Sloth: Being Lazy Is a Virtue (When Issuing Database Queries)

ALVIN CHEUNG, University of Washington
SAMUEL MADDEN and ARMANDO SOLAR-LEZAMA, MIT CSAIL

Many web applications store persistent data in databases. During execution, such applications spend a significant amount of time communicating with the database for retrieval and storing of persistent data over the network. These network round-trips represent a significant fraction of the overall execution time for many applications (especially those that issue a lot of database queries) and, as a result, increase their latency. While there has been prior work that aims to eliminate round-trips by batching queries, they are limited by (1) a requirement that developers manually identify batching opportunities, or (2) the fact that they employ static program analysis techniques that cannot exploit many opportunities for batching, as many of these opportunities require knowing precise information about the state of the running program.

In this article, we present SLOTH, a new system that extends traditional *lazy evaluation* to expose query batching opportunities during application execution, even across loops, branches, and method boundaries. Many such opportunities often require expensive and sophisticated static analysis to recognize from the application source code. Rather than doing so, SLOTH instead makes use of dynamic analysis to capture information about the program state and, based on that information, decides how to batch queries and when to issue them to the database. We formalize extended lazy evaluation and prove that it preserves program semantics when executed under standard semantics. Furthermore, we describe our implementation of SLOTH and our experience in evaluating SLOTH using over 100 benchmarks from two large-scale open-source applications, in which SLOTH achieved up to a 3× reduction in page load time by delaying computation using extended lazy evaluation.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications

General Terms: Performance, Languages

Additional Key Words and Phrases: Application performance, lazy evaluation, query optimization

ACM Reference Format:
Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2016. Sloth: Being lazy is a virtue (when issuing database queries). ACM Trans. Database Syst. 41, 2, Article 8 (June 2016), 42 pages.
DOI: http://dx.doi.org/10.1145/2894749

1. INTRODUCTION

Many web applications are backed by database servers that are physically separated from the servers hosting the application. Such applications are usually hosted on an application server, interact with users via webpages, and store persistent data in a database server. Even though the application server and the database server tend to reside in close proximity (e.g., within the same data center), a typical page load

This work is supported by the Intel Science and Technology Center for Big Data and the National Science Foundation, under grant SHF-1116362 and SHF-1139056.
Authors' addresses: A. Cheung, Computer Science & Engineering, University of Washington, Box 352350, Seattle, WA 98195; email: akcheung@cs.washington.edu; S. Madden and A. Solar-Lezama, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, MA 02139; emails: {madden, asolar}@csail.mit.edu.

DOI: http://dx.doi.org/10.1145/2894749

spends a significant amount of time issuing queries and waiting for network round-trips to complete, with a consequent increase in application latency. The situation is exacerbated by object-relational mapping (ORM) frameworks such as Hibernate and Django, which access the database by manipulating native objects rather than issuing SQL queries. These frameworks automatically translate accesses to objects into SQL, often resulting in multiple queries (thus round-trips) to reconstruct a single object. For example, with the applications used in our experiments, we found that many webpages spend 50% or more of their time waiting on query execution and network communication as they load, even with the application and database servers hosted in the same data center.

Latency is important for many reasons. First, even hundreds of milliseconds of additional latency can dramatically increase the dissatisfaction of web application users. For example, a 2010 study by Akamai suggested that 57% of users will abandon a webpage that takes more than 3s to load [Akamai 2010]. As another example, Google reported in 2006 that an extra 0.5s of latency reduced the overall traffic by 20% [Linden 2006]. Second, ORM frameworks can greatly increase load times by performing additional queries to retrieve objects that are linked to the one that was initially requested; as a result, a few tens of milliseconds per object can turn into seconds of additional latency for an entire webpage [StackOverflow 2014b, 2014c, 2014d, 2014e]. Though some techniques (such as Hibernate's "eager fetching") aim to mitigate this, they are far from perfect, as we discuss later. Finally, decreasing latency often increases throughput: as each request takes less time to complete, the server can process more requests simultaneously.

There have been two general approaches for programs to reduce application latency due to database queries: (i) hide this latency by overlapping communication and computation, or (ii) reduce the number of round-trips by fetching more data in each one. Latency hiding is most commonly achieved by prefetching query results so that the communication time overlaps with computation and the data is ready when the application really needs it. Both of these techniques have been explored in prior research. Latency hiding, which generally takes the form of asynchronously "prefetching" query results so that they are available when needed by the program, was explored by Ramachandra and Sudarshan [2012], who employed static analysis to identify queries that will be executed unconditionally by a piece of code. The compiler can then transform the code so that these queries are issued as soon as their query parameters are computed and before their results are needed. Unfortunately, for many web applications, there is simply not enough computation to perform between the point when the query parameters are available and the query results are used, which reduces the effectiveness of this technique. Also, if the queries are executed conditionally, prefetching queries requires speculation about program execution, and can end up issuing additional useless queries.

In contrast to prefetching, most ORM frameworks allow users to specify "fetching strategies" that describe when an object or members of a collection should be fetched from the database. The goal of such strategies is to allow developers to indirectly control the number of network round-trips. The default strategy is usually "lazy fetching," in which each object is fetched from the database only when it is used by the application. This means that there is a round-trip for every object, but the only objects fetched are those that are certainly used by the application. The alternative "eager" strategy causes all objects related to an object (e.g., that are part of the same collection or referenced by a foreign key) to be fetched as soon as the object is requested. The eager strategy reduces the number of round-trips to the database by combining the queries involved in fetching multiple entities (e.g., using joins). Of course, this eager strategy can result in fetching objects that are not needed and, in some cases, can actually incur *more* round-trips than lazy fetching. For this reason, deciding when to label entities as "eager" is a nontrivial task, as evidenced by the number of questions on online forums

regarding when to use which strategy, with "it depends" being the most common answer. In addition, for large-scale projects that involve multiple developers, it is difficult for the designer of the data-access layer to predict how entities will be accessed in the application and, therefore, which strategy should be used. Finally, fetching strategies are very specific to ORM frameworks and fail to address the general problem, which is also present in non-ORM applications.

