A Unified Platform for Data Driven Web Applictions with Automatic Client-Server Partitioning

Fan Yang[1], Nitin Gupta[1], Nicholas Gerner[1], Xin Qi[1], Alan Demers[1], Johannes Gehrke[1], Jayavel Shanmugasundaram[2]

[1] Cornell University, Ithaca, NY
[2] Yahoo!, Santa Clara, CA

{yangf, niting, nsg7, qixin, ademers, johannes}@cs.cornell.edu, jaishan@yahoo-inc.com

ABSTRACT

Data-driven web applications are usually structured in three tiers with different programming models at each tier. This division forces developers to manually partition application functionality across the tiers, resulting in complex logic, suboptimal partitioning, and expensive re-partitioning of applications.

In this paper, we introduce a unified platform for automatic partitioning of data-driven web applications. Our approach is based on Hilda [25, 13], a high-level declarative programming language with a unified data and programming model for all the layers of the application. Based on run-time properties of the application, Hilda's run time system automatically partitions the application between the tiers to improve response time while adhering to memory or processing constraints at the clients. We evaluate our methodology with traces from a real application and with TPC-W, and our results show that automatic partitioning outperforms manual partitioning without the associated development overhead.

1. INTRODUCTION

An important class of applications is *data-driven web applications*, i.e., web applications that run on top of a backend database system. Examples of such applications are b2c portals such as online shopping sites and online auctions, and various b2b portals. Data-driven web applications are usually structured in three tiers: 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 (see Figure 1).

Current development platforms use different programming models at each tier. For example, server side application development frameworks such as J2EE and Enterprise Java Beans (EJBs) wrap relational data as Java objects. PHP and ASP.NET bridge the difference between the data model at the lowest tier and the middle tier in similar ways. The top tier usually uses a different programming model, such as AJAX or FLASH [1], which allows the developer to build rich clients. The difference in programming model between the different tiers forces the developer to decide manually how to partition application functionality across the tiers, and to implement functionality at each tier separately using different programming languages and models.

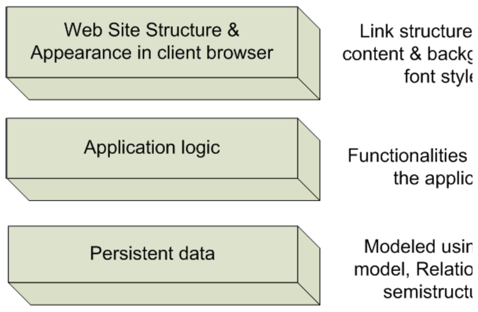

Figure 1: Tiers in a Data-Driven Web Application

Exposing the boundaries between tiers to the programmer in this way has four significant drawbacks.

Increased Development Time. Having different programming models in different tiers makes it hard to develop, maintain, and optimize applications, as the developer must manually bridge the differences between the individual models (for example, the relational model, EJBs, and HTML forms).

Complex Logic due to Partitioning. Partitioning application logic across the tiers requires complex logic to synchronize the state of the application. For example, in order to enable partial updates (a well known strategy in AJAX [2]), data can be cached at the client side. However, client-side caching can allow content shown in the browser to become outdated due to concurrent updates to the application state from different users. Such application-level conflicts are difficult to detect, and existing systems do not provide automatic support for conflict detection.

Suboptimal Partitioning. Since the decision of how to partition the application is left to the developer (who may have little data on which to base her decisions), the resulting division of the application may be be suboptimal in terms of system performance.

Expensive Re-Partitioning. Once a partitioning of the application has been implemented, moving functionality between layers is complex. For example, consider an application that allows a user to sort on a column of a table: this may initially be implemented in the application server as an SQL **order by** query issued over a relational database. If the developer later decides to move sorting to the client side to improve responsiveness, the sort functionality must be reimplemented in a different programming model such as JavaScript.

In this paper, we introduce a unified platform for automatic partitioning of data-driven web applications. Our approach is based on Hilda, a high-level declarative programming language with a unified data and programming model for all layers of the application [25]. In particular, we show in this paper how to automatically partition a Hilda application between the client and middle tier based on runtime behavior of the application — all of this completely transparent to the developer. Our way of partitioning automatically synchronizes state between client and server without the developer having to write any additional code to achieve this. A web applcation developer thus can focus on the core application logic without worrying about the parititioning of the application or changes to the parition. The Hilda system is available as open-source software at `http://www.cs.cornell.edu/database/hilda`.

In summary, this paper makes the following contributions.

- We have developed a run-time environment for Hilda that allows us to automatically partition a data-driven web application dynamically between client and middle tier in a way that is completely transparent to the developer. (Section 2)
- We model client-middle tier partitioning as an optimization problem. The resulting problem is NP-hard with the client side space constraint, but we give an approximation algorithm that is provably within a factor of three of the optimal solution of the problem. (Section 3)
- We show how we can use trace data to instantiate the optimization model and how to derive practical decisions about client-middle tier partitioning. In a thorough experimental evaluation using a technical benchmark and a real application, we show the efficacy of our techniques. (Section 4)

We discuss related work in Section 5, and we conclude in Section 6.

2. AUTOMATIC CLIENT-SERVER PARTITIONING

2.1 Hilda Overview

Hilda is a high-level declarative language designed for developing data-driven web applications [25]. It is based on UML [5] and the relational data model [18]. Instead of using different data models and languages for different layers of the application stack, Hilda presents a unified programming model for all layers (see Figure1).

First, Hilda is based on UML [5], a well-accepted modeling framework. Hilda's main construct are **AUnits**, which correspond to UML classes. The local state of an AUnit corresponds to UML class attributes. As classes can have operations, AUnits can have *Activators*. With data and associated operations, the Hilda programming model is state-based in that a Hilda programmer specifies what operations are allowable in a given state of the program. The main difference from the traditional use of UML is that the object creation and operations are specified declaratively[1], 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 problem 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 to enable highly concurrent execution. The server maintains the current state of the application, and each client sees a (possibly out-of-date) version of it locally. Whenever a client wants to perform an update operation, it checks with the server to see if this operation is still valid in the current system state (to avoid application conflicts). Notice that this separation between client and server state is only *conceptual*. The real separation can be different and should be done by the Hilda compiler or runtime environment based on certain optimization criterion, e.g., sanity checks can be pushed to client side to save bandwidth and round trip time.

Fourth, Hilda models the application logic and associated control flow as a hierarchy. This decision is based on our experience in developing data-driven web applications: since navigation can be very complex, and since the operations that a user can perform at any time depend on complex conditions that have to be satisfied by the current state of the user's session, we need a way to cleanly specify these preconditions. Hilda specifies preconditions hierarchically; this helps the programmer to think in high-level abstractions which are then further broken down into smaller steps further down in the hierarchy. Hilda's hierarchical structure 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.

Fifth, Hilda uses inheritance to separate application logic from web site structure. Specifically, application developers can derive a web site AUnit by inheriting from the corresponding application logic AUnit. The use of inheritance for this purpose has two advantages: (1) the same structured programming model can be used for both application logic and web site structure, and (2) the same application logic can be reused for multiple web site structures.

Finally, Hilda provides a HTML-based presentation construct called a PUnit (Presentation Unit), which is associated with an AUnit and describes how the content of the

[1]This is also the main reason we use different names for otherwise standard object-oriented concepts, so that declarative and non-declarative constructs are easily distinguished.

