MANTICORE: a Framework for Partitioning Software Services for Hybrid Cloud

Nima Kaviani
Department of Computer Science
University of British Columbia
Vancouver, Canada
nkaviani@cs.ubc.ca

Eric Wohlstadter
Department of Computer Science
University of British Columbia
Vancouver, Canada
wohlstad@cs.ubc.ca

Rodger Lea
Media and Graphics Interdisciplinary Centre
University of British Columbia
Vancouver, Canada
rlea@magic.ubc.ca

***Abstract*—Hybrid cloud deployment can be an attractive option for companies wanting to deploy software services on scalable public clouds, while still assuming local control over sensitive data resources. A hybrid deployment, despite providing better control, is difficult to design since code must be partitioned and distributed efficiently between public and private premises. This paper describes our research into automated partitioning of software services for hybrid clouds. We have identified two specific shortfalls of existing partitioning research which are important to a hybrid cloud setting: (i) inflexibility in placement of software function execution between public/private hosts and (ii) no support for making explicit tradeoffs between monetary cost and performance. We propose a new software profiling and partitioning framework (called MANTICORE) which addresses these problems. Experiments on an open-source Web application show that the new approach ensures better performance without increasing costs.**

I. Introduction

Pressure to assure data confidentiality, integrity, and auditability in the cloud, as well as concerns on data lock-in, have increased the interest in *hybrid cloud architecture* [1]. In a hybrid deployment, software developers have control over which components to place in the public cloud and which to keep privately on premise. However, a hybrid deployment is difficult to design since a service's software implementation must be partitioned and distributed efficiently between public and private hosts. Consequently, utilizing automated techniques to help with analysis and deployment of service implementation in a hybrid cloud is useful. Such techniques have been explored for other domains and are referred to as *application partitioning*. Unfortunately, existing techniques [2], [3], [4] fail to capture certain software and business parameters important to a hybrid cloud deployment.

Problem #1: Existing research on application partitioning provides techniques to determine the optimal mapping of software functions to network hosts (e.g. client or server) [2], [3], [4]. This research only supports a simple one-to-one mapping of functions to hosts. However, in our research we have found this simple mapping to be inadequate because the optimal placement of a function depends on the context in which that function is used. In short, sometimes it is better to execute a particular function in the public cloud and sometimes it is better to execute it on premise. We call such distinction, *context-sensitive partitioning*, and describe our solution to this problem in this paper.

Problem #2: At its core, application partitioning is a method for applying mathematical optimization to distributed software development. The primary objectives for optimization are performance, i.e. request processing latency, and the monetary cost of deployment. However, previous work does not provide a means for developers to explicitly make tradeoffs between these two objectives. We provide a *flexible cost modeling* technique which allows developers to make tradeoffs between request latency and monetary cost.

In this paper, we discuss MANTICORE, a framework that allows software developers to analyze a monolithic (i.e single host) software service to make it suitable for hybrid cloud. We provide a motivating example (Section II); discuss our approach for modeling software behavior (Section III-A), cost (Section III-B), and partitioning (Section III-B); provide an evaluation (Section IV), related work (Section V); and conclusion (Section VI).

II. Motivating Example

Consider a hypothetical company that wishes to deploy a hybrid version of a stock trading service. As a running example, we consider the Apache DayTrader [5] application. DayTrader is a benchmark that simulates the operations of a stock trading service and has been investigated in other cloud computing research [6], [7]. DayTrader implements seven service request types (i.e. operations) allowing users to login (`doLogin`), view/update their account information (`doAccount` & `doAccountUpdate`), view their portfolio (`doPortfolio`), lookup stock quotes (`doQuotes`), and buy/sell (`doBuy` & `doSell`) stock shares.

In this example, we suppose that the company wishes to shield their sensitive data resources by keeping them on-premise, while still making use of some public cloud computational resources. The task of MANTICORE is to take an application which was originally built to run in a single location and partition it across a distributed hybrid setting. This partitioning process - as illustrated in Figure 1 - works as follows:

First, the original application is executed in a test environment (or profiling capable production environment [8]), collecting traces of resource usage for each function (top of Figure 1). We use an application execution profiler (jip-osgi [8], [9]) to instrument the bytecode for the application with extra monitoring code. The outcome of profiling is then captured as a dependency graph. This dependency graph is

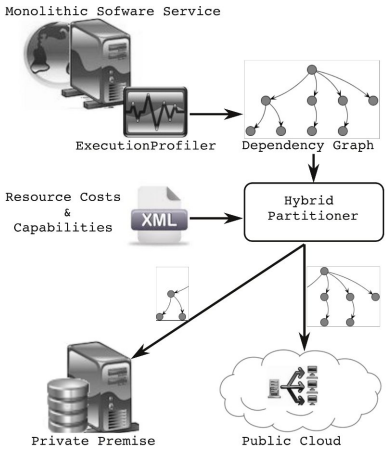

Fig. 1. Overview of the partitioning process: from monolithic software service (top) to hybrid software service (bottom).

a high-level abstraction capturing the relationships between functions relevant for partitioning.

Second, this graph model of an application and an objective function are provided to a graph partitioning algorithm for an optimized deployment to be decided upon (middle of Figure 1). The output is a mapping of functions to network hosts which optimizes for the objective.