This article describes a new approach for reducing the latency of database-backed applications that combines many features of the two strategies described. The goal is to reduce the number of round-trips to the database by *batching* queries issued by the application. Rather than relying entirely on static analysis, as in the case of prior work, the key idea is to collect queries using a new technique that we call *extended lazy evaluation* (or simply "lazy evaluation" in the rest of the article). As the application executes, queries are batched into a query store instead of being executed right away. In addition, nondatabase-related computation is delayed until it is absolutely necessary. As the application continues to execute, multiple queries are accumulated in the query store. When a value that is derived from query results is finally needed (say, when it is printed on the console), then all queries that are registered with the query store are executed by the database in a single batch. The results are then used to evaluate the outcome of the computation. The technique is conceptually related to traditional lazy evaluation as supported by functional languages (either as the default evaluation strategy or as program constructs) such as Haskell, Miranda, Scheme, and OCaml [Jones and Santos 1998]. In traditional lazy evaluation, there are two classes of computations: those that can be delayed and those that force the delayed computation to take place because they must be executed eagerly. In our extended lazy evaluation, queries constitute a third kind of computation because, even though their actual execution is delayed, they must eagerly register themselves with the batching mechanism so that they can be issued together with other queries in the batch.

Compared to query extraction using static analysis [Wiedermann et al. 2008; Iu et al. 2010; Cheung et al. 2013], our approach batches queries dynamically as the program executes, and defers computation as long as possible to maximize the opportunity to overlap query execution with program evaluation. As a result, it is able to batch queries across branches and even method calls, which results in larger batch sizes and fewer database round-trips. Unlike fetching strategies, our approach is not fundamentally tied to ORM frameworks. Moreover, we do not require developers to label entities as eager or lazy, as our system only brings in entities from the database as they are originally requested by the application. Note that our approach is orthogonal to other multiquery optimization approaches that optimize batches of queries [Giannikis et al. 2012]; we do not merge queries to improve their performance, nor do we depend on many concurrent users issuing queries to collect large batches. Instead, we optimize applications to extract batches from a *single* client, and issue those in a single round-trip to the database; the database still executes each of the query statements in the batch individually.

We have implemented this approach in a new system called Sloth. The system is targeted to general applications written in an imperative language that use databases for persistent storage. Sloth consists of two components: a compiler and a number of libraries for runtime execution. Unlike traditional compilers, Sloth compiles the application source code to execute using lazy evaluation, and the runtime libraries implement mechanisms for batching multiple queries as the application executes. This article makes the following contributions:

—We devise a new mechanism to batch queries in database-backed applications based on a combination of a new "lazyifying" compiler, and combine static and dynamic

program analysis to generate the queries to be batched. We formalize lazy evaluation in the presence of database queries, and prove that our transformation preserves the semantics of the original program, including transaction boundaries.
—We propose a number of optimizations to improve the quality of the compiled lazy code.
—We built and evaluated SLOTH using real-world web applications totaling over 300k lines of code. Our results show that SLOTH achieves a median speedup between 1.3× and 2.2× (depending on network latency), with maximum speedups as high as 3.1×. Reducing latency also improves maximum throughput of our applications by 1.5×.

While web applications represent one kind of application that uses databases for persistent storage, we believe our techniques are applicable to other database applications as well. For instance, our extended lazy evaluation techniques can be used to optimize applications that interact with databases via interfaces such as JDBC [Andersen 2014] and ODBC [Microsoft 2015], and similarly for applications that issue individual queries to the database. An interesting area for future research is to apply such techniques beyond database applications, such as batching remote procedure calls in distributed computing.

In the following, we first describe how SLOTH works through a motivating example in Section 2. Then, we explain our compilation strategy in Section 3. Next, we discuss the optimizations that are used to improve generated code quality in Section 4, followed by a formalization of the semantics of extended lazy evaluation in Section 5. We describe our prototype implementation in Section 6, and report our experimental results using both real-world benchmarks in Section 7. We review related work in Section 8 and present our conclusions in Section 9. The current article is an extension of the previously published conference version [Cheung et al. 2014a]. In particular, in this article, we describe our experiments in detail. Furthermore, we describe a formalism to model program execution in SLOTH, and prove semantic correspondence between the original program execution and that translated by SLOTH.

2. OVERVIEW

In this section, we give an overview of SLOTH using the code fragment shown in Figure 1. The fragment is abridged from OpenMRS [2014], an open-source patient-record web application written in Java. It is hosted using the Spring web framework and uses the Hibernate ORM library to manage persistent data. The application has been deployed in numerous countries worldwide since 2006.

The application is structured using the Model-View-Control (MVC) pattern. The code fragment shown in Figure 1 is part of a controller that builds a model to be displayed by the view after construction. The controller is invoked by the web framework when a user logs in to the application to view the dashboard for a particular patient. It first creates a model (a `HashMap` object), populates it with appropriate patient data based on the logged-in user's privileges, and returns the populated model to the web framework. The web framework then passes the partially constructed model to other controllers that may add additional data and, finally, to the view creator to generate HTML output.

As written, this code fragment can issue up to four queries; the queries are issued by calls of the form `get...` on the data-access objects, that is, the `Service` objects, following the web framework's convention. The first query in Line 8 fetches the `Patient` object that the user is interested in displaying and adds it to the model. The code then issues queries on Lines 12 and 14, and Line 17 to fetch various data associated with the patient, with the retrieved data all added to the model.