AUnit is to be presented. PUnits ensure a clear separation of application logic from presentation because they deal only with presentation issues like page layout, font size and background color, while AUnits deal only with application logic and web site structure.

2.2 Design Details

Hilda models application logic using building blocks called AUnits (for Application Units), analogous to UML classes. Each AUnit models a functional component of the application, and encapsulates the operations and the data associated with a web page, subpage or a frame of a webpages. The AUnit is a single-entry single-exit programming construct that is associated with an (optional) *input schema* and an (optional) *output schema*. The input and output schemas are both relational schemas. Given an AUnit, one or more instances of the AUnit can be created. Each instance of an AUnit takes in an input conforming to the input schema of the AUnit and returns an output conforming to its output schema. The act of creating an instance of an AUnit is called *activation*, and the act of destroying an instance of an AUnit is called *deactivation*.

There are three types of AUnits: *Basic AUnits*, *User-Defined AUnits* and *External AUnits*. Basic AUnits are predefined by the system and provide functionality to interact with end users. For example, an instance of the *ShowRow* AUnit shows the attribute values of the input(a single row) to the user and returns no output. Similarly, an instance of the *GetRow* Basic AUnit returns a row of values entered by a user; it takes in no input and returns a single row as an output. Other Basic AUnits for other common interaction tasks are defined similarly.

A User-Defined AUnit corresponds to a functional component in the system. Just as components can have subcomponents, each instance of a User-Defined AUnit also contains zero or more instances of child (User-Defined or Basic) AUnits, which are called *child AUnit instances*. AUnits (like sub-components) can be reused in more than one place. The definition of a User-Defined AUnit contains the application logic of activating and deactivating child AUnit instances, preparing input for child AUnit instances, updating local state and processing output of child AUnit instances and it's own input and output schemas.

External AUnits are used to express small parts of the application logic that do not lend themselves to declarative specification. For example, if an application requires the use of a max-flow min-cut algorithm, it will be awkward to program this using SQL (even though it can theoretically be done with order-based functions and recursion in SQL'99). External AUnits support the same API as other AUnits, but are specified in an imperative language such as Java. Since most data manipulation can be specified declaratively, we expect only a small part of the code to be written using External AUnits; in fact, applications such as CMS do not need External AUnits at all.

One AUnit in the hilda program is designated as the *root AUnit*, which intuitively corresponds to the "main" function in a program. A new instance of the root AUnit is activated each time a new user connects to the Hilda application, and this instance is deactivated when the user disconnects.

Rendering logic is defined for every AUnit to specify the visual appearance of the AUnit at the client. Just as components can have subcomponents, AUnits can form parent-

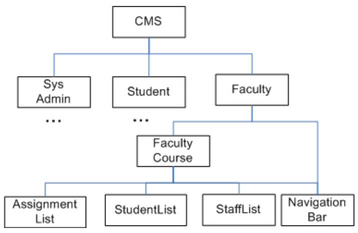

Figure 2: A part of the AUnit hierarchy that models the Course Management System

child relationships, resulting in a hierarchical structure as shown in Figure 2. Each AUnit takes input data from its parent AUnit, and returns output data back to the parent. An AUnit contains *activators* which control activation of instances of its child AUnits. Each activator specifies the following information: (i) the child AUnit of which it creates instances; (ii) an *activation condition*, which defines when and how instances of the child AUnit should be activated; (iii) an *activation ID* that uniquely identifies each child AUnit instance, (iv) input data for instances of the child AUnit; and (v) the operations triggered by the outputs of the child AUnit instances. An activator can activate multiple instances of the same child AUnit that are identified by their ID, which serves as the primary key to distinguish between instances of the same child AUnit. The *KEY* of an instance is then defined as the concatenation of its ID with the IDs of its ancestor AUnit instances. In Hilda, the activation conditions and various operations for the business logic are specified as SQL queries.

Example: A Course Management System. Figure 2 shows part of the AUnit hierarchy for a Course Management System we developed at Cornell that is currently being used by over 2000 students. The *CMS* AUnit represents the application, and contains AUnits for faculty, students and system administrators, which are modeled by child AUnits *Faculty*, *Student* and *Sys Admin*, respectively. A faculty member can view students, add and remove staff, edit assignments and perform other course related operations, each of which is implemented as a child of the *Faculty* AUnit. For example, the *StaffList* AUnit encapsulates the data corresponding to the list of staff members associated with the current course. This AUnit allows the faculty member to view and update, in a browser, the complete list of staff members. The ID of a *Faculty* AUnit instance is the faculty's NetID; the ID of a *FacultyCourse* AUnit instance is the name of the course; its KEY is a tuple consisting of the course ID and the faculty's NetID. The activation condition for *Faculty* is that the current user logs in as a course faculty member.

The hierarchical structure formed by AUnits models the hierarchical structure of a website (Figure 2). The leaf level AUnits represent basic components such as HTML forms that allow user interaction. For example, the *Navigation Bar* AUnit represents a form containing options, one of which can be selected by a user. One of the AUnits in a Hilda program is designated as the *root AUnit*, which intuitively corresponds to the "main" function in a program. For example, the *CMS* AUnit is the root AUnit of the application in Figure 2. A new instance of this root AUnit is activated each time a new user connects to the application, and this instance is deactivated when the user disconnects. The pro-

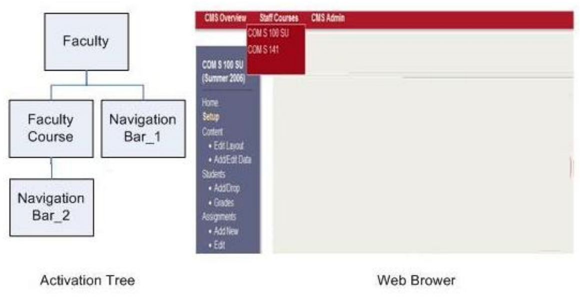

Figure 3: Activation tree and the corresponding webpage

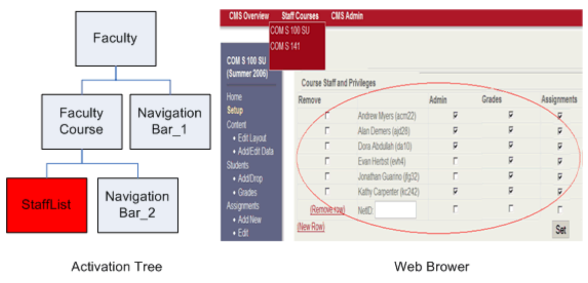

Figure 4: Activation tree and the corresponding webpage after selecting editing staff

gram then recursively activates children of the root AUnit and constructs a tree of AUnit instances. The system maintains this activation tree for each user session. At any time, the activation tree represents the part of the application currently available to users through a web browser.

When a user performs an operation, such as submitting a form, the leaf level AUnit corresponding to that operation returns data to its parent, which then performs operations to update the application state, and/or returns data to its parent, and so forth. AUnit instances can only get data and return data to their parents, which follows the similar data flow for function invocations. After the return chain terminates, a new activation tree is constructed by reevaluating every activation condition based on the updated state. Please notice that reconstructing the whole activation tree after each return chain terminates is the semantic of the execution model. The real implementation can optimizes the process by skipping reevaluate irrelevant part of the tree and will not build the whole tree from scratch.

In Hilda, a web application is no longer considered as a connected graph of individual web pages that allows users to navigate from any page to any other page. Instead, execution of a web application is modelled as a sequence of transitions from one activation tree to another. The transition is triggered by users' interaction and each activation tree corresponds to a webpage shown to the user and the operations the users can perform on that page.

