CHAPTER 1
INTRODUCTION TO PROBLEM DOMAIN

Network Applications

Networked applications allow users access to the computing resources they require regardless of their physical location. Networked applications can take on many forms; some of these include:

- Time-sharing terminals
- Front-ends to remote databases
- Applications executing remotely, forwarding their interface via X Windows or Metaframe
- Thin clients such as VNC and Remote Desktop
- Applications utilizing RPC (remote procedure call)
- Web applications such as email and shopping

As the Internet becomes increasingly pervasive, users become more accustomed to being able to access their data and applications from any computer. Users can also trust that their data and applications will not be lost due to mishap or malicious software; the role of the application end-user as system administrator diminishes and computers become more consumer-oriented [Greschler and Mangan 2002]. For application providers, allowing users to access computing resources as a service prevents software piracy and promotes consistent revenue generation.

Generally, networked applications fall into two categories: thick client applications, and thin client applications (Figure 1-1). These are described in detail in the following sections.

Thick Client Applications

A “thick” client application is characterized by a remote server containing a data source and having little functionality. The client application is almost entirely responsible for

---

manipulating the data on the server, as well as managing the user interface to the application (Figure 1-1).

![Diagram of thin client versus thick client.](image)

Figure 1-1. Thin client versus thick client. A thick client application will include most of the application logic, whereas a thin client relies on the server to perform application logic.

Most thick client applications are business-oriented, Local-Area Network (LAN) or Virtual Private Network (VPN) database front-end applications, such as those responsible for inventory management or payroll. Development tools typically consist of Rapid Application Development (RAD) Integrated Development Environments (IDEs) with built-in database support such as those provided by Visual Basic or Delphi. The Java Application Windowing Toolkit (AWT) is also a popular platform for thick client applications. Consumer oriented thick client applications are less common, although email client applications such as Microsoft Outlook and Mozilla Thunderbird qualify as thick clients: an email server is directed to perform specific actions such as sending and deleting emails based on instructions from its clients, and many actions, such as updating an address book or the composing of an email may be completely independent of the server.

The primary advantage of a thick client is that it facilitates the creation of a rich, natively implemented user interface. Database access, even though remote, is not any different for the developer than a server-side application, since even then, the database client library still needs to
connect to the database server application, even if they are on the same server. These two primary developer tasks, user interface design and database programming, are often further simplified by the advanced developer tools available.

Thick client applications do have several disadvantages, however, with security as the primary problem. A thick client must be implicitly trusted by the server since the client essentially has unfiltered access to the database (and/or other server resources, such as RPC services), except for limited macroscopic controls such as table permissions.

The second problem with thick clients is that of availability. A thick client must be manually installed on each client machine. If any changes are made to a thick client application, the upgraded version must be re-installed on each machine. This can sometimes be mitigated by adding automatic update capabilities, which require more developer and server resources, or by running off a network share, which requires additional server and network resources.

The third problem with thick clients is that of performance. While a native interface may elevate the user experience, any data or service related activity requires a server response. A long sequence of actions may have relatively simple or linear logic behind it, but will still need to transfer data between the client and server between each step. A database report may result in the transmission of a great deal of information, while the user may be only interested in a small subset. Facilities to narrow the size of reports ahead of time require additional development. The fundamental performance problem is that, even with technologies such as SQL and RPC that allow some of the application logic on the server, the client is dealing with the server through a series of relatively simple requests. The request/response mechanism introduces latency between each step. In addition, the client may require a great deal of data from the server which is used to compute a relatively small result.
**Thin Client Applications**

In contrast to a thick client, a “thin” client application has most of its functionality on the server, alongside the server resources. The user interface is also managed primarily by the server. A thin client typically consists of a generic “dumb” client terminal. The client terminal is “dumb” in the sense that it maintains no information on the state of the application being run and can perform no or extremely limited portions of application logic. The client only receives user input and sends it essentially verbatim to server, while obeying server instructions on how to present the user interface (Figure 1-1).

Thin clients include applications such as VNC and Remote Desktop which forward the entire graphical operating system shell from the server to the remote client, as well as remote text terminals such as rsh and ssh. Protocols such as Metaframe and X allow the user interface of a single application to be forwarded to a remote client. Most online games count as thin client applications because, despite a typically high use of client resources and specialized client applications, the client is basically sending inputs to the server and rendering what the server tells it, while all of the game-state related logic and information is on the server. A web browser, which only sends requests for and renders web pages, is a thin client; however, recent changes have blurred that line by moving additional functionality client-side.

The primary advantage of a thin client application is that of availability. Thin clients are usually generic, so that the same client can be installed on all the client machines once, and the server-side application can be upgraded or new applications deployed without needing to update the client machines. This independence also allows server-side applications to run automatically on as many platforms as for which the thin client technology is available.
Thin clients also lack the potential performance disadvantages of thick client application, as more complex actions can be carried out atomically on the server without needing to communicate with the client.

However, since a thin client protocol is “dumb”, and does not know anything of the application state or logic, every user action effectively results in a request/response transaction that must occur before the user sees the result of their action. In addition, thin client protocols can be inefficient and may result in transmitting insignificant client inputs to the server, as well as redundant user interface updates to the client. For example, the client may send information on mouse movement that is not used by the application, and the server may continually resend instructions to redraw the same dialog box as it is moved around the screen.

Even if the protocol is extremely efficient bandwidth-wise, there is a finite minimum latency period between request and response. While most network latency today is attributable to the limits of switching and routing technology, even a perfect speed-of-light link from New York to Los Angeles would have a round-trip time of approximately 25 milliseconds, which approaches the range of minimum human perception [Bays et al. 2005]. This latency between action and reaction interferes with the flow of the application and lowers the user experience.

Thin clients may use some techniques to hide this latency for idempotent actions; for example, a web browser may cache GET requests, short-circuiting the interaction with the server to retrieve cached static documents. A text terminal may run in half-duplex (not echoing input) so that the client can immediately output text as typed. A graphical terminal may hide the server mouse cursor position and instead show the client host's mouse cursor, even though the server's cursor position may lag behind and be infrequently updated. Similarly, an online game may allow the player to move his avatar around freely and smoothly, and only occasionally update the
server – the server and other clients may interpolate and predict smoother animation by using splines. However, these are specialized techniques that must be developed into the client and are not applicable to non-idempotent actions that the user may take.

**Web Applications**

The first consumer-oriented Internet applications used WWW browsers to affect the user interface. This technique had two primary shortcomings that negatively impacted the user experience versus a standalone application.

First, WWW uses the Hyper-Text Markup Language (HTML) to describe individual web pages. HTML is designed to label different parts of the document with certain attributes, for example, a section of text may require emphasis (<em>) or be computer code (<code>). Elements also exist to create tables, input fields, and links to other HTML documents. However, HTML is a document markup-language, not a document layout language (such as Postscript). How the document is actually rendered is up to factors that the developer cannot control, such as the browser's interpretation of the HTML and the user's individual preferences.

The second major problem with using the web browser as a thin client or application platform is the synchronous request/response mechanism that must be employed every time the user takes an action inside the application. A single web page presents the interface to the user. The user performs actions by selecting links or submitting forms to the server. The server is entirely responsible for maintaining the application state. The document-oriented browser interprets any interaction with the server as a request for a new document. Therefore, for the user to affect the application state, the browser must send a request to the server for a new document to display. The user must wait for the updated user interface to be received by the client browser before taking any additional actions.
Web applications, due to the stateless request/response mechanism, force the developer to transform what would otherwise be straightforward imperative logic for user input and output into a continuation passing style\(^2\) [Wand 1980]. To preserve state, the developer must choose data to be stored in server-side sessions or passed through hidden HTML inputs. Neither is completely appropriate for all data. Application state data stored in the session is not affected by the user opening multiple windows, which then splits the application state, or using the navigation features of the browser. However, passing data through hidden HTML inputs can be relatively unwieldy compared to session storage, and impractical for large data or secure information, such as passwords in plain text or the cost of an item to be billed that a malicious user might modify before sending back. Two additions to web browser technologies have been employed to partially resolve these problems.

First, Cascading Style Sheets (CSS)\(^3\) is a markup that describes exact attributes of how a given HTML element is rendered. Second, ECMAscript\(^4\) (generally referred to by the name of one of its variants, JavaScript) provides the capability for the browser to maintain and modify the application state and for the HTML Document Object Model (DOM) to be modified from within the document itself. JavaScript also provides a class, XMLHttpRequest, which allows asynchronous requests to the server from within the document. This combination of technologies is known as Asynchronous JavaScript and XML, or “AJAX”\(^5\). AJAX enables the developer to


\(^3\) Lie, H. K., and Bos, B. 1999. Cascading Style Sheets, level 1. W3C Recommendation. \texttt{http://www.w3.org/TR/REC-CSS1} (June 2009)


create a much improved user experience: JavaScript allows some client-side functionality, and asynchronous requests can improve or eliminate some of the delay of synchronous requests while eliminating the need for the entire document to be reloaded after every user action.

AJAX still has limitations, however. Currently, no web browser is completely standards compliant, and each browser has its own limitations and features. As such, each browser must be targeted individually as a platform by the developer. Further, the browser itself becomes a vestigial frame for the application, since the fundamental features of the browser -- the address bar, bookmarking, and forward/back buttons -- become meaningless; use of forward/back navigation features may even interfere with the application.

AJAX is a “sandboxed” environment. This can be helpful in ensuring security and platform independence. However, a sandbox is no guarantee of either. There have been several security holes in AJAX browsers [Lam et al. 2006; Livshits and Erlingsson 2007], and cross-site scripting (XSS) vulnerabilities [Shanmugam and Ponnavaikko 2007] are a persistent venue of attack. As already discussed, the platform dependence of an AJAX application is merely transferred from the operating system and machine architecture to the browser, as the browser becomes the platform.

In a more general sense, the downsides of a sandboxed environment are performance and access to native host resources. Usually, performance is touted as the main disadvantage; however, due to Moore's law and a merely linear penalty from virtual machine or interpreted execution, sandboxed applications are usually just slightly behind the curve of constantly improving native performance.

However, limited access to other native resources negatively impacts the user experience. The use of non-native GUI components can create an inconsistent and confusing user interface.
Even with CSS, the user interface capabilities of the browser are still very limited and inconsistent on different platforms and browsers. There is simply no access to multimedia capabilities (audio, video, accelerated or 3D graphics). Input devices are limited to those supported by the browser: mouse and keyboard. Local files must be sent through the server in their entirety to reach the AJAX application; similarly, files must be saved via a single download from the server. This potentially raises privacy and security concerns since any document the client uses must go through the server. Also, client-side random-access files and databases are impossible.

This dependence on the server and the limited functionality of the AJAX browser also make it nearly impossible for all but the simplest of AJAX applications to run offline, without access to the server.

Fundamentally, even an AJAX application is still dependent on the client to server request/response mechanism, since the XMLHttpRequest is still precisely that, an HTTP request. Therefore, an application can be notified asynchronously of events occurring on the server only by frequent polling requests. This is an inefficient use of server and network resources and may still yield unacceptable delays between an event occurring and the client notification.

The final drawback to AJAX applications is that the application must be developed in two components, a JavaScript/HTML client and a server written in another language. However, these two components are inextricably linked and intermingled, with the server application and the client JavaScript both generating HTML and modifying the DOM, and in some cases the server application itself generates the JavaScript code with which it later communicates. Managing these complex interactions and interdependencies falls on the application developers and maintainers on top of the application requirements.
Rich Internet Applications

The term “Rich Internet Application” (RIA) originally described a narrow set of technologies that included offline execution, component based design, and multimedia capabilities\textsuperscript{6}. Today, the term is used more loosely to describe a range of applications that may have only one or two of the original requirements.

The most common use today, and for the definition for the context of this research, a “Rich Internet Application” (RIA) is a networked application, intended to be openly accessible over the Internet, that has some state and application logic client-side to improve the user experience over a standard web application. Conversely, since the RIA is an openly accessible Internet application, for security and performance, much of the logic and application state must be confined to the server. The goal of an RIA is to maximize the user experience by creating the illusion that the Internet application is executing locally, by hiding server requests and utilizing local resources as much as possible.

Goals

To the effect that an RIA creates the illusion to the user that the application is a cohesive, client-side application, the RIA development tools should strive to create the same illusion for the RIA developer.

I propose the development of a platform for creating and deploying Rich Internet Applications that will meet the following goals:

- Simplify the software engineering process by allowing the creation of a single application in one language.
- Simplify the software engineering process by making the locality of the code and data as transparent to the developer as possible. In particular, the developer should be removed

from the task of engineering communications mechanisms between the client and server. The developed application architecture should also not reflect nor be based on a client/server methodology.

- Provide built-in facilities to support applications running offline or with intermittent connections. These facilities should be as transparent as possible to both the developer and consequently the user.

- Improve the user experience by allowing the developer to easily access native functionality of the client. The framework should also be easily extensible to support additional native functionality.

- Provide for asynchronous server to client communications.

- Provide simple facilities to the developer for creating multiple threads of execution to minimize unavoidable fundamental request/response delays. Even in non-networked applications, multiple threads are useful for long operations such as writing files or printing documents.

- Applications will not be explicitly installed or upgraded on the client hosts. This goes along with transparent code locality; the developer will not know or care where the code is located, and by deduction, neither will the user. The system will automatically download and upgrade code as needed.

- The system should not be inherently insecure and should provide facilities to promote development of secure applications.

The platform will be primary intended for applications rather than as a method to present interactive documents. There are many technologies sufficient for presenting interactive content. Instead, the system will provide a simple and unobtrusive foundation for creating applications. The focus is to address the needs of applications, such as threading and extensible native libraries, which current RIA technologies do not sufficiently address.

**Host-Independent Execution**

The primary mechanism through which the simultaneous resolution of these goals will be possible is host-independent execution and thread migration.

The developer will create a single cohesive application that is compiled into a virtual machine (VM) bytecode format. While the virtual machine is running, a thread's execution state,
along with the relevant code if needed, can be transmitted from the server to client, or vice versa. This allows the thread of execution to go to the resources it needs, rather than the developer needing to explicitly invent mechanisms to retrieve data from or modify the state of an application on a remote host.

Since threading facilities are provided, resources can be manipulated or retrieved asynchronously from other tasks or the user interface. Similarly, a thread executing server-side can wait on server-side resources, then transfer client-side or signal a client-side thread when ready to provide asynchronous client to server notifications and communications.

When the execution state is transferred, it is possible for the bytecode necessary to continue execution to be transferred with it. It would not be necessary to download the entire code of the application ahead of time and, in fact, it would probably not be necessary (nor secure) to transfer the entire application code at all. The client would keep track of and cache code segments in much the same way a web browser caches static documents: the client would maintain server time-stamps and verify before executing that the client's code cache was up-to-date with the server. This removes the need to install and upgrade applications and provides for on-demand, just-in-time code migration.

Host-Independent Execution facilitates migration, allowing threads of execution to travel to the resources they require. This allows the application to operate on a higher level, where it does not matter to the application where resources are located. This allows the developer to focus on creating the application itself, and use an architecture appropriate to that of the specific application, rather than force a client/server architecture with specialized requests for specific data. This simplification will result in creating RIAs in less time with simpler code and, therefore, less maintenance. Further, hiding the client/server interactions from the developer will
automatically result in the interactions being hidden from the user, improving the user experience. Thin versus thick client performance issues will be minimized by automatically running code using primarily server resources on the server and code using primarily client resources on the client. Finally, by moving the client/server architecture to a lower level, enforcing security policies will become more automated and fewer security issues will be present in developed applications.

In the next chapter we examine newer RIA technologies and look at Mobile Code Languages from which we may be able to derive some techniques that may be useful in the RIA domain.
CHAPTER 2
BACKGROUND RESEARCH

Other RIA Technologies

As discussed in Chapter 1, the AJAX-enabled web browser is currently the most popular platform for developing Rich Internet Applications (RIAs). However, there are many new technologies for developing RIAs.

These newer technologies vary in the aspects of the Rich Internet Application on which they seek to improve. Generally, newer technologies can be categorized by those extending the AJAX/browser-based architecture, and those providing an alternative to the browser-based model of application development.

Improvements to AJAX

Several technologies build upon AJAX to simplify application development, help ensure cross-browser compatibility, and add facilities for offline functionality and asynchronous server to client messaging.

AJAX frameworks

An AJAX framework provides a suite of pre-written JavaScript cross-browser components. The framework is either the basis of the server-side application that is responsible for dynamically generating the client-side JavaScript, or the framework generates an independent client-side application. It has even been suggested that JavaScript serves as an “assembly language” of the web [Puder 2007], and that a developer’s exposure to it should be minimized.

Google Web Toolkit and Gears. Google Web Toolkit\(^1\) client-side applications are written in Java and then compiled into client-side AJAX. Server-side functionality follows the RPC

---

model. Google Gears\(^2\) provides facilities for allowing AJAX applications to operate offline. Gears consists of a back-end that must be installed onto the client host. The back-end consists of an SQLite database server and another server process with which the client application interacts with when it cannot communicate with the remote server.

**Dojo Toolkit.** The Dojo Toolkit\(^3\) provides pre-built cross-browser AJAX components, and a library of improved cross-browser JavaScript functionality. This includes methods for making asynchronous requests and an event system that can trigger on DOM updates and JavaScript method calls.

The Dojo Toolkit provides some limited capabilities for client-side storage. A Flash applet may be employed to store data using Flash's Local Shared Objects, along with cookies and Firefox's persistent storage mechanism.

**OpenLaszlo.** OpenLaszlo\(^4\) is a compiler which accepts code written in LZX, which is essentially JavaScript code wrapped in XML. OpenLaszlo compiles the code into the client application in both a Flash binary format and a DHTML version. When the application is initialized, it will choose the most appropriate version to load.

**Comet**

Comet\(^5\) is a technique for allowing AJAX client-side applications to receive asynchronous messages from the server. The Comet technique involves a connection being made by the client to the web server. The client typically makes a request and the server responds; however, the


server does not terminate the connection and may send additional responses encapsulated as part of the continuing original response.

Several models of asynchronous communication may be used. Typically, a “Subscribe/Publish” architecture is used where the client indicates interest in particular classes of events. When an event occurs, the server sends notification to all subscribed clients.

For AJAX applications, XML may be used to describe the events, as XML is commonly used for asynchronous RPC. However, JSON (JavaScript Object Notation) is a simpler alternative in that the browser's JavaScript parser can automatically (though somewhat insecurely) evaluate the response.

Depending on the application, use of Comet may require extensions to the web server daemon already in use or may require a specialized Comet daemon to be run in addition to the web server.

**Multimedia Extensions**

One of the major focuses on improving the user experience with Internet applications has been moving beyond the user interface constraints of the browser. Technologies such as Flash were created as an extension of the web browser, providing animated graphics and sound capabilities that the browser and HTML lack.

**Flash**

Flash\textsuperscript{6} was created in 1996 as a method to easily create and distribute vector-based graphics and animations via websites. As opposed to raster graphics, vector graphics scale

---


\textsuperscript{7} Waldron, R. 2002. The Flash History. \textit{Flashmagazine}. \texttt{http://www.flashmagazine.com/413.htm} (June 2009)
seamlessly to the dimensions of the browser window and often consume less storage space, which translates into decreased required bandwidth and therefore faster load times.

Since then, Flash has added ActionScript, a variant of ECMAScript, to allow application logic to be integrated into the Flash files. The term “Rich Internet Application” was created in 2002\(^8\) to define the type of applications that the then-upcoming Flash MX was targeting as a development platform.