Finally, a new system is deployed where the partitioned code is executed on different hosts according to the mapping provided by the algorithm. For functions mapped to the same host, execution happens exactly as it had in the single host version of the system. For function calls where caller and callee have been mapped to different hosts, a remote procedure call is injected into the system in place of the traditional function call. This injection is achieved transparently through binary instrumentation. While previous research has provided a similar partitioning approach, it fails to address two specific problems which we describe here using the DayTrader example.

Problem #1 as it applies to DayTrader (Context-Sensitive Partitioning): The source code for DayTrader implements basic functionality of a stock trading application. For example, there is a function called `getQuote` which takes as an argument a stock name and returns the current market information about that stock. As is normal practice for programming, this function is called from many different places in the application to reuse the function in the processing of different request types. After inspecting this particular application in detail, we realized that executing this function in the public cloud for some request types but on the private premise for other request types makes the overall execution of DayTrader more efficient. We call this capability, context-sensitive partitioning. This new approach is described in Section III-A and evaluated in our experiments.

Problem #2 as it applies to DayTrader (Flexible Cost-Modeling): There are often tradeoffs that need to be made between the provision of lowest monetary cost deployment and best performance in hybrid deployments. The essence of this dilemma is that on one hand, companies may need to keep control over (sensitive) data by keeping it on premise. On the other hand, managing scalable software systems in the public cloud is generally cheaper than serving up software from your own private infrastructure. Some reports claim a typical 80% savings using public cloud versus on-premise private systems [10]. However, putting the code in the cloud and the data on the premise introduces extra latency that degrades the performance.

For example, in DayTrader, many of the functions in implementation code make heavy and repeated use of data resources which would be kept on premise. From the simple perspective of CPU cycles, it would be more cost efficient to lease CPU resources for these functions from the public cloud. However, since those particular functions are tightly bound to data, moving them across the network may introduce unacceptable request processing latency. The key point is that we cannot simply improve upon request latency and cost simultaneously by pushing all software to the cloud, but rather must trade off the benefit of moving particular software functions to the cloud with the negative effect the move has on the overall end-to-end system execution. Our approach to this problem is described in Section III-B and evaluated in our experiments.

III. Approach

In order to present the details of MANTICORE we organize the discussion along the phases of the tool chain (Figure 1): dependency analysis (Section III-A), cost modeling (Section III-B), and partitioning (Section III-C).

A. Application Dependency Analysis

In this section we introduce three different graph models of application behavior. The first two models described here were explored in previous research (presented for background and used for a baseline comparison), whereas the third model is our own context-sensitive model. Common to all three models is the fact that vertices represent execution components in the application with vertex weights corresponding to aggregated CPU usage during the profiling of the application. Edges represent data-links with edge weights capturing latency and data transfer. The models differ in how the notions of *component* and *data link* are realized. For example, nodes might represent low-level function definitions from source code - or - high-level service request types. While both of these capture some notion of an execution component, these different choices have an important effect on what kind of optimization can be achieved. In the next three paragraphs we will refer to Figure 2 to illustrate the details. This figure is simply a high-level comparative mockup and does not correspond to the actual models used by our tool. A real example of a dependency model can be seen in Figure 3.

Request-based Model (RBM): In this model nodes represent either request types or data objects (e.g. relational tables). Edges are created between each request-node and all of the data objects the request operates on [11], [12].

An illustration inspired by DayTrader is provided in Figure 2(a). Two request types `doPortfolio` and `doBuy` are

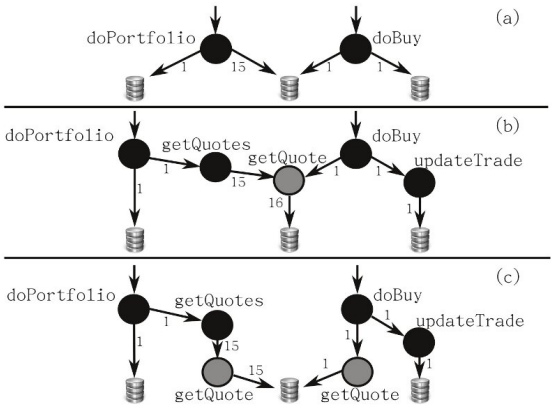

Fig. 2. Application Dependency Models: a) RBM, b) SSM, c) CSM. Circles are application nodes and cylinders are database tables. The edge weights represent number of edge traverses between nodes. For simplicity we do not present the amount of data transferred on each link.

shown, each dependent on some tables from the DayTrader database (the identity of the tables is not relevant for this example). In a hybrid setting, the job of an application partitioner is to decide whether the code for each request type should be executed in the public cloud or the private premise. The decision is based on the CPU demands of processing each request type and the various data dependencies. Executing code in the cloud will yield to higher CPU scalability at a decreased cost as compared to manually administered private CPU resources. However, separating request types from the data they depend on could introduce too much additional latency or bandwidth requirements. For example, in Figure 2(a), we see that `doPortfolio` accesses the middle database table an average of 15 times per request.