CMS example: Consider a case in the CMS, when a user logs in as course faculty and come to a course page. Figure 3 shows the current activation and the page in the browser. Each activated AUnit instance corresponds to a sub-page of the content shown in the browser. NavigationBar_1 corresponds to the navigation bar at the top of the page, and NavigationBar_2 corresponds to the one at the left. Then she wants to view and edit the list of staff members in current course and select that option from the navigation bar. NavigationBar_2 returns the selected option to its parent FacultyCourse and updated the local state[2] of its parent. When return chain finished (in this case, the chain only has one step), the new activation tree is constructed based on the updated state. One StaffList instance will be activated based on its activation condition. Figure 4 shows the resulting activation tree and its appearance in a web browser. Again if the user wants to view the student list for this course and choose "Student" from the navigation bar, the NavigationBar_2 AUnit instance returns to the FacultyCourse instance and updates the state, which then deactivates the StaffList instance and activates a new instance of the StudentList AUnit. The webpage gets updated and will show students information.

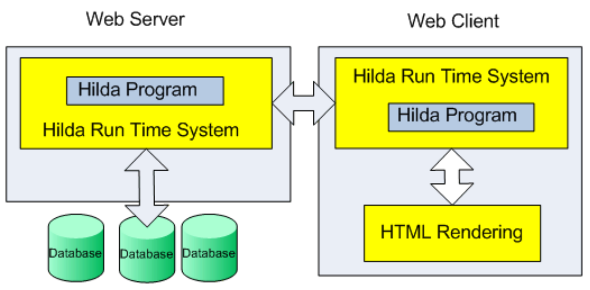

Figure 5: System Architecture

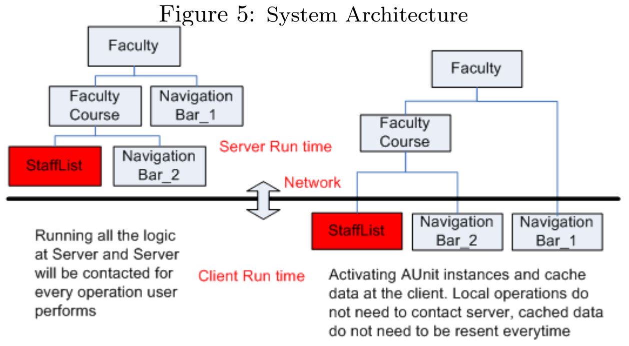
Faculty
Faculty Course
Navigation Bar_1
StaffList
Navigation Bar_2
Server Run time
Network
Faculty
Faculty Course
StaffList
Navigation Bar_2
Navigation Bar_1
Running all the logic at Server and Server will be contacted for every operation user performs
Client Run time
Activating AUnit instances and cache data at the client. Local operations do not need to contact server, cached data do not need to be resent everytime

Figure 6: Activation tree with different partitions

2.3 Run time system

The Hilda run time systems, for both the server and the client, are evaluation engines for Hilda programs. They execute the application logic specified in a program by maintaining the activation trees, and maintain consistency between the client and server states. We use RTSS to refer to the run time system residing at the server, and RTSC for the system residing at the client. The RTSS is a Java servlet running in some application server (such as JBOSS or Weblogic), and has connections to the back end database. It communicates with the RTSC. The RTSC runs as "sticky" applet which resides in the secondary cache of the client, and is available for quick loading by the browsers [15].

Based on the semantics of Hilda, the RTSS and RTSC coordinate with each other to maintain the activation tree. An AUnit instance is activated when its activation condition is satisfied and is deactivated when the conditions fails to be satisfied. The location (client or server) where an instance is activated is not predefined by the application developer; instead, it is determined and can be changed at run time by

[2]We omit the details of local state of AUnit here, which are also expressed tables. In this case, we have a CurrentChoice table which keep track of which option users selected. Please refer to [25] for more details

the run time system.

The RTSC caches local data at the client. It uses this data to generate webpages dynamically (e.g. student info list), and to store a user's temporary input (e.g. items in a shopping cart). This temporary data is stored in main memory, and reused by the RTSC. The RTSC contacts the RTSS to check for updates to its input data. The system imposes an upper bound on how out-of-date the client state can be by periodically contacting the server using *heartbeat* messages. To avoid sending the cached data back and forth between the client and the server, the RTSS maintains a copy of the data sent to each client. On receiving an update request, the server checks for updates to the client input data, and responds with only the updated data. To limit the amount of server-side data required for each client session, the developer can specify a maximum life span for the data in the server cache, as well as the heartbeat frequency of the client. Our cache consistency strategy is similar to a detection based approach for transactional client server caching [11, 24, 17], although the system allows for the integration of other strategies in the future.

Client-server partitioning is done based on the *activation profile* of an application. The activation profile specifies which AUnit instances, identified by their unique key, should be activated in the client. When the run time system activates an instance of some AUnit, it refers to the configuration profile to determine whether the RTSS or the RTSC should activate the instance. The activation profile is generated automatically, based on the observed workload of the system. We discuss activation profile generation in the next section.

CMS example: Figure 6, shows two different partitions of the same activation tree. In the first case, we keep the complete activation tree at the server side, while in the second case, we keep part of it at the client side. The main drawback of running everything at the server side is that the client must contact the server for every operation the user performs. The server then resends the entire refreshed page in HTML format to the client. However, if the navigation bar and *Stafflist* instances are executed at the client, the run time system can cache the data needed by them. Then, if the user adds or removes staff, the list of staff members is updated locally in the client, and the server is contacted only when the *Submit* button is clicked by the user. Only after this step are the updates in the staff list sent to the server, which updates the database. Note that the client already has all the data and code for the new staff list, unless it has been updated by other users — which is detected by the run time synchronization algorithm. This allows the RTSC to create HTML at the client without waiting for the server to respond. Other parts of the page, such as navigation bars, are normally unaffected by the transmitted data, and therefore keep their place in the page.

Maintaining AUnit instances at the client can therefore result in better system response time and a better user experience. This can also be seen from our experiments, discussed in Section 4. Similar caching logic for partial updating of pages can be implemented in frameworks like AJAX only by extensive client side coding. In our framework, such partitions are automatic.

3. MODEL OF CLIENT-SERVER PARTITIONING

In this section, we present a cost model for client-server partitioning. We first define the problem and formulate it as an optimization problem. We show that the problem is NP-hard. We then give an algorithm that approximates the optimum partition, and prove a bound on the approximation error.

3.1 Partitioning Philosophy

A plausible method of solving the client-server partitioning problem would be partition at the granularity of AUnit definitions; i.e., to partition the set of AUnit definitions into two sets, one corresponding to AUnits whose instances will run on the server, and the other corresponding to AUnits whose instances will run on the client. Such method does not capture the fact that different instances of the same AUnit may require very different amounts of computation and data transfer. For example, in the Course Management System, the EditCourse AUnit provides the functionality for course staff to edit courses. The amount course-related data, such as the number of students enrolled, the number of assignments, etc., can differ substantially between courses. We may want to ship the data and computation to the client for small courses while keeping big courses at the server side to save bandwidth. This motivates our decision to group similar instances together, profile their execution and then partition programs at the level of AUnit instances based on the profiles.