Flash now provides facilities for persistent offline storage, access to web services and RPC through XML, and streaming audio and video. Flash is a native extension to the browser, and it is available not only for Windows, OS X, Linux, and Solaris, but also some mobile platforms such as Windows Mobile and Symbian.

**Microsoft Silverlight**

Silverlight\(^9\) is Microsoft's response to Flash. The fundamental concepts are the same; Silverlight is a native browser extension (for Windows, OS X, and an independently developed Linux implementation) that adds multimedia capabilities through a subset of the Windows Presentation Foundation (WPF).

WPF describes an application's user interface in XAML (Extensible Application Markup Language). Silverlight is the browser extension that renders and executes the XAML. Silverlight also supports functionality written in JScript, Microsoft’s implementation of ECMAScript. Future versions of Silverlight will include support for the Common Language Runtime (CLR) so that .NET languages such as C# can be used to create Silverlight applications.

---


Scalable Vector Graphics

Scalable Vector Graphics (SVG)\textsuperscript{10} is another Flash-alternative. SVG provides advanced vector graphics and animation capabilities, described in XML. It also allows scripting in ECMAScript or SMIL (Synchronized Multimedia Integration Language)\textsuperscript{11}. Native browser support is not always present and usually incomplete. Adobe has discontinued their SVG browser extension since acquiring Flash.

Java applets

Java applets\textsuperscript{12} are a particular type of Java application that runs inside the browser, with the Java Runtime Environment (JRE) as a browser extension.

Java applets may do anything that a normal application would, including create windows outside the browser, access local files, and make network connections. However, these capabilities to access native resources are usually constrained unless specific permissions are granted.

Java applets enjoy the relatively distinct advantage that they are not restricted to operating above the HTTP protocol layer for their server interaction – applets may make unrestricted TCP connections to their hosting domain\textsuperscript{13}. In addition, they may be integrated with a servlet or interacting with a Java RMI server which simplifies tasks for the developer: for example, Java


\textsuperscript{13} 2007. Practical Considerations When Writing Applets. \textit{The Java Tutorials}. Sun Microsystems, Inc. \url{http://java.sun.com/docs/books/tutorial/deployment/applet/practicalindex.html} (June 2009)
objects may be serialized and reconstituted automatically. The code for a class must only be written once since the same language is used both client- and server-side.

JavaFX

JavaFX\(^\text{14}\) is a new technology that Sun is introducing. JavaFX is a declarative language designed to create application user interfaces, along the lines of XAML, MXML, and SVG. Running on the Java platform, JavaFX can be used with Java applets and applications, or be used by itself to create RIAs much the same as Flash and Silverlight.

Alternate Application Frameworks

The previous sections described technologies which seek to extend the functionality of the web browser to improve its usefulness as a platform for Rich Internet Applications. This section examines some technologies that dispense with the browser model.

Flex and Adobe Integrated Runtime

Flex\(^\text{15}\) is Adobe's evolution of Flash into a more proper application development platform. Since Flash was initially developed as an artistic tool to create website graphics, the development process was not appropriate for application development. Flex includes an IDE and RAD GUI designer. MXML is the XML-based declarative language used to describe Flex user interfaces, replacing the proprietary binary format used by Flash.

The Adobe Integrated Runtime (AIR)\(^\text{16}\) is a client-side runtime environment that hosts applications developed with Flash, HTML, and JavaScript. The AIR includes such native functionality as file access, GUI components, and clipboard/drag and drop. As well, an SQLite


A database system is included for persistent storage much like Google Gears. The AIR is only available for Windows, OS X, and Linux.

**XULRunner**

XML User Interface Language (XUL)\(^{17}\) is another declarative language designed to create user interfaces for applications. Currently, XUL is used by Mozilla applications such as Firefox and Thunderbird to implement their user interfaces. XUL works with JavaScript to create application functionality. The XULRunner is a runtime engine designed to allow XUL to be executed independently of a browser.

**Mobile Code**

Any RIA platform needs a mechanism to transport the client-side portion of the logic to the client host. Carzaniga et al. [1997] describe a taxonomy of code mobility that allows categorization of technologies based on how they transport the code and execution state of the application. Essentially, a *weakly mobile system* is capable of executing code on a remote site, but a *strongly mobile* system is one in which the thread of execution itself is mobile. Application mobility is necessary to access resources on a remote site. From the developer's perspective, the types of mobility fall under four paradigms:

- A static web document or RPC request are examples of a *client-server* paradigm, where the server contains both the application code and resources.
- All of the above described RIA technologies fall under the *code-on-demand* paradigm, where the client site requests from a remote site the code to affect the user interface.
- *Remote evaluation* involves requesting that a remote site execute provided code.
- A *mobile agent* migrates its own code and state to the resources it requires.

As opposed to current RIA technologies that are all weakly mobile, a strongly mobile architecture would allow a developer to create a more cohesive application architecture not reliant on a client/server paradigm and explicit remote evaluation of the user interface.

Techniques for remote evaluation are typically used to implement explicit parallelism such as in the Parallel Virtual Machine (PVM) [Beguelin et al. 1991] framework. While RIAs may be considered a type of grid computing in that the task of executing the application may be divided over several heterogeneous sites, a user-oriented application cannot generally make use of the kind of parallelism provided since a user can only focus on a one task at a time: an application generally spends most of its time waiting for user input, while the rest of the time performing a single task to satisfy a user request. Effectively, a user-oriented application is single threaded.

We are more interested in the application developer's point of view. To simplify the developer's task, this effective (if not actual) single thread will run through each task a user may request and then return where it started: waiting for user input. A mobile agent paradigm would allow the developer to think in terms of the application thread moving from the client site to the server site as needed to access server resources, then finally arriving back at the client site to present the results and await further input. Therefore, one cohesive application can be written instead of dividing the application into separate client and server components.

Several languages have been created for use in distributed systems of various types to implement mobile agent paradigms. For the application developer, a mobile agent paradigm allows the developer to write a cohesive application that can migrate to resources as it needs them. Systems such as Agent TCL [Gray 1996; Gray et al. 1996], Tycoon [Matthes and Schmidt 1992], and Telescript [Thorn 1997; Carzaniga et al. 1997] have been developed to facilitate
creation of strongly mobile applications. Examining this prior work may give some insight into desirable features or problems that may be encountered developing a new system.

Agent TCL

Agent TCL is a modification to the Tool Command Language (TCL) that adds a `jump` command which causes the executing thread to migrate to a remote execution engine. Agents may pass messages or establish direct connections to one another. Tk can be used to create a GUI. Timeout and retry mechanisms are also added to TCL to allow the developer to compensate for partial failures.

Tycoon

The Tycoon (Typed COmmunicating Objects in Open eNvironments) Language (TL) was designed to assist with database systems and applications programming. TL is a functional language that compiles to an intermediate bytecode which is itself an intrinsic type. It is, therefore, possible for closures to be created that can then be transmitted to a remote host for execution. Tycoon is particularly notable for its simple integration of and dynamic linking to native extensions [Thorn 1997; Mathiske et al. 1997].

Telescript

Telescript was developed for a PDA created by the General Magic corporation. With Telescript, the cell service provider also provided a remote Telescript execution engine on their servers. A Telescript application running on a user's device could explicitly execute a `go` command that would migrate the execution of the process to the provider's servers. There, the application could perform operations such as accessing persistent storage, searching databases, accessing the Internet, or communicating with other Telescript users. The application migrates back to the device when complete. The user would not necessarily need to remain connected to the network during remote execution.
Obliq

Obliq [Thorn 1997] is an object-oriented, multi-threaded, dynamically typed language for distributed application development, built on top of Modula 3. Primitive values – including closures – in Obliq can be transmitted to remote execution engines. Objects must be explicitly migrated by first serializing the object, ensuring that no other threads are executing in the object.

Visual Obliq

Visual Obliq [Bharat and Brown 1994] extends Obliq by adding an event driven RAD (Rapid Application Development) GUI builder, similar to those provided by Flex and Silverlight. The applications built with Visual Obliq provide a callback-based framework for responding to UI events. Visual Obliq emphasizes the concept of “transparent location,” where the underlying distributed computing mechanisms are hidden from the developer. However, Visual Obliq is intended to be used to build groupware with homogeneous distributed clients. That is, the UI and functionality is to be duplicated on many hosts, rather than a heterogeneous client/server model. The transparency mechanism relies on accessing other client's UI “forms” through an indexed array, with each index corresponding to a client. The transparency breaks down when complex non-UI data needs to be transmitted between hosts, since objects must then be explicitly serialized and transmitted. Visual Obliq also does not sufficiently address the concerns raised in Chapter 4.

Limbo

Limbo [Thorn 1997] is the programming language used with Lucent's network operating system Inferno. Limbo is an imperative language which is compiled into a platform-independent virtual machine bytecode called Dis. Limbo modules are cryptographically signed by their providers and explicitly loaded at runtime. Inferno services are provided to Limbo applications via a uniform filesystem interface. Services available to an application appear within the
application's filesystem namespace. The concept of typed “channels” facilitates interprocess communications. Memory is automatically managed in Inferno via reference counting.

Summary

In this chapter, I examined several RIA technologies and distributed computing technologies which may have aspects applicable to the RIA domain. Current RIA technologies tend to focus on client-side presentation issues and present the developer with a heterogeneous distributed environment with layers of complexity. To simplify the single task of application development, it is necessary to construct a simple, homogenous distributed environment by providing unobtrusive distributed computing facilities. The next chapter describes such a system.
CHAPTER 3
SYSTEM OVERVIEW

To meet the goals described in Chapter 1, we propose the development of the Pip system.

Pip is a framework for developing client/server distributed applications as a single cohesive application.

System Requirements

The Pip system is designed to meet the following requirements:

- A network application, using resources on two heterogeneous hosts ("server" and "client"), must be able to be written as a single cohesive program.
- Code must be mobile on-demand, allowing clients to start an application without installation.
- Code must be host-independent, capable of executing on both the client and server.
- It must be possible to indicate code that must be executed on a particular host, for security or resource utilization.
- Callee functions that must be run remotely should operate like RPC.
- Asynchronous server to client communication must be possible.
- The host environment must be capable of migrating threads as they are running.
- It must be possible to easily extend the framework to provide native functionality.
- The host environment must support threading in the hosted application to not hinder responsiveness during synchronous requests. This will help the developer mask unavoidable latency in certain situations.

To help meet these goals, Pip includes the following components (Figure 3-1):

- A simple dynamically typed imperative language which is a derivative of PHP.
- A bytecode compiler for the Pip language.
- A virtual machine host environment which runs on both the server and client.
Figure 3-1. Architecture overview

Usage Narrative

This section provides a general use-case scenario for Pip.

Developing the Application

The application developer writes the application in the Pip language. He makes use of a native cross-platform UI extension on the client, and a database extension on the server.

Deploying the Application

When the application is complete, the application provider creates a Pip daemon process on the server pvm.example.com. The Pip daemon is compiled with just the required server database extension. The provider adds the compiled virtual machine object files and any other binary resources into a path so that the daemon can find them.

The provider then creates a web site for his application. The web site contains an HTML link to launch the application:

pvm://pvm.example.com/application

Launching the Application

A user with the Pip host installed visits the application provider's website. He decides to launch the application, and clicks on the link. The client-side engine, which is a registered
protocol handler in the browser, contacts the daemon at pvm.example.com and instructs it to begin executing a new session of application.

Application in Use

The server runs the program until it encounters a function that is part of the client-side UI library. Then, it makes a request to the client to call this function. The client looks for and then executes the function and returns the result. The application may make several client-side calls in a row. At this point, the server engine decides to migrate the thread to the client to avoid the delay from latency. The server makes a request to execute the code on the client and the client responds affirmatively. However, the client immediately finds that it does not have the code for the current function. It makes a request back to the server for the function code, which the server fulfills. The client stores this code on disk along with the server-provided time-stamp so that it does not need to request it the next time the application is run.

When the user prompts the application to save his work, the client creates a new thread which makes a request to the server to execute the server-side function to save the work in persistent storage. The main application thread remains running in the UI library's input loop, ensuring the responsiveness of the application during this possibly lengthy operation. The application uses events to re-join the save thread with the main thread. The input loop is hooked into the VM to notify it of events.

The user chooses to exit the application. The client engine sends a request to the server to terminate the session, and then exits.

Upgrading the Application

As with web applications, Pip applications may be upgraded solely on the server. The modified code is compiled and the old object binary files are replaced with the new.
Applications that are already running will have already loaded the modules into memory and will not be affected by the upgrade unless the system administrator specifically resets them. New sessions will use the new files.

When a new session is begun, a client storing cached code from previous sessions will query the server to ensure that the client cache is up to date. If it is not, the client will purge the cache, ensuring that its copy of the code is synchronized with the server’s copy and is up to date.

**System Architecture**

The underlying Pip architecture is client/server based on the concept of transaction dialogs similar to the Session Initiation Protocol (SIP).

![Diagram](image)

Figure 3-2. A nested series of transactions composes a dialog.

One host makes a request to which the other host must respond. A transaction consists of a request and its corresponding final response. The responding host may delay the final response to make its own request. A nested series of transactions composes a dialog (Figure 3-2).
PVM Protocol

The Pip Virtual Machine (PVM) protocol describes how the request and response messages are composed. PVM is loosely based on RFC 3261 (SIP)\(^1\) which is based on RFC 2616 (HTTP)\(^2\). However, as SIP is not an extension of HTTP, PVM is not an extension of SIP.

PVM is a text-based, human-readable protocol. PVM can be encapsulated into TCP or UDP messages, although the prototype implementation does not support UDP and the necessary retry and acknowledgment mechanism from SIP.

A sample request to perform a remote procedure call might look like:

```
RPC example_function PVM/1.0
Session-ID: 12345678
Thread-ID: 2
CSeq: 123 RPC
Content-Type: construct/xml
Content-Length: 456
<< XML Argument Data >>
```

The corresponding final response:

```
PVM/1.0 200 OK
Session-ID: 12345678
Thread-ID: 2
CSeq: 123 RPC
Content-Type: construct/xml
Content-Length: 789
<< XML Return Value Data >>
```

Responses may also be provisional, as opposed to final, and do not end the transaction:

```
PVM/1.0 100 RUNNING
Session-ID: 12345678
Thread-ID: 2
CSeq: 123 RPC
```

Upon receiving a provisional response, the requesting host will continue to wait for a final response, but it will be advised of the progress of its request's fulfillment, as provisional

---


responses act as an acknowledgment. Provisional responses are indicated by a response status code between 100 and 199 inclusive. In the example above, the responding host would first send a provisional response before beginning execution of the procedure – in fact, it is necessary to send a provisional response before the responding host can introduce a new transaction into the dialog.

**Sessions**

A session encompasses the state of the application and the distributed virtual machine on which it is running (Figure 3-3). Sessions are essentially the same concept as from web application programming.

![Session Diagram](image)

Figure 3-3. A session conceptually encapsulates the entire distributed machine state.

When an application is started and the first request is sent to the server, the server constructs a Session-ID string and attaches it to the response. All messages from then on will have the Session-ID attached. If the TCP connection is lost and re-established, or a connectionless protocol such as UDP is used, then the Session-ID will allow the server, which may be maintaining several sessions simultaneously, to associate the message with the proper session.
Threads

A single session may include several concurrent threads of execution (Figure 3-4). These are identified in messages by a session-unique Thread-ID string. Thread creation must take place on the client, so unlike the Session-ID string, the Thread-ID string is assigned by the client. In a TCP implementation, each thread corresponds to at most one connection. All messages relating to a particular thread occur within that connection.

Figure 3-4. A Session-ID string refers to a session that encapsulates threads referred to by session-unique Thread-ID.

Messages are passed synchronously between hosts. It is not possible to interrupt thread execution with an asynchronous message. This means that the host that originally made a request must block the execution of the thread until a response is received. Therefore, within a dialog, the responding host is considered to be in control of the thread execution, even if that host does not actually proceed to affect any execution (e.g., it may be responding to a request for code). This seeming limitation does not mitigate the developer's ability to affect asynchronous communication: similar to local application development, the developer may create multiple threads and use the built-in events mechanism.
Storage

Local variables

The virtual machine allocates thread storage in a stackframe based on the space needs of the current function scope. Even though the language is dynamically typed, first order storage is allocated before a function begins execution. Higher order storage such as individual records in an array type are allocated as needed.

For communication between hosts, Pip encodes data types into an XML-based format. XML allows structured data such as arrays and classes to be easily encoded due to its recursive nature.

Pip was designed with bandwidth use as only a minor concern. The assumption is that the real concern in distributed systems is latency, which is a hard physical limitation, while available bandwidth will continue to expand along the lines of Moore's law for the foreseeable future.

However, from a pragmatic viewpoint, bandwidth use cannot be ignored. For some applications, such as cell phones for which bandwidth may be well behind the curve of performance and may imply additional costs (and may have a hard limit from limited spectrum), it is critical to provide facilities to minimize bandwidth usage.

The first way this can be done is by using compression beneath the application layer, or a compressed encoding for serialized data.

Higher level methods to conserve bandwidth when migrating a thread include allowing the migrating to choose to omit portions of the stackframe. These omissions may include large variables in the current scope, or the stackframe may be truncated below a certain point. Omitted single variables can be requested as needed; for a truncated stackframe, execution will migrate back when the bottom of the stackframe is encountered.
It is possible for a host to take a snapshot-based difference of a large variable and only transmit changes to a structure, or pipeline changes by sending provisional responses with updates to variables as they take place. These techniques will be investigated as part of this research.

**Resource handles**

Functions that modify large data structures need a pass-by-reference mechanism that will work across hosts without requiring the entire structure be transmitted. Similarly, native functions returning pointers or other native resources need a mechanism to encapsulate these resources without sacrificing security.

Figure 3-5. Host B refers to Host A’s resources by handle number from Host A’s resource table.

Each host maintains a table of resources (Figure 3-5). Each resource record contains a raw pointer to the resource or internal representation of a referenced data structure, and a unique resource handle number to identify the resource. The handle number is used in place of the raw pointer in communications between hosts.
The Pip Language

Pip is a derivative of PHP. The syntax is essentially the same although it is current lacking class support and many of the built-in functions. However, it is straightforward to implement these features. While similar to PHP, Pip modifies and extends the PHP syntax.

Removal from Web Page Context

Since Pip is not intended to be used to generate HTML output, the HTML embedding features are removed. <? and ?> tokens, indicating to start and stop parsing as PHP are absent. A Pip program will not begin and end with these tokens as a PHP program would.

Site Bindings

Pip also includes features for indicating whether certain functions are “bound” to the client or server site.

A function can be declared normally:

```php
function example()
```

Modifiers can optionally be used in the declaration:

```php
server function example()
client function example()
auto function example()
```

The `auto` keyword means that the function is not bound to the client or server and the VM can decide the appropriate execution site. Absence of a binding keyword implies `auto`.

The import Keyword

As a Pip program is not parsed at runtime, the language omits the PHP `include` and `require` statements. In their place is an `import` keyword that is similar to Java's package `import`. At the top of a source file, function libraries that are needed are declared via `import` statements:
Note that the import statement includes an identifier for a host site binding. If this is `client` or `server`, the library is only loaded by the VM on the specified site. This may be done for security, performance, or because resources used by a library are specific to a site.

Imported libraries can consist of other Pip object files or natively implemented extensions that are linked to the VM host. A program may import a library bound to a specific host even if that library is not available on the other host.

**Threading Capabilities**

PHP is intended to be used in the one-off, continuation-based Web application model, where a program is run to service a specific request, and then terminates. Multi-threading is generally unnecessary and can be implemented only by spawning additional PHP interpreter processes or by the requesting client making simultaneous requests.

