

Automatic Client-Server Partitioning of Data-Driven Web Applications

[Demonstration Paper]

Nicholas Gerner, Fan Yang, Alan Demers, Johannes Gehrke, Mirek Riedewald,
Jayavel Shanmugasundaram
Cornell University
Ithaca, New York

{nsg7, yangf, ademers, johannes, mirek, jai}@cs.cornell.edu

ABSTRACT

Current application development tools provide completely different programming models for the application server (e.g., Java and J2EE) and the client web browser (e.g., JavaScript and HTML). Consequently, the application developer is forced to partition the application code between the server and client at the time of writing the application. However, the partitioning of the code between the client and server may have to be changed during the evolution of the application for performance reasons (it may be better to push more functionality to the client), for correctness reasons (data that conflicts with multiple clients cannot always be pushed to clients), and for supporting clients with different computing power (browsers on desktops vs. PDAs). Since the client and server use different programming models, moving application code from client to server (and vice versa) reduces programmer productivity and also has the potential to introduce concurrency bugs. In this demonstration, we advocate an alternative solution to this problem: we propose developing applications using a unified declarative high-level language called Hilda [3], and show how a Hilda compiler can automatically (and correctly) partition Hilda code between the client and the server using a real Course Management System application [2]. We illustrate our techniques using two clients: a powerful laptop machine and a less powerful PDA.

1. INTRODUCTION

An important class of applications are *data-driven web applications*, i.e., web applications that run on top of a back-end database system. Examples of such applications include online shopping sites, online auctions, and business-to-business portals. Data-driven web applications are usually structured in three tiers, with a database system that stores persistent data as the lowest tier, an application server that contains most of the application logic as the middle-tier, and the

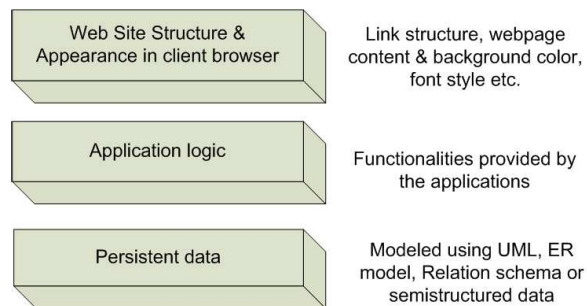


Figure 1: Tiers in a Data-Driven Web Application

client web browser that contains some client-specific application logic and presentation as the top tier (Figure 1).

The vast majority of current application development tools support entirely different programming models at the application server and the client browser. For instance, the part of the application that is executed at the application server is often written using Java and Enterprise Java Beans (EJBs) (which wrap relational data as Java objects), while many clients are written using a combination of JavaScript, PHP and HTML. Consequently, the application developer is forced to decide which part of the application is to be run in the server and which part is to be run the client, *at the time of writing this application*. We argue that forcing a programmer to make this decision at application development time is a suboptimal strategy, both from a programmer productivity and an application correctness point of view.

With respect to programmer productivity, it is often necessary to move various parts of the application from the client to the server (and vice versa) during the evolution of an application for a variety of reasons, including performance concerns and supporting multiple clients (ranging from powerful desktops to less powerful PDAs). A simple example of client-server partitioning is allowing users to sort on a column of a table: it may initially be implemented in the application server as a SQL `order by` query issued over a relational database, but it may later have to be reimplemented as a specialized sorting method in JavaScript to reduce communication with the server. For more complex pieces of code, this amounts to reimplementing a significant part of

the application logic. With respect to application correctness, it is not always possible to run some application server code at the client, and still maintain the correctness of the application. A simple example is fulfilling product orders: running this code at multiple clients may give each client the impression that there are sufficient items to fulfill her order, but there may not be sufficient items to fulfill all the clients' orders. More generally, evaluating EJB/database transactional code in the client may require careful programming to avoid concurrency problems.

In this demonstration, we advocate an alternative solution for partitioning applications between the client and server. We propose developing data-driven web applications using a high-level declarative programming language called Hilda [3], which has a unified data/programming model for all layers of the application stack (database, application server, client browser). We show how a Hilda compiler can take in a high-level Hilda program and *automatically* generate correct code for the three application layers. We also illustrate how the Hilda compiler can optimize this client-server partitioning based on the client capabilities and the run-time properties of the application. The Hilda compiler is available as open-source software at <http://www.cs.cornell.edu/database/hilda>.

We will demonstrate the benefits of Hilda using a real application: a Course Management System (CMS) [2] used by more than 2000 students, staff and faculty at Cornell University. The original version of the system was developed using traditional application development tools (relational databases, J2EE, JavaScript, HTML), while a new version has been developed using Hilda. We will demonstrate how the Hilda compiler automatically generates the appropriate code for two different clients – a powerful laptop computer and a less powerful PDA. We will also illustrate how many of the performance enhancements “hard-coded” in the original version of CMS are automatically generated by the Hilda compiler, and show how the Hilda compiler avoids some of the concurrency bugs present in the original version of CMS.

The rest of this demonstration is organized as follows. In Section 2, we describe the Hilda language and in Section 3, we describe what we plan to show in the demo.