Different types of clients can have very different computing and storage resources. For example, moving computing and data to a powerful desktop client may be desirable, while doing the same for a PDA client may adversely affect its response time. This motivates us to partition the application based on the types of clients. We make the assumption that the cost associated with a partition is independent of the load on the server. So the partitions corresponding to different client do not interfere with each other on the performance. We assume the load on the server can be reduced using existing load-balancing techniques and improving the scalability of the system are out of the scope of the paper. The solutions for each client type thus obtained can be combined to yield an overall optimal solution for the application. Therefore, we describe the cost model only for a single client type.

3.2 Terminology

Recall that Hilda models an application in a hierarchical manner, where each AUnit contains other AUnits. Let *aid* be a unique identifier associated with each AUnit definition. Then, we define the *class graph* of a Hilda program P as:

Definition 1: $ClassGraph(P) = (V, E)$ where $V = \{v | v$ is an AUnit definition in $P\}$, and $E = \{(v, w) | v, w \in V$ and w is a child AUnit of $v\}$ $\diamond$

For a valid Hilda program, the class graph must be a DAG. However, instances of an AUnit may be activated for different keys. These instances can be uniquely identified by the pair (aid, key), where *key* corresponds to the set of evidence that leads to the activation of a given AUnit instance. This leads us to the definition of the *key tree* of a given Hilda program P:

Definition 2: $KeyTree(P) = (V, E)$ where $V = \{(aid, key) |$ *aid* is the identifier of some AUnit definition in $P\}$, and

$E = \{(v,w) | v, w \in V$ and $w.aid$ corresponds to a child AUnit of the AUnit corresponding to $v.aid\}$ $\diamond$

Note that each AUnit corresponds to a different key, where the key includes the key of the AUnit's parent node. Therefore, an instance of any AUnit, except the root, is activated by exactly one parent. Thus the key tree must be a tree. Note that the class graph of a Hilda program is effectively an aggregated version of the key tree, obtained by merging nodes that have the same *aid*. In order to estimate the response time of a system, we require for each node of a key tree, various annotations such as the expected time for processing the AUnit instance and expected data to be processed for that instance. We next define an annotation function for the key tree of a Hilda program P as:

Definition 3: The *annotation function* $\mathcal{A}\ (P) : V \to \mathbb{R}^4$ for the key tree (V, E) of a hilda program P is a function that maps each node v of the key tree onto a 4-tuple, where the fields correspond to the following: the probability p_v that a randomly chosen activation over the space of all executions of P is on the AUnit that v is associated with; the expected time t_v for processing queries of the node, the expected sum d_v of the size of input and output data, and the expected number l_v of connections established by this node between the client and the server, respectively. By the definition of the probabilities p_v we also have

$$\sum_v p_v = 1.$$

. $\diamond$

We describe here the partitioning of a Hilda program into the client part and the server part, at the granularity of its key tree. Whether an AUnit instance is activated and evaluated at the client or the server depends on how the partitioning is done. Let $\zeta : V \in KeyTree(P) \to \{client, server\}$ be a function specifying where AUnit instances in the key tree of program P are located. Then, we define a parameter α, which expresses the proportion blowup in computation time between the client and the server, as:

$$\alpha = \frac{t_u}{t_v} \text{ where } \zeta(u) = server \text{ and } \zeta(v) = client.$$

The data size of any given AUnit instance is assumed to be independent of ζ. This is because the input and output data of any instance remains the same, regardless of whether the instance is located at the server or the client. The partition of a hilda program P, then, is defined by a cut C as:

Definition 4: $Partition(P) \equiv C = (G_s, G_c)$ a cut in the tree $KeyTree(P) = (V, E)$ s.t. G_s and G_c are disjoint, G_s is connected, the root node belongs to G_s, and $V_s \cup V_c = V$. $\diamond$

In this definition, $G_s = (V_s, E_s)$ is the part that runs on the server and $G_c = (V_c, E_c)$ is the part that runs on the client, i.e. an AUnit instance a will be activated and maintained at client side iff $\exists v \in V_c \ni a.key = v.key$. We denote the set of edges between the two sides of the partition by $E_{cut} = E - (E_s \cup E_c)$.

3.3 Cost Model

We now define our cost model. In this paper, our goal is to optimize the average response time for users. Optimizing other goals, such as system throughput, would involve a similar analysis but a different cost model. We leave this as future work. Recall that we assume that the key trees corresponding to different types of clients are independent of each other, and do not affect the cost model for any given tree. We therefore consider the key tree for only a single type of client.

Given the key tree $KeyTree(P) = (V, E)$ of a program P, the annotation function $\mathcal{A}$, and the cut $C = (G_s, G_c)$ that partitions the tree into server and client subgraphs, we define the expected user response time as:

$$cost_C(P) = \sum_{v \in KeyTree(P)} p_v \times t_v^C$$

The time to perform AUnit instance processing, given a partition, includes the time to process the AUnit instance at the client (return queries and later reactivation queries), the time to send query results to and from the server and the time to process the queries at the server:

$$t_v^C = t_v^{client} + t_v^{data} + t_v^{server}$$

where t_v^{client} is the expected time for processing v at the client, t^{data} is the expected time for sending result sets to and from the server, including the time for preparation of the data, and t_v^{server} is the expected time for processing queries at the server. Based on our earlier assumption that the time to process an AUnit instance at the client is proportional to the time to process the same instance at the server, and assuming that the data transmission time for transferring a result set between client and server is proportional to the size of that result set, we have:

$$t_v^{client} = \begin{cases} 0 & \text{if } \zeta(v) = server \\ \alpha \times t_v & \text{if } \zeta(v) = client \end{cases}$$

$$t_v^{server} = \begin{cases} t_v & \text{if } \zeta(v) = server \\ 0 & \text{if } \zeta(v) = client \end{cases}$$

We also have

$$t_v^{data} = \begin{cases} \gamma \times d_v + L \times l_v + d_v/\beta & \text{if } \exists u \text{ s.t. } (u,v) \in E_{cut} \\ 0 & \text{otherwise} \end{cases}$$

Here, the data transmission cost t_v^{data} consists of three parts: the expected time for preparing the data to transfer, expected overhead of the handshaking process for establishing TCP connections, and the expected time for transferring the data. We assume that the expected time for preparing and transferring data is proportional to the expected amount of data transferred, with proportionality constants γ and β, respectively. L is the expected overhead for the handshaking process(initial round trip time), which allows us to take into account the number of connections.

These definitions yield the following optimization problem to choose a cut C for a program P:

$$\arg\min_C cost_C(P)$$

We define an additional constraint to take into account client memory limitations. Let $M_C(T)$, the memory usage at the client given the cut C, be given by:

$$M_C(P) = \sum_{v \in KeyTree(P), \zeta(v)=client} m_v$$

where m_v is the maximum memory that is used by any query of the AUnit instance v. Then, if $\hat{M}$ is the maximum memory available for the application at the client, we have the constraint $M_C(T) \leq \hat{M}$.

Before presenting our solution for the problem, we want to justify several simplification we made in our cost model. First, we ignore the cost at server side for synchronization and processing heart beat messages. Because it is done asynchronous to users' actions and do not noticeably affect users' respond time. Second, we don't consider the cost for transferring the run time system and Hilda code to the client side. They are implemented as sticky applet and can be reloaded from the client machine after the first time. Last, the web browser rendering time would be the same across different partitioning scenarios and we don't include it in the cost model.

3.4 Solution For Partitioning

The problem of finding an optimal partition with constraints for a given key tree can be proven to be NP-hard.

Theorem 1: The Hilda client/server partitioning problem is NP-hard.