In dispensing with the Web continuation model and in the interest of creating a single, cohesive application, more advanced tools for multi-threading may be necessary. In most application development instances, the developer will want a simple way to “farm off” to another thread a task that he knows may take a long time to maintain responsiveness of the application UI. As opposed to local applications, a distributed application may have many more instances of resource use that take a noticeable period of time to complete due to unavoidable communication latency. However, the practice of limited use of multi-threading to complete specific tasks while retaining responsiveness is common even in local application development. Therefore, it may be desirable to introduce methods of asynchronous operation and communication.
Events

Events offer the developer another option to parallelize synchronous operations. Events are similar to messages in the Actor model [Agha 1986; Hewitt 1977]: Threads only have the “address” of threads which they themselves create; messages (events) can only be sent to actors (threads) for which the sending thread has an address. Addresses can be communicated in messages between actors, however.

Figure 3-6. Events example. (1) Thread 1 creates Thread 2. (2) Thread 1 communicates the reference of Thread 2 to Thread 3. (3) Events issued by Thread 2 may be received by Threads 1 and 2.

Instead of explicitly sending messages, however, an event is analogous to a message broadcast. Recipient threads must be subscribed to an event to receive it.

A thread handle is an intrinsic type. New threads can be easily created:

$t = thread(ExampleFunction());

ExampleFunction will now run in the thread referenced by $t. Threads can broadcast events:

event("event name", $data);

$data can be of any intrinsic type, including arrays. A listening thread must register:

$r = event_register($t, "event name", OnEventFunction);

It's also possible to declare an event handler to automatically execute whenever any child thread issues a specific event.

function OnEventFunction($data) on "event name"
This could help reduce the scatter of references depending on the application implementation.

To receive an event, the subscribing thread must be in an idle state or must explicitly poll or wait on (block for) pending events. This is because of the synchronous transaction-based nature of the VM threads as discussed previously. It also simplifies the problem of avoiding deadlocks and race conditions since a thread will be in a known base state when executing an event handler.

Threads can unregister functions from events with the handle that is returned by the initial registration:

```php
event_unregister($r);
```

Threads can also reference their own thread handle with the $thread identifier. This can be used to communicate to child threads so they can receive events from the parent, or a thread can register for its own events.

Events complement futures: Futures block synchronously while events may occur asynchronously during a thread's execution. This makes each useful in somewhat different situations.

**Interleaved site-dependent expressions**

Consider the following code to populate a client-side GUI component with rows from a server-side database query result set:

```php
while ($x = database_fetch_row())
    gui_populate_list($x);
```

Simply executing this code linearly, as-is would create a delay which grows linearly with the number of rows in the result set due to latency.
When it is known that the host-dependent expressions – in this case, `database_fetch_row` and `gui_populate_list` – only have side effects on their respective sites and are only dependent on each other, several optimizations are possible.

The developer could rewrite this code to cache all the database rows in one loop and then store them into the GUI component in a second loop. This requires the most work from the developer, and requiring the developer to implement solutions as such trends too close to defeating the purpose of the system as a clean way to develop distributed client/server applications as local applications. Another developer, unfamiliar with the underlying problem being solved, examining code written in such a way would not be provided any contextual information about the purpose of that particular structuring of the code. An additional negative of this technique is that it may not work well on the general problem if the amount of data being cached grew to be very large.

It must be emphasized that this code works correctly, if sub-optimally. Allowing the developer to write code which engenders this performance problem is perfectly fine but an advanced developer may realize the problem and wish to optimize; the language should provide the facilities to easily and cleanly do so.

As discussed, futures can be used to pipeline remote requests. Consider that the example code given is executing on the server. In this case, since `gui_populate_list` has no local side effects, the developer can easily rewrite the code as such:

```php
while ($x = database_fetch_row())
    future gui_populate_list($x);
```

Now, the main thread will not block waiting for `gui_populate_list` to return from execution on the client. The problem of unbounded growth of a cache is also solved – if resources are low,
the VM can block on a futures expression until resources have been freed, presumably by fulfilling outstanding futures.

In the case that the loop is running on the client, that is, the loop guard expression is evaluated remotely, futures are not helpful because the loop guard needs to be evaluated before the next loop iteration can begin. In this case, the only viable solution is to move execution of the loop to the remote site. If future expressions are used as above, however, the VM will only see that the main thread is repeatedly making requests for remote evaluation of the guard expression and will migrate the thread, security considerations allowing. The local-only (gui_populate_list) requests will be made in another thread, and will not be considered. Therefore, after the migration, performance will be the same.

Events could also be used to execute the remote loop guard expression in a thread to be executed remotely, which could then issue events back to the local thread to process the results as they occur. Even though this would only use two threads total, the event requests made by the remote thread would be queued by the receiving thread, and the pipelining effect would be the same.

The use of events requires a significant structural change in the program by the developer, however, an additional advantage is that the main thread is free to proceed and will asynchronously receive the updates from the remote thread. This is particularly useful in that it allows the main thread to fall through to the UI loop, maintaining responsiveness of the application while the UI is updated asynchronously.

We will discuss more effective solutions to these problems in Chapter 5.

**Native Extensions**

A crucial requirement is that the developer be able to access native resources in a secure manner. Pip supports development of native function libraries to augment the language. The
prototype system only supports static linking of extension libraries to the VM host, but the system architecture also allows for dynamic linking to extensions.

**Extension Use**

To make use of an extension, the developer includes an import statement at the beginning of the source file that will make use of the extension:

```plaintext
import server:mysql;
```

As discussed previously, the import statement includes the site for which the extension will be loaded. Only the VM at the specified site will try to load the extension. Upon encountering a call to an undefined function, the VM's default behavior is to make an RPC request (Figure 3-6). Therefore, the host specified as the extension's binding site will receive the request to execute its extension functions.

Figure 3-7. Extensions are only loaded on the binding site. RPCs are automatically dispatched for remote-bound extensions.

Import statements for extensions may also include arguments for requested security descriptors:

```plaintext
import client:fileaccess READ_ANY, CREATE_PATH "%app%/*", WRITE_PATH "%app%/*";
```
Security descriptor names are defined by the extension developer to parameterize levels of access granted to the application. The application developer requests permissions by specifying the security descriptors corresponding to desired permissions in the import declaration.

Extension function names will be imported into the application namespace, and may be called as any other function. Extension functions may return resource handles which other functions implemented in the extension may accept as arguments.

**Extension Development**

Extensions are implemented in, or at least wrapped by, ANSI C. The extension source is compiled with and linked to the VM host executable.

The extension source consists of:

- An initialization function that the VM host calls on startup. This registers the extension with the host program so that the VM is aware of its presence.

- A second initialization function that the VM calls upon a session import of the extension. This function registers the extension functions with the VM session. The function receives the applicable set of security descriptors that are to be used to filter registered extension functions.

- The implementation of the individual extension functions. These are individual C functions which are supplied as function pointers to the VM. The number of arguments and a pointer to the arguments list is supplied by the VM when the functions are called. Macros are used to convert the VM's internal representations of the argument values into C primitives including integers, strings, and resource pointers, as well as convert the return value into the appropriate internal representation.

**Extension Security**

As mentioned previously, extensions are supplied with security descriptors indicating which functions to register. The final set of security descriptors which determine the set of functions to register is the intersection of the requested descriptors declared with the import statement, and those permissions that the user grants to the application or the trusted application publisher (Figure 3-8).
Figure 3-8. Registered capabilities are determined by the intersection of developer requested and user granted security descriptors.

The extension author must be implicitly trusted, not only because the extension will execute natively “outside the sandbox” on the host machine, but also to ensure that the security descriptors are accurate and are accurately applied to the extension functions.

For most common end-user applications, this should not be a major shortcoming. Application providers and extension providers should be two different parties except for very specialized applications. The creator of a certain native technology may supply the extension, and the application provider will simply inform the user of the requirement of the technology's extension.

Furthermore, in general-purpose applications, the VM host program that the end user receives should include the most common cross-platform UI and multimedia extensions, vetted for security by the VM provider. Since the application developers would know that the end users would have these particular extensions already, they would be most likely to rely on them to develop their application rather than choose a non-standard extension. Therefore, it should be rare for an Internet application provider to require installation of new client extension libraries.
Extension functions not imported as a result of not matching specified security descriptors will simply not exist in the application namespace. The executing host, upon encountering a call to an undefined function, will make a request to the remote host which will fail with an “offline” failure type (Figure 3-9). This is the general fallback mechanism for a function being unavailable for any reason. At this point the developer may make use of the offline functionality features to either: detect insufficient security settings and notify the user (an inelegant solution that should be a last resort for critical functionality), or to fail gracefully and fall back on another method.

In the next chapter, we will examine potential problems with the concept of transparent locality.
CHAPTER 4
THE PROBLEM WITH TRANSPARENT LOCALITY

In the canonical “A Note on Distributed Computing,” [Waldo et al. 1996] it is argued that it is impossible to create a system which makes distributed computing subsystems completely transparent to the distributed application developer. Most of their points remain valid, although their criticisms with respect to Pip are tempered by two facts:

- Pip is not intended to completely isolate the developer from all distributed computing issues. Instead, it provides a framework which keeps distributed computing issues from intruding on the architecture of the application. Facilities are provided to deal with distributed computing issues, although it is possible for the developer to naively ignore these issues for simple or non-critical applications.

- The authors do not consider a pure client/server relationship true distributed computing as with NFS (Network File System): “…failure modes are simpler than in the more general case of peer-to-peer…”

That said, the authors raise several valid concerns in several areas which Pip attempts to address: Latency, Memory Access, and Partial Failure and Concurrency.

**Latency**

Latency is a concern because accessing remote resources may take substantially longer than access to similar local resources. This presents a problem to the developer when the delay crosses the line from one not usually noticeable by the end-user, or noticeable but tolerable, to an intolerable delay that interferes with the user experience.

For example, saving a document to persistent storage can be a lengthy process that may cause a noticeable delay. Most local applications will create a new thread to save the document, while the main thread maintains the responsiveness of the user interface, even if some features may be disabled while the document saves.

A distributed word-processor, however, might have a spell check database residing on the remote server. Assume that, on an entirely local application, the delay when a user clicks on a
misspelled word to see suggested spelling is on the order of 10ms. This is an acceptable delay in that it is not noticeable to the user and therefore does not interfere with the user experience. On the distributed application, however, the round-trip time for a single request and response must be added in to the normal processing time. Further, an application server might be under a much higher load than a single-user host. The response time is now increased by a variable delay depending on the round-trip time for the request/response mechanism and the load of the server, and may well interfere with the user experience – the application will appear to “freeze” as it retrieves word spellings. It is generally preferable to inform the user that the application is working by a status message or graphical indicator and allow unrelated components of the user interface to remain responsive.

As demonstrated by the document-saving example, it is common practice for application developers to create threads to perform specific tasks to maintain the responsiveness of the application. What breaks the illusion of creating a local application is that the application developer must now compensate for delays in performing many other tasks which previously, in a local application, did not result in a noticeable delay.

Pip provides several mechanisms to mitigate latency-related issues while allowing the developer to architect the application the same as a local application:

- Futures, enabling pipelining as well as simple “one off” worker threads.
- Events provide a simple mechanism for the server to asynchronously update the client UI.
- Automatic thread migration can autonomously transfer the execution of a thread using site-specific resources to those resources.

Automatic thread migration is the most transparent to the developer in that it requires no changes to the program over the local equivalent. However, its use is the most limited as it is most effective when resources specific to only one site are repeatedly accessed. It cannot by itself solve the Interleaved Host-Dependent Expression problem.
Independent expressions can be evaluated concurrently using futures. The simplest use of a future expression would be a remote data sink; for example, saving a document on the remote site. In this case simply prepending “future” to the function call that performs the save would be sufficient. The concurrent thread would be capable of notifying the user of progress or error states (assuming a thread-safe UI extension). For dependent expressions, pipelining may be applicable.

Events are the least transparent; however, they are are structured similarly to the way a typical local application would synchronize with a concurrent worker thread, with event handlers synchronized with the UI loop. AJAX developers will be familiar with the asynchronous structure since it is similar to that of XMLHttpRequest.

**Memory Access**

Raw memory access is not necessary to create an application. Many powerful, interpreted higher-level languages lacking raw pointers are commonly used to create applications.

Access to native resources across host boundaries and referencing parts of complex data structures is a concern, however. Pip’s resource handles encapsulate resources such as these and ensure that the host that owns the resource is the one to manipulate it.

**Partial Failure and Concurrency**

There are two additional types of failures that can occur in a distributed application versus a local application (Figure 4-1). The first occurs where one host cannot connect to the other host to make a request. Pip considers this type of failure to be an “offline” failure and provides facilities for the developer to handle this situation. The other type of failure is where a host is able to connect and make a request, but the connection terminates or the originating host never receives a response. This is a “partial failure” which presents several difficulties stemming from not knowing if the request completed.
Pip's protocol layer will remember the CSeq for the last request issued by a thread. The remote host servicing the request will keep a persistent (file- or database-based) record of requests each thread has received. It will also record when a request is completed. On a partial failure disconnect, the requesting host will continue to retry the request. If the remote host receives the retry request and finds the request was never completed (but did terminate), it returns a fatal error. This seems on the surface like an inelegant solution, but if we consider that the application process is a single process, the fact that it crashed on one host implies that it crashes on the other, much as an unhandled exception can propagate up through an entire application. While consistent with local application development, this situation is still to be avoided when possible.
If the retry times out and the request cannot be resubmitted, the requesting host has encountered a true partial failure and cannot determine if the thread crashed without completing, never started (offline failure), or if the request completed successfully. Here, the host simply must continue retrying the request. The application layer cannot be trusted to handle the failure correctly as it does not have information about at what point during execution the failure occurred. The user can be prompted by the lower level system to continue retrying or to restart the application.

This seems grim. However, most requests will not be of a critical nature. Failures will present to the developer on the function call level, even if control was transferred midway into the function execution. The developer can declaratively specify functions that fail in less severe ways. A function declared idempotent, for example, will merely generate an offline failure as the information desired is simply not available, regardless of how far the request progressed. This allows the application developer to be naively isolated from the underlying system, however a more advanced developer will be able to take advantage of simple available features to easily optimize the application and user experience.

Generally, the only partial failures of concern are those that non-idempotently manipulate server resources. Non-idempotent manipulation of thread-owned data structures is not a concern because the change in the data structure is only realized by the requesting agent when it receives the final response. Internet application developers are already accustomed to distinguishing functions based on idempotence; HTTP distinguishes POST from GET.

Side effects from partial failures can be minimized while still keeping the underlying mechanisms relatively transparent; use of declarative structures prevents the mechanisms from interfering with the imperative application logic.
In this chapter we examined some of the unique distributed computing issues need to be dealt with when attempting to create a traditional application development environment for RIAs. In the next chapter, we summarize progress so far on the Pip system, what remains to be implemented, and outstanding issues requiring further research.
CHAPTER 5
OPTIMIZATION

This chapter discusses the performance optimization techniques developed for distributed network applications which are the focus of the research.

Factors Influencing Performance

To optimize a network application, we must first understand the factors affecting its performance, which we define to include wall-clock execution time and responsiveness to user actions:

- Total execution time,
- Bandwidth consumption,
- Network latency.

Total Execution Time

The total execution time of a program is the total amount of time spent executing on a processor. There already exist many compiler and runtime optimization techniques for reducing execution time of a local program. These techniques are equally applicable to distributed applications. Examples of execution-time reduction optimizations include exploiting instruction level parallelism, processor cache hints, and just-in-time compilation. Since these techniques are effectively no different for distributed programs, they are not explored in this work.

Bandwidth Consumption

Bandwidth consumption is a major factor in the performance of network applications. A receiving party experiences additional network latency corresponding to the length of a received message divided by available bandwidth. Therefore, we can attribute the bandwidth performance penalty of a network application to three factors:

- Available bandwidth,
- Length of network messages,
- Number of network messages.
Available bandwidth

Available bandwidth is the rate at which data travels over a network connection between two hosts. It determines the minimum time delay from receiving the first byte of a message until the message has been completely received.

Available bandwidth is a factor outside of a single software component’s control. Software within a system may work cooperatively to manage bandwidth. Quality-of-service (QOS) controls may be implemented to guarantee a minimum available bandwidth. However, for Internet applications, this is not something over which we have much control.

In general, however, Internet bandwidth has been following an ever increasing trend. There is theoretically no limit on bandwidth increases since more links can always be added with a linear cost increase.

Since available network bandwidth tends to grow in general, and there is not realistically anything that can be done from our point of view to increase available bandwidth, we will ignore this factor.

Length of network messages

The length of a received message is also proportional to the minimum delay experienced by the receiver. Unlike available bandwidth, however, there is plenty that can be done to shorten network messages.

Efficient encodings such as Google’s Protocol Buffers¹ can be used to attempt to minimize message lengths. At a lower layer, data compression techniques can further shorten transmitted messages.

In the distributed execution context, hosts may be able to cache received data – mobile code or RPC argument values. The transmitting host could then send an identifying symbol corresponding to the cached data instead of re-sending the data.

**Number of network messages**

As the effects of latency from individual messages accumulate, the number of messages transmitted plays a role in total wall-clock execution time.

Messages can be eliminated by grouping multiple requests into one message and sharing common data such as message headers. Runtime optimizations could migrate threads to the host where resources are being most frequently accessed. Compiler optimizations can group host-dependent RPCs into one remote procedure.

**Network Latency**

Network latency is the minimum amount of time from the first byte of a message being sent until the first byte is received by the remote host.

While advances in reducing latency have come from new switching technologies and improving infrastructure with more direct links with fewer switches, network latency has a fundamental limit determined by the speed of light: the fastest rate at which a message could possibly travel from one point to another. As discussed in the introduction, this delay is significant, and there is not much performance left to be extracted.

The only factor influencing performance due to network latency under our control is the number of messages transmitted and acknowledged. The length of a message typically does not matter as each packet is pipelined on the network, although some QOS systems may simulate limited bandwidth by inducing latency for or dropping subsequent packets.
### Optimizing Network Applications

The dominating factors in reducing wall-clock execution time and maximizing responsiveness of a network application are the number of messages that a host must wait to receive and the length of each message. Certainly execution time plays a role, although this is somewhat diminished in end-user applications where user-interface responsiveness is key and many low-level user interface operations are handled by fast native libraries. Further, in an Internet application, where network latencies over 100 milliseconds are common, an order of magnitude difference in execution time of most functions will not be as noticeable except when they are very processor intensive.

The primary goal in optimizing the Pip network application system is to minimize the effects of latency on the user experience. Bandwidth is a concern, but since available bandwidth is always increasing and techniques for reducing bandwidth consumption are much more general in scope, we will not be focusing on bandwidth reduction. In our benchmarks, we will treat the system as if it has nearly infinite bandwidth.

Execution time, deemed a relatively insignificant factor, subject to ever increasing hardware improvements, and much more general than the network application domain, will also not be a research focus. However, there will be some exploration of parallel distributed heterogeneous execution.

Therefore, the fundamental factor influencing performance and the one we strive to minimize is the number of synchronous network messages.

### Static versus Dynamic Optimization

There are two general approaches to automatically optimizing application performance: static optimizations that are made at build-time by the compiler or other pre- or post-processing tools, and dynamic optimizations that are made at runtime by the execution environment.
While there are many types of optimizations that fall into both categories, we only consider techniques that will help with the goal of reducing the number of network transactions in the distributed execution environment. Network messages are necessary whenever data or a thread of execution is transferred from one host to the other.