It is important to observe that, of the four round-trips that this code can incur, only the first one is essential—without the result of that first query, the other queries cannot

```
ModelAndView handleRequest(...) {
  Map model = new HashMap<String, Object>();
  Object o = request.getAttribute("patientId");
  if (o != null) {
    Integer patientId = (Integer) o;
    if (!model.containsKey("patient")) {
      if (hasPrivilege(VIEW_PATIENTS)) {
        Patient p = getPatientService().getPatient(patientId);
         model.put("patient", p);
         ...
         model.put("patientEncounters",
                  getEncounterService().getEncountersByPatient(p));
         ...
         List visits = getVisitService().getVisitsByPatient(p);
         CollectionUtils.filter(visits, ...);
         model.put("patientVisits", visits);
         model.put("activeVisits", getVisitService().getActiveVisitsByPatient(p));
         ...
      }
      ...
    }
    ...
  }
  ...
  return new ModelAndView(portletPath, "model", model);
}
```

Fig. 1. Code fragment abridged from OpenMRS.

be constructed. In fact, the results from the other queries are stored only in the model and not used until the view (i.e., the webpage that corresponds to the dashboard) is actually rendered. This means that, in principle, the developer could have collected the last three queries in a single batch and sent it to the database in a single round-trip. The developer could have gone even further, collecting in a single batch all the queries involved in building the model until the data from any of the queries in the batch is really needed—either because the model needs to be displayed or because the data is needed to construct a new query. Manually transforming the code in this way would have a big impact on the number of round-trips incurred by the application, but would also impose an unacceptable burden on the developer. In the rest of the section, we describe how SLOTH automates such a transformation with only minimal changes to the original code, and requires no extra work from the developer.

An important ingredient to automatically transform the code to batch queries as described earlier is *lazy evaluation*. In most traditional programming languages, the evaluation of a statement causes that statement to execute; thus, any function calls made by that statement are executed before the program proceeds to evaluating the next statement. In lazy evaluation, by contrast, the evaluation of a statement does not cause the statement to execute; instead, the evaluation produces a *thunk*: a placeholder that stands for the result of that computation, and it also remembers what the computation was. The only statements that are executed immediately upon evaluation are those that produce output (e.g., printing on the console, generating the textual representation of a webpage) or those that cause an externally visible side effect (e.g., reading from files). When such a statement executes, the thunks corresponding to all the values that flow into that statement will be *forced*, meaning that the delayed computation that they represented will finally be executed.

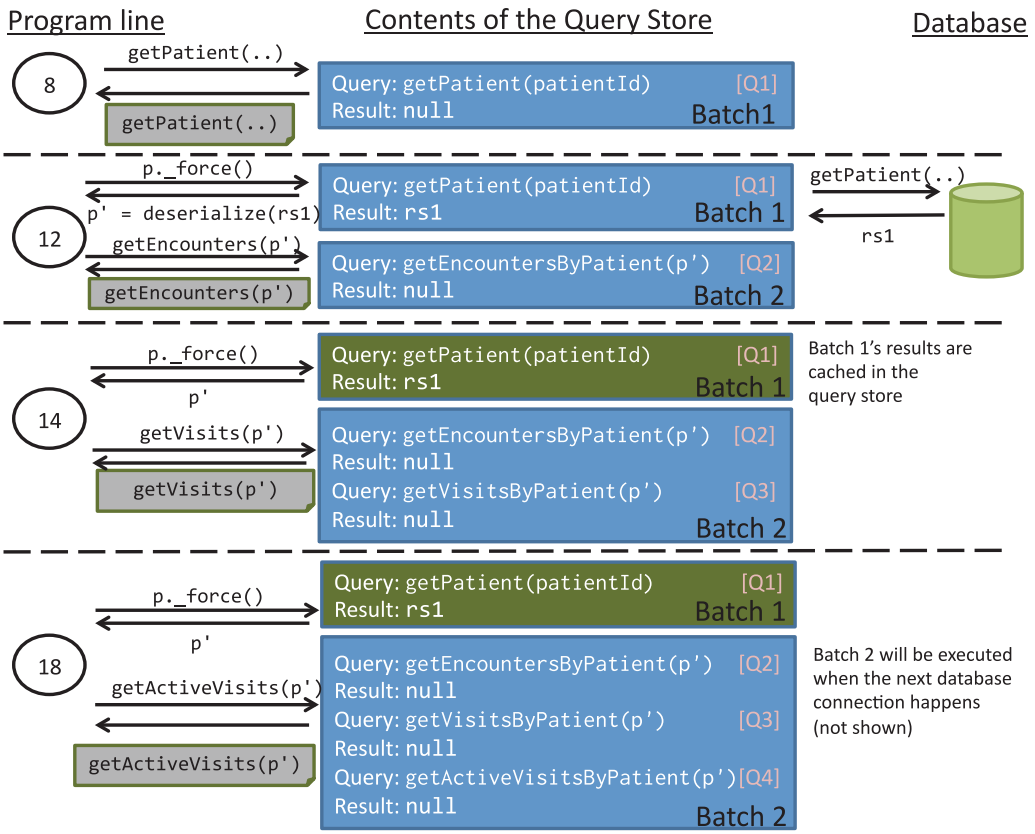


Fig. 2. Operational diagram of the example code fragment.

The key idea behind our approach is to modify the basic machinery of lazy evaluation so that, when a thunk is created, any queries performed by the statement represented by the thunk are added to a *query store* kept by the runtime in order to batch queries. Because the computation has been delayed, the results of those queries are not yet needed; thus, the queries can accumulate in the query store until any thunk that requires the result of such queries is forced. At that point, the entire batch of queries is sent to the database for processing in a single round-trip. This process is illustrated in Figure 2. During program execution, Line 8 issues a call to fetch the `Patient` object that corresponds to `patientId` (Q1). Rather than executing the query, SLOTH compiles the call to register the query with the query store instead. The query is recorded in the current batch within the store (Batch 1), and a thunk is returned to the program (represented by the gray box in Figure 2). Then, in Line 12, the program needs to access the patient object `p` to generate the queries to fetch the patient's encounters (Q2) followed by visits in Line 14 (Q3). At this point, the thunk `p` is forced, Batch 1 is executed, and its results (`rs1`) are recorded in the query cache in the store. A new nonthunk object `p'` is returned to the program upon deserialization from `rs1`, and `p'` is memoized in order to avoid redundant deserializations. After this query is executed, Q2 and Q3 can be generated using `p'` and are registered with the query store in a new batch (Batch 2). Unlike the patient query, however, Q2 and Q3 are not executed within `handleRequest` since their results are not used (thunks are stored in the model map in Lines 12 and 16). Note that, even though Line 15 uses the results of Q3 by filtering it,

our analysis determines that the operation does not have externally visible side effects and is thus delayed, allowing Batch 2 to remain unexecuted. This leads to batching another query in Line 17 that fetches the patient's active visits (Q4), and the method returns.