Static Structure Model (SSM): A drawback of the *RBM* is that it does not separate request types into individual programming language functions used in the implementation. This means that a partitioner does not have the flexibility to execute part of a request in the cloud and part of it on premise, even if that would provide an advantage. Some other research on application partitioning ([2], [3], [13]) deals with a more implementation-level model of a service, we refer to as *SSM*.

In this model nodes in the graph represent the *definition* of a function or the existence of some data object. By *definition* we mean that a node corresponding to some function occurs only once in the graph, regardless of how many times the function is typically executed during the processing of a request. For data objects, this distinction is not necessary (because data does not execute). An edge between two nodes means that there is at least one call between those functions. During execution profiling, the execution time of each function call is aggregated to compute a total weight for the corresponding node. Likewise for edges, the weight corresponds to the total aggregate data passed between the two functions as well as the number of times the associated edge is traversed, for all calls between the functions.

An illustration inspired by DayTrader is provided in Figure 2(b). Here we see `doPortfolio` and `doBuy` broken down into the functions that implement them. Using this model, the job of an application partitioner is to decide whether each function should execute code in the public cloud or the private premise. However, both request types make use of the function `getQuote`. This causes the edge weights induced by `doPortfolio` and `doBuy` to become conflated, resulting in an overly abstract representation. We found that having only one node representing functions such as `getQuote` may over-constrain the partitioning optimizer causing it to produce suboptimal code placement suggestions. Referring back to Figure 2b, the conflated edge from `getQuote` to the database table bears a weight of 16 for which the profiler is not able to distinguish the weight induced by `doPortfolio` and the weight induced by `doBuy`.

Context-Sensitive Model (CSM): In order to address the problems of the previous two models we have built MANTICORE on a context-sensitive model of software behavior. In this model each node represents the *execution* of a function in a distinct *calling context*. A calling context captures the transitive set of callers to each function execution (aka. an execution stack). Using the execution of a function rather than a function's static definition as the basis for MANTICORE allows our runtime to execute the same function on premise or in the public cloud depending on the situation. Note, however, if we simply tried to apply automated optimization to a graph which contained one node for each distinct function execution in a captured execution profile, the graph would quickly grow too large. Our intuition was that we can capture the most important differences in each function execution by only considering each execution different if it occurs in a different calling context.

An illustration inspired by DayTrader is provided in Figure 2(c). Now we can see that two copies of `getQuote` are created, one where it is in the context of `getQuotes` and one where it is in the context of `doBuy`. Consider, as is the case for DayTrader, that `getQuotes` calls `getQuote` several times in a loop; an average of 15 times as indicated by the edge weight. If `getQuote` was executed only on the public cloud, this would cause a large number of round-trips over the network, so in an optimal deployment it should be placed on premise where it is close to the data. However, in the context of `doBuy`, `getQuote` is only called once, so for this case executing `getQuote` in the public cloud does not incur extra latency, and yet we can take advantage of the public cloud's pricing benefits. Different from previous research, MANTICORE's approach provides this capability for such context-sensitive partitioning.

B. Cost Modeling

Cost modeling involves associating quantifiable metrics to the execution footprint of a given application. Since the overall time of execution for an application directly contributes to the cost of deployment, a simplistic optimization strategy would be to only minimize the total time of execution. However, this does not account for the costs billed by cloud providers. To reflect the costs of a hybrid deployment we augment the

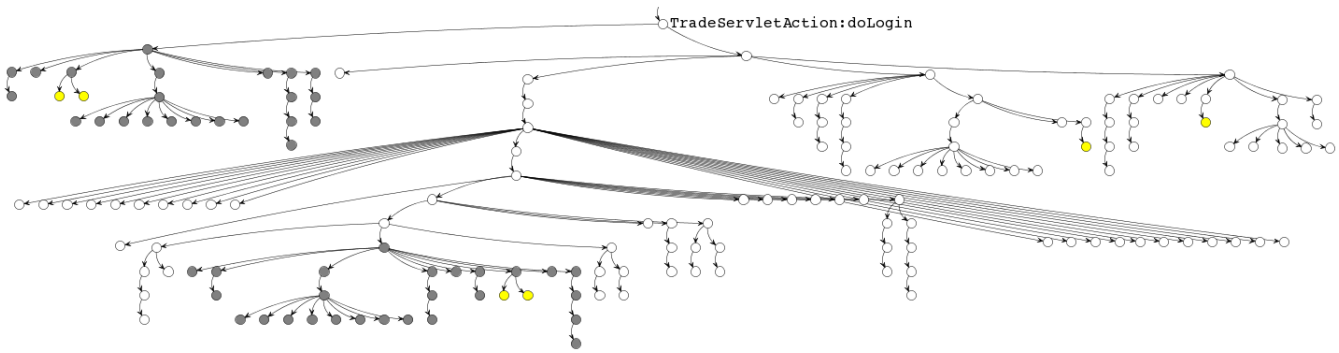

Fig. 3. CSM Visualization of partitioning results for the `TradeServletAction:doLogin` request type (root node of the tree) in the DayTrader example provided by MANTICORE. Yellow nodes (alternatively light gray nodes) represent database tables and all the other nodes represent individual function executions with dark gray nodes being chosen to be placed on premise and white nodes to be placed in the cloud. Note in this example that functions are pulled to the private premise when there is more database access under the partitioned subgraph (although this is not always the case because it also depends on other factors such as CPU usage of functions and size of data transferred).

dependency graphs from Section III-A with cost implications of a hybrid deployment. This is done in two phases.