**Dynamic Optimization**

Dynamic optimizations include a variety of techniques such as:

- Just-in-time (JIT) compilation, which compiles bytecode or interpreted programs into native instruction code as it is executed.
- Partial evaluation, where a program is re-compiled to be optimized for some known inputs.
- Memoization, where pure function return values for given inputs may be cached for reuse.
- Thread migration, specific to distributed computing, where a thread migrates to a host with the resources it needs or to distribute CPU load.

JIT compilation and partial evaluation techniques are not really applicable to our problem because they are equally effective local optimizations. However, memoization and thread migration have potential to reduce the number of remote requests made during execution.

**Memoization**

If the compiler can identify pure functions, there are two optimizations that can be made. One is that the virtual machine can cache calls to pure functions and return the cached values. If the function call is remote, this can prevent redundant network requests. The other is a static optimization: pure functions in most cases will not rely on host-specific resources, and therefore can be inlined, removing the possibility that a remote request would need to be made to execute a pure function.
Thread migration

Thread migration allows a thread of execution to be suspended, serialized, transmitted to a remote host, and then seamlessly resumed on the remote host [Mathiske et al. 1997]. Threads are lazily migrated when execution begins as soon as possible on the remote host and needed code or data are streamed to the remote host, while eager migration transmits all of the necessary code and data before remote execution begins.

Dynamic thread migration has some distinct disadvantages, however. First, there must be a period of sub-optimal performance that the heuristics can recognize before the migration is forced. In addition, use of heuristics on real-time data may result in a decision being made shortly before conditions change such that the situation again becomes sub-optimal, meaning the migration will have actually harmed performance. In short, the use of heuristics is inexact and in many situations may lead to incorrect “guesses” about how to extract optimal performance.

Another issue with dynamic thread migration is that of what data to migrate, i.e., how much of the current thread state to migrate. While we have decided that bandwidth use is not a major performance issue, obviously always transmitting the entire thread state whenever execution passes from one host to another would be an extremely sub-optimal situation: it is unlikely except in very simple cases that both hosts will need access to the entire thread state, and that the thread state would be so volatile as to need to be retransmitted every time execution passed from one host to the other.

Migrating a minimal state and then lazily migrating remaining state on-demand creates additional network transactions, which would undermine our goal. So, additional heuristics must be employed to examine data such as previous migration points and to attempt to associate pieces of data with hosts. This increases the complexity of the heuristics and creates more
situations where performance may be suboptimal until the heuristic data is sufficient to rectify them.

Finally, since dynamic thread migration relies on statistics gathered during runtime to affect optimization, there may be poor-performing corner cases when unusual or erratic inputs are supplied. Dynamic optimization may engender bugs and security issues that only manifest under very particular circumstances. Essentially, the use of many heuristics in runtime operation would make developed applications exponentially more complex to thoroughly test since the program may execute completely differently from one run to the next.

**Static Optimization**

Static optimization includes such techniques as:

- Peephole optimization, which optimizes inside a small window of instructions.
- Inlining function calls to reduce overhead.
- Dependence analysis to remove unreachable code.
- Partial evaluation of expressions given constant values.

In our work, static optimization places the responsibility of assigning code to execute on specific hosts on the compiler. Whereas dynamic analysis is based on information about what the program has done in the past, static analysis takes into account all possible paths of execution. While dynamic optimization allows execution to adapt to different runtime environments and inputs, static optimization must remain conservative to optimize for the general case.

There are several advantages of applying static instead of dynamic optimization techniques. First, data-flow analysis can determine the maximum subset of state necessary to transfer to (and from) the remote host. That is, the compiler can see all data that can possibly be accessed or modified in advance, and transfer only that set of data. It is still possible that some data transferred is never accessed due to conditional statements or runtime array access.
calculations, but the transferred subset could still be much smaller than the entire thread state, and no additional transactions due to lazy migration are necessary.

The other key advantage of static optimization is that there are no latency periods of sub-optimal performance before the heuristics provide enough information to take action: the analysis is all completed in advance, and the program as compiled is already in a more optimal state.

Finally, static optimization creates a program which executes the same every time, instead of modifying execution based on accumulated runtime heuristics. This makes the program much easier to test and debug than a dynamically optimized program, since behavior should remain consistent for different input cases and subsequent runs.

Our goal in static optimization is to transform a naively-written program to have similar performance as a program written to maximize performance, while maintaining correctness as defined by the original program’s behavior.

**Static Optimization Methods**

We believe static optimization techniques will provide the biggest performance improvements with the smallest penalties in terms of complexity, maintainability, and minimizing instances of counter-productive changes, since we will be optimizing for the general case, and there is no heuristic accumulation latency, nor reliance on possibly flawed heuristics.

Here we explore several static optimization strategies to minimize network transactions:

- Function binding,
- Asynchronous RPC,
- Proxy return value type,
- General dependence analysis,
- Functional transformation.
**Function Binding**

In this optimization, functions bodies are scanned for function calls. The sole reliance on function calls bound to a particular host binds the analyzed function to that same host. Functions making both calls to server and client bound functions are left available to be executed on either host. Pure functions, not making calls to functions bound to any host, may be inlined.

This technique is simple to implement, but leaves much of the work in the hands of the virtual machine since the virtual machine must decide where to execute functions with mixed bindings. Our virtual machine implements a simple policy:

1. If the code is available locally and not bound to the remote host, execute it locally,
2. Otherwise, make a remote procedure call (RPC) request,
3. If an RPC request cannot be executed because the code is bound to the remote host, send the code back to the remote host.

In many cases simple function binding provides adequate performance and has well defined security properties in terms of data shared between hosts as a result of virtual machine execution policy. However, there are some significant performance drawbacks that must be addressed.

**Asynchronous RPC**

Consider the case of a loop, containing calls to server-bound and client-bound functions $fs$ and $fc$, respectively:

```plaintext
while ($x = fs()) fc($x)
```

The surrounding function will not be bound to either host. Whichever host executes the function will need to make at least as many RPCs as loop iterations: if the server executes the loop, every call to $fc$ will be an RPC, and if the client executes the loop, then every call to $fs$ will be an RPC. The executing host must wait for the remote request to complete in each iteration of the
loop. Even if the remote procedure takes no time to execute, the local host will wait for twice the network latency time for each RPC.

In this example, we note that the server-side function acts as a data-source while the client-side function acts as a data-sink. That is, data originates from the server, and is ultimately consumed by the client. The implication of this is that, if the loop were executed on the server (i.e., the data source), it should not need to wait for a response from the client (i.e., the data sink) to continue execution.

Given a remote function whose return value is discarded, and said remote function has no global side effects, it can be executed asynchronously. Asynchronous execution means that the requesting host does not need to wait for a response from the remote host after making the request. Execution will proceed on the local host without any delay caused by network latency. Extra performance may even be gained due to concurrent execution of the remote request with the local execution.

However, an asynchronously executed remote function may have side-effects. For example, it may be writing to a file in which the records must be in sequential order. Therefore, the receiving host must queue each asynchronous request in the order in which it was issued. Synchronous requests must also be processed in the same queue. Functions which have global or control side effects (e.g., call remote functions) must still be called synchronously, since they may engender more remote function calls which must be executed in order.

In our implementation of Pip, the native extension interface allows the extension author to specify if an extension function may be called asynchronously. Since Pip does not provide global variables, most functions are unlikely to have global side-effects. The compiler will call functions tagged as “asynchronous” synchronously when the return value is used or a pass-by-
reference argument specified. This limits native functions that may not be called asynchronously to those that have global control side-effects, i.e., those that use callbacks that may lead back to execution on the remote host.

**Related work: Co.** Microsoft’s Cω language\(^2\) provides a similar queued asynchronous procedure call mechanism, with declarations describing which functions (C# methods) may be called asynchronously [Benton et al. 2004]. In Cω, asynchronous methods are queued until a corresponding synchronous method is called. The synchronous method declaration specifies which asynchronous methods must have completed before the synchronous method can be executed. Asynchronous calls are either queued or executed concurrently with the main thread. If the main thread reaches a synchronous call with unfulfilled prerequisite methods, the main thread blocks until all corresponding asynchronous methods have completed.

**Proxy Type**

Consider the following code fragment, where `fs1` and `fs2` are server-bound function calls, and `fc1` and `fc2` are client-side function calls:

```bash
$x = fs1();
$y = fc1($x);
$z = fs2();
fc2($y, $z);
```

In this example, our sink function `fc2` can be dispatched asynchronously, but this is not ideal because a call to `fc1` must first complete synchronously in order to supply the value of the argument `$y` to `fc2`. Executing the code fragment client-side does not help since we must complete two synchronous server-side calls to retrieve the values assigned to `$x` and `$z`.

Futures

A future [Baker and Hewitt 1977; Flanagan and Felleisen 1995] is a concurrent programming construct which specifies that the result of an expression is not needed until a later point during the execution of the thread. Specifically, a lazy future is not evaluated until it is needed while an eager future is evaluated concurrently with the continued execution of the originating thread.

Suppose we were to add eager futures to Pip. We will denote futures with the `future` operator. The following examples illustrate its use:

- \$x = future \$y + \$z; computes \$y + \$z concurrently,
- \$x = future ExampleFunction(\$y + \$z); creates a new thread to evaluate \$y + \$z and call ExampleFunction.

The result of a future expression is a marker or proxy (called a promise) that stands in for the result until it is available. References made to the promise before the concurrent evaluation is complete block the referencing thread (Figure 5-1).

![Figure 5-1](image)

Figure 5-1. A future spawns a concurrent thread to evaluate the given expression.
In distributed applications, futures can be particularly useful because they can be easily used to pipeline sequential dependent remote evaluations [Blelloch and Reid-Miller 1997]. The net effect of the pipelining is to force remote evaluation of the entire sequence in one transaction.

![Diagram of nested RPCs with futures](image)

Figure 5-2. Transactions to evaluate nested RPCs. A. The expression evaluated without futures. B. Futures allow the requests to be sent nearly simultaneously and the evaluation to be performed remotely with a minimum of latency.

Adding eager futures to Pip to enable concurrent remote evaluation would alleviate the bottleneck of supplying remote function return values to subsequent remote function calls by pipelining the calls.

```php
$x = fs1();
$y = future fc1($x);
$z = fs2();
future fc2($y, $z);
```

In our example, we will add the future keyword in front of the call to `fc1` to concurrently evaluate the remote call to `fc1` while the local (server-side) code continues to execute. When the future thread is created, a proxy value is placed in `$y`. When local execution reaches the call to `fc2`, another thread is created to evaluate the arguments and perform the call. If the first future thread has finished evaluating the remote call to `fc1`, then `$y` will contain `fc1`’s return value. Otherwise, `$y` will contain a proxy type with a value corresponding to the `fc1` future expression. If the remote request for `fc1` contained this proxy value, then the remote host can know to
associate the proxy value with the request’s return value. Therefore, the proxy value can be sent to the remote host as the argument for $fc_2$. Using futures, the requests can be issued virtually concurrently, halving the effective network latency to complete evaluation (Figure 5-2).

**Simplifying proxy types**

We can distill these concepts quite a bit to reduce overhead and complexity. Our primary goal in introducing future evaluation (and with all our optimizations) is to minimize synchronous remote requests. The primary property of futures we are interested in is the ability to pipeline synchronous remote requests. This is made possible by the proxy type, a value of which we can supply as a remote function call argument to represent a previous synchronous remote function call’s return value.

In the implementation of Pip, the proxy value is generated locally when the VM encounters the remote function call and is immediately returned as the result. The proxy value is also sent with the request. When the remote host has finished executing the function, the return value is stored in a table associated with the proxy value instead of being sent back to the originating host.

When the remote host receives an RPC request, it checks the arguments to see if any are proxy values. If they are, they are replaced with the actual values before the function call is made.

Functions that have been declared as being able to be called asynchronously are automatically called asynchronously; only now, they can be called asynchronously even when the return value is used. It is not necessary to specify a future keyword: the compiler can determine if it is feasible to make the remote procedure call into an asynchronous call with a proxy return value. To do so, first, the remote function can only be used as an immediate expression of either a remote function call argument or as an R-value to an assignment. Second,
if the return result is used in an assignment, the variable it is assigned to must only be used as the sole assignment value or as a remote function call argument. This condition applies to any subsequent assignments.

The remote requests must be completed in the order they appear in the program to maintain the correctness of the original program with respect to side effects. The remote host must serially execute the asynchronous tasks, just as before. Therefore, we do not need the complexity of actually creating parallel threads of execution for each future expression: the asynchronous procedure call mechanism will queue requests, allowing execution of the main thread to continue without delay. However, even without creating new threads, the remote host will perform these tasks concurrent to the local host’s main thread, just in serial order with respect to one another.

**General Dependence Analysis**

The proxy type works well to help turn what would be synchronous RPCs into asynchronous calls when the return value is directly used as an argument to another remote function. However, if the value is transformed in any way, using a proxy value to substitute for the value is no longer possible.

Consider our previous example code fragment, but now we add 1 to the argument of fc2:

```
[1] $x = fs1();
[2] $y = fc1($x);
[3] $z = fs2();
[4] fc2($y + 1, $z);
```

Previously, fc1 was called asynchronously with a proxy value stored in $y. With this change, however, it is no longer possible unless we wait to retrieve the value of $y before the call to fc2, so that we can add 1 to it. If we retrieve or wait for $y, however, we give up the performance gained by using the proxy value, since we add a synchronous request.
Examining the dependences of the statements upon one another, we note that statement 4 (the remote call to \texttt{fc2}) is immediately dependent upon statements 2 and 3. However, there is no immediate dependence between statements 2 and 3. Therefore, statements 2 and 3 may be swapped:

\begin{verbatim}
[1] $x = fs1();
[3] $z = fs2();
[2] $y = fc1($x);
[4] fc2($y + 1, $z);
\end{verbatim}

Now, statements 1 and 3 are server-bound, while statements 2 and 4 are client-bound. We can eliminate synchronous requests by executing the beginning of the program on the server and then making an asynchronous request to the client to execute the remainder of the program. The $y + 1$ argument to \texttt{fc2} can be computed client-side.

Chapter 6 describes the general implementation of this type optimization to identify concurrent, independent tasks, and chapter 7 describes the application of this technique to the problem of automatically distributing execution.
CHAPTER 6
FUNCTIONAL TRANSFORMATION

Introduction

Imperative languages are commonly used in high-level application development, however, functional languages are more easily automatically parallelized [Jones 1989]. With multi-core CPUs becoming commonplace, we believe the ability of high level language compilers to automatically exploit parallelism opportunities will become more important.

Consider how the expression $sqr(a) + sqr(b)$ might be evaluated in an imperative language versus a functional language. The imperative version is restricted to first evaluate $sqr(a)$, then $sqr(b)$, then add the results together. On the other hand, a pure functional language, with only pass-by-value functions and no shared mutable state, lacks this restriction. The two $sqr$ functions may be evaluated in any order, or even concurrently.

An imperative interpreter should not need to evaluate the $sqr$ operations strictly in-order, either, since the $sqr$ function does not modify shared state. However, imperative languages make a guarantee to the programmer that operations will be evaluated in the order in which they are specified. The reason for this guarantee is the existence of side-effects, that is, changes in state that may affect the program, but are not explicitly defined in the program logic.

Causes of side-effects include manipulation of shared state, for example, by pointers, pass-by-reference arguments, or global variables, and manipulation of a program’s environment, for example, writing to a file. Pure functional programs do not have side-effects; instead, shared state is explicitly tracked through mechanisms such as monads [Wadler 1990].

Side-effects are explicitly defined in a functional program, allowing the compiler to order operations when necessary. In an imperative program, the programmer must manually specify the order of all operations to take into account side-effects. The fundamental problem with the imperative model is that the programmer must specify an order of operations whether or not the operations are independent.

The imperative programming model reflects the way the hardware processes instructions. In this sense, imperative programming can be considered low-level. For low-level programming, the strict requirement that every operation be executed in the order specified is important because the compiler has to view interaction with, for example, hardware devices, the operating system, and raw memory manipulation as black boxes which can affect any other part of the program. However, consider a sufficiently high-level imperative programming language, in which every operation is pure, shared state is able to be tracked (no ambiguous pointer operations), and interaction with the outside system is done only through APIs of which the compiler is aware. This imperative language would now have many properties of a pure functional language, and the strict in-order evaluation requirement could be removed.

In this paper we introduce techniques used in transforming a program written in a high-level imperative language into a functional program:

- We describe the concept of “side-effect domains” wherein functions may be specified as only having side-effects upon other functions in the same domain.
- We describe how to use a dependence graph representation of a program to identify task-level parallelism.
- We discuss techniques for transforming imperative programs into their functional equivalents, which aids in creating the dependence graph representations.
Side-Effect Domains

As an example, we will discuss a simple program to connect to a remote database, query some information, and write the information to a local file (Figure 6-1). We will make the assumption that the remote database and the local file system cannot interact with one another. We will also assume for simplicity that the runtime environment does not do anything to interrupt program execution, such as throw exceptions. Therefore, any side-effects of the file system functions do not affect the database functions, and vice versa. Without involving the semantics of what the functions actually do, we can safely classify them into two separate domains of side-effects: functions that affect or may be affected by file system operations, and functions that affect or may be affected by database operations.

When the programmer wrote the program, he took into account the side-effects of each operation in deciding the order in which to call each function. For example, he would have to first open the file before closing it. But, some of that order may not always be necessary: it is not necessary to connect to the database before opening the file. These operations are independent of one another.

The programmer could have written any version of the program which had the same topological ordering of operations as dictated by the side-effects and data-dependences of the functions called. The imperative programming model forces the programmer to choose a valid topological order of operations, and the compiler must obey that order when evaluating the operations in fear of violating hidden dependences due to side-effects. However, if the compiler is aware that certain classes of operations cannot interfere with other specified classes of operations, it removes the artificial constraint that those operations must be executed sequentially and in the original order specified.
A side-effect domain is a class of functions that are only dependent on other functions in the same domain. Functions may belong to more than one side-effect domain. Functions may also belong to all side-effect domains if they are capable of modifying the flow of the program. For example, a function that throws an exception may have side-effects on all subsequent operations since whether or not the subsequent operations are executed is dependent on the function in question successfully completing.

Pure functions, which neither have side-effects nor are affected by any external influences, for example, the $\text{sqr}$ function, belong to no side-effect domains. Since pure functions lack side-effects, the number of times they execute does not influence the correctness of the program. Therefore, identification of pure functions allows optimizations such as memoization and speculative execution.

Specifying side-effect domains gives the compiler the freedom to use valid topological orders of operations other than the order specified in the program. The order of operations specified in the program is still important, however, since the compiler does not know the semantics of the operations. In Figure 6-1, the compiler does not know that it is necessary to open a file before closing it, but the side-effect domain (Figure 6-2) tells the compiler that any file operations must be performed in the same order as originally specified with respect to one another. If two operations are independent of one another, not only can they be executed in any order, but they can be executed in parallel. We can rewrite the program in Figure 6-1 as a multi-threaded program (Figure 6-3).

Thread B may need to block to receive the value stored in $x$, but aside from that, the two threads are independent. Theoretically there would be a performance gain as the blocking database and file system operations execute in parallel. While the programmer could have
written the program as a multi-threaded program, in most imperative languages, this adds a lot of overhead to the development process in terms of the program architecture and the developer needing to manually micromanage thread creation and synchronization details. The developer may find it is not worth the development overhead to break small tasks up to be processed by separate threads. This is especially so if the tasks are not performed inside tight loops, which they seldom are in an end-user application, where performance is often measured as responsiveness in performing small tasks. Side-effect domains allow the compiler to identify these opportunities for parallelism.