Depending on the subsequent program path, Batch 2 might be appended with further queries. Q2, Q3, and Q4 may be executed later when the application needs to access the database to get the value from a registered query, or they might not be executed at all if the application has no further need to access the database.

This example shows how SLOTH is able to perform much more batching than either the existing "lazy" fetching mode of Hibernate or prior work using static analysis [Ramachandra and Sudarshan 2012]. Hibernate's lazy fetching mode would have to evaluate the results of the database-accessing statements such as `getVisitsBy-Patient(p)` on Line 14, as its results are needed by the filtering operation, leaving no opportunity to batch. In contrast, SLOTH places thunks into the model and delays the filtering operation, which avoids evaluating any of the queries. This enables more queries to be batched and executed together in a subsequent trip to the database. Static analysis also cannot perform any batching for these queries, because it cannot determine what queries need to be evaluated at compile time as the queries are parameterized (such as by the specific patient id that is fetched in Line 8), and also because they are executed conditionally only if the logged-in user has the required privilege.

There are some languages, such as Haskell, that execute lazily by default, but Java has no such support. Furthermore, we want to tightly control how lazy evaluation takes place so that we can calibrate the trade-offs between execution overhead and the degree of batching achieved by the system. We would not have such tight control if we were working under an existing lazy evaluation framework. Instead, we rely on our own SLOTH compiler to transform the code for lazy evaluation. At runtime, the transformed code relies on the SLOTH runtime to maintain the query store. The runtime also includes a custom JDBC driver that allows multiple queries to be issued to the database in a single round-trip, as well as extended versions of the application framework, ORM library, and application server that can process thunks (we currently provide extensions to the Spring application framework, the Hibernate ORM library, and the Tomcat application server, to be described in Section 6). For monolithic applications that directly use the JDBC driver to interact with the database, developers just need to change such applications to use the SLOTH batch JDBC driver instead. For applications hosted on application servers, developers only need to host them on the SLOTH extended application server instead after compiling their application with the SLOTH compiler.

3. COMPILING TO LAZY SEMANTICS

In this section, we describe how SLOTH compiles the application source code to be evaluated lazily. Figure 3 shows the overall architecture of the SLOTH compiler, the details of which are described in this section and the next.

3.1. Code Simplification

To ease the implementation, the SLOTH compiler first simplifies the input source code. All loop constructs are converted to `while (true)`, where the original loop condition is converted into branches with control flow statements in their bodies, and assignments are broken down to have at most one operation on their right-hand side. Thus, an assignment such as `x = a + b + c;` will be translated to `t = a + b; x = t + c;`, with `t` being a temporary variable. Type parameters (generics) are also eliminated, and inner and anonymous classes are extracted into stand-alone ones.

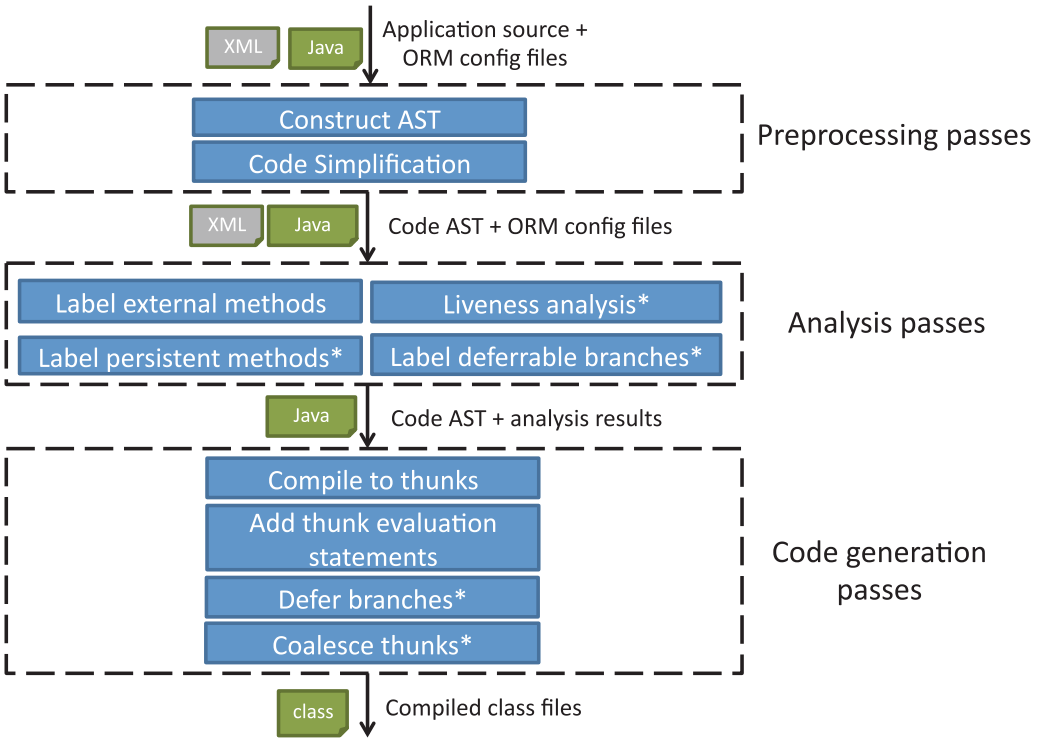


Fig. 3. Architecture of the SLOTH compiler, with * marking those components used for optimization.

3.2. Thunk Conversion

After simplification, the SLOTH compiler converts each statement of the source code into extended lazy semantics. For clarity, in the following, we present the compilation through a series of examples using concrete Java syntax. However, beyond recognizing methods that issue queries (such as those in the JDBC API), our compilation is not Java-specific, and we formalize the compilation process in Section 5 using an abstract kernel language.