In the first phase, vertex weights and edge weights need to be updated to reflect the overall time of application execution when deployed across private premise and public cloud. For a given vertex i in the dependency graph, we generate two weights indicating the execution time of any corresponding function on-premise and in the cloud (based on the CPU capabilities of the host machines) by applying linear fitting techniques [14] to its measured CPU usage cycles ($t_{measured_i}$) during the profiling phase:

$$t_{exec_i} = t_{measured_i} \times \frac{CPU_{target}}{CPU_{source}} \tag{1}$$

where CPU_{target} represents CPU capabilities of the target deployment host and CPU_{source} represents CPU capabilities of the machine used during the profiling phase.

The edge weights are set to reflect the amount of time it takes for data to be communicated between two components i, j in the application when those components are split between the cloud and the on-premise data center. We utilize a model similar to the communication latency model suggested in [15] as follows:

$$t_{comm_{i,j}} = \left[\frac{\frac{d_{i,j}}{\phi}}{\left(\frac{\beta}{\phi}\right)^2 - \frac{\beta \times d_{i,j}}{\phi^2}} + \frac{d_{i,j}}{\beta} + \lambda \right] \tag{2}$$

where $d_{i,j}$ indicates the amount of data communicated on the edge e, β is the communication bandwidth between the premise and the cloud provider, and ϕ is the frequency of sending d bytes of data between i and j in each time unit. The first expression determines the queuing delay based on the assumption that the requests for data transfer on the network follow a Poisson distribution; $\frac{d_{i,j}}{\beta}$ is the data transmission time between the premise and the cloud; and λ is a constant representing the communication latency for a given public cloud.

In the second phase, we include implications of cloud deployment. A realistic model starts by accounting for the actual cost of deployment by applying cost schemes offered by a cloud provider. We have encoded the pricing models for several popular platforms in an XML format that is provided as input to MANTICORE (center of Figure 1). Given this information, MANTICORE updates vertex weights for the graph model as follows:

$$cost_{exec_i} = \alpha \times cost_{exec_{unit}} \times \left(\frac{t_{measured_i}}{T_{unit}} \right) \times \frac{CPU_{target}}{CPU_{source}} \tag{3}$$

where T_{unit} represents the time unit for which cloud charges apply, $cost_{exec_{unit}}$ indicates cloud charges for each T_{unit}, and α is the cost ratio of running each function on a target premise machine versus running it on the cloud machine. α is configurable by the system architect. In our evaluations we change it from 1 to 25 for varied premise deployment costs, evaluating a 0 to 25 times cost saving for public cloud deployment.

Next we model the monetary cost of communication between vertices i and j as follows:

$$\begin{aligned} cost_{comm_{i,j}} &= \gamma \times cost_{exec_{unit}} \times \left(\frac{t_{comm_{i,j}}}{T_{unit}} \right) \\ &+ \frac{d_{i,j}}{D_{unit}} \times cost_{comm_{unit}} \end{aligned} \tag{4}$$

The first part of the above equation accounts for charges due to the latency introduced in the hybrid cloud deployment (i.e., by introducing remote function calls), while the second part of the equation accounts for charges directly related to transferring data between functions deployed in the cloud and the ones on premise. In the above formula, D_{unit} represents the data unit for which cloud charges apply and $cost_{comm_{unit}}$ indicates cloud charges for each D_{unit}.

Finally, we introduce γ as a configurable parameter reflecting the effect of latency on the cost of deployment. This allows developers to use our cost model to make flexible tradeoffs between monetary cost and latency. The larger a developer chooses the value of γ, the algorithm will work towards minimizing communication latency and improving the round trip time; whereas a smaller γ diminishes the effect of latency in searching for the optimal monetary deployment cost.

We can formulate γ in the equation below:

$$\gamma = \frac{T_{unit} \times cost_{latency_{unit}}}{cost_{exec_{unit}} \times T_{latency_{unit}}} \quad (5)$$

with $cost_{latency_{unit}}$ being the monetary effect of having the latency time ($T_{latency_{unit}}$) incurred during an end-to-end execution of a request. The formulation of Equation 5 defines γ in relation to T_{unit} and $cost_{exec_{unit}}$, allowing for latency costs to be tied into the cost measures given in a public cloud provider's cost schema. For example, given a system with 100 req/sec and a latency of 10msec/req, there will be 1 second of latency for every second of system execution. If a software developer defines a cost-to-latency policy indicating that every hour of time wasted on latency ($T_{latency_{unit}}$) is worth \$0.16 ($cost_{latency_{unit}}$), and given $cost_{exec_{unit}}$ is \$0.32 per hour (T_{unit}), Equation 5 reveals the value of γ to be set equal to 1 for the cost formulation to account for the overhead.