Side-effect domains should be defined for all functions that interact with the outside environment. In a sufficiently high-level language, all external I/O would be performed through APIs defined at compile time. The ability to alias memory locations through pointers and access raw memory would be extremely limited. In this high-level environment, the application would be pure logic with all I/O performed through APIs which would have predefined side-effect domains. The application developer should not need to concern herself with the concept of side-effect domains for the compiler to take advantage of them.

**Dependence Analysis**

In our example program, there is one data dependence: \( \text{file\_write}(x) \) is dependent on \( x = \text{database\_query()} \). There are also several side-effect dependences. Figure 6-4 illustrates all the dependences between the statements of the program in the form of a program dependence graph (PDG) [Ferrante et al. 1987]. Our goal is to be able to generate the PDG of all the operations in an imperative program. The PDG can then be used to identify opportunities for parallelism.
Side-Effect Dependences

Most side-effect dependences are identified using side-effect domains. A function call in a given domain is dependent on all previously defined function calls within that domain.

For example, `file_close()` is dependent on `file_write(x)` and `file_open()` (Figure 6-4). In constructing the dependence graph, however, it is sufficient to link a function call only to the most recent previous function call within each common domain: `file_close()` only needs to be dependent on `file_write()` since `file_write()` is dependent on `file_open()`.

Data Dependences

Variables store state for later use. In Figure 6-1, the variable `x` is used as a place-holder for the value read from the database. If the programmer had composited the functions:

```c
file_write(database_read())
```

then the data dependence of the `file_write(...)` node on the `database_read()` node would be clear.

Instead, to track this dependence, the imperative parser must tag the use of `x` as an l-value when it is assigned a value (*defined*). Traversing the imperative abstract syntax tree (AST) in the same order as it would be compiled, subsequent r-value nodes of `x` are linked to the definition node by a data dependence.

Control Dependences

Operations that may execute a variable number of times are control dependent. For example, in the statement `if (x) f()`, the function call `f()` is control dependent upon the `if` operation. The `if` operation determines if `f` is called 0 or 1 times. Similarly, a `while` operation determines if its dependent code is executed 0 or more times, and its guard expression 1 or more times.
Every AST node is control dependent upon the most recent conditional expression ancestor node.

```plaintext
if (x) {
    y = f(x + l);
    if (y) g();
}
```

In this example, the = operator, the f function call, +, x and l expressions, and the nested if and y evaluation are all control dependent upon the outermost if. The g() function call, however, is only control dependent upon the nested if.

**Generalized algorithm**

The PDG is built as the AST is traversed in the same way it would be traversed being compiled or interpreted. When an AST node representing an operation is visited, a new PDG node is created. Edges are created between dependence nodes to represent the different types of dependences.

1. An expression node is data-dependent on its children (sub-expressions): in the expression 2 + 3, the + node has a data-dependence on both the 2 and 3 nodes.
2. Function calls are dependent upon the most recently encountered function calls within each of the side-effect domains to which it belongs.
3. R-value references to variables are data dependent on the definition of the variable, e.g., the assignment operator. L-values must be side-effect dependent upon the most recent previous definition and its subsequent r-value references. Prior transformation to SSA form can be used to minimize occurrences of re-definition of variables.

**Compiling**

Compiling the program is a matter of deciding on a topological order in which to evaluate the AST nodes. The PDG allows us to enumerate possible orderings. Research into techniques to compile PDGs into efficient parallel programs [McCreary and Gill 1992] already exists, so we will just describe a simple evaluation scheme.
Task scheduling with side-effect dependences

We wish to execute independent tasks in their own threads. Before the compilation pass through the dependence flow graph, we first identify paths through the graph to be executed in their own threads. This is done by looking for data-flow paths between side-effect dependent operations, and labeling every node in the path as belonging to the same thread. For example, in the expression \( f(g() + 1) \), assuming \( f \) and \( g \) share side-effect domains, the + expression along the data-flow path between \( g() \) and \( f(...) \) would be labeled the same as the function calls.

After all the inter-function call paths are labeled, the paths from the leaf nodes are followed in a depth-first traversal until a labeled node is reached, and the unlabeled portion of the path is labeled the same as the reached labeled node. The 1 constant leaf node in the example expression traces back to the labeled + node, and therefore is evaluated in the same thread as the rest of the expression. Sometimes a side-effect free expression may have multiple immediate dependents that lie in different side effect domains. In this case, the algorithm will have arbitrarily chosen the label to assign the expression branch. Similarly, the label for pure code along paths between two distinctly labeled nodes is arbitrarily chosen.

Inter-thread data dependences

While the side-effect dependences are used to assign execution paths to threads, there may remain data-dependences between threads. The compiler can treat the data dependences as messages passed between threads. When the compiler evaluates a node with a data dependence on a node in another thread, it injects code to receive the required data before proceeding with the evaluation of the node. Similarly, after evaluation of a node that has an external dependent node, the compiler injects code to send the required data. Other types of dependences may lie between threads; these can be considered semaphores for the dependent thread to proceed.
In our initial sample program (Figure 6-1), a value $x$ is read from a database and written to a file. Labeling the PDG, the database operations and file operations will be performed in two different threads. The storage of the value into $x$ itself is a pure operation, and thus, will be arbitrarily assigned to a thread.

We will label the database thread Thread A and the file thread Thread B. We will chose to label the assignment to and evaluation of $x$ as executing in Thread A. Now, when the program is executed, Thread A and Thread B run concurrently. Thread B reaches the file_write operation and blocks awaiting its argument. Thread A reads the value from the database, assigns it to $x$, evaluates $x$ as an r-value, and then dispatches the value to Thread B. Thread B receives its argument, and both threads execute concurrently until completion.

**Functional Transformation**

PDGs are a common tool for identifying parallelism, and there are varied techniques for the analysis of imperative control dependences [Towle 1976], which present difficulties in PDG analysis since, for example, they may create cycles. Our approach is unique in that we effectively transform the imperative program into its functional equivalent.

**Mutable State**

Pure functional programs lack mutable state. While eliminating all mutable state from many programs is impossible, we can attempt to minimize it to give us more flexibility in optimizing a program.

**Eliminating place-holder values**

In the sample program (Figure 6-1), the variable $x$ is used as a place-holder for the value read from the database. The algorithm to break this program into two communicating threads leaves vestigial code of an assignment to and subsequent evaluation of $x$. It is common to find variables used as place-holders when operations must be executed sequentially; values may need
to be saved to be used in code further down or to be used in multiple places. It may even simply be a matter of aesthetics in not nesting function calls.

Any number of compiler optimizations can be used to eliminate needlessly storing values into memory. The simplest, since we are operating on the PDG, is to eliminate the assignment and dependent evaluation nodes, and directly linking the dependent nodes to the definition of the value. This results in an evaluation that is much more akin to a memoized function result in a functional program.

**Variable re-definition**

As mentioned previously, transformation to SSA form eliminates a lot of dependences if variables are redefined. Array access analysis can be performed to attempt to determine if two array operations access the same elements, and if they do not, they do not need to be dependent upon one another. Ambiguous array writes must be side-effect dependent as if the variable is redefined.

**Pass-by-reference**

A pure functional language is only capable of pass-by-value. However, many lower level API functions use pass-by-reference arguments. In most cases, this is used as a mechanism to return multiple values, but in some cases the passed value may or may not be modified. (In the cases where the pass-by-reference arguments are never modified, e.g., strings, the high-level API binding should mask this.) Unfortunately, the compiler has to assume that any pass-by-reference arguments will be modified. In performing the dependence analysis, the function call is considered a redefinition of its pass-by-value arguments. Therefore, any subsequent evaluation of a variable used as a pass-by-value argument is data-dependent on the function call.
**Conditionals**

An imperative conditional `if` statement specifies that a sequence of instructions is only executed if some condition is true. For the program in Figure 6-6, assume that the `a_` prefixed functions are in one side-effect domain while the `b_` functions are in another. To make this code multi-threaded, we have to break up the one `if` statement into two, without re-evaluating the conditional expression (Figure 6-7). To do this, the compiler must store the result of the conditional expression and communicate this as a data dependence [Allen et al. 1983].

When the compiler encounters a control dependent node, it injects code to check the conditional expression result before evaluating the node. To handle nested conditionals, the compiler must also check the result of the conditional node’s conditional parent.

**Loops**

Loops are problematic because operations in the loop body may affect other operations that appear before it, as well as after, as the loop body may be executed more than once. This creates cycles in the PDG, which our evaluation scheme cannot handle.

In functional programming, loops are handled by tail recursion. Recursion eliminates the forward dependences that would be necessary in an imperative loop structure. All the forward dependences are now encapsulated in the recursive function call. Therefore, cycles in the PDG are eliminated. Tail recursion places the recursive call at the end of the function which eliminates the need to preserve state on a stack.

To implement this, our compiler must transform loop structures into recursive functions. For example, a while statement becomes a function accepting two lambda functions as arguments:

```haskell
while(cond, body) {
    if (cond()) { body(); while(cond, body); }
}
```
The loop function forms a closure with its parent function. Data-flow analysis must be used to track values defined inside the loop function that are subsequently used outside the loop.

```plaintext
x = 0;
while (x < 10) x = x + 1;
print(x);
```

Here, \( x \) is modified inside the loop body. \( x \) must be passed by reference to the loop function. This ensures that the loop is dependent on the initial \( x = 0 \) definition and that the print statement is dependent on the loop. However, a loop may be independent of surrounding code, and arguments to the loop function may be pass-by-value. If \( \text{print}(x) \) were not present, \( x \) could be passed by value. In that case, the loop may be executed in a separate thread, and further loop analysis may yield data-parallel optimizations.

**Subroutine Functions**

Our compiler needs to be able to perform dependence analysis on application logic subroutine calls. We are assuming that the high-level language is pure, but the subroutine functions may call functions with external side-effects. To account for this, a subroutine function must inherit the side-effect domains of all the functions it calls.

To achieve ideal performance, subroutine functions must be analyzed in-line with the caller function. For example, if a function calls two different subroutines which each call functions in two different side-effect domains, the compiler will have to run the two subroutines sequentially. However, inlining the subroutines may allow the compiler to break out the API functions in each side-effect domain from the two subroutines, and run them concurrently. In Figure 6-8 the \( \text{init} \) and \( \text{write_value} \) functions each belong to both the database and file side-effect domains. Therefore, \( \text{write_value} \) will have a side-effect dependence on \( \text{init} \), and the compiler will have to schedule them sequentially. By inlining the two subroutines, however, the compiler can perform the analysis as previously demonstrated.
The compiler should not always need to inline everything for optimal performance. In general, most developers practicing separation of concerns will seldom mix different types of low-level functionality in a single subroutine. Therefore, mixing calls to functions with different side-effect domains will usually only occur at the higher architectural levels of a program.

The compiler only needs to inline functions that belong to multiple side-effect domains shared by other function calls in the subroutine. Inlining may yield additional benefits such as enabling rematerialization of pure expressions that might otherwise be inter-thread dependences.

Related Work

The idea of having compilers optimize programs to take advantage of parallel computation is not new, and some systems are in common use today. The OpenMP API, for example, works with a pre-processor to allow simple, nearly transparent parallelization of loops [Sato 2002] in imperative programs. Some functional languages such as Erlang are inherently parallel [Armstrong 1997].

There also exists work in transforming imperative logic to allow parallelization. Chu and Carver [1994] implemented a pre-processor to parallelize sequential Fortran code. Their system uses dependence analysis to restructure code. Like our system, subroutines may be partially evaluated in-line. As compared to our system, it could not safely parallelize code with different control dependencies, but it did perform extensive data-dependence analysis including array access analysis.

Generally, there is a lot of work using PDGs to analyze programs to extract parallelism, including static analysis tools [Allen et al. 1988] and determining optimum granularity of parallel tasks [McCreary and Gill 1992]. Several different techniques exist for dealing with problem of acyclic control dependences [Allen et al. 1983; Towle 1976]. The techniques in working with acyclic PDGs are closely related to those in program slicing [Choi and Ferrante 1994].
Work that seeks to identify heterogeneous domains of parallelism includes Neubauer and Thiemann’s [2005] system to decompose a program into different processes by tier. They describe formal methods using program slicing to separate the program into two communicating processes.

In pure functional languages, which lack side-effects, monads [Wadler 1990], witnesses [Terauchi and Aiken 2008], and uniqueness [Barendsen and Smetsers 1993] types were introduced to make external side-effects explicit. There has been work to blend the advantages of imperative and functional programs [Gifford and Lucassen 1986], and to allow imperative languages to borrow the explicit side-effect type systems [Fahndrich and DeLine 2002].

**Conclusion**

We have presented a general method to allow compilers to identify opportunities for task-level parallelism in imperative programs, rooted in functional programming principles. We believe this could help end-user applications, typically written in high-level imperative languages, automatically take advantage of small opportunities for parallelism that developers would not otherwise find worthwhile from a development cost, complexity, and maintenance perspective. Combined with loop parallelization techniques, compilers should be able to make much more use of increasingly common multi-core processors for end-user applications.

The techniques are based on the idea of side-effect domains. In a sufficiently high-level language, the only access to the outside world is through predefined APIs. While more specific declarative methods could be used to describe the semantics of API operations, we hope the simplicity of side-effect domains would make them simpler to implement in practice.
database_connect();
x = database_query();
database_close();
file_open();
file_write(x);
file_close();

Figure 6-1. Sample imperative program.

<table>
<thead>
<tr>
<th>file domain</th>
<th>database domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_open</td>
<td>database_connect</td>
</tr>
<tr>
<td>file_write</td>
<td>database_query</td>
</tr>
<tr>
<td>file_close</td>
<td>database_close</td>
</tr>
</tbody>
</table>

Figure 6-2. API functions designated to distinct side-effect domains.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>database_connect();</td>
<td>file_open();</td>
</tr>
<tr>
<td>x = database_query();</td>
<td>x = receive();</td>
</tr>
<tr>
<td>send(x);</td>
<td>file_write(x);</td>
</tr>
<tr>
<td>database_close();</td>
<td>file_close();</td>
</tr>
</tbody>
</table>

Figure 6-3. Multi-threaded version of sample program.
Figure 6-4. Dependence graph of statements in sample program (Figure 6-1). Dashed lines are side-effect dependences while solid lines are data dependences.
Figure 6-5. Dependence graph of AST nodes in sample program.
b_f();
if a_f() {
    b_g();
    a_g();
}

Figure 6-6. Program with if statement and its dependence graph. Dotted lines are control dependences.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = a_f();</td>
<td>b_f();</td>
</tr>
<tr>
<td>send(x);</td>
<td>x = receive();</td>
</tr>
<tr>
<td>if (x) a_g();</td>
<td>if (x) b_g();</td>
</tr>
</tbody>
</table>

Figure 6-7. Multi-threaded version of Figure 6-6.
function init() {
    database_connect();
    file_open();
}

function write_value() {
    x = database_query();
    file_write(x);
}

function main() {
    init();
    write_value();
    file_close();
}

Figure 6-8. Program (Figure 6-1) with subroutines.
CHAPTER 7
IMPLEMENTATION OF FUNCTIONAL TRANSFORMATION IN PIP

The preceding chapter described a technique for decomposing a program into sub-programs of concurrent, communicating tasks. Parallel computing and distributed computing are very closely related – distributed computing is effectively parallel computing with higher latencies and lower bandwidth in communication between computation units. This allows many of the concepts discussed in the previous chapter to easily carry over to Pip. This chapter discusses the details of how Pip is implemented.

Client/Server Model as Heterogeneous Distributed Environment

With Pip, we are trying to optimize performance of client/server applications. In the client/server model, there are two hosts with a different set of resources. For client/server end-user applications, the client typically manages the user interface, while the server manages persistent data.

Computation tasks may be allocated to a site to optimize responsiveness. For example, a mortgage calculator website may perform the computations to determine a mortgage payment amount on the client with Javascript after the needed values are entered by the user. Meanwhile, a street map website would choose to perform the computations to find the shortest route on the server, rather than make multiple remote queries to a large street database on the server. With respect to developing a client/server application, one would choose to allocate a task to run on the client or server based on the site whose resources are used by the task.

We treat the client and server hosts as two separate computation units, since we are focused only on reducing network transactions, and not creating concurrent tasks on the same host. This allows us to define the very large-grain side-effect domains of client and server. This simplifies the task for the API developer, who does not need to declare specific side-effect
domains for each API function. Instead, he only needs to declare whether or not a function may be called asynchronously. The application developer then uses the `import` keyword to declare an entire API library as bound to the client or server side-effect domains.

**Domain-Specific Tasks as Function Calls**

The major change to the functional transformation for Pip’s compiler implementation is to code generation: instead of independent segments of the dependence graph being allocated to different CPUs, segments of the dependence graph colored to a side-effect domain (client or server, in this case) are outlined by the compiler into their own client- or server-bound functions.

**Arguments**

Arguments to the outlined function calls are determined by the data dependences of the nodes in the segment that make up the function. Therefore, all of the necessary data dependences must be satisfied by the beginning of the function call, instead of being communicated as they are satisfied. If subsequent dependent nodes outside of the function are data dependent upon any nodes in the function, a variable to hold the result is passed by reference into the function.

**Asynchronous Calls**

There are two requirements for an outlined function call to be able to be made asynchronously. First, it must not contain any pass-by-reference arguments. Second, it must not contain any function calls that must be made synchronously, as flagged in the API. This applies whether the call is made directly in the outlined function, or anywhere else deeper in the call graph of the outlined function.

**Compiler Architecture**

This section describes the details of the Pip compiler given the background of the previous chapters. The virtual machine is described in Chapter 3. The Pip language, also discussed in Chapter 3, is based on PHP. It is important to note that PHP is a dynamic, weakly typed language
with function level scoping, and Pip inherits these properties. Unlike PHP, there is no global scope: this aids our transformation to a functional representation by eliminating implicit side-effects.

**Compiler Front End**

The Pip parser is implemented with the UNIX lex and yacc tools (or their free equivalents). The parser generated by these tools constructs an AST which is then analyzed and transformed into the functional PDG intermediate representation. As each node of the AST is created by the yacc-generated parser, it is “registered:” a pointer to the node is stored in an array, and the corresponding index of the array is stored as part of the AST node structure. This serves several purposes:

- It simplifies traversing the AST when each node needs to examined or we need to search for particular types of nodes.
- The node number serves as an indirect pointer to reference nodes from other nodes.

The node number is human-readable, simplifying examination of debug output and compiler-generated visual graphs.

In addition to building the AST node tree and list, the front-end also creates a list of function definitions.

**Function definitions**

A function must be defined (not just declared) in a program before a function call to that function may be made. Extension functions are defined by the extension author, and these are loaded from meta-information in the comments of the extension header file that interfaces to the compiler (Figure 7-1). Function definitions found in the code take precedence over extension function definitions; i.e., the application developer may override extension functions. This is done intentionally for two reasons: so that a newly introduced extension function does not break
an existing application, and so that a developer can override a basic native function to provide additional functionality, such as sanitizing input.

Extension authors may tag their functions with several different properties (Figure 7-1), such as `proxy`, `async`, `pure`, and `idempotent`. In theory, the compiler would be able to take advantage of all of this information to improve performance. However, currently, only the `async` and `proxy` tags actually affect optimization. The `async` tag tells the compiler that the function is free of global side-effects and, therefore, may be called asynchronously with respect to the main thread execution on the remote host. The `proxy` tag implies the `async` tag, but also tells the compiler that it can assume that the return value is only useful on the local host since it is tied to a local resource. Therefore, it acts as a guarantee to the compiler that an asynchronous remote request immediately returning a proxy type may be safely made without analysis. These two tags translate into the back-end emitting the `ACALL` and `PCALL` bytecode instructions, respectively, rather than a synchronous `CALL`. However, `ACALL` is only emitted for a given function call instance if the return result is not used and no arguments are pass-by-reference. A `PCALL` may also not have any pass-by-reference arguments.