Proof: We can prove this theorem by a reduction from the well-known 0-1 Knapsack problem, which is NP-hard.

Consider a Knapsack problem instance with n items, each having a profit p_i and a weight w_i $(i = 1, 2, \ldots, n)$. The goal is to find a subset of items with their total weight no more than a given bound W, and their total profit maximized. We can construct the following instance of the client/server partitioning problem: the key tree will contain $n + 1$ items, n of which are leaves, and correspond to the n items in the Knapsack problem; the memory cost for node i being on the client side is w_i, and the client-side memory bound is W; the computing cost for node i being on the server side is p_i, and the computing costs for client side are all 0; the data transfer cost are all 0.

The client/server partitioning problem is minimizing the sum of server-side computing costs for nodes at server side, which is equivalent to maximizing the sum of server-side computing costs for nodes at client side. It is then obvious that solving the client/server partitioning problem instance is equivalent to solving the Knapsack problem instance. Therefore the Hilda client/server paritioning problem is NP-hard. □

Therefore, we design an approximation algorithm, which guarantees to give a result which is within three times of the optimal in the worst case. The technique we use is Randomized Rounding[20]: we first formulate the problem as an Integer Programming (IP) porblem, relax it to a Linear Programming (LP) problem, solve it, and use a randomized algorithm, similar to that in [14], to round the solution to an integral one that is not much worse.

Given a key tree $KeyTree(P) = (V, E)$, for every node $v \in V$, we define a variable x_v and for every edge $e \in E$, we define a variable y_e. The optimal partition problem for a given Hilda program P, with the annotation function $\mathcal{A}$ can then be formulated as the following IP problem:

$$\text{Min} \sum_{v \in V} x_v * s(v) + \sum_{v \in V} (1 - x_v) * c(v) + \sum_{e \in E} y_e * n(e)$$

subject to:

$x_{root} = 1$	$root$ is the root of KT
$x_{v_1} \geq x_{v_2}$	$\forall e(v_1, v_2) \in E$
$y_e \geq x_{v_1} - x_{v_2}$	$\forall e(v_1, v_2) \in E$
$\sum_{v \in V} (1 - x_v) * M(v) \leq \hat{M}$	
$x_v \in \{0, 1\}$	$\forall v \in V$
$y_e \in \{0, 1\}$	$\forall e \in E$

For each node $v \in V$ and edge $e = (u, v) \in E_{cut}$,

$c(v) = \alpha \times t_v$ is the computing cost at client side
$s(v) = t_v$ is the computing cost at server side
$M(v) = m_v$ is the memory cost at client side
$n(e) = (1/\beta + \gamma) \times d_v + L \times l_v$ is the data transfer cost

The optimal solution for above integer programming will give us an optimal partition $c = (G_s, G_c), E_{cut}$ in the following way:

$$x_v = \begin{cases} 0 & \text{if } v \in V_s \\ 1 & \text{if } v \in V_c \end{cases} \text{ and } y_e = \begin{cases} 0 & \text{if } e \notin E_{cut} \\ 1 & \text{if } e \in E_{cut} \end{cases}$$

We can relax the above problem, by allowing $x_v \in [0, 1]$ and $y_e \in [0, 1]$, and get an LP problem that is solvable in polynomial time, with solution X^*. We can then round each x_v^* to 0 or 1 with a threshold uniformly randomly chosen from $[1/3, 2/3]$. This special rounding technique guarantees that the objective function and constraints are still within a reasonable bound. The following algorithm find a number t so we can round each x_v to 0 if $x_v \leq t$ and to 1 if if $x_v > t$

```
1: RoundingCut:
   Input: (KT, C, S, N, M)
   {KT is the key tree, C, S, N, M are the client cost, server cost, bandwidth cost, main memory cost for each node and edge in KT}
   Output: c {estimated optimal partition on KT}
2:   Construct the linear programming problem as mentioned above on (KT, C, S, N, M)
3:   Solve the linear programming problem and get optimal solutions (X, Y) where $X[v]$ gives the optimal solution for variable $x_v$ and $Y[e]$ gives the optimal solution for variable $y_e$
4:   $X_{optimal} \leftarrow NULL$
5:   $min \leftarrow 0$
6:   for all t in X do
7:     construct X' where X'[v] = 0 if if $x_v \leq t$ and and X'[v] = 1 if $x_v > t$
8:     construct Y' where Y'[e] = X'[v] - X'[w] and e=(v,w) $optimal \leftarrow$ evaluate minimizing function on X' and Y'
9:     if $min \leq optimal$ then
10:      $min \leftarrow optimal$
11:      $X_optimal$ = X'
12:    end if
13:  end for
14:  Construct c based on $X_{optimal}$ according to the method mentioned above
```

Theorem 2: The approximated solution produced by RoundingCut algorithm is at most 3 times as much as the optimal partition solution. The p, m under the approximated solution are at most 3 times as much as $\hat{p}, \hat{m}$.

Proof: If all the variables $x_v (v \in V)$ and $y_e (e \in E)$ are constrained to be 0 or 1, then the integer programming will give the optimal solution to the partition problem. By relaxing the variables to take real values in $[0, 1]$, we get a linear program, whose solution gives a lower bound to the value of the optimal partition. So we only need to construct an integral solution that has value within a constant factor of the optimal solution to the linear program.

Consider the following randomized rounding algorithm:

- Solve the LP optimally, and denote the optimal solution to it as $x_v^*(v \in V)$ and $y_e^*(e \in E)$.
- Generate t uniformly at random from $[\frac{1}{3}, \frac{2}{3}]$.
- For all $v \in V$ s.t. $x_v^* \geq t$, put v at the server side, and the remaining nodes are at the client side.

If we were doing the rounding with t chosen uniformly at random from $[0, 1]$, the *expected* solution will satisfy all the constraints and have the value as the LP optimal. However each particular rounded solution might not have the optimal value, and to be worse, it might also violate the memory constraints. We want to show that there exists one rounded solution, which *simultaneously* have the following two properties:

- The value is within a constant factor of the optimal solution.
- The memory bound is violated at most by a constant factor.

Let us denote the rounded variables by $\bar{x}_v$ and $\bar{y}_e$, which are random variables depending on t.

Claim: The following inequalities hold for the randomized rounding algorithm:

- $\sum_{v \in V} (1 - \bar{x}_v) * M(v) \leq 3\hat{M}$
- $\sum_{v \in V} \bar{x}_v * s(v) \leq 3 \sum_{v \in V} x_v^* * s(v)$
- $\sum_{v \in V} (1 - \bar{x}_v) * c(v) \leq 3 \sum_{v \in V} (1 - x_v^*) * c(v)$
- $E[\sum_{e \in E} \bar{y}_e * n(e)] \leq 3 \sum_{e \in E} y_e^* * n(e)$

Here $E[\cdot]$ means expectation.

Proof: We will prove the first, the second, and the fourth inequalities. The proof of the third one is the same as the second.