2. THE HILDA LANGUAGE

Hilda [3] is a high-level language designed for developing data-driven web applications. The design of Hilda embodies several key concepts.

First, Hilda is based on UML [1], a well-accepted modeling framework. Hilda provides an application building block called an *AUnit* (for Application Unit), analogous to a UML class. AUnits support encapsulation like a regular UML class, but the creation and manipulation of AUnits is specified declaratively and provides natural support for conflict detection in the face of concurrent application updates. AUnits are single-entry and single-exit, which facilitates structured programming. The main difference from the traditional use of UML is that the object creation and operations are specified declaratively, which enables the Hilda compiler to automatically perform various optimizations without burdening the user with performance issues.

Second, Hilda uses a single data model – the relational model – to represent the state of all parts of the application, including the database, application logic and the client. This eliminates the impedance mismatch between different programming models and also enables the application logic to be specified declaratively using SQL. The choice of the relational model also allows for a practical and efficient implementation since most existing database systems are relational.

Third, Hilda *logically* separates server and client state by separating persistent states and local states to enable highly concurrent execution. The server maintains the current state of the application which is shared among clients, and each client sees a (possibly out-of-date) version of it locally. Notice that this separation between client and server state is only *conceptual*. The real separation can be different and is done by the Hilda compiler or runtime environment based on various performance and correctness criteria. While in traditional system, this separation is done at programming time and is difficult to change during system evolution.

Fourth, Hilda models the application logic and associated control flow as a hierarchy (Activation Tree) which captures the application logic of the system. It also enables encapsulation as the hierarchy naturally limits the scope of the data access of an object. Hilda's control flow goes along the same hierarchy. It is like structured programming, with a tree-like execution structure. It is powerful enough to capture complex graph control flows, but makes the specification of operations more structured and confined to small parts of the code.

Finally, Hilda provides a HTML-based presentation construct. It ensures a clear separation of application logic from presentation

3. CMS AND ITS IMPLEMENTATION IN HILDA

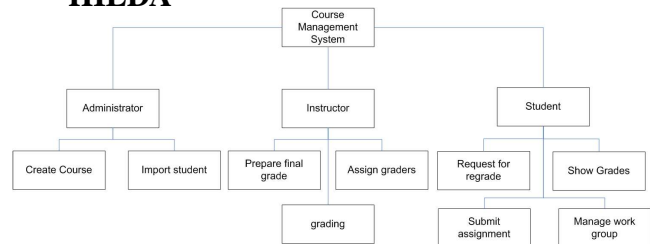


Figure 2: Functionality provided by CMS

We will demonstrate the benefits of Hilda using a real Course Management System (CMS) [2] application. CMS is a secure and scalable web-based course management system that was designed to simplify, streamline, and automate many aspects of the work flow associated with running a large course (Figure 2). CMS is currently being used by over 2000 students in 40 courses in computer science, physics, economics and engineering. CMS uses the standard three-tier architecture and the original implementation of CMS was written in Java, using enterprise java beans, session beans, servlets, java server pages, and the XDoclet code generation system. To illustrate the benefits of the Hilda language, we reimplement

CMS in Hilda.

We now describe some key features of CMS and show how the Hilda compiler automatically performs various client-server partitioning optimizations(Figure 3). To illustrate these optimizations, we use two clients – a powerful laptop and a PDA. To visualize the effects of partitioning, we use a server monitor show which parts of the application logic are evaluated at the server. More specifically, we show the message for AUnits activation/deactivation, push and pull the application logic to and from clients, updating persistent data, messages returned from the clients etc.

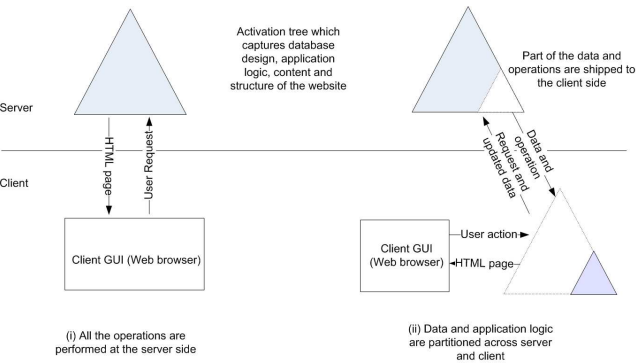


Figure 3: Server-Client Partitioning

Viewing Grades. CMS allows course staff to view students' submissions and grades (Figure 4), and allows them to sort this information on different attributes. As an optimization, the original version of CMS performed the sorting at the client using JavaScript code: approximately 380 lines of specialized JavaScript code was required to support table sorting and this code had to be explicitly shipped to the client on every request involving sortable tables. On the other hand, in Hilda, the sorting logic is specified declaratively as a single SQL statement with an "order by" clause. Depending on the client capabilities, the statement will either be executed at the server or compiled into Java code and executed at the client. As shown in the server monitor (Figure 5), the operation and data are shipped to the client for a laptop, while they are performed at the server for the PDA.