The final piece of information tied to a function definition is the host to which it is bound. The import declaration that identified the extension library contains a `client` or `server` decoration making this distinction.

For functions defined as Pip source code, these same properties are tracked. However, they are inferred after the function has been parsed. The AST nodes comprising the function are scanned for function calls. Unless the function has been specifically designated by the developer as being tied to a particular host through the use of a `server` or `client` decoration, it will be tied to a particular host if it contains only calls tied to one host or the other. In that case, the function
is tagged as asynchronous-capable if it contains only asynchronous or proxy calls. Beyond this, no further optimization will be done for functions bound to one host.

The compiler has very strong conditions that must be met to emit `ACALL` and `PCALL` instructions; however, this may be overridden by the application developer by specifying the `async` or `proxy` decorations in the front of a function call definition:

```plaintext
async example_function($x);
```

It is also worth mentioning that the `ACALL` and `PCALL` instructions only have an effect if they are executed on remote functions: the VM will search for a locally available function matching the given name before resorting to a remote request. Further, functions called asynchronously are only asynchronous with respect to the execution of the main thread on the other host, and as discussed in Chapter 5, must be executed in FIFO order with respect to other asynchronous and synchronous requests.

**Pass-by-reference**

By default, every function call passes arguments by value. This helps in the transformation to a functional representation, since pure functional languages only pass-by-value to eliminate side-effects. However, it is sometimes necessary to still pass arguments by reference. Syntactically, a function may need to return multiple values and, other than returning an array or object, there is no other way to do so aside from using pass-by-reference arguments. Pass-by-reference arguments are also necessary to allow a function to directly modify a piece of information without having to duplicate it in the process.

In Pip, the ampersand (`&`) operator is used to show that a variable is passed by reference:

```plaintext
example_function($x, &$y);
```

In this case, `$y` is passed by reference, while `$x` is passed by value. It is not valid to use the `&` operator anywhere but prefacing a variable in a function argument. So, for example, it is illegal
to assign a reference to a variable, or to use a reference as a return value. Function declarations do not need to distinguish between pass-by-value and pass-by-reference parameters; however, for application source code readability, this should be added in the future.

The compiler tags AST identifier nodes decorated with the ampersand as references. Normally, the compiler emits a `PUSHS` – push value of symbol – instruction when it encounters an identifier in an expression R-value context (an argument list is an expression). However, when it encounters an identifier tagged as a reference in an expression R-value, it instead emits a `PUSHR` instruction – push reference to symbol. This instruction tells the VM to create a reference record to put on the stack. The reference record contains several pieces of information, including the position of the actual value in the stack, as well as the host that “owns” the value – when a reference record is created, this is the same as the current host. If the current host does not own the value, then it contains a pointer to a copy of the value.

If the runtime engine tries to `PUSHR` a symbol that contains a reference (i.e., a function that has been passed an argument by reference attempts to pass that value to another function by reference), then the existing reference record is copied on to the top of the stack. It is not possible to create a reference to a reference.

When the runtime engine tries to `PUSHS` a symbol containing a reference, the reference is dereferenced, and the actual value is placed on the stack, or, if the current host does not own the value, the local copy of the value is placed on the stack.

When an identifier is evaluated in the context of an L-value, a `STOI` (store into) instruction is emitted. The runtime engine first looks to see if the symbol contains a reference, and if so, then it is first dereferenced and the actual value is overwritten. The local copy of the reference record has a flag set indicating that it has been modified. When the function returns, the
stackframe is scanned for modified references – references can only exist there if they were arguments -- and this flag, as well the new value, propagate to the caller. If the caller resides on the same host, the value propagates automatically since they will both have pointers pointing to the same value, whether that is a copy or the original entry on the stack. If the caller resides on the remote host, then the new value is transmitted as part of the final response to the synchronous RPC request.

Also, the virtual machine marks values passed into a function by value as read-only. Unlike most languages, an attempt to write to a passed-by-value parameter variable does not result in modifying the function-local copy; instead, the virtual machine raises an error. This is intended to aid in debugging an application since the language is weakly typed and the compiler does not do any type checking on the arguments. While it is possible to add a static check to ensure that all arguments passed to another function defined in the source code are passed-by-reference if the function might possibility attempt to write to the value, we opted to give the developer the option, in the interest of performance, to pass-by-value if he knows that the function will not attempt to write to the parameter given the supplied arguments.

**Volatile identifier nodes and symbols**

AST identifier nodes used as L-values or referenced with the ampersand operator are marked as volatile, i.e., the values the corresponding variables contain are subject to change during execution. The function-level symbol associated with the identifier is also marked as volatile if any of the instances of the corresponding identifier in the function body are volatile. This helps the middle-end properly construct the dependence graph since it can tell if it needs to account for data dependences between instances of the identifier.
Compiler Middle End

The middle end of the compiler is responsible for transforming the AST into the functional representation and performing the outlining optimizations. For every function definition AST node, the compiler may perform several tasks:

- The function definition is added to the table of functions.
- AST nodes are scanned to count the number of site-dependent function calls. At this point, function calls can be dependent on multiple sites. For example, a function may call both server- and client-bound functions, and therefore would be dependent on both.
- The function is analyzed to determine if it is dependent only on one site. If it is, or if the function is determined to be pure, or the developer has marked the function as bound to a particular site, then all the future dependency analysis will be skipped.
- The function is analyzed to determine if it contains only asynchronous calls or calls to functions on the same site to which it is bound. If so, then the function is flagged as able to be called asynchronously.
- The loop separation algorithm is invoked if the function is not explicitly bound to a site and contains both client- and server-dependent function calls.
- The PDG of the function is constructed.
- The PDG is partitioned into separate sub-functions; i.e., contiguous parts of the PDG that are dependent only on one host are outlined into calls to separate functions.
- If no functions can be outlined, the AST is compiled as-is.

This is a broad overview of the middle-end procedure that is performed on each function. We now discuss the details of the algorithms that are used in this procedure.

Scanning the AST to count site-dependent calls

The compiler recursively traverses the AST of the function and keeps track of several pieces of information:

- The total number of function calls bound to the client and those bound to the server. If a function call invokes a previously defined function that is dependent on both the client and server, then that function’s cumulative count of both types of dependent functions is added to the count we maintain for the current function. The current function’s total is stored as part of the function definition.
• The total number of site-bound function calls of each type that may be made asynchronously. This is also stored as part of the function definition, and is less than or equal to the number of remote functions of each type.

• The total number of site-bound function calls in the current expression. The expression evaluation terminates when the depth-first search encounters a statement node on the way back to the root: the expression totals are reset to zero. The totals for each expression branch are stored in their respective AST root nodes.

If the function only contains calls to functions dependent on one site, then the function can be bound to that site. Further, if all of those functions are flagged as being able to be called asynchronously, then the current function is also flagged as being able to be called asynchronously. This information can also be used to aid in determining opportunities to inline functions when that functionality is implemented.

Loop outlining

This algorithm traverses the AST of the given function searching for \texttt{while} statements. When other loop types are added, it searches for those as well. When it finds a loop statement, it terminates the branch at the loop statement, and replaces the loop operator of the parent statement node with a function call operator.

A new function is defined named \_loop\_x, where \textit{x} is a unique number identifying the loop. The function definition record contains a flag indicating to the back-end that the function is a tail-recursive loop; this flag is set. The new function definition inherits the parent’s site binding, although it may be changed later after analysis. The symbol table of the parent function is cloned as the symbol table of the new function.

The root node of the new function AST is the original \texttt{while} operator. This is changed to an \texttt{if} operator, and a field in the function definition is set to point to this node to indicate to the back-end the target of the tail-recursive jump.
The new function is then analyzed to determine the necessary arguments from the original parent function to properly form the closure. The ASTs of both the parent function definition and the new closure function definition are traversed to scan for identifiers. Since we cloned the symbol table of the parent function as the new closure function’s symbol table, we have a convenient mapping of identifiers to their respective counterparts in the other function. Using this mapping, we create an array of flags indexed by symbol number. Figure 7-2 enumerates and describes these flags. When an identifier node is found during a traversal, an appropriate flag is set depending on the context in which the identifier appears. A general flag is set to indicate to the algorithm when it has passed the closure function call node in the parent AST, this determines whether CLOSURE_ID_HOST_PRE_ or CLOSURE_ID_HOST_POST_ flags are set.

Once both ASTs have been scanned for identifiers, the parameter list of the new closure function and the argument list of the closure function call are constructed. Generally, any identifier that appears in both must be an argument. The AST of the parent function call has a new argument list expression branch appended; the AST of the closure function root node has a branch appended with the parameter list. Any identifier appearing anywhere in the parent function and used as an L-value is passed by reference. Technically, this condition is too weak: if the identifier is re-defined prior to use in either the closure function or after the closure call in the parent function, it should not need to be passed by reference (except for a potential WAR dependence after the closure). Prior transformation to SSA form will automatically strengthen the pass-by-reference requirements.

Finally, the new closure function definition is recursively fed back into the top level of the middle-end function analysis procedure; therefore, any nested loops are automatically outlined. The algorithm then continues to traverse the original AST to find subsequent loops.
Constructing the function PDG

Each entry in the array of AST nodes created by registering nodes when the AST is built also contains two lists: a list of nodes that are dependent upon the corresponding node as well as the dependence type, and a list of nodes that the current node is dependent upon and the respective dependence types. This algorithm fills out those lists on a per-function basis, creating the function’s PDG. In addition, each entry also contains a pointer to the nearest parent node upon which the node is conditionally (control) dependent. Thus, each node has at most one control dependence, but the control dependence node can itself be conditionally dependent.

The algorithm traverses the AST depth-first, tracking control dependences on the way down (pre-order), and data and side-effect dependences on the way up (post-order).

The most recent control dependence is passed down recursively and assigned to each node as the traversal descends. Each node encountered is assigned the most recent control dependence. When the current node is itself a control dependence, then that node is still labeled with the previous control dependence, but then the current node is passed down as the most recent control dependence. Control dependences can be further distinguished from one another. For example, a while loop guard expression needs to be treated slightly different than the while loop body during code generation.

Data dependences in expressions are straightforward: the parent expression node is dependent upon the children. For example, in the expression $2 + 3$, the $+$ node is data dependent upon the $2$ and $3$ nodes. Data dependence of identifiers is tracked with a per-identifier record indicating the most recent expression involving a volatile AST identifier node of the respective identifier. This includes assignment ($=$) and reference ($\&$) operators. Non-volatile identifiers are not tracked (e.g., a pass-by-value parameter is non-volatile). When a volatile identifier is used in a non-volatile context, a data dependence is created linking the R-value to the volatile expression
node. When an identifier is re-defined, a side-effect dependence is created linking the two definitions. The R-values that occur in between the two definitions will still be correct without an explicit dependence of the re-definition on the R-values because their data dependencies will be fulfilled before the side-effect dependence can be fulfilled. SSA form should reduce or eliminate these types of side-effect dependences when identifiers are re-defined.

As the algorithm traverses the AST, it keeps separate track of the most recently encountered function calls dependent upon the client and those dependent upon the server, as defined by the scan of the callee’s AST for dependences. If a called function is dependent upon both, than both the most recent server and most recent client function calls are updated to point to the newly encountered function. If the callee is dependent upon neither client nor server (it is pure), than neither most recent call pointer is updated. Before updating the pointers, however, a side-effect dependence is created between the current call and the most recent calls with the same site dependences. So, each function call may have between zero to two side-effect dependences. This is where inlining a mixed dependence function may help reduce client/server boundaries in the PDG.

**Binding PDG nodes to a site**

The first step in the partitioning process is to accumulate the “inputs” to a function. These act as starting points in the evaluation of the PDG, and include constant values, parameters, and function calls with no arguments. These are nodes that themselves have no dependences.

The algorithm to bind each node to a specific site is then invoked. Starting with a list of inputs, the algorithm picks an input, and traverses the dependence tree starting from that input. It searches for a node explicitly bound (i.e., a call to a server-bound function) to a particular host. If the input node already has a binding, then that is the host binding that is searched for, otherwise, it searches for a default binding: if the function has any server-side dependences, it searches for
that; otherwise, it searches for a client-side dependence. If the input has a control dependence, it uses the binding that the control dependence was assigned.

As the algorithm searches it keeps track of the path from the input node to the found node. If it cannot find a path to a dependent node with the binding it would like, it inverts its choice of binding and tries again. If it still cannot find a bound node of either type, it adds the input node to a list for later processing. If it is successful in finding a path, it binds every node in the path to the chosen site.

The algorithm for finding the path starts at the given node and then performs a depth-first search of all dependent nodes with all dependences already visited and assigned a binding by the algorithm except for the current node. It stops if it finds a node with the explicit binding for which it was looking. If it cannot find either type of dependent (i.e., child) node, it goes back up the recursion stack to try alternate branches, until it has visited every node, at which point it gives up.

The algorithm to bind the path accepts the stack from the searching algorithm, and uses it to implicitly bind every node in the found path to the chosen host. In the process, it also appends newly orphaned nodes to the list of input nodes to use as starting points for the binding algorithm to continue.

Once all input nodes have been processed, the binding algorithm processes the input nodes from which no path to an explicitly bound node could be found – these are the input nodes which were added to the list for later processing.

First, the algorithm tries to determine an appropriate binding type for which to search. The algorithm checks the type of the control dependence of the node, if any, and looks to see if any previous adjacent node, upon which the current node is dependent, is implicitly bound the same
as the control dependence node. This would be ideal since it would minimize client/server interaction. If not, it looks to see if it can find a previous adjacent dependence with the opposite binding of the control dependence. If there is no implicitly bound control dependence, the control dependence binding value is assumed to be the default binding value passed into the algorithm. If it cannot find an adjacent implicitly-bound dependence, it looks to find a dependent node bound to the control dependence binding.

The search algorithm is then instructed to first search for explicitly bound, and, failing that, implicitly bound nodes that were previously assigned a site binding by the algorithm. If it cannot find a path, it searches again but for the opposite of the desired binding. If it still cannot find anything, it is instructed to search again, but this time it may accept unbound ‘terminal’ nodes – those with no dependents.

When it finds a path, it labels it with the same algorithm used previously, adding newly orphaned nodes to the list for processing until it runs out of nodes.

The host to bind the path is determined as follows: if a terminal node was with no binding found, the control dependence binding is used. If the found node has a binding then that is used to bind the entire path.

When the second input list is exhausted, the algorithm will have implicitly bound every node in the function. However, some touch-up is needed to optimize certain situations.

First, the argument list construction for a function call is bound to the same host as the function callee. The individual expressions are free to be bound to either host, but the actual operations to set up the arguments for the call are made local to the function to avoid multiple unnecessary nested function calls – i.e., so the compiler does not create a function that does nothing but call a function.
Conditional operators are bound to the same host as the conditional expressions upon which they are data-dependent. This ensures that loops do not make unnecessary synchronous remote calls in evaluating the loop guard. The implicit bindings of the function are then re-evaluated with the default site preference given as the loop’s chosen site. Keep in mind that by this point loop outlining has been done, so there is at most one loop per function.

**Partitioning the PDG into site-dependent groups**

Each group is a list of dependent nodes bound to the same site. The algorithm first puts all input nodes into two lists, one for each site. The algorithm then arbitrarily picks a site. To create a group, the algorithm picks a node with no dependences that matches the chosen site, and adds it to the current group, removing it from the list. That node’s dependents are then inspected to see if they have any dependences that are not already in a group; if not, that node is added to the appropriate list of nodes with no dependences. The algorithm continues to grow the current group until it has exhausted the list of independent nodes of the chosen site. Then, it inverts the chosen site and starts a new group. This is continued until both lists are exhausted.

This algorithm allows the groups to be executed serially and ensure that all dependences are met, while minimizing the number of client/server transitions.

Finally, any function argument operators that are in different groups from the function call are moved to the same group as the function call.

**Creating functions out of groups**

The first step is to propagate function call attributes to the group level. Each group has a flag indicating whether or not it may be called asynchronously. First, each group is scanned for function calls and if any of those function calls are prohibited from being made asynchronously (due to side effects), then the group may not be called asynchronously.
Then, for each group, a list of input and output nodes is created. The input nodes are nodes that are in a group upon which the current group is dependent; the output nodes are in the current group. Multiple groups may use the same output node of one group as an input. The dependence type of each input and output is recorded as part of the group record. Now, each group is part of a broader PDG.

The parameter list for each group is then created. First the inputs of the group are scanned for external data and control dependences. If a data dependence is an identifier, than that identifier is used as the parameter identifier, otherwise, a new identifier is created based on the unique number of the input node. This way, we ensure that parameters are not duplicated if an input is used in multiple places in the group.

Then, the output list for the current group is scanned for data and control dependences. If any are found, then a pass-by-reference parameter is added to the parameter list for the group. If the parameter already existed as pass-by-value, then it is simply switched to pass-by-reference. Finally, the parameter list of the parent function is scanned for any identifiers used inside the group, since these will not have explicit dependences, and these are added as pass-by-value or pass-by-reference if volatile as appropriate.

**Compiler Back End**

The back end of the compiler emits instructions into a text file which an assembler accepts as input and translates into a more compact binary file.

The back end is written to be somewhat generic in that it is set up with callbacks to allow different types of traversal: pre- and post-order callbacks allow additional processing, and a callback is used to determine where to go when the compiler reaches the bottom of an expression tree. This allows the same back-end assembler output functionality to work for compiling both from the AST and from the PDG.
Compiling the AST

For pure functions or functions only dependent on one host, the AST is traversed depth-first and compiled as normal.

Compiling the PDG

First, each group is compiled. The function declaration is written along with the parameter setup on the stackframe. Then, the compilation of the code is done by choosing a node with no unevaluated dependencies and traversing depth-first with the code-generator. If the code generator finds it is compiling a node with a data or control dependence upon a node that it did not just visit (and therefore, the value is not on the stack), it will substitute in a temporary register identifier named with the unique identifier for the node upon which the current node is dependent. Similarly, if the code generator sees it is not going to visit a data-dependent node, it will pop the value on the stack into a register variable named for the current node.

As the code-generation moves through each node, it checks the stack of control dependences for that node. As previously discussed, control dependences link to one another, ensuring they stay in a proper nested order. Before the compiler emits instructions for a node, it traverses the linked control-dependence list to compiler a stack of control dependences for the current node. It also has the stack of dependences from the previous node it compiled. The compiler compares the top of the stack to see if they are the same and, if not, pops the top of the larger stack and writes an appropriate label target (for the previous stack) or jump (for the current stack) into the assembly to match. This continues until the stacks match. This allows operations that share control dependences to be widely scattered from one another while maintaining correctness.
The compiler then chooses another node with no uncompiled dependences to begin compilation again, until it has compiled the entire group. Interestingly, these different branches could theoretically be run concurrently.

The main function, which calls each group function, is then directly output. First, if the function is a loop, the label for the tail recursion is created at the beginning. Then, each group is called in the order it was created to preserve dependences. First all the arguments are pushed onto the stack, then the appropriate CALL or ACALL instruction is emitted depending on if the group may be called asynchronously. The return value from the last evaluated group is the one actually returned by the main function.

If the function is a loop, then the loop conditional expression node number that we stored as part of the function definition is used to created a loop conditional value that is explicitly stored in a register variable named for the node. The compiler will then emit code at the end of the main function to compare this value and jump to the loop label if the condition is true.

**Assembler**

The assembler accepts the text assembly and compiles into a more compact binary representation. The binary representation simplifies execution by the VM and reduces the amount of bandwidth required when sending code. The binary file is contains individual records for each function that are loaded separately. When a client requests function code, the function record from the binary file is transmitted.