In concrete syntax, each statement in the original program is replaced with an allocation of an anonymous class derived from the abstract `Thunk` class after compilation, with the code for the original statement placed inside a new `_force` class method. To "evaluate" the thunk, we invoke this method, which executes the original program statement and returns the result (if any). For example, the statement

```
int x = c + d;
```

is compiled into lazy semantics as

```
Thunk<Integer> x =
  new Thunk<Integer>(c, d) {
    Integer _force() { return c._force() + d._force(); }
  };
```

There are a few points to note in the example. First, all variables are converted into `Thunk` types after compilation. For instance, `x` has type `Thunk<Integer>` after compilation (the type parameter `int` means that the thunk will return an integer upon evaluation), and likewise for `c` and `d`. As a consequence, all variables need to

be evaluated before carrying out the actual computation inside the body of `_force`. Second, to avoid redundant evaluations, we memoize the return value of `_force` so that subsequent calls to `_force` will return the memoized value instead (for brevity purposes, details are not shown here).

While the compilation is relatively straightforward, the mechanism presented earlier can incur substantial runtime overhead, as the compiled version of each statement incurs allocation of a `Thunk` object, and all computations are converted to method calls. Section 4 describes several optimizations that we have devised to reduce the overhead. Section 7.6 quantifies the overhead of lazy semantics, which shows that, despite some overhead, it is generally much less than the savings we obtain from reducing round-trips.

3.3. Compiling Query Calls

Method calls that issue database queries, such as JDBC `executeQuery` calls and calls to ORM library APIs that retrieve entities from persistent storage, are compiled differently from ordinary method calls. In particular, we want to extract the query that would be executed from such calls and record it in the query store so that it can be issued when the next batch is sent to the database. To facilitate this, we designed a query store that consists of the following components: (a) a buffer that stores the current batch of queries to be executed and associates a unique id with each query, and (b) a result store that contains the results returned from batches previously sent to the database within the same transaction. The result store is a map from the unique query identifier to its result set. The query store API consists of two methods:

—`QueryId registerQuery(String sql)`: Add the sql query to the current batch of queries and return a unique identifier to the caller. If sql is an `INSERT`, `UPDATE`, `ABORT`, `COMMIT`, or `SELECT... INTO`, then the current batch will be immediately sent to the database to ensure that these updates are not left lingering in the query store. On the other hand, the method avoids introducing redundant queries into the batch; thus, if sql textually matches another query already in the query buffer, the identifier of the first query will be returned.
—`ResultSet getResultSet(QueryId id)`: Check if the result set associated with `id` resides in the result store; if so, return the cached result. Otherwise, issue the current batch of queries in a single round-trip, process the result sets by adding them to the result store, and return the result set that corresponds to `id`.

To use the query store, method calls that issue database queries are compiled to a thunk that passes the SQL query to be executed in the constructor. The thunk registers the query to be executed with the query store using `registerQuery` in its constructor and stores the returned `QueryId` as its member field. The `_force` method of the thunk then calls `getResultSet` to get the corresponding result set. For ORM library calls, the result sets are passed to the appropriate deserialization methods in order to convert the result set into heap objects that are returned to the caller.

Note that creating thunks associated with queries require evaluating all other thunks that are needed in order to construct the query itself. For example, consider Line 8 of Figure 1, which makes a call to the database to retrieve a particular patient's data:

```
Patient p = getPatientService().getPatient(patientId);
```

In the lazy version of this fragment, `patientId` is converted to a thunk that is evaluated before the SQL query can be passed to the query store:

```
Thunk<Patient> p = new Thunk<Patient>(patientId) {
  { this.id = queryStore.regQuery(getQuery(patientId._force())); }

  Patient _force() { return deserialize(queryStore.getResultSet(id)); }
}
```

Here, `getQuery` calls an ORM library method to generate the SQL string and substitutes the evaluated `patientId` in it, and `deserialize` reconstructs an object from an SQL result set returned by the database.

3.4. Compiling Method Calls

In the spirit of laziness, it would be ideal to delay executing method calls as long as possible (in the best case, the result from the method call is never needed; therefore, we do not need to execute the call). However, method calls might have side effects that change the program heap, for instance, changing the values of heap objects that are externally visible outside of the application, such as a global variable. For the purpose of thunk compilation, we divide methods into two categories: "external" methods are those for which the Sloth compiler does not have source-code access; we refer to the other methods as "internal" ones. Method labeling is done as one of the analysis passes in the Sloth compiler; the thunk conversion pass uses the method labels during compilation.

Internal Methods without Side Effects. This is the ideal case in which we can delay executing the method. The call is compiled to a thunk with the method call as the body of the `_force` method. Any return value of the method is assigned to the thunk. For example, if `int foo(Object x, Object y)` is an internal method with no side effects, then

```
int r = foo(x, y);
```

is compiled to

```
Thunk<Integer> r = new Thunk<Integer>(x, y) {
    Integer _force() { return this.foo(x._force(), y._force()); }
};
```

Internal Methods with Externally Visible Side Effects. We cannot defer the execution of such methods due to their externally visible side effects. However, we can still defer the evaluation of its arguments until necessary inside the method body. Thus, the Sloth compiler generates a special version of the method in which its parameters are thunk values, and the original call sites are compiled to calling the special version of the method instead. For example, if `int bar(Object x)` is such a method, then

```
int r = bar(x);
```

is compiled to

```
Thunk<Integer> r = bar_thunk(x);
```

with the declaration of `bar_thunk` as `Thunk<Integer> bar_thunk(Thunk<Object> x)`.

External Methods. We cannot defer the execution of external methods unless we know that they are side-effect free. Since we do not have access to their source code, the Sloth compiler does not change the original method call during compilation, except for forcing the arguments and receiver objects as needed. As an example, Line 3 in Figure 1,

```
Object o = request.getAttribute("patientId");,
```

is compiled to

```
Thunk<Object> o = new LiteralThunk(request._force().getAttribute("patientId"));.
```

As discussed earlier, since the types of all variables are converted to thunks, the (nonthunk) return value of external method calls is stored in `LiteralThunk` objects that simply returns the nonthunk value when `_force` is called, as shown in the earlier example.