C. Partitioning

Integer programming (IP) is commonly leveraged as the underlying optimization procedure for partitioning [4], [13], [16]. We take the augmented dependency graph from Section III-B and convert it to an IP. For every node i we consider a variable x_i in the IP formulation, where set s refers to functions executed on premise and set t executed in the cloud.

$$\begin{aligned} x_i &\in \{0,1\} \\ \forall x_i \in s, x_i &= 0 \\ \forall x_i \in t, x_i &= 1 \end{aligned} \quad (6)$$

With the above constraints we have the following objective defined (The quadratic expression in the objective function can be relaxed by making the expansion suggested in [4]):

$$\min \sum_{i \in V} x_i cost_{exec_i} + \sum_{(i,j) \in E} (x_i - x_j)^2 cost_{comm_{i,j}} \quad (7)$$

Finally, with this information our partitioner provides an optimal mapping of function execution to hosts for each request type (bottom of Figure 1) using guidance from developers on their preference of γ. As a convenience to developers, MANTICORE provides a visualization of the partitioned CSM; an example of this for DayTrader is shown in Figure 3.

D. Implementation

MANTICORE is implemented in Java and consists of two sub-components: *i*) The *jip-osgi* profiling framework [8], [9] which combines profiling capabilities of the Java interactive profiler (JiP) [17] with extra instrumentation capabilities for measuring cross-method data transfer and data exchange with database engines, and *ii)* The MANTICORE *Analyzer* [18] which performs post-analysis of profiling data by generating the dependency models, cost models, application partitioning and distribution models. Implementation details of MANTICORE can be found online [19].

IV. Evaluation

We performed experiments to test the validity of MANTICORE using DayTrader as described in Section II. For our public cloud machine we used a Large Amazon EC2 instance from the US West region (Oregon), with 7.5 GB of memory, and 4 EC2 Compute Units. For our premise machine we used a 3.5 GHz dual core machine with 4.0 GB of memory at our lab in Vancouver. For our database server, we also setup MySQL 5.1 on a third premise machine with a 2.5 GHz dual core CPU, 4.0 GB of memory, also in Vancouver. All three machines were running Ubuntu 11.10. The machines were connected with a data link of 100 Mb/sec and we measured the latency to be 15 milliseconds between the cloud and the premise.

For all the experiments, unless otherwise stated, we set the cost of execution on a cloud machine to be \$0.32 per hour, while the cost of data transfer is \$0.12 per GByte when data is going from the cloud to the premise and \$0 per GByte when data is going from premise to cloud[1].

Throughout this section we refer to three types of deployments: *i) premise:* meaning all functions are executed on the premise; *ii) cloud:* meaning all functions are executed in the cloud with data on the premise; and *iii) manticore-hybrid-deployment (MHD):* implying a deployment with cuts in the execution such that some function execution takes place on the premise and some in the cloud, with data on premise.

A. Cost Models vs Measured Deployments

In order to verify the accuracy of the cost models, we compare the execution and data transfer measurements of models generated by MANTICORE to those of real deployments for DayTrader. Since our profiling data was collected on the premise machine, the models and the real deployment for the machine on premise are identical. In a hybrid or a full cloud deployment, the models are generated by applying linear fitting techniques as described in Section III-B. We compared the generated cost models with the real deployment of the DayTrader application for settings where the deployment is fully in the cloud or is *MHD*. Figure 4 compares the results for MANTICORE generated models and practical deployments for *overall execution time*. The results are averaged over 1000 requests to each request types.

As can be seen from Figure 4, the execution time for the generated models in case of the *MHD* provides 81.3% average estimation accuracy (σ=0.013) while models for full cloud deployments provide 86.1% average estimation accuracy (σ=0.0052) compared to a practical deployment. Similarly, we compared the actual data transfer to the modeled data transfer. For data transfer the *MHD* model provides 87.4% average estimation accuracy compared to the real deployment (σ=0.0061) while for a full cloud deployment we get 86.3% average estimation accuracy (σ=0.0076). In the following sections, we discuss real deployments of DayTrader as suggested by MANTICORE, and as results follow, we verify that using

[1]The cost scheme used for our evaluations is identical to the on-demand cost scheme offered by Amazon EC2 [20].

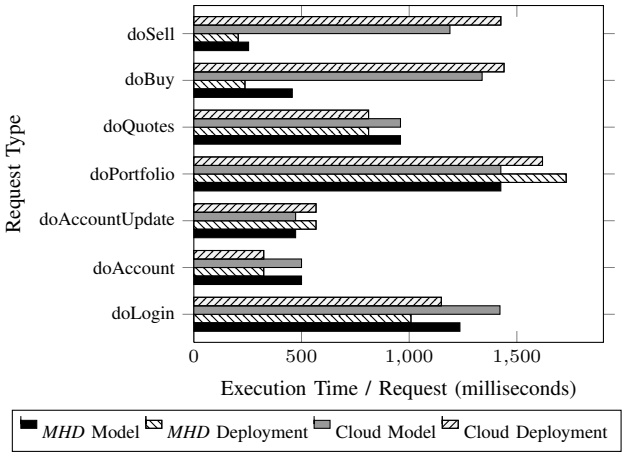

Fig. 4. Simulated and measured execution times for hybrid and cloud code placements for each DayTrader request type

cost models of this accuracy can lead to real benefits in actual deployments.