**Conclusion**

This chapter detailed the Pip compiler implementation of the concepts discussed in previous chapters. The next chapter demonstrates the performance of our implementation.
/* ... 
* #func mysql_connect          proxy
* #func mysql_select_db        async
* #func mysql_query            proxy
* #func mysql_fetch_array      async
* #func mysql_num_rows         async
* #func addslashes             proxy   pure
*/

Figure 7-1. MySQL extension function definitions used by the compiler are located in the comments of the VM’s header file for that extension.

CLOSURE_ID_UNUSED       0 /* Identifier is never referenced. */
CLOSURE_ID_HOST_PARAM   1 /* Id. is a parameter of the parent */
CLOSURE_ID_HOST_PRE_LVALUE  2 /* Appears before closure as L-value */
CLOSURE_ID_HOST_PRE_RVALUE  4 /* Appears before closure as R-value */
CLOSURE_ID_HOST_POST_LVALUE   8 /* Appears after closure as L-value */
CLOSURE_ID_HOST_POST_RVALUE 16 /* Appears after closure as R-value */
CLOSURE_ID_CLOSURE_RVALUE   32 /* Appears in closure as R-value */
CLOSURE_ID_CLOSURE_LVALUE   64 /* Appears in closure as L-value */
CLOSURE_ID_CLOSURE_APPEARS (32 | 64) /* Appears in closure */
CLOSURE_ID_HOST_APPEARS   (1 | 2 | 4 | 8 | 16)/*Appears in parent*/

Figure 7-2. Flags used to mark identifiers found during loop outlining.
This chapter describes a simple application we created to demonstrate the Pip system and the testing methodology and the tests that were used to verify performance increases.

Application

As an example of the system’s potential usefulness in creating end-user network applications, we created two versions of a simple online text editor program (Appendices B and C). Both applications allow the user to create and edit a plain text document. The document may be saved in the server’s database with a name and retrieved for further editing in another session. Both versions of the editor application support collaborative editing of the same document – a key feature of an Internet enabled application – but in different ways.

Collaboration through Differencing

The first version of the editor application (Appendix B) enables collaborative editing by differencing the document when the user chooses to save. If the three-way difference indicates that the user’s copy of the document cannot be merged with the remote copy, the user’s changes are discarded. This can happen when another user edits the same area of the document as the local user, and then saves it before the local user. It is possible to add options to discard the remote copy or manually merge the documents.

Collaboration through Thread Events

The second version of the editor application (Appendix C) takes advantage of Pip’s flexibility in architecting applications. Pip considers each instance of a client-server connection with a given session id to be an independent thread of execution within that session: two connections supplying the same session-id will be part of the same virtual machine “process,” and may communicate using inter-thread communication mechanisms.
In this case, we implemented the collaborative editor using the signal message-passing system. Each thread of the application subscribes to receive a `SIGNAL_UPDATE_TEXT` signal. When a user makes changes, the signal is sent, via the server, to all the subscribed clients, along with the changed text. The communicated changes override any local changes that have not yet been sent out, though two or more users typing simultaneously will create a race condition as to which client’s changes have authority.

**Performance Testing Methodology**

In devising the performance tests, we chose some common programming tasks that might be found in a network application. We implemented these naively of the underlying network performance issues, and in some instances, intentionally worst-case. This allowed us to make performance comparisons of the effectiveness between different configurations of the performance features. In some cases, optimized programs were also coded, to allow us to compare the automatic optimizations of the sub-optimal implementations with the ideal cases.

The Pip virtual machine execution engine contains a profiling feature that reports the time taken to complete a transaction. To time the total runtime of a program, we can use the transaction time of the initial **EXECUTE** request. This transaction represents the total running time of a program and a final response is issued only when the program has terminated. The request is made even if it must be fulfilled locally (when the code is cached on the client and the server is unavailable). We are also capable of analyzing the response times of individual synchronous requests, and the time until the first provisional response.

**Simulating Latency**

Consistent test results would be difficult to obtain using the Internet since latency between two hosts may fluctuate greatly. Further, we wanted to be able to repeat tests with different fixed latencies. Therefore, it was necessary to simulate latency in a controlled environment. As an
additional requirement, since we were concerned only with latency and not with the effects of bandwidth consumption, available bandwidth had to be virtually unlimited – or at least, high enough such that the impact on performance was negligible compared to latency.

Therefore, we decided to run both hosts on a single machine. We had to run one host inside a VMWare virtual machine since it appeared Nagle’s algorithm could not be disabled on the local interface (our implementation of Pip uses TCP and Nagle’s algorithm must be disabled for optimum performance due to its use of small, sporadic messages [Mogul and Minshall 2001]).

With latency minimized and bandwidth maximized, we needed a way to easily and consistently add latency to the system. To do so, we used the Linux Traffic Control utility, tc\(^1\). This allowed to us easily change latency from the command line:

```
tc qdisc add dev eth0 root netem delay 10ms
```

The above command adds 10 milliseconds of one-way latency to any messages sent or received over the first Ethernet interface. Since tc operates on a very low level in the network stack, the effect of this command can be verified with the ping command, which reports twice the additional latency from tc. The time to receive the final response from a null transaction in the Pip engine will report approximately the same time as ping.

**Test Parameters**

Our test configuration gave us close to zero base latency and unlimited bandwidth, save for occasional, insignificant (less than 10 millisecond), latency spikes due to system load. These hiccups can be thrown out of the raw data as outliers, or left to be averaged in since the impact on overall performance is negligible. Generally, extreme outliers (time differences of more than 10\%) were discarded, and the first runs were always discarded to allow for the disk cache to have

\(^1\) Hubert et al. Linux Advanced Routing & Traffic Control [http://lartc.org/](http://lartc.org/) (June 2009)
cached the necessary files for the tests. Excluding first runs and outliers, for each test, the times of five runs were averaged together to obtain the final results.

**Tests and Results**

**Benchmark 1 - Interleaved Independent Client and Server Function Calls**

We constructed a simple program which makes a series of alternating calls to a server-side function and a client-side function (Figure 8-1). The server-side calls are each dependent upon the previous server-side call, and the client-side function calls are each dependent upon the previous client-side function call. However, the client-side functions are never dependent upon any server-side functions, and vice versa. This is the purest example of a program that would benefit from dependence analysis optimizations: the imperative logic dictates that the remote calls be interleaved with local ones, while the dependence analysis shows that they are independent and therefore may be grouped resulting in one local and one remote call.

**Effect of dependence analysis and pre-caching mobile code**

This test compares the program running with and without pre-cached mobile code. If the client does not have the code for a function designated to run on the client, a transaction is initiated to retrieve it. Code is pre-cached when the client already has the code on disk when the program is started. In this case, the request to retrieve the code will not be made.

Executing the program when compiled with and without dependence analysis is also compared. Dependence analysis should transform all the separate client and server functions into one function each for the client and server, greatly reducing the number of remote requests made.

The cumulative effect of these optimizations can be seen in Figure 8-2.

**Effect of number of transactions**

For this test we modified the benchmark program to contain different numbers of paired client and server calls, and ran the program with no optimizations with different simulated
latency values. The intent is to determine the impact that the number of transactions combined with latency has on performance.

At higher latencies, there is a clear linear impact from both latency and number of transactions on execution time (Figure 8-3). If latency is the dominating performance factor, then the run time should be asymptotic with latency multiplied by number of transactions.

**Benchmark 2 – Loop Execution and Asynchronous RPC**

The second benchmark (Figure 8-4) is designed to test how the system behaves executing loops. The first loop should execute entirely on the server; the second loop executes on the server but make asynchronous requests to the client to execute the `sys_print` function.

**Effect of asynchronous versus synchronous call in loop body.** For this test, we simulated 25 milliseconds of one-way latency and measured total execution time of the program for different numbers of loop iterations. This test was conducted both with and without asynchronous requests enabled (Figure 8-5).

The results demonstrate that, again, synchronous requests correlate linearly with execution time. Enabling asynchronous requests makes a huge difference in performance, and appears to scale slightly non-linearly (Figure 8-6), probably due to TCP flow control.

**Benchmark 3 – Loops with Different Guard Dependences**

For benchmark 3 (Figure 8-7), we modified benchmark 2 so that the first loop had a guard dependent on the client and a loop body dependent on the server. We left the second loop in the same function with a guard dependent on the server and a loop body dependent on the client.

**Effect of dependence analysis versus asynchronous calls.** For this test, we ran the benchmark with 50 loop iterations. We used 0 and 25 milliseconds of simulated one-way latency, since we have already established the behavior with respect to synchronous and asynchronous
requests as latency increases further. The test was run with asynchronous requests and dependence analysis enabled, and with only asynchronous requests.

Figure 8-8 shows the results: dependence analysis was able to turn the loop with the client-dependent guard into a client-bound function and, therefore, execute the loop body as asynchronous requests. Without dependence analysis, the entire function is executed server-side, and a synchronous request must be made each loop iteration to check the loop guard. In this case, since there were 50 loop iterations, this was 50 synchronous requests that can be executed asynchronously due to the dependence analysis and loop transformation.

Conclusion

This chapter demonstrated the performance improvements due to our optimization techniques. The next chapter discusses the overall effectiveness of these optimizations and possible improvements as future work.
server fs($x) {
    return $x * 2;
}
client fc($x) {
    return $x + 1;
}
main() {
    $c = 1;
    $s = 2;
    $s = fs($s);
    $c = fc($c);
    $s = fs($s);
    $c = fc($c);
    $s = fs($s);
    $c = fc($c);
}

Figure 8-1. Benchmark program 1.

![Graph showing effect of pre-caching and dependence analysis on completion time versus one-way latency]  
Figure 8-2. Effect of dependence analysis and pre-caching of mobile code from benchmark 1.
Figure 8-3. Performance impact of latency as the number of synchronous requests increase.

```php
import server:mysql;
import client:system;

main()
{
    $link = mysql_connect("localhost", "root", "123");
    mysql_select_db($link, "TestData");
    mysql_query($link, "DELETE FROM BenchCounter");
    $x = 0;
    while ($x < 50)
    {
        mysql_query($link,
            "INSERT INTO BenchCounter SET Counter = " . $x);
        $x = $x + 1;
    }
    $result = mysql_query($link, "SELECT * FROM BenchCounter");
    while ($row = mysql_fetch_array($result))
    {
        sys_print($row["Counter"]);
    }
    return 0;
}
```

Figure 8-4. Benchmark program 2.
Figure 8-5. Execution times for benchmark 2 with and without asynchronous requests for different numbers of loop iterations.

Figure 8-6. Closer examination of benchmark 2 with asynchronous requests reveals slightly non-linear behavior.
import server:mysql;
import client:system;
main()
{
$link = mysql_connect("localhost", "root", "123");
mysql_select_db($link, "TestData");
mysql_query($link, "DELETE FROM BenchCounter");
while (($x = sys_get_number()) < 50)
{
    mysql_query($link,
        "INSERT INTO BenchCounter SET Counter = ". $x);
}
$result = mysql_query($link, "SELECT * FROM BenchCounter");
while ($row = mysql_fetch_array($result))
{
    sys_print($row["Counter"]);
}
return 0;
}

Figure 8-7. Benchmark program 3.

Figure 8-8. Execution times of benchmark 3 with and without dependence analysis loop transformation.
CHAPTER 9
CONCLUSIONS AND FUTURE WORK

This chapter summarizes the work completed and possibilities for improving the system in the future.

Conclusions

Overall, the system works as expected: clients can connect and run server-side applications through code-on-demand and RPC mechanisms. There is limited offline support: unavailable functions return null.

As for performance, which is the focus of the study, we have demonstrated large improvements in performance of code that has been written without consideration of the underlying distributed mechanisms versus naïve RPC. The mechanisms that have enabled these transparent improvements include:

- Automatically identifying functions that may be called asynchronously,
- Use of a proxy type for values that must be locally referenced but only remotely evaluated, and
- Dependence analysis and functional transformation to isolate independent tasks.

We have also implemented several other features in Pip towards the goal of demonstrating the feasibility of developing Rich Internet Applications using local application development models.

Extension Libraries

A basic GTK+ library and MySQL client library have been created as sample extensions. GTK+\(^2\) is a cross-platform GUI toolkit which supports Windows, Linux, and OS X; MySQL\(^3\) is an SQL RDBMS popular in web application development.


Glade\textsuperscript{4} is an application to develop GTK+ user interfaces. Glade creates XML files that the libglade library can parse and run. The Glade system has been integrated into Pip to enable RAD support similar to Flex and Silverlight’s user interface designers.

The system library enables access to the server or client file systems; the signal library enables inter-thread communication via events.

Future Work

The Pip language still lacks some features such as floating point capabilities, object classes, and more advanced offline execution features.

Performance

While we have defined a general framework of automatic optimization, there is room to build on the foundation we have established.

Loop structures

There is still room for improvement with respect to performance. Consider the case of a loop with a host-independent guard. The body of said loop contains two independent, host-dependent tasks, e.g., \( i = 0; \) while \( (i < 10) \) \{ \( f_s(); \) \( f_c(); \) \}. In this case, the loop will make 10 remote requests (albeit, they may be asynchronous). However, we know in advance that the call will be made 10 times.

Currently, Pip only supports a basic while loop structure. However, it would be straightforward to implement several higher-level loop structures which the compiler can easily optimize. If designed correctly, these loop structures, while designed to improve performance, may be completely transparent to developer, since they will simplify his job, as well.

\textsuperscript{4} Glade – a User Interface Designer for GTK+ and GNOME. http://glade.gnome.org/ (June 2009)
Our previous example could be re-written with such a structure: `foreach (i in [0..9]) { .. }`. Here, we have devised a new loop structure, `foreach`, which iterates over a fixed set of values. The structure could also accept arrays and variable values, so long as the guard was defined to remain invariant during execution of the loop. The developer will use this structure because it is easier and cleaner in the code; therefore, by giving it a secondary purpose for the developer, its primary purpose of enabling optimization remains transparent and automatic.

**Inlining and SSA form**

We have discussed the merits of inlining and transformation to SSA form in Chapter 5, but this currently remains unimplemented. We believe there would be a small performance gain to be seen in larger programs and those programs that frequently re-define variables; however, this would be an incremental improvement.

**Code Mobility**

Currently, the client may request function code from the server. The client will cache the code in memory for that session, but it does not cache the code in persistent storage: the next time the application is run, the client-side code must be retrieved again. The client should cache the code it retrieves persistently. Once per session, instead of retrieving code, a request can be made to retrieve the date on which the server-side version of the module was compiled. If the server is determined to be running a newer version of a module, the client will purge that module's cached function code. This is similar to an HTTP HEAD request for cached GET responses.

In addition, to increase performance and usability, the server should be able to asynchronously send function code to the client in anticipation of future function calls. The compiler should create a list of functions that are called by each function. This list can be used
by the server to prioritize the order in which code is sent to the client, based on the currently executing function.

**Resource Management**

The resource handle system described previously will be augmented by a distributed reference counting system. Resource handles transmitted to the remote site will be marked, and the remote site will be responsible for reference counting the remote resource handle. If the count of the remote handle reaches zero, the originating site will be notified.

Resource types will be registered by either the VM or an extension. The reference counting engine will free extension-defined resources through a callback supplied when the type is registered. References to dynamic structures will also be counted. Currently, the system uses only pass-by-value, but the addition of pass-by-reference will necessitate reference counting.

**Security**

Security is an important issue when developing general use Internet applications. Both the client and the server must remain secure. However, in a heterogeneous environment, there will be different requirements for securing each.

The client host must be protected from unauthorized use of native resources, or, more generally, non-permitted behavior. A malicious application must not be allowed to “escape” the VM sandbox environment. This must be balanced with the need for applications to effectively use native resources.

The server host must be protected from malicious users who would seek unauthorized access to native functionality, and also from those who would seek to disrupt the availability of the server.

In addition, the client and server may each have access to confidential data. These data may vary from explicitly compromising data such as passwords and bank account numbers, to
potentially revealing information such as SQL queries, file paths, and data addresses to private user data. These data can be categorized as either data that are shared and must be protected from third parties, or data that must also be protected from the other host.

The security requirements are:

4. The client must allow only explicitly permitted access to native resources.

5. To ensure the security of (1), the client must be able to grant access to only specific applications or providers.

6. Requirement (2) necessitates the existence of a mechanism whereby the client can be guaranteed that the application is provided by a trusted source and that the application code that the client executes has not been modified.

7. Confidential data transmitted between parties must be secure from third parties.

8. Confidential data on either host site must be protected from transmission to or modification by the corresponding remote site.

9. Host resources must not be able to be manipulated in ways that would compromise the security of the sandbox environment.

10. Server resources must not be able to be manipulated in ways that would compromise the availability of the server.

Requirements (1) and (2) are satisfied by the security descriptors system: the user must explicitly grant an application access to resources.

Requirement (3) would be satisfied by a code-signing system: application code is transmitted with a cryptographic hash to verify its authenticity. The hash is encrypted with a private key by the provider who distributes their public key to be used to decrypt and authenticate the hash.

Requirement (4) can be satisfied by use of a secure lower-level protocol such as SSL.

Requirement (5) is taken care of by the resource handle system. Instead of raw pointers or other data being transmitted to the remote server when migrating state or passing RPC arguments, the unique resource handle numbers are used. The different resources themselves are
typed, so an extension has the ability to ensure that a resource handle passed as an argument to one of its functions originated in that same extension and is being used as intended. This also partially satisfies requirement (6).

A security advantage of web development’s continuation model is that allocated resources, such as database connections and file handles, are usually freed before exiting to the client. That is, each server operation will atomically allocate and release its resources. A developer would have to go out of his way to store resources in a session, creating the opportunity for a denial-of-service attack in which repeated requests could be made by clients to allocate many server resources without freeing them. However, dispensing with the continuation model and making the distinction between client and server more transparent to the developer creates opportunities to exploit server resources to affect denial-of-service attacks. A modified client engine could make repeated requests to run code on the server to allocate database connections, for example.

First, we can attempt to mitigate this problem through an artificial constraint on the developer: execution of server-native extension functions must be called only from within functions that are declared server-bound. The advantage is that the server will be able to deny any client requests to directly call native functions. This prevents the client from directly manipulating server resources. A naïve developer will still be able to create security vulnerabilities by creating shallow wrapper functions around native functionality, constructing queries and file paths in site-neutral code. The advantage of removing transparency in this situation is that the developer should be more aware that by by-passing these restrictions, he is going out of his way to create a vulnerability, to the point where this system would be no less inherently insecure than a request/response architecture. However, this technique interferes with
the location transparency and may not be necessary in secured environments with more trusted client hosts. The restriction could be easily toggled in server-side configuration.

A more advanced and rigorous technique to prevent exploitation of server-side resources, in addition to the above described restriction, involves the server utilizing information about the order that native calls are to be made. Consider the case that the server knows all of the possible paths of execution a given program might take. Whenever the server processes a native function call, it can check the paths of execution to verify that the call history up to and including the latest call matches at least one of the possible paths. However, this is not a feasible solution: due to loops and conditionals, there may be a near infinite number of paths. Remember that we are primarily interested in ensuring that functions that allocate resources are not called haphazardly – in most properly written applications, they will be followed later by a function that frees the resource before another is allocated.