3.5. Class Definitions and Heap Operations

For classes that are defined by the application, the SLOTH compiler changes the type of each member field to `Thunk` to facilitate accesses to field values under lazy evaluation. For each publicly accessible `final` field, the compiler adds an extra field with the original type, with its value set to the evaluated result of the corresponding thunk-converted version of the field. These fields are created so that they can be accessed from external methods. Publicly accessible nonfinal fields cannot be made lazy.

In addition, the SLOTH compiler changes the type of each parameter in method declarations to `Thunk` to facilitate method call conventions discussed in Section 3.4. Like public fields, since public methods can potentially be invoked by external code (e.g., the web framework that hosts the application or by JDK methods such as calling `equals` while searching for a key within a `Map` object), the SLOTH compiler generates a "dummy" method that has the same declaration (in terms of method name and parameter types) as the original method. The body of such dummy methods simply invokes the thunk-converted version of the corresponding method. If the method has a return value, then it is evaluated on exit. For instance, the following method,

```
public Foo bar (Baz b) { ... },
```

is compiled to two methods by the SLOTH compiler:

```
// to be called by internal code
public Thunk<Foo> bar_thunk (Thunk<Baz> b) { ... }

// to be called by external code
public Foo bar (Baz b) {
  return bar_thunk(new LiteralThunk(b))._force();
}
```

With that in mind, the compiler translates object field reads to simply return thunks. However, updates to heap objects are not delayed in order to ensure consistency of subsequent heap reads. In order to carry out a field write, however, the receiver object needs to be evaluated if it is a thunk. Thus, the statement

```
Foo obj = ...
obj.field = x;
```

is compiled to

```
Thunk<Foo> obj = ...
obj._force().field = x;
```

Notice that, while the *target* of the heap write is evaluated (`obj` in the example), the *value* that is written (`x` in the example) is a thunk object, meaning that it can represent computation that has not been evaluated yet.

3.6. Evaluating Thunks

In previous sections, we discussed the basic compilation of statements into lazy semantics using thunks. In this section, we describe when thunks are evaluated, that is, when the original computation that they represent is actually carried out.

As mentioned in the last section, the target object in field reads and writes are evaluated when encountered. However, the value of the field and the object that is written to the field can still be thunks. The same is applied to array accesses and writes, in which the target array and index are evaluated before the operation.

For method calls in which the execution of the method body is not delayed, the target object is evaluated prior to the call if the called method is nonstatic. While our compiler could have deferred the evaluation of the target object by converting all member methods into static class methods, it is likely that the body of such methods (or further methods that are invoked inside the body) accesses some fields of the target object and will end up evaluating the target object. Thus, it is unlikely that there are any significant savings in delaying such an evaluation. Finally, when calling external methods, all parameters are evaluated as discussed.

In the basic compiler, all branch conditions are evaluated when `if` statements are encountered. Recall that all loops are canonicalized into `while (true)` loops with the loop condition rewritten using branches. We present an optimization to this restriction in Section 4.2. Similarly, statements that throw exceptions, obtain locks on objects (`synchronized`), or spawn new threads of control are not deferred. Finally, thunk evaluations can also happen when compiling statements that issue queries, as discussed in Section 3.3.

3.7. Limitations

There are three limitations that we do not currently handle. First, because of delayed execution, exceptions that are thrown by the original program might not occur at the same program point in the SLOTH-compiled version. For instance, the original program might throw an exception in a method call, but in the SLOTH-compiled version, the call might be deferred until the thunk corresponding to the call is evaluated. While the exception will still be thrown eventually, the SLOTH-compiled program might have executed more code than the original program before hitting the exception.

Second, since the SLOTH compiler changes the representation of member fields in each internal class, we currently do not support custom deserializers. For instance, one of the applications used in our experiments reads in an XML file that contains the contents of an object before the application source code is compiled by SLOTH. As a result, the compiled application fails to re-create the object, as its representation has changed. We manually fixed the XML file to match the expected types in our benchmark. In general, we do not expect this to be common practice, given that Java already provides its own object serialization routines.

Finally, SLOTH currently has limited support for dynamically loaded code via reflection. If the loaded code has previously been compiled by SLOTH, then the queries issued within the loaded code will be buffered as usual. Otherwise, the issued queries will not be buffered. Implementing thunk logic into the bytecode loader will allow SLOTH to fully support reflection.

4. BEING EVEN LAZIER

In the previous section, we described how SLOTH compiles source code into lazy semantics. However, as noted in Section 3.2, there can be substantial overhead if we follow the compilation procedure naïvely. In this section, we describe three optimizations. The goal of these optimizations is to generate more efficient code and to further defer

$$
\begin{aligned}
v \in \text{values} &::= \text{True} \mid \text{False} \mid \text{number or string literal} \\
e_l \in \text{assignable expressions} &::= x \mid e.f \\
e \in \text{expressions} &::= v \mid e_l \mid \{f_i = e_i\} \mid e_1 \; op \; e_2 \mid uop \; e \mid f(e) \mid e_a[e_i] \mid R(e) \mid W(e) \\
s \in \text{statements} &::= \text{skip} \mid e_l := e \mid \text{if}(e) \text{ then } s_1 \text{ else } s_2 \mid \text{while(True) do } s \mid s_1 \; ; \; s_2 \\
op \in \text{binary operators} &::= \wedge \mid \vee \mid > \mid < \mid = \\
uop \in \text{unary operators} &::= - \mid ! \\
p \in \text{programs} &::= s
\end{aligned}
$$

Fig. 4. Abstract language to model input programs.

computation. As discussed in Section 2, deferring computation delays thunk evaluations, which, in turn, increases the chances of obtaining larger query batches during execution time. As in the previous section, we describe the optimizations using concrete Java syntax for clarity, although they can all be formalized using the language to be described in Figure 4.