Let set $C = \{v \in V | \bar{x}_v = 0\}$. For any node $v \in C$, $x_v^* < t \leq \frac{2}{3}$, i.e., $3(1 - x_v^*) \geq 1$. Then we have

$$\begin{aligned}
\sum_{v \in V} (1 - \bar{x}_v) * M(v) &\leq \sum_{v \in C} (1 - \bar{x}_v) * M(v) \\
&= \sum_{v \in C} M(v) \\
&\leq \sum_{v \in C} 3(1 - x_v^*) * M(v) \\
&\leq \sum_{v \in V} 3(1 - x_v^*) * M(v) \\
&\leq 3\hat{M}
\end{aligned}$$

Let set $S = \{v \in V | \bar{x}_v = 1\}$. For any node $v \in S$, $x_v^* \geq t \geq \frac{1}{3}$. Then we have

$$\begin{aligned}
\sum_{v \in V} \bar{x}_v * s(v) &\leq \sum_{v \in S} \bar{x}_v * s(v) \\
&= \sum_{v \in S} s(v) \\
&\leq \sum_{v \in S} 3x_v^* * s(v) \\
&\leq 3 \sum_{v \in V} x_v^* * s(v)
\end{aligned}$$

Now let us prove the last inequality. It is easy to see that $y_e^* = x_{v_1}^* - x_{v_2}^* (\forall e = (v_1, v_2))$. So $\bar{y}_e = 1$, i.e., edge e is included in the cut, iff $x_{v_2}^* < t \leq x_{v_1}^*$. Since t is uniformly picked from $[1/3, 2/3]$, the probability for t to fall into the range $(x_{v_2}^*, x_{v_1}^*]$ is at most $\frac{x_{v_1}^* - x_{v_2}^*}{2/3 - 1/3}$, which is $3y_e^*$. Then the inequality follows. □

From the above claim, we know that the first three inequalities are satisfied *absolutely*, and only the last one is about expectation. Therefore *all* the possible rounded results of the algorithm can at most violate the first two constraints by a factor of 3, they will also be within factor of 3 of the optimal value on the first two parts of the objective function. The property of expectation implies that there exists at least one particular rounded solution that satisfies the last inequality. That solution is the one that is guaranteed to be simultaneously within factor 3 from the optimal solution and the bounding constraint. □

Note that the theoretical bound given here is a worst-case bound. In practice, we found that the response time obtained using our algorithm is very close to the optimal response time for the applications that we considered in our experimental evaluation.

4. EXPERIMENTAL EVALUATION

In this section, we first describe the setup for the experiments we performed to evaluate the performance our Hilda system(Section 4.1). We then compare the performance of a Hilda and a J2EE implementations of a real world application (CMS) and a technical benchmark (TPC-W). These comparisons show the benefits of automatic client-server partitioning (Section 4.2).

4.1 Experimental Setup

We first discuss how we estimate the annotation of a key tree using a trace of the running application. We then describe how we apply the result of the optimization problem to achieve a partition of the application, and we give an overview of the physical setup for the experiments.

4.1.1 Parameter Estimation

A trace consists of a sequence of AUnit activations, along with meta data for the time, data and number of connections associated with each activation.

Definition 5: Let P be a Hilda program. A *trace* $Trace(P) = \langle (i, v_i, t_i, d_i, l_i, t_i^{\gamma}) | 1 \leq i \leq n \rangle$ of P is a sequence of five-tuples called *events*. The number i is the sequence number of the event, $v_i = (aid, key)$ uniquely identifies an AUnit instance in P, t_i is the time taken to process the queries in this instance, d_i is sum of the size of the input and output data for the instance, l_i is the number of connections established between the client and the server, and t_i^γ is the time spent to prepare the data by this instance. $\diamond$

Given the above definition, the annotation function of the keytree of program P can be estimated through an aggregated version of the trace. Since multiple events in the trace may be associated with the same node v of the key tree, we can estimate the value of v's annotation by counting and aggregating the trace data for each node. More precisely, we estimate the annotation function for a node $v \in KeyTree(P)$ as follows. Let $\mathcal{A}(v) = (p, t, d, l)$. Then we can estimate (p, t, d, l) with $(\hat{p}, \hat{t}, \hat{d}, \hat{l})$ as follows:

$$\begin{aligned}
\hat{p} &= \frac{|\{i|\exists(i, v', t', d', l', t'') \in Trace(P)\}|}{n}, \\
\hat{t} &= \frac{\sum\{t|\exists(i', t, d', l', t'') \in Trace(P)\}}{p \times n}, \\
\hat{d} &= \frac{\sum\{d|\exists(i', t', d, l', t'') \in Trace(P)\}}{p \times n}, \\
\hat{l} &= \frac{\sum\{l|\exists(i', t', d', l, t'') \in Trace(P)\}}{p \times n}.
\end{aligned}$$

The other parameters for optimization were specified according to the physical setup. We ran the experiments on the PlanetLab network. Given that only powerful desktop clients are used in PlanetLab, we assumed that the client and the server have similar computing power. Therefore, we set parameter $\alpha = 1$, and no bound was imposed for the memory available at the client. The bandwidth (β) of the network was roughly 300KB, and the round trip time L was approximated as 10ms. We could also have estimated these parameters automatically at runtime; this is left as future work. We also estimated γ as follows:

$$\gamma = \frac{1}{n} \sum_{i \le n} \frac{t_i^\gamma}{d_i}.$$

4.1.2 Partitioning Logic

The client-server partitioning for a program P is done at the granularity of key trees. Given a cut $C = (G_s, G_c)$ in the key tree, we ship the data of the AUnit instances in V_c to the client. However, note that our constructed annotation function assumes that the future workload is very similar to the one seen before. In practice, the future workload can contain AUnit instances that have never been encountered before. Therefore, the partitioning is also done at the class graph level, using nodes from the class graph as representatives for instances not yet seen in the trace. For unseen instances, we will position the instance based on the computed partitions for the class graph.

4.1.3 Physical Setup

We illustrate the benefits of Hilda using a Course Management System and an Online Book Store application that is based on the TPC-W benchmark. We compare responsiveness of the system (a.k.a. average users' response time) of a Hilda implementation and a J2EE implementation of the two applications. The applications were deployed in a JBOSS application server setup on a 2.66Ghz machine having 4GB of RAM, and used MS SQL 2005 as the backend database management server. The client simulators were deployed on the PlanetLab network, and included the Hilda RTSC.

Operation	Description	Number
O1	View CMS homepage	24994
O2	View course management system summary	244
O3	Add/remove courses	18
O4	View course property page(as instructor)	219
O5	View course property page(as admin)	83
O6	Edit course property	91
O7	View course homepage(as student)	7912
O8	View course homepage(as instructor)	1858 1858
O9	View student list page	9
O10	View add students page	133
O11	Add/edit students	867
O12	Drop students	48
O13	Update students final grades	25
O14	View adding assignment page	158
O15	View editing assignment page	841
O16	view assignment list	846
O17	View assignment details	20923
O18	Editing assignment	497
O19	View adding category page	205
O20	View edit category schema page	120
O21	View edit category content page	150
O22	Add/remove/edit columns in category schema	103
O23	Add/remove/edit rows of category content	16

Table 1: Operations in the CMS Application

We measured the response time for each operation, i.e. the time taken to submit a request, process it at the server/-client and receive the resulting page from the server. Therefore, this measure includes the time spent on the server to process the request, the time spent at the client and the network transmission time. However, we did not take into account the time taken by the web browsers to render the resulting HTML pages. Also, in order to reduce the error due to the erratic nature of the PlanetLab network, the experiments were conducted twice. The values we present in the next section are therefore averages over two runs of the simulation.

4.2 Experimental Results

We now present experimental results from two applications: a CMS and an Online Book Store.