B. Evaluation of Context-Sensitive Modeling

Next, we evaluate the three dependency models from Section III-A to see which one contributes to a more performant deployment under varied premise cost and fixed cloud cost. We used MANTICORE to decide about function placements for DayTrader as the cost of deployment to a premise machine linearly changes from \$0.16 per hour to \$2.5 under an expected load of 100 requests per second. For each cost value, we took the mapping of function executions suggested by MANTICORE for each of the models and physically deployed it. We then measured the average perceived latency across all the requests to see how the partitioning affects the overall performance of the system. Results are shown in Figure 5. The line at the top of the graph shows the full deployment of the code to the cloud whereas the bottom line indicates a full premise deployment.

We observe that *SSM* quickly yields to a full deployment to the cloud (cf. Figure 5). This is mainly because this model

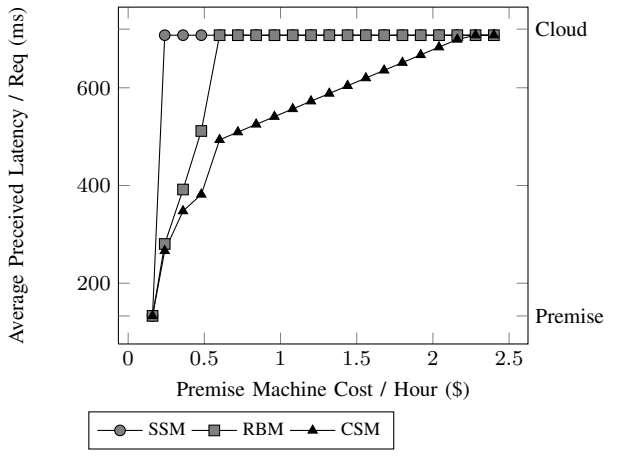

Fig. 5. Comparison of latency adjustments for the SSM, RBM, and CSM as the premise cost changes.

does not account for function replication and thus, separation of code elements would imply cutting a lot of code dependency links which would significantly increase the communication latency to a bigger average value compared to a full cloud deployment. To prevent this, upon premise cost increase, the analysis framework quickly chooses to have all the code in the cloud and only pay for the communication latency when it comes to retrieving data from database tables.

For *RBM*, there is more freedom in choosing code placement as the dependency model consists of subgraphs separated by request types. Compared to the SSM, this model is more tolerant of changes to the cost of deployment on premise and compensates between cost and performance by gradually pushing the code for different request types from premise to the cloud to the point where all the code is in the cloud.

CSM provides the most performant choices for a hybrid deployment. The fine level of granularity in the model allows for replication of code while enabling the partitioning algorithms to find the edges with the lowest performance overhead to be cut. As a result, unlike SSM and RBM, cuts made in the CSM do not solely separate code from data but also separate code units with low communication overhead from one another thus optimizing the deployment.

To summarize Figure 5, when cost of deployment to premise is very cheap, all three models choose to have the code deployed on premise. Similarly, when cost of deployment to the premise is very expensive, all three models choose to push all the code to the cloud. However, between these two spectrums, CSM provides more latency aware deployments by not only separating code from data but also by separating code units with lower performance overhead from one another, yielding to better performing deployments.

C. Evaluation of Flexible Cost Modeling

Performance degradation is one of the biggest concerns when it comes to separation of *code from data* or *code from code* for a distributed application deployment. The degradation is mostly concerned with the extra latency added to the overall execution process as a matter of having the data sent and received over the network. In this section we show how by changing γ from Equation 5, MANTICORE allows for latency and cost to be traded for one another.

With a measured roundtrip communication latency of 15ms between the cloud machine and our premise machine, we gradually increased γ from 0.5 to 15 (i.e. $\times 30$) to see how it affects function placement decisions and the corresponding deployment costs. To measure deployment costs, we set $\alpha = 5$ representing 80% cost saving (cf. Equation 3). Table I shows code placement decisions made by the partitioning framework for different request types.

As shown in Table I, with $\gamma = 0.5$, the partitioning algorithm chooses to have all functions in the cloud where the deployment would be the cheapest (low γ favors monetary cost over latency). As γ increases, the algorithm gradually pushes more and more functionality to the premise in order to accommodate a more performant deployment, either by making a cut in the code (*MHD*) or by pushing the entire

code for a request type to the premise (*premise*). Figure 6 shows the increase in cost of deployment as the functions are gradually pushed to the premise. As we already discussed in Section IV-C, cost of deployment is increased as the partitioner moves more code entities to the premise.

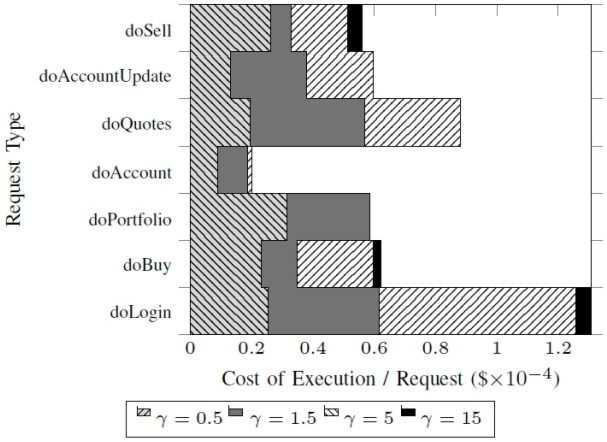

Fig. 6. Monetary cost of deploying requests of various type for DayTrader with respect to changes in Gamma (γ)