4.1. Selective Compilation

The goal of compiling to lazy semantics is to enable query batching. Obviously, the benefits are observable only for the parts of the application that actually issue queries; otherwise, compiling to lazy semantics will simply add runtime overhead for the remaining parts of the application. Thus, the SLOTH compiler analyzes each method to determine whether it can possibly access the database. The analysis is a conservative one that labels a method as using persistent data if it

—issues a query in its method body.
—calls another method that uses persistent data. Because of dynamic dispatch, if the called method is overridden by any of its subclasses, we check if any of the overridden versions is persistent and, if so, we label the call to be persistent.
—accesses object fields that are stored persistently. This is done by examining the static object types that are used in each method, and checking whether it uses an object whose type is persistently stored. The latter is determined by checking for classes that are populated by query result sets in its constructor (in the case of JDBC) or by examining the object mapping configuration files for ORM frameworks.

The analysis is implemented as an interprocedural, flow-insensitive dataflow analysis [Kildall 1973]. It first identifies the set of methods m containing statements that perform any of the mentioned items. Then, any method that calls m is labeled as persistent. This process continues until all methods that can possibly be persistent are labeled. For methods that are not labeled as persistent, the SLOTH compiler does not convert their bodies into lazy semantics—they are compiled as is. For the two applications used in our experiments, our results show about 28% and 17% of the methods do not use persistent data, and those are mainly methods that handle application configuration and output page formatting (see Section 7.5 for details).

4.2. Deferring Control Flow Evaluations

In the basic compiler, all branch conditions are evaluated when an `if` statement is encountered, as discussed in Section 3.6. The rationale is that the outcome of the branch affects the subsequent program path; hence, we need the branch outcome in order to continue program execution. However, we can do better based on the intuition that, if neither branch of the condition creates any changes to the program state that are externally visible outside of the application, then the *entire* branch statement can be deferred as a thunk like other simple statements. Formally, if none of the statements

within the branch contains calls that issue queries or thunk evaluations as discussed in Section 3.6 (recall that thunks need to be evaluated when their values are needed in operations that cannot be deferred, such as making changes to the program state that are externally visible), then the entire branch statement can be deferred. For instance, in the code fragment

```
if (c) a = b; else a = d;,
```

the basic compiler would compile the code fragment into

```
if (c._force())
  a = new Thunk0(b) { ... };
else
  a = new Thunk1(d) { ... };,
```

which could result in queries being executed as a result of evaluating c. However, since the bodies of the branch statements do not make any externally visible state changes, the whole branch statement can be deferred as

```
ThunkBlock tb = new ThunkBlock2(c, b, d) {
  void _force () {
    if (c._force())
      a = b._force();
    else
      a = d._force();
  }
}

Thunk<Integer> a = tb.a();
```

where the evaluation of c is further delayed. ThunkBlock2 implements the ThunkBlock class (just like Thunk0 is derived from the Thunk class). The ThunkBlock class is similar to the Thunk class, except that it defines methods (not shown here) that return thunk variables defined within the block, such as a in the example. Calling _force on any of the thunk outputs from a thunk block will evaluate the entire block, along with all other output objects that are associated with that thunk block. In sum, this optimization allows us to further delay thunk evaluations, which, in turn, can increase query batch sizes.

To implement this optimization, the SLOTH compiler first iterates through the body of the if statement to determine if any thunk evaluation takes place, and all branches that are deferrable are labeled. During thunk generation, deferrable branches are translated to ThunkBlock objects, with the original statements inside the branches constituting the body of the _force methods. Variables defined inside the branch are assigned to output thunks, as described earlier. The same optimization is applied to defer loops as well. Recall that all loops are converted into while (true) loops with embedded control flow statements (break and continue) inside their bodies. Using similar logic, a loop can be deferred and compiled to a ThunkBlock if all statements inside the loop body can be deferred.

4.3. Coalescing Thunks

The basic compilation described in Section 3.2 results in new Thunk objects being created for each computation that is delayed. Due to the temporary variables that are introduced as a result of code simplification, the number of operations (thus the number of Thunk objects) can be much larger than the number of lines of Java code. This

can substantially slow down the compiled application. As an optimization, the thunk coalescing pass merges consecutive statements into thunk blocks to avoid allocation of thunks. The idea is that, if for two consecutive statements s_1 and s_2, and that s_1 defines a variable v that is used in s_2 and not anywhere after in the program, then we can combine s_1 and s_2 into a thunk block with s_1 and s_2 inside its _force method (provided that both statements can be deferred as discussed in Section 3). This way, we avoid creating the thunk object for v that would be created under basic compilation. As an illustrative example, consider the following code fragment:

```
int foo (int a, int b, int c, int d) {
  int e = a + b;
  int f = e + c;
  int g = f + d;
  return g; }
```

Under basic compilation, the code fragment is compiled to

```
1 Thunk<Integer> foo (Thunk<Integer> a, Thunk<Integer> b,
2                     Thunk<Integer> c, Thunk<Integer> d) {
3   Thunk<Integer> e = new Thunk0(a, b) { ... }
4   Thunk<Integer> f = new Thunk1(e, c) { ... }
5   Thunk<Integer> g = new Thunk2(f, d) { ... }
6   return g;
7 }
```

Note that three thunk objects are created within the code fragment, with the original code performed in the _force methods inside the definitions of classes Thunk0, Thunk1, and Thunk2, respectively. However, in this case, the variables e and f are not used anywhere, that is, they are no longer *live*, after Line 5. Thus, we can combine the first three statements into a single thunk, resulting in the following:

```
Thunk<Integer> foo (Thunk<Integer> a, Thunk<Integer> b,
                    Thunk<Integer> c, Thunk<Integer> d) {
  ThunkBlock tb = new ThunkBlock3(a, b, c, d) { ... }
  Thunk<Integer> g = tb.g();
  return g;
}
```

The optimized version reduces the number of object allocations from 3 to 2: one allocation for ThunkBlock3 and another for the Thunk object representing g that is created within the thunk block. In this case, the _force method inside the ThunkBlock3 class consists of statements that perform the addition in the original code. As described earlier, the thunk block keeps track of all thunk values that need to be output, in this case, the variable g.

This optimization is implemented in multiple steps in the SLOTH compiler. First, we identify variables that are live at each program statement. Live variables are computed using a dataflow analysis that iterates through program statements in a backwards manner to determine the variables that are used at each program statement (therefore, must be live).