4.2.1 Course Management System

Our first experiments were performed on CMS, a Course Management System developed at the Cornell Computer Science Department which is currently in use by more than 2000 students, staff and faculty [6]. The original version of CMS was developed using traditional application development tools such as J2EE/EJB, JavaScript and HTML, while a new version has been developed using Hilda.

The J2EE version of the CMS was developed by experienced programmers, and therefore included extensive client-server partitioning that was done manually. Most of the client-side application logic was implemented using Javascript, and thus allowed updating the webpages dynamically. For example, features such as sorting tables based on selected column values, showing or hiding portions of a web page, and caching users' input temporarily in the browser were

System	Average Response Time(ms)	Average Data Transmission(KB)
J2EE	278.80	17.99
Server Only	312.64	19.09
Client Server	270.01	12.74

Table 2: CMS: Response Time and Data Transmission

already implemented at the client side.

To calculate the average response time, we emulated the operations performed on the CMS in one semester. A usage log consisting of 60000 operations was collected from the J2EE version of the system, along with the necessary parameters. Table 1 lists the operations that the users performed. The first three thousand operations from this log were used as a trace to construct the annotation function, which was then used to calculate a partition for the application. The rest of the operations where then tested based on the calculated partition. The average response time shown in Table 2 does not include the time to collect the trace. Table 2 also presents the performance measure of the application when it is deployed at the server without any partitioning.

It is evident from Table 2 that the Hilda version of CMS with automatic partitioning is comparable to the J2EE version in average response time. The automatically partitioned version, however, reduces the average data transferred between the client and the server by roughly 30%.

Figure 8 shows the average data transfer for each operation. Owing to the fact that caching user input at the client reduces the amount of data transferred between the client and the server, operations such as O3, O11, O12, O13, O14, O19, O22, O23 that involve updates result in comparatively less data transfer. For example, consider O3 – after the system administrator creates a new course, the page is refreshed with a new list of courses. However, if the AUnit for the course list gets pushed to the client, the page generated at client side is able use locally cached data.

The J2EE version of the CMS allows a web browser to cache webpages for later visit, at the page level, while the Hilda run time system caches data at the AUnit (subpage) level. For example, a navigation bar that is present on most pages includes the list of available courses, and contains the assignment and category list corresponding to each course. After partitioning, the Hilda run time system keeps the AUnit instances for the navigation bar at the client, including the data and the logic to generate HTML segments for navigation bars. Such partial updating yields benefits in the response time for the operations O1, O7, O8, O16, O17, O19, O20. The Hilda run time system also makes sure that the data for a navigation bar (list of assignments, courses and categories) is up to date, by periodically checking with the server for any changes.

Bad design decisions may sometime result in suboptimal performance. In the J2EE version of the CMS, the logic for users to sort tables based on different columns is always pushed to the client. However, all pages in the system are assembled dynamically, and the Javascript generated on the fly is embedded in the HTML pages. This Javascript makes the size of pages with sortable tables very large (600K on average). It increases the network transmission time and results in poor response time even compared to the Hilda version without any partitions (Figure 7: O10, O11, O12 and O13).

4.2.2 Online Book Store

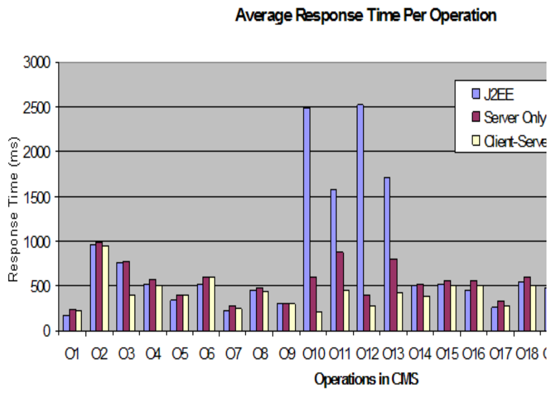

Figure 7: Average Response Time for Different Operations in CMS

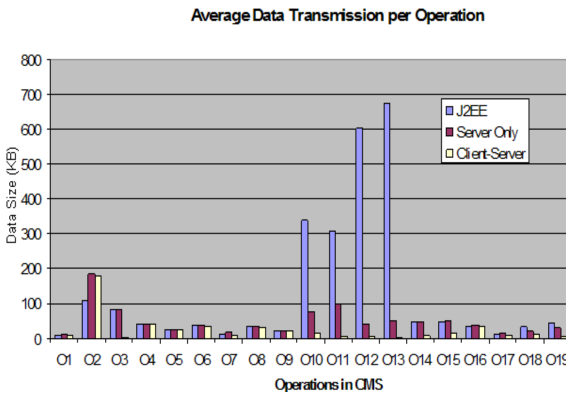

Figure 8: Average Amount of Data Transmitted for Different Operations in CMS

The TPC-W [9] benchmark specifies an online book shop application as the test case for evaluating application server performance. In this application, users can register, view book details, manage their shopping carts and check out, while managers can add new book details into their inventory. We implemented the application using both J2EE and Hilda, and evaluated the average response time of the two systems using a trace synthesized according to the specifications in the benchmark. In the J2EE version, we did not implement any application logic at the client side except for the basic HTML presentations. We took the first 5 percent of the workload as training set for the system to collect statistics, and then measured the response time after the application ran with the computed optimal partition for the Hilda version with partitioning enabled.

Table 4, Figure 9 and Figure 10 show the average response time and the average data transmission for each operation of the application, in the J2EE version and the Hilda versions with and without automatic partitioning. The Hilda system benefits from activating instances of shopping cart AUnit at the client side. A user can add the book she viewed (O5) into the shopping cart (O6) and view the details at a later time (O7), possibly before checkout. The shopping cart and the details about the books in the shopping cart are cached along with the AUnit instance, which make the add to the cart (O6) and view detail (O7) operations locally executable, resulting in a much better response time.

Operation	Description	Number
O1	View website homepage	118
O2	Register as new user	999
O3	Add a book to product list	2098
O4	Register an author of a book	970
O5	View book details	1542
O6	Add a book into shopping cart	4593
O7	View shopping cart details	814
O8	View checkout page	918
O9	Checkout	1799
O10	View order status	920

Table 3: Operations in TCP-W Online Bookstore Application

System	Average Response Time(ms)	Average Data Transmission(KB)
J2EE	221.80	21.7
Server Only	231.88	21.9
Client Server	143.48	3.3

Table 4: Average Response Time and Data Transmission for TPC-W

5. RELATED WORK

In recent years, many programming models and frameworks [10, 4, 8, 7, 12] have been proposed for designing and developing web applications. A few of these frameworks also propose a high level programming model to develop application logic for different tiers of an application. Hilda takes the further step of automating the process of client-server partitioning using a quantitative approach.

Caching data and query results at clients is a concept that has been studied in relational and object-oriented database systems. Work in this area has focused on Transactional Client-Server Cache Consistency [11, 24, 17], a technique that evaluates part of a transaction at the client by shipping it the required data. This work is concerned with guaranteeing the ACID properties of a transaction, and proposes many different approaches such as the Avoidance Based Approach (Adaptive CallBack Locking) and the Detection Based Approach (Adaptive Optimistic Concurrency Control). However, the work assumes a predefined partition of transactions across the server and the client, and thus is complementary to what Hilda achieves.