Assignment Creation. CMS allows course staff to create/edit assignments (Figure 6). The corresponding web page contains many fields, and as the course staff makes

	Group	Extension	Files Rem.	Assigned To		Grade	
				OneProblem	TwoProblem		
1	Grubbs, Katherine (kg42)	(Add)	0	da10	da10	150.0	(Grade Files Ugroup)
2	doering, Jane (ahd4)	(Add)	1	da10	da10	119.0	(Grade Files Ugroup)
3	Smythe, John (as98)	(Add)	1	rs94	rs94	90.0	(Grade Files Ugroup)
4	Carden, Wm (ac255)	(Add)	1	ds9	ds9	85.0	(Grade Files Ugroup)
5	Kinnball, Aaron (as333)	(Add)	1			75.0	(Grade Files Ugroup)
6	Bentick-smith, John (as92)	(Add)	1	rs94	rs94	65.0	(Grade Files Ugroup)
7	Smilie, John (as28)	(Add)	1	rs94	rs94	65.0	(Grade Files Ugroup)
8	Doerner, Thomas (as22)	(Add)	0	da10	da10	57.0	(Grade Files Ugroup)
9	Pennington, Csten (as99)	(Add)	1	da10		35.0	(Grade Files Ugroup)
10	Kleinmitch, (hs2)	(Add)	1	rs94	rs94		(Grade Files Ugroup)
11	Goldsmith, (hs25)	(Add)	1	da10	da10		(Grade Files Ugroup)
12	Lee, Pei-chen (as25)	(Add)	1	da10			(Grade Files Ugroup)
13	Jr., John (as58)	(Add)	1				(Grade Files Ugroup)
14	Guarino, Jonathan (hs32)	(Add)	1	rs94	rs94		(Grade Files Ugroup)
15	Doelgaast, Virginia (as27)	(Add)	1	ds9	ds9		(Grade Files Ugroup)

Figure 4: Student's information

```
ACTIVATE CourseMSG7_1_1 from client 128.253.18.60
ACTIVATE AssignmentMSG7_1_1 from client 128.253.18.60
ACTIVATE GradeRowMSG7_1_1_1 from client 128.253.18.60
ACTIVATE StudNameMSG7_1_1_1 from client 128.253.18.60
ACTIVATE StudGradeMSG7_1_1_1 from client 128.253.18.60
ACTIVATE GradeRowMSG7_1_1_2 from client 128.253.18.60
ACTIVATE StudNameMSG7_1_1_2_1 from client 128.253.18.60
ACTIVATE StudGradeMSG7_1_1_2_1 from client 128.253.18.60

PUSH AssignmentMSG7_1_1 from client 128.253.18.60
PUSH GradeRowMSG7_1_1_1 from client 128.253.18.60
PUSH StudNameMSG7_1_1_1 from client 128.253.18.60
PUSH StudGradeMSG7_1_1_1 from client 128.253.18.60
PUSH GradeRowMSG7_1_1_2 from client 128.253.18.60
PUSH StudNameMSG7_1_1_2_1 from client 128.253.18.60
PUSH StudGradeMSG7_1_1_2_1 from client 128.253.18.60

RETURN AssignmentMSG7_1_1.ReleaseGrades from client 128.253.18.60

ACTIVATE CourseFY26_1 from client 64.59.24.100
ACTIVATE AssignmentFY26_1_1 from client 64.59.24.100
ACTIVATE GradeRowFY26_1_1_1 from client 64.59.24.100
ACTIVATE StudNameFY26_1_1_1 from client 64.59.24.100
ACTIVATE StudGradeFY26_1_1_1 from client 64.59.24.100
ACTIVATE GradeRowFY26_1_1_2 from client 64.59.24.100
ACTIVATE StudNameFY26_1_1_2_1 from client 64.59.24.100
ACTIVATE StudGradeFY26_1_1_2_1 from client 64.59.24.100

RETURN StudNameFY26_1_1_1.Sort from client 64.59.24.100
RETURN GradeRowFY26_1_1_1.Sort from client 64.59.24.100
RETURN StudGradeFY26_1_1_1.Sort from client 64.59.24.100
RETURN GradeRowFY26_1_1_2.Sort from client 64.59.24.100
RETURN AssignmentFY26_1_1.ReleaseGrades from client 64.59.24.100
```

Figure 5: For user nsg7(from desktop client), sorting logic and the table data is pushed to the client while for user fy26(from PDA), operations are performed at the server.

Figure 6: Assignment creation/editing

changes, various sanity checks have to be performed (e.g. the due date of an assignment should be earlier than the release date). In the original version of CMS, the checks were explicitly replicated at the client to provide better user-feedback, and then rechecked at the server to preserve database integrity. Using Hilda, all the checks are specified declaratively in a single location, and it is the compiler's responsibility to ensure that the appropriate implementations are generated. The run time messages are shown in Figure7.

User Preferences. CMS allows users to store their preferences (such as when to receive email notifications). Such preferences are shared among all instances of a given user, say, if the same user logs in to two separate windows in different computers for two different courses. Since there is a potential for sharing the preference data across multiple clients, the original version of CMS takes the conservative