After thunks are generated, the compiler iterates through each method to combine consecutive statements into thunk blocks. The process continues until no statements can be further combined within each method. After that, the compiler examines the _force method of each thunk block and records the set of variables that are defined. For each such variable v, the compiler checks to see if all statements that use v are

also included in the same thunk block by making use of the liveness information. If so, it does not need to create a thunk object for v. This optimization significantly reduces the number of thunk objects that need to be allocated, thus improves the efficiency of the generated code, as shown in Section 7.5.

5. FORMAL SEMANTICS

We now formalize the compilation to extended lazy evaluation outlined in the last section. For the sake of presentation, we describe the semantics in terms of an abstract language that we describe in the following section. Using that language, we furthermore prove that extended lazy evaluation preserves the semantics of the original program. We do that by showing the correspondence of programs executed under standard and extended lazy semantics, except for the limitations stated in Section 3.7.

5.1. An Abstract Language

Figure 4 shows the abstract language that we use to define extended lazy evaluation semantics. The language is simple, but will help us illustrate the main principles behind extended lazy evaluation that can be easily applied not just to Java, but to any other object-oriented language.

The main constructs to highlight in the language are the expression $R(e)$, which issues a database *read* query derived from the value of expression e, and $W(e)$, which is an expressive that issues a query that can mutate the database, such as an `INSERT` or `UPDATE`, and returns a status code that indicates whether the operation was successful.

The abstract language models the basic operations in imperative languages and is not specific to Java. To convert the original Java source code into the abstract language, the SLOTH compiler first simplifies the original source code, as described in Section 3.1. Such simplifications include semantic-preserving rewrites such as breaking down compound expressions and canonicalizing all loops into `while (true)` loops. The compiler then translates the simplified source code into the abstract language shown earlier. In the abstract language, we model unstructured control flow statements such as `break` and `continue` using Boolean variables to indicate if control flow has been altered. Such unstructured control flow statements are then translated into Boolean variable assignments [Zhang and D'Hollander 2004], and the statements that can be affected by unstructured control flow (e.g., statements that follow a `break` statement inside a loop) are wrapped into a conditional block guarded by the appropriate indicator Boolean variable(s).

In addition, the language models the return value of each method using the special variable @. Statements such as `return e` in the input code are translated into two variable assignments: one that assigns to @, and another that assigns to an indicator Boolean variable to represent the fact that control flow has been altered. Statements that follow the original `return e` statement are wrapped into conditional blocks similar to those discussed earlier for loops.

5.2. Program Model

To model program execution, we use the tuples $\langle D, \sigma, h\rangle$ and $\langle Q, D, \sigma, h\rangle$ to model the program state under standard and extended lazy evaluation, respectively, where:

—$D : v \rightarrow v$ represents the database. It is modeled as a map that takes in an SQL query v and returns a value, which is either a single value (e.g., the numerical value of an aggregate) or an array of values (e.g., a set of records stored in the database).
—$\sigma : x \rightarrow e$ represents the environment that maps program variables to expressions. It is used to look up variable values during program evaluation.

—$h : e_l \rightarrow e$ represents the program heap that maps variables that can be used as memory locations (i.e., program variables or fields) to the expression stored at that location. We use the notation $h[e]$ to represent looking up value stored at location e, and $h[e_1, e_2]$ to denote looking up the value stored at heap location e_1 with offset e_2 (e.g., a field dereference). The same notation is used to denote array value lookups as well.
—$Q : id \rightarrow e$ represents the query store under extended lazy semantics. It maps unique query identifiers (as discussed in Section 3.3) to the query expressions that were registered with the query store when the respective identifier was initially created.

In addition, we define the auxiliary function update : $D \times v \rightarrow D$ to model database modifications (e.g., as a result of executing an `UPDATE` query). It takes in a database D and a write query v (represented by a string value), and returns a new database. Since the language semantics are not concerned with the details of query execution, we model such operations as a pure function that returns a new database state.

5.3. Language Semantics

Given the abstract language, we now discuss how to evaluate each language construct under standard execution semantics. Expression evaluation is defined through a set of rules that takes a program state s and an expression e, and produces a new program state along with the value of the expression. The state s of the program is represented by a tuple (D, σ, h), where D is the database that maps queries to their result sets, σ is the environment that maps program variables to expressions, and h is the program heap that maps addresses to expressions.

As an example, the rule to evaluate the binary expression $e_1\ op\ e_2$ is shown here:

$$\frac{\langle s, e_1\rangle \rightarrow \langle s', v_1\rangle \quad \langle s', e_2\rangle \rightarrow \langle s'', v_2\rangle \quad v_1\ op\ v_2 \rightarrow v}{\langle s, e_1\ op\ e_2\rangle \rightarrow \langle s'', v\rangle} \quad \text{[Binary].}$$

The notation above the line describes how the subexpressions e_1 and e_2 are evaluated to values v_1 and v_2, respectively. The result of evaluating the overall expression is shown below the line; it is the result of applying op to v_1 and v_2, together with the state as transformed by the evaluation of the two subexpressions.

As another example, the evaluation of a read query $R(e)$ must first evaluate the query e to a query string v, then return the result of consulting the database D' with this query string. Note that the evaluation of e might itself modify the database, for example, if e involves a function call that internally issues an update query; thus, the query v must execute on the database as it is left after the evaluation of e:

$$\frac{\langle (D, \sigma, h), e\rangle \rightarrow \langle (D', \sigma, h'), v\rangle}{\langle (D, \sigma, h), R(e)\rangle \rightarrow \langle (D', \sigma, h'), D'[v]\rangle} \quad \text{[Read Query].}$$

The rest of the evaluation rules are standard, and are included in the appendix.

To describe lazy evaluation, we augment the state tuple s with the query store Q, which maps a query identifier to a pair (q, rs) that represents the SQL query q and its corresponding result set rs. rs is initially set to null ($\emptyset$) when the pair is created. We model thunks using the pair (σ, e), where σ represents the environment for looking up variables during thunk evaluation, and e the expression to evaluate. In our Java implementation the environment is implemented as fields in each generated `Thunk` class, and e is the expression in the body of the `_force` method.

As discussed in Section