Comparing each deployment with its predecessor and successor deployments illustrates that changing γ for the partitioning algorithm can balance between cost and performance of the deployed application. For example, the deployment when γ equals 5 is (on average) 50.9% more expensive than the successor deployment suggested when γ equals 1.5, yet its average performance is only 21.7% slower (performance details are not shown in a graph, due to space limits). However, the deployment suggested for when γ equals 5, compared to its predecessor deployment when γ equals 15, is on average 24.6% slower but 2.6% more cost effective. As we showed here, software developers are able to decide about the proper value of γ for their distribution model based on the latency and cost policies of their system.

D. Evaluation of Scalability

We also performed a scalability analysis for DayTrader to see how different code placement choices affect application throughput. DayTrader comes with a client workload generator that models user behaviors for browsing and purchasing stocks. For the deployment tests, we used a range of 500 to 4000 simulated clients over a period of 5 minutes and measured throughput. Figure 7 shows the throughput of the system under varied user load. As shown in the figure, the machine on the premise starts to be the most efficient when number of user threads sending requests is below 2500. Once we passed this threshold, the premise machine got overloaded and was unable to properly handle incoming requests.

RequestType	**Deployment choice for varied Gamma (γ)**			
	0.5	**1.5**	**5**	**15**
doLogin	*cloud*	*MHD*	*MHD*	*premise*
doBuy	*cloud*	*MHD*	*MHD*	*premise*
doPortfolio	*cloud*	*premise*	*premise*	*premise*
doAccount	*cloud*	*cloud*	*premise*	*premise*
doQuotes	*cloud*	*cloud*	*premise*	*premise*
doAccountUpdate	*cloud*	*cloud*	*premise*	*premise*
doSell	*cloud*	*MHD*	*MHD*	*premise*

TABLE I
CODE PLACEMENT FOR DIFFERENT REQUEST TYPES AS GAMMA (γ) CHANGES FROM 0.5 TO 15.

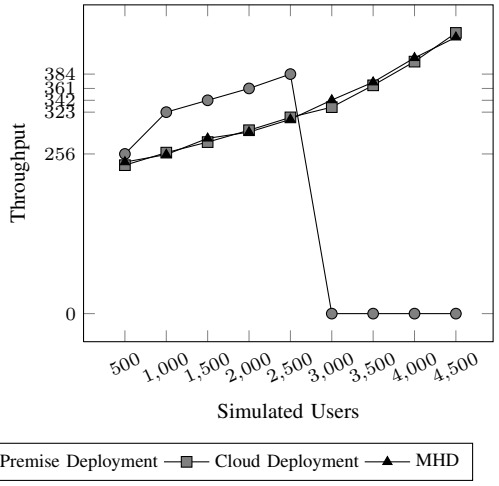

Fig. 7. Scalability tests for full premise, full cloud, and hybrid deployments

In contrast, for the *MHD* and the cloud deployment, the extra latency introduced as a result of partitioning reduced the overall system throughput compared to the premise deployment when the server was handling less than or equal to 2500 user threads. However, As more user threads were added those deployments scaled better than a full premise deployment.

Finally, we notice that the hybrid and cloud deployment had very similar scalability. However, as shown in Section IV-B a hybrid deployment using our *CSM* provides better response time when CPU is not a bottleneck. So for the DayTrader case study we see clear advantage in using *Manticore* over simple public cloud or private premise deployments.

V. Related Work

Application partitioning has been explored extensively for client-server architectures in a variety of network settings: *i)* in a wide-area setting [2], *ii)* within a LAN [3], or *iii)* across a wireless network [4]. These client-server systems are different from our setting since we are in essence working towards partitioning between backend servers, not server and client. At a low-level this difference manifests itself as different optimization goals for partitioning. In client-server partitioning the goal is to partition software by pushing code tightly coupled to front-end interaction towards the client and pushing the code that works on shared persistent data to the server. In our hybrid cloud partitioning, our goals are driven by maximizing the use of elastic public cloud infrastructure while constraining the placement of private resources.

Partitioning for hybrid clouds differs in that extra constraints on geographical placement of code, types of instances leased from the cloud, and their associated charges in the public

cloud must be taken into consideration. Within the context of cloud, Volley [11] increases performance and reduces data center traffic by data relocation, yet it does not deal with code partitioning. Efforts like Conductor [21] and HybrEx [22] suggest hybrid deployments for cost optimizations or security considerations in the cloud, but their main focus in on Map/Reduce type of applications. Other approaches such as CloudCmp [23], CiteSeer^x [24], and the work by Truong & Dustdar [25] look into cost savings of software service deployment to the cloud by analyzing resource consumption; yet their cost-considerations do not drive deployment decisions using partitioning techniques.