The code can be statically analyzed by the compiler to determine the order of function execution – this feature would already be used by the asynchronous mobile code caching system previously described. For each native function, a list would be compiled consisting of function calls that might precede or follow – that is, the function calls are behind conditionals or loops. The creation of the list in either direction terminates when encountering a function call that is not behind a conditional or loop. The idea is that any operations for which the order is important, like allocation and freeing of resources, would book-end lists of order-independent operations. Now, when the server encounters a request for a native function execution, it can traverse the resulting tree structure to verify both that the requested call follows the previous call and that the previous call precedes the requested call.
Consider the function pseudo-code below:

```
allocate
A:    ...
    if x
        modify_1
    if y
        modify_2
    if z
        goto A
free
```

In this example, the `allocate` and `free` operations book-end permutations of `modify_1` and `modify_2` operations. Assuming that this function is called arbitrarily in the higher level program, `allocate` must be preceded by either nothing (first native call made) or `free`, since `free` must be called before exiting the function, which may be called repeatedly. Similarly, `free` must be preceded by `allocate`, `modify_1`, or `modify_2`. Conversely, it cannot be preceded by another `free`. The `modify_X` functions must be either preceded by another `modify_X` or an `allocate`.

Extension authors may also implement some level of security in this realm, by keeping track of allocated resources on a per-session basis and denying requests for excessive resources.

These techniques work on single sessions or single threads. There is nothing intrinsic to stop an attacker from instantiating many sessions or threads except perhaps a hard limit on the number of sessions or number of threads per session. However, attacks of this type would not be unique to this system and methods exist to prevent and mitigate such attacks. Resources allocated by partially created sessions without extant connections would be automatically freed by the reference counting system.
APPENDIX A
LANGUAGE GRAMMAR

The Pip language is based on PHP and has a similar grammar.

program:
  directive_list function_list

directive_list:
  directive_list directive
  | ε

directive:
  import
  | define

import:
  KW_IMPORT import_binding ':' IDENTIFIER ';

define:
  KW_DEFINE '(' IDENTIFIER ',' constant_expr ')' ';

function_list:
  function_list function
  | ε

function:
  function_binding IDENTIFIER '(' parameter_list ')' stmt_block

parameter_list:
  VARIABLE
  | parameter_list ',' VARIABLE
  | ε

function_binding:
  binding

import_binding:
  binding

binding:
  'server'
  | 'client'
  | 'auto'
  | ε

stmt:
  declr_stmt
  | expr_stmt
  | KW_RETURN ';
  | KW_RETURN expr ';
  | KW_PRINT '(' expr ')' ';
  | KW_WHILE '(' expr ')' stmt
  | KW_DO stmt KW_WHILE '(' expr ')' stmt
  | KW_FOR '(' expr_stmt expr_stmt expr ')' stmt
  | KW_IF '(' expr ')' stmt
| KW_IF '(' expr ')' stmt KW_ELSE stmt
| stmt_block

stmt_list:
  stmt
  | stmt_list stmt

stmt_block:
  '{' stmt_list '}'

argument_list:
  expr
  | argument_list ',' expr
  | ε

function_call:
  IDENTIFIER '(' argument_list ')

constant_expr:
  INTEGER
  | STRING
  | VARIABLE
  | IDENTIFIER

primary_expr:
  function_call
  | constant_expr
  | '(' expr ')

postfix_expr:
  primary_expr
  | postfix_expr '[' expr ']
  | postfix_expr OP_INC
  | postfix_expr OP_DEC

unary_expr:
  postfix_expr
  | OP_INC unary_expr
  | OP_DEC unary_expr
  | '-' expr
  | '!' expr

multiplicative_expr:
  unary_expr
  | multiplicative_expr '*' unary_expr
  | multiplicative_expr '/' unary_expr
  | multiplicative_expr '%' unary_expr

additive_expr:
  multiplicative_expr
  | additive_expr '.' multiplicative_expr
  | additive_expr '+' multiplicative_expr
  | additive_expr '-' multiplicative_expr

relational_expr:
  additive_expr
| relational_expr '<' additive_expr
| relational_expr '>' additive_expr
| relational_expr LE additive_expr
| relational_expr GE additive_expr

equality_expr:
    relational_expr
    | equality_expr EQ relational_expr
    | equality_expr NE relational_expr

logical_and_expr:
    equality_expr
    | logical_and_expr OP_AND equality_expr

logical_or_expr:
    logical_and_expr
    | logical_or_expr OP_OR logical_and_expr

conditional_expr:
    logical_or_expr
    | logical_or_expr '?' expr ':' conditional_expr

assignment_expr:
    conditional_expr
    | unary_expr '=' assignment_expr

expr:
    assignment_expr

declr_stmt:
    KW_GLOBAL VARIABLE ';

expr_stmt:
    expr ';
    | ';'
APPENDIX B
EDITOR APPLICATION WITH COLLABORATION BY DIFFERENCING

/*
 * Simple text editor demo application.
 */

import client:gtk;
import server:mysql;
import server:system;

define(W_MAIN, "main_window");
define(W_TEXTVIEW, "textview");
define(W_ABOUTDIALOG, "about_dialog");

define(AS_FILENAME, "filename");
define(AS_MODIFIED, "modified");
define(AS_BUILDER, "builder");
define(AS_LASTTEXT, "last-text");

update_title_bar($app_state)
{
    $title = "[" . $app_state[AS_FILENAME] . "] - Pip Text Editor";
    if ($app_state[AS_MODIFIED])
        $title = "* " . $title;

    gtk_window_set_title(gtk_builder_get_object($app_state[AS_BUILDER], W_MAIN), $title);
}

check_quit($app_state)
{
    if (!$app_state[AS_MODIFIED] || GTK_RESPONSE_YES ==
        gtk_message_box("Are you sure you wish to quit?", "Pip Editor",
        GTK_MESSAGE_QUESTION, GTK_BUTTONS_YES_NO))
    {
        gtk_main_quit();
        return TRUE;
    } else
        return FALSE;
}

on_filequit_activate($widget, $app_state)
{
    check_quit($app_state);
}

on_delete_event($widget, $event, $app_state)
{
    return (!check_quit($app_state));
}

on_helpabout_activate($widget, $app_state)
{
$dialog = gtk_builder_get_object($app_state[AS_BUILDER], W_ABOUTDIALOG);
    gtk_dialog_run($dialog);
    gtk_widget_hide($dialog);
}

database_connect()
{
    $link = mysql_connect("localhost", "root", "123");
    mysql_select_db($link, "TestData");
    return $link;
}

load_file($filename)
{
    $link = database_connect();
    $query = "SELECT docText FROM TextDocuments WHERE docName = " . addslashes($filename) . "";
    $result = mysql_query($link, $query);
    if (mysql_num_rows($result))
        $text = mysql_fetch_array($result)['docText'];
    else
        $text = FALSE;
    return $text;
}

build_file_list($treeview)
{
    gtk_tree_view_list_new($treeview);
    $link = database_connect();
    $query = "SELECT docName FROM TextDocuments ORDER BY docModified DESC";
    $result = mysql_query($link, $query);
    while ($row = mysql_fetch_array($result))
    {
        gtk_tree_view_list_append($treeview, $row['docName']);
    }
}

set_document_text($app_state, $text)
{
    $textview = gtk_builder_get_object($app_state[AS_BUILDER], W_TEXTVIEW);
    gtk_text_view_set_text($textview, $text);
}

on_fileopen_activate($widget, $app_state)
{
    $dialog = gtk_builder_get_object($app_state[AS_BUILDER], "server_open_dialog");
    $treeview = gtk_builder_get_object($app_state[AS_BUILDER], "server_open_treeview");
    build_file_list($treeview);
}
gtk_dialog_run($dialog);
gtk_widget_hide($dialog);

$file = gtk_tree_view_list_get_selection($treeview);
if ($file)
{
    $text = load_file($file);
    set_document_text($app_state, $text);
    $app_state[AS_FILENAME] = $file;
    $app_state[AS_MODIFIED] = FALSE;
    $app_state[AS_LASTTEXT] = $text;
    update_title_bar($app_state);
}

get_document_text($app_state)
{
    $textview = gtk_builder_get_object($app_state[AS_BUILDER],
    W_TEXTVIEW);
    $text = gtk_text_view_get_text($textview);
    return $text;
}

save_file($document_data, $filename)
{
    $link = database_connect();

    $query = "SELECT docId FROM TextDocuments WHERE docName = '' .
    addslashes($filename) . '''";
    $result = mysql_query($link, $query);
    if (mysql_num_rows($result))
    {
        $docId = mysql_fetch_array($result)["docId"];
        $query = "UPDATE TextDocuments SET docText = '' .
    addslashes($document_data) . ''' , docCreated = docCreated, docModified = NOW() WHERE docId = '' . $docId;
    } else
    {"query = "INSERT INTO TextDocuments SET docName = '' .
    addslashes($filename) . ''' , docText = '' . addslashes($document_data) . ''' ,
    docCreated = NOW() , docModified = NOW()"";
    mysql_query($link, $query);
    return TRUE;
}

app_save($app_state)
{
    /* attempt to merge user's changes with what's in the database. */
    $t_new = get_document_text($app_state);
    $t_original = $app_state[AS_LASTTEXT];
    $t_db = load_file($app_state[AS_FILENAME]);

    $t_merged = $t_new;
    $t_excluded = FALSE;
if ($t_db)
{
    file_put_contents("/tmp/cedit-original", $t_original);
    file_put_contents("/tmp/cedit-new", $t_new);
    file_put_contents("/tmp/cedit-db", $t_db);
    $t_merged = shell("diff3 -m -3 /tmp/cedit-db /tmp/cedit-
    original /tmp/cedit-new");
    $t_excluded = shell("diff3 -x /tmp/cedit-db /tmp/cedit-
    original /tmp/cedit-new");
}
if (save_file($t_merged, $app_state[AS_FILENAME]))
{
    $app_state[AS_MODIFIED] = FALSE;
    $app_state[AS_LASTTEXT] = $t_merged;
    set_document_text($app_state, $t_merged);
    update_title_bar($app_state);
    if ($t_excluded)
        gtk_message_box("Some of your changes could not be
merged.\n\nThe following changes were discarded:\n\n" . $t_excluded);
}

on_filesaveas_activate($widget, $app_state)
{
    $dialog = gtk_builder_get_object($app_state[AS_BUILDER],
    "server_saveas_dialog");
    gtk_dialog_run($dialog);

    $app_state[AS_FILENAME] =
    gtk_entry_get_text(gtk_builder_get_object($app_state[AS_BUILDER],
    "server_saveas_entry"));
    app_save($app_state);
    gtk_widget_hide($dialog);
}

on_filesave_activate($widget, $app_state)
{
    if ("Untitled" == $app_state["filename"])
        return on_filesaveas_activate($widget, &$app_state);
    else
        app_save($app_state);
}

on_buffer_changed($widget, $app_state)
{
    if (!$app_state[AS_MODIFIED])
    {
        $app_state[AS_MODIFIED] = TRUE;
        update_title_bar($app_state);
    }
}

main()
gtk_init("cedit1");
$builder = gtk_builder_new();

$link = database_connect();
$query = "SELECT uiXml FROM UI WHERE uiName = 'cedit1'";
$result = mysql_query($link, $query);
$ui = mysql_fetch_array($result)['uiXml'];

/* global application state */
$app_state[AS_BUILDER] = $builder;
$app_state[AS_MODIFIED] = FALSE;
$app_state[AS_FILENAME] = "Untitled";
$app_state[AS_LASTTEXT] = "";

gtk_builder_add_from_string($builder, $ui);
gtk_builder_connect_signals($builder, &$app_state);

$buffer = gtk_text_view_get_buffer(gtk_builder_get_object($builder, W_TEXTVIEW));
g_signal_connect($buffer, "changed", on_buffer_changed, &app_state);

$window = gtk_builder_get_object($builder, W_MAIN);
gtk_widget_show($window);

gtk_main();
APPENDIX C
EDITOR APPLICATION WITH COLLABORATION BY THREAD EVENTS

/*
 * Simple text editor demo application.
 */

import client:gtk;
import server:mysql;
import server:system;
import server:signal;

define(W_MAIN, "main_window");
define(W_TEXTVIEW, "textview");
define(W_ABOUTDIALOG, "about_dialog");

define(AS_FILENAME, "filename");
define(AS_MODIFIED, "modified");
define(AS_BUILDER, "builder");
define(AS_LASTTEXT, "last-text");

define(SIGNAL_UPDATE_TEXT, "update-text");

update_title_bar($app_state)
{
    $title = "[" . $app_state[AS_FILENAME] . "] - Pip Text Editor";
    if ($app_state[AS_MODIFIED])
        $title = "* " . $title;
    gtk_window_set_title(gtk_builder_get_object($app_state[AS_BUILDER], W_MAIN), $title);
}

check_quit($app_state)
{
    if (!$app_state[AS_MODIFIED] || GTK_RESPONSE_YES ==
        gtk_message_box("Are you sure you wish to quit?", "Pip Editor",
        GTK_MESSAGE_QUESTION, GTK_BUTTONS_YES_NO))
    {
        gtk_main_quit();
        return TRUE;
    } else
        return FALSE;
}

on_filequit_activate($widget, $app_state)
{
    check_quit($app_state);
}

on_delete_event($widget, $event, $app_state)
{
    return (!check_quit($app_state));
}
on_helpabout_activate($widget, $app_state)
{
    $dialog = gtk_builder_get_object($app_state[AS_BUILDER],
W_ABOUTDIALOG);
    gtk_dialog_run($dialog);
    gtk_widget_hide($dialog);
}

database_connect()
{
    $link = mysql_connect("localhost", "root", "123");
    mysql_select_db($link, "TestData");
    return $link;
}

load_file($filename)
{
    $link = database_connect();
    $query = "SELECT docText FROM TextDocuments WHERE docName = '" .
addslashes($filename) . '"";
    $result = mysql_query($link, $query);
    if (mysql_num_rows($result))
        $text = mysql_fetch_array($result)["docText"];  
    else
        $text = FALSE;
    return $text;
}

build_file_list($treeview)
{
    gtk_tree_view_list_new($treeview);
    $link = database_connect();
    $query = "SELECT docName FROM TextDocuments ORDER BY docModified
DESC";
    $result = mysql_query($link, $query);
    while ($row = mysql_fetch_array($result))
    {
        gtk_tree_view_list_append($treeview, $row["docName"]);
    }
}

set_document_text($app_state, $text)
{
    $textview = gtk_builder_get_object($app_state[AS_BUILDER],
W_TEXTVIEW);
    gtk_text_view_set_text($textview, $text);
}

on_fileopen_activate($widget, $app_state)
{
    $dialog = gtk_builder_get_object($app_state[AS_BUILDER],
"server_open_dialog");
}
$treeview = gtk_builder_get_object($app_state[AS_BUILDER],
"server_open_treeview");
build_file_list($treeview);

gtk_dialog_run($dialog);
gtk_widget_hide($dialog);

$file = gtk_tree_view_list_get_selection($treeview);
if ($file)
{
    $text = load_file($file);
    set_document_text($app_state, $text);
    $app_state[AS_FILENAME] = $file;
    $app_state[AS_MODIFIED] = FALSE;
    $app_state[AS_LASTTEXT] = $text;
    update_title_bar($app_state);
}

get_document_text($app_state)
{
    $textview = gtk_builder_get_object($app_state[AS_BUILDER],
"W_TEXTVIEW");
    $text = gtk_text_view_get_text($textview);
    return $text;
}

save_file($document_data, $filename)
{
    $link = database_connect();
    $query = "SELECT docId FROM TextDocuments WHERE docName = " . 
            addslashes($filename) . "";
    $result = mysql_query($link, $query);
    if (mysql_num_rows($result))
    {
        $docId = mysql_fetch_array($result)["docId"];
        $query = "UPDATE TextDocuments SET docText = " . 
                 addslashes($document_data) . ", docCreated = docCreated, docModified = NOW()" . 
                 WHERE docId = " . $docId;
    } else
    {
        $query = "INSERT INTO TextDocuments SET docName = " . 
                 addslashes($filename) . "", docText = "" . addslashes($document_data) . "", 
                 docCreated = NOW(), docModified = NOW()";
        mysql_query($link, $query);
    
        return TRUE;
    }

app_save($app_state)
{
    /* attempt to merge user's changes with what's in the database. */
    $t_new = get_document_text($app_state);
    $t_original = $app_state[AS_LASTTEXT];
    $t_db = load_file($app_state[AS_FILENAME]);
if ($t_db)
{
    file_put_contents("/tmp/cedit-original", $t_original);
    file_put_contents("/tmp/cedit-new", $t_new);
    file_put_contents("/tmp/cedit-db", $t_db);
    $t_merged = shell("diff3 -m -3 /tmp/cedit-db /tmp/cedit-
original /tmp/cedit-new");
    $t_excluded = shell("diff3 -x /tmp/cedit-db /tmp/cedit-
original /tmp/cedit-new");
} else
{
    $t_merged = $t_new;
    $t_excluded = FALSE;
}

if (save_file($t_merged, $app_state[AS_FILENAME]))
{
    $app_state[AS_MODIFIED] = FALSE;
    $app_state[AS_LASTTEXT] = $t_merged;
    set_document_text($app_state, $t_merged);
    update_title_bar($app_state);

    if ($t_excluded)
        gtk_message_box("Some of your changes could not be
merged.\n\nThe following changes were discarded:\n\n" . $t_excluded);

    return TRUE;
}

on_filesaveas_activate($widget, $app_state)
{
    $dialog = gtk_builder_get_object($app_state[AS_BUILDER],
"server_saveas_dialog");
    gtk_dialog_run($dialog);

    $app_state[AS_FILENAME] =
    gtk_entry_get_text(gtk_builder_get_object($app_state[AS_BUILDER],
"server_saveas_entry"));
    app_save(&$app_state);
    gtk_widget_hide($dialog);
}

on_filesave_activate($widget, $app_state)
{
    if ("Untitled" == $app_state["filename"])
        return on_filesaveas_activate($widget, &$app_state);
    else
        app_save(&$app_state);
}

client on_buffer_changed($widget, $app_state)
{
    if (!$app_state[AS_MODIFIED])
    {
        $app_state[AS_MODIFIED] = TRUE;
}
update_title_bar($app_state);
}

signal(SIGNAL_UPDATE_TEXT, get_document_text($app_state));
}
don_signal_update_text($signal, $text, $textview)
{
    gtk_text_view_set_text($textview, $text);
}
main()
{
    gtk_init("cedit2");
    $builder = gtk_builder_new();
    $link = database_connect();
    $query = "SELECT uiXml FROM UI WHERE uiName = 'cedit1'";
    $result = mysql_query($link, $query);
    $ui = mysql_fetch_array($result)['uiXml'];

    /* global application state */
    $app_state[AS_BUILDER] = $builder;
    $app_state[AS_MODIFIED] = FALSE;
    $app_state[AS_FILENAME] = "Untitiled";
    $app_state[AS_LASTTEXT] = "";

    gtk_builder_add_from_string($builder, $ui);
    gtk_builder_connect_signals($builder, &app_state);

    $buffer = gtk_text_view_get_buffer(gtk_builder_get_object($builder, W_TEXTVIEW));
    g_signal_connect($buffer, "changed", on_buffer_changed, &app_state);
    $window = gtk_builder_get_object($builder, W_MAIN);
    gtk_widget_show($window);

    $textview = gtk_builder_get_object($app_state[AS_BUILDER], W_TEXTVIEW);
    signal_register(SIGNAL_UPDATE_TEXT, on_signal_update_text, $textview,
    $textview);
    gtk_main();
}
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Shawn Outman was born in Virginia but quickly immigrated to Florida for the weather. He has spent two thirds of his life in Gainesville and two thirds of his time in Gainesville attending the University of Florida, where he received his bachelor’s and master’s in computer engineering, and now, hopefully, his doctorate. He now lives on the beach in Melbourne, Florida, where the weather is slightly better.