Hybrid Shipping Architectures have been proposed to run queries in a distributed setting [21, 22]. The motivation behind these systems is that data shipping (query execution at clients) and query shipping (query execution at servers)

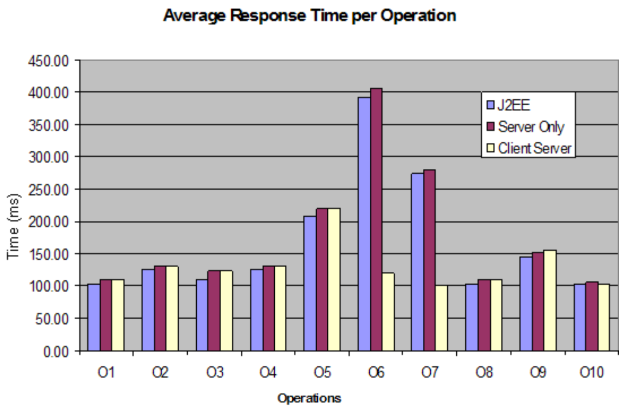

Figure 9: Average Response Yime for Different Operations in TPC-W

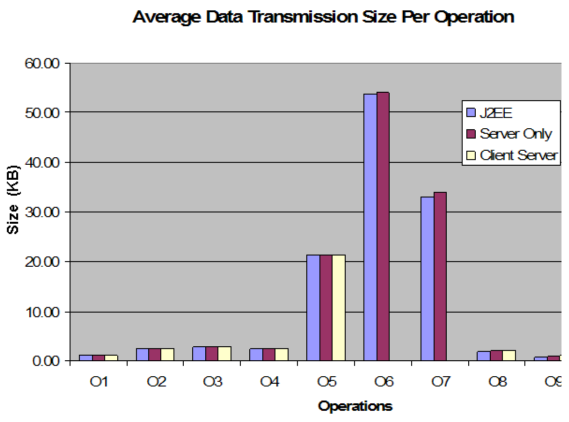

Figure 10: Average Amount of Data Transmitted for Different Operations in TPC-W

can be done together. However, these architectures only consider the partitioning of a single read-only query. They decompose each query into operators such as join, scan and display etc. and then distribute these operations across different sites, taking into account the parallelism and communication costs. They use standard optimization techniques to achieve this. Our goal, on the other hand, is to partition queries in one transaction across the server and the client and to cache data for multiple queries.

Another related area of research is Mobile Code, which aims at transforming a centralized program into a distributed architecture and utilizing resources in distributed systems [16, 23, 3, 19]. The system in [16] takes the binary code of a program and distributes the components and the procedures among a cluster in order to optimize the communication cost. Wang et al. address the problem of partitioning programs in the context of mobile devices [23]. They represent a program in the form of a Task Control Flow Graph (TCFG), i.e. a directed graph, where each node represents a task, and each edge represents data transfer between the tasks. Their cost model includes computation time, communication time, scheduling time, and data registration time. They formulate the optimization problem as a parameterized min-cut/max-flow problem, where common parameters include buffer size, input size, command-line options, etc. The Abacus system [3] consists of a programming model and a run-time system. The proposed programming model encourages the programmer to develop data-intensive applications using small, functionally independent components or objects. The run time system automates the placement of the objects in data-intensive applications and file systems among the nodes of a cluster. The J-Orchestra[19] system partitions Java applications into distributed ones using Java RMI. By rewriting the code using Java RMI, their system can distribute components which share data in memory and thus result in finer granularity for partitioning. However, none of this work consider the concepts of consistency and conflicts for the cached table data between client and server sides. Another drawback is that the language model used by all of this work is not declarative, and therefore the efficacy of the system is limited by how programmers code the components and the procedures.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduced a unified platform for data driven web applications. The platform is based on Hilda, a high level declarative language that allows dynamic partitioning of the web application between the client and the server in a manner that is completely transparent to the developer. This automatic partitioning helps in avoiding manual application partitioning decisions, which can be ad hoc and suboptimal. Based on the observed workload, the Hilda run time system determines a client-server partition of the application, which is close to the optimal partition, using a quantitative method. We also illustrated the benefits of Hilda and automatic client-server partitioning by comparing it with J2EE, using two web applications — a Course Management System with a real workload and an Online Book Store with a benchmark workload. We showed that the performance of the CMS is comparable for both Hilda and J2EE, and that Hilda gains on the amount of data transferred between the client and the server.The TPC-W benchmarked Online Book Store illustrated a 35 percent improvement in response time for Hilda over a J2EE implementation of the same.

The current Hilda optimization model treats each user operation independently, but does not take into account the client side operations performed by the users. Interesting techniques such as asynchronous prefetching and anticipating user actions to prefetch data are not supported. The optimization goal currently focuses only on improving a user's experience and the system's response time. It would be interesting to consider other goals for optimization, such as system throughput by automating load balancing at server side.

7. REFERENCES

[1] Adobe flash. http://en.wikipedia.org/wiki/Macromedia_Flash.

[2] Asynchronous javascript and xml. http://en.wikipedia.org/wiki/Ajax_(programming).

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *USENIX 2000 Annual Technical Conference, San Diego, CA, June 2000.*, pages 307–322, 2000.

[4] A. Bongio, S. Ceri, P. Fraternali, and A. Maurino. Modeling data entry and operations in webml. In *The World Wide Web and Databases (WebDB, Selected Papers)*, pages 201–214, 2000.

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

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

[7] M. Brambilla and others. Declarative specification of web applications exploiting web services and workflows. In *Proc. SIGMOD*, pages 909–910, 2004.

[8] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. In *Proc. the ninth International World Wide Web Conference*, 2000.

[9] T. W. Commerce. Tpc benchmark http://www.tpc.org/tpcw/.

[10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Submitted to ESOP 2007.*

[11] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Trans. Database Syst.*, 22(3), 1997.

[12] P. Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(3):227–263, 1999.

[13] N. Gerner, F. Yang, A. Demers, J. Gehrke, M. Riedewald, and J. Shanmugasundaram. Automatic clientserver partitioning of data driven web applications. In *Proc. SIGMOD*, 2006.

[14] A. Hayrapetyan, D. Kempe, M. Pál, and Z. Svitkina. Unbalanced graph cuts. In *European Symposium on Algorithms (ESA), Mallorca, Spain*, 2005.

[15] http://java.sun.com/j2se/1.4.2/docs /guide/plugin/developer_guide/applet_caching.html.

[16] G. C. Hunt and M. L. Scott. The coign automatic distributed partitioning system. In *Operating Systems Design and Implementation*, pages 187–200, 1999.

[17] M. Ozsu, K. Voruganti, and R. Unrau. An asynchronous avoidance-based cache consistency algorithm for client caching dbmss, 1998.

[18] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, 3 edition, 2003.

[19] E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning. *European Conference on Object-Oriented Programming (ECOOP), Malaga, June 2002.*

[20] V. V. Vazirani. *Approximation Algorithms.* Springer-Verlag, Berlin, 2001.

[21] K. Voruganti, M. T. Ozsu, and R. C. Unrau. An adaptive hybrid server architecture for client caching ODBMSs. In *The VLDB Journal*, pages 150–161, 1999.

[22] K. Voruganti, M. T. Özsu, and R. C. Unrau. An adaptive data-shipping architecture for client caching data management systems. *Distrib. Parallel Databases*, 15(2):137–177, 2004.

[23] C. Wang and Z. Li. Parametric analysis for adaptive computation offloading. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, 2004.

[24] K. Wu, P. fei Chuang, and D. J. Lilja. An active data-aware cache consistency protocol for highly-scalable data-shipping dbms architectures. In *CF '04: Proceedings of the 1st conference on Computing frontiers*, 2004.

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