CloneCloud [16] optimizes for high performance and minimum resource usage for applications in mobile and embedded devices by offloading the execution to the cloud; but optimizing towards a cheaper deployment is not the focus for CloneCloud. Cloudward Bound [13] and COPE [26] optimize for cost of deployment in a hybrid setting, however there are several differences between these approaches and MANTICORE. For these approaches, partitioning and relocation of components happens at the level of application servers (or VMs) not the finer level of code entities (i.e. functions) as is the case with MANTICORE. Furthermore, none of these approaches supports context sensitivity. Finally, while COPE does not account for latency, Cloudward Bound enforces the accepted latency by defining an upper limit constraint, whereas MANTICORE allows for software developers to decide about their preferred cost-to-latency ratios.

VI. Conclusion and Future Work

In this work, we proposed an extension to existing application partitioning techniques to provide for hybrid deployment of software services. The evaluation on DayTrader showed that the new approach can more effectively contribute towards an optimized hybrid cloud deployment. In particular, it showed that: the costs of a hybrid deployment extrapolated from monitoring a single-host test version of the service were at least 81.3% accurate (Section IV-A); the context-sensitive modeling of service behavior provided a better representation to optimize placement of software function execution (Section IV-B); our formulation of the objective function for optimization allows developers to tune the tradeoff between end-to-end round-trip time and deployment costs (Section IV-C); and that a hybrid deployment using MANTICORE, while providing similar scalability as a full cloud deployment, offers better round-trip latency under comparable load (Section IV-D).

In the future, we have two specific plans to address some limitations of the current research. First, our current framework addresses the partitioning of code, but not the partitioning of data, since we have focused on the scenario where data is kept on premise. However, for some businesses it might only be required that a subset of data be kept on premise. In that case, it would be advantageous for MANTICORE to guide the developer in the decision of which data entities could be moved to the cloud without separating tightly coupled data entities (e.g. database tables that are frequently joined). Second, while our context-sensitive approach is better at capturing the important dynamics of an implementation compared to previous research, it is not completely adaptive to dynamic fluctuations in workload. In the future it may be possible to change partitioning decisions "just-in-time" during the execution of the system in case workload differs significantly from that observed during the use of our profiling instrumentation.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[2] S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Building secure web applications with automatic partitioning," in *Proc. of the Symposium on Operating Systems Principles (SOSP)*, 2009.

[3] G. Hunt and M. Scott, "The Coign automatic distributed partitioning system," in *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[4] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based Partitioning for Sensornet Applications," in *Proc. of the NSDI*, 2009.

[5] Apache DayTrader Benchmark - Online: https://cwiki.apache.org/GMOxDOC20/daytrader.html.

[6] Y. Ueda and T. Nakatani, "Performance variations of two open-source cloud platforms," in *the Int'l Symp. on Workload Characterization*, 2010.

[7] W. Iqbal, M. N. Dailey, and D. Carrera, "SLA-driven dynamic resource management for multi-tier web applications in a cloud," in *Proc. of the Int'l Conf. on Cluster, Cloud and Grid Computing (CCGRID)*, 2010.

[8] N. Kaviani, E. Wohlstadter, and R. Lea, "Profiling-as-a-Service: Adaptive Scalable Resource Profiling for the Cloud in the Cloud." in *Int'd Conf. on Service Oriented Computing (ICSOC)*, 2011.

[9] JiP-OSGi - Online: http://code.google.com/p/jip-osgi/.

[10] Microsoft, "The Economics of the Cloud," USA, November 2010.

[11] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman, "Volley: Automated data placement for geo-distributed cloud services." in *NSDI*, 2010.

[12] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *Proc. of the Int'l Conf. on Management of Data*, 2004.

[13] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, and M. Tawarmalani, "Cloudward bound: planning for beneficial migration of enterprise applications to the cloud," in *SIGCOMM*, 2010.

[14] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *Proc. of the NSDI*, 2005.

[15] J. M. Johansson, "On the impact of network latency on distributed systems design," *Information Technology and Management*, vol. 1, pp. 183–194, 2000.

[16] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud." in *Proc. of the European Symposium on Operating Sytems (EuroSys)*, 2011.

[17] The Java Interactive Profiler - Online: http://jiprof.sourceforge.net/.

[18] The Manticore Source Repository - Online: https://github.com/nkaviani/ca.ubc.magic.partitioning.analyzer.

[19] Manticore Homepage - Online: http://nima.magic.ubc.ca/manticore.

[20] Amazon Web Services (AWS) Pricing - Online: http://aws.amazon.com/ec2/pricing/.

[21] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues, "Orchestrating the deployment of computations in the cloud conductor," in *NSDI*, 2012.

[22] S. Y. Ko, K. Jeon, and R. Morales, "The HybrEx model for confidentiality and privacy in cloud computing," in *Proc. of HotCloud*, 2011.

[23] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Shopping for a Cloud Made Easy," *HotCloud*, 2010.

[24] P. Teregowda, B. Urgaonkar, and C. Giles, "CiteSeerx: a Cloud Perspective," in *Proc. of the HotCloud Workshop*, 2010.

[25] H. L. Truong and S. Dustdar, "Composable cost estimation and monitoring for computational applications in cloud computing environments," *Procedia CS*, vol. 1, no. 1, pp. 2175–2184, 2010.

[26] C. Liu, B. T. Loo, and Y. Mao, "Declarative automated cloud resource orchestration," in *Proc. of the Symposium on Cloud Computing*, 2011.