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

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 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 fullfil 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.

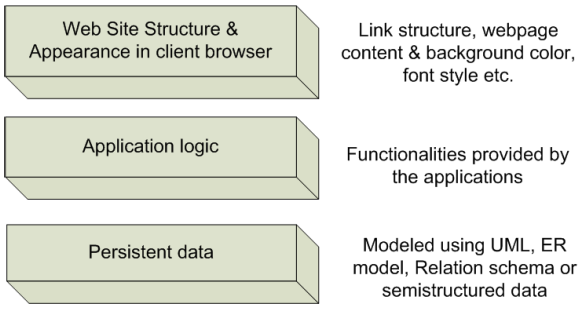

Figure 1: Tiers in a Data-Driven Web Application

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

SIGMOD'06, June 27–29, 2006, Chicago, Illinois, USA.

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. In traditional systems, this separation is hard-wired during the programming stage, hence it is difficult to change when the system evolves.

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 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

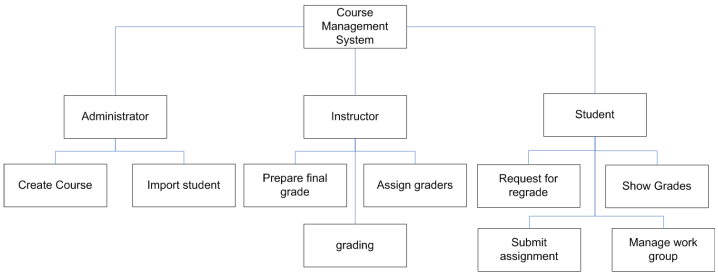

Figure 2: Functionality provided by CMS

3. CMS AND ITS IMPLEMENTATION IN HILDA

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, servelets, java server pages, and the XDoclet code generation system. To illustrate the benefits of the Hilda language, we reimplemented 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 AUnit activation/deactivation, push and pull the application logic to and from clients, updating persistent data, messages returned from the clients etc.

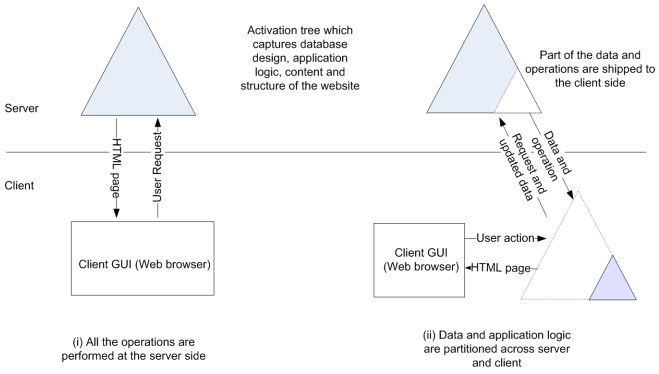

Figure 3: Server-Client Partitioning

Viewing Grades. CMS allows course staff to view students' grades for assignments(Figure 4[1]), 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

[1]The grades shown are for illustration purpose and do not represent real student grades.

	Last Name	First Name	A1	A2	A1P	Total	Final Grade	
			Stats	Stats	Stats			
1	Bentinck-smith	John	65.0	*	*	0.7	A+	Drop
2	Cardoen	Wim	85.0	*	*	0.9	A	Drop
3	Doellgast	Virginia	30.0	*	*	0.3	A	Drop
4	doering	Jane	119.0	*	*	1.2	A	Drop
5	Doermer	Thomas	57.0	79.0	*	2.2	A+	Drop
6	Goldsmith		*	*	*	*	A	Drop
7	Grankin	Sergey	57.0	79.0	*	2.2	A	Drop
8	Grubbs	Katherine	150.0	71.0	*	2.9	A	Drop
9	Guarino	Jonathan	*	*	*	*	A	Drop
10	Jr.	John	*	*	*	*	A+	Drop
11	Kimball	Aaron	76.0	*	*	0.8	A+	Drop
12	kleinsmith		*	*	*	*	A+	Drop

Figure 4: Student's information

```
ACTIVATE CourseNSG7_1   from client 128.253.18.60
ACTIVATE AssignmentNSG7_1_1     from client 128.253.18.60
ACTIVATE GradeRowNSG7_1_1_1     from client 128.253.18.60
ACTIVATE StudNameNSG7_1_1_1_1   from client 128.253.18.60
ACTIVATE StudGradeNSG7_1_1_1_1  from client 128.253.18.60
ACTIVATE GradeRowNSG7_1_1_2     from client 128.253.18.60
ACTIVATE StudNameNSG7_1_1_2_1   from client 128.253.18.60
ACTIVATE StudGradeNSG7_1_1_2_1  from client 128.253.18.60

PUSH AssignmentNSG7_1_1 from client 128.253.18.60
PUSH GradeRowNSG7_1_1_1 from client 128.253.18.60
PUSH StudNameNSG7_1_1_1_1       from client 128.253.18.60
PUSH StudGradeNSG7_1_1_1_1      from client 128.253.18.60
PUSH GradeRowNSG7_1_1_2 from client 128.253.18.60
PUSH StudNameNSG7_1_1_2_1       from client 128.253.18.60
PUSH StudGradeNSG7_1_1_2_1      from client 128.253.18.60

RETURN AssignmentNSG7_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_1   from client 64.59.24.100
ACTIVATE StudGradeFY26_1_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_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_1.Sort       from client 64.59.24.100
RETURN GradeRowFY26_1_1_1.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 fy26(from PDA), operations are performed at the server.

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. Based 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. The corresponding web page contains many fields, and as the course staff makes 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.

Group Management. CMS allows students to create and disband groups in order to work on group projects. A student can extend an invitation to other students to join her group, and the other students can either accept or decline this invitation. Students can also withdraw outstanding invitations and otherwise disband groups. Consequently, the act of accepting an invitation can potentially have many concurrency conflicts: the invitation can be withdrawn, the student extending the invitation could have dropped the course, or the group assignment itself could have been cancelled. In the original version of CMS, all of these potential violations had to be explicitly checked in a database transaction, and there were many bugs (most of which are fixed now) that resulted from forgetting to check certain conditions. In contrast, Hilda specifies operations and their preconditions in a declarative high-level manner. Consequently, the Hilda run-time can automatically monitor when an operations pre-conditions are violated due to concurrent actions, and thereby disallow such operations. Figure 6 illustrates this deactivation using the server monitor.

```
RETURN InviteNSG7_1_1_1_1.ButtonPush from client 128.253.18.60
RETURN InvitationsNSG7_1_1_1.Invite from client 128.253.18.60
RETURN AssignmentNSG7_1_1.Invite from client 128.253.18.60
RETURN CourseNSG7_1.Invite from client 128.253.18.60
PERSIST CMS.Invitations
REACTIVATE CourseNSG7_1
REACTIVATE AssignmentNSG7_1_1
REACTIVATE InvitationsNSG7_1_1_1
REACTIVATE WithdrawNSG7_1_1_1_1


ACTIVATE CourseFY26_1   from client 64.59.24.100
ACTIVATE AssignmentFY26_1_1     from client 64.59.24.100
ACTIVATE InvitationsFY26_1_1_1  from client 64.59.24.100
ACTIVATE InvitationFY26_1_1_1_1 from client 64.59.24.100
ACTIVATE AcceptFY26_1_1_1_1     from client 64.59.24.100

RETURN WithdrawNSG7_1_1_1_1.ButtonPush from client 128.253.18.60
RETURN InvitationsNSG7_1_1_1.Revoke from client 128.253.18.60
RETURN AssignmentNSG7_1_1.Revoke from client 128.253.18.60
RETURN CourseNSG7_1.Revoke from client 128.253.18.60
PERSIST CMS.Invitations
REACTIVATE CourseNSG7_1
REACTIVATE AssignmentNSG7_1_1
REACTIVATE CourseFY26_1
REACTIVATE AssignmentFY26_1_1
REACTIVATE InvitationsFY26_1_1_1
DEACTIVATE InvitationFY26_1_1_1_1
DEACTIVATE AcceptFY26_1_1_1_1

RETURN AcceptFY26_1_1_1_1.Accept from client 64.59.24.100
        stale execution context, aborting
```

Figure 6: After user nsg7 withdraws the invitation, fy26's acceptance of the same invitation will be disallowed

4. CONCLUSION

The high-level declarative nature of Hilda enables an exciting optimization opportunity: client-server partitioning can be done automatically and correctly by a compiler instead of having the programmer write low-level and error-prone code for this purpose. Consequently, the programmer only needs to focus on developing the core logic of the system, and the Hilda compiler can automatically compile it into code depending on the capabilities of the client and other factors such as bandwidth limitations and concurrent actions. We will illustrate the benefits of Hilda using a real-world course management system application.

5. REFERENCES

[1] G. Booch et al. *The Unified Modeling Language User Guide, The Addison-Wesley Object Technology Series.* Addison Wesley, 1998.

[2] C. Botev et al. Supporting workflow in a course management system. In *Proc. SIGCSE*, 2005.

[3] F. Yang et al. Hilda: A high-level language for data-driven web applications. In *Proc. ICDE*, 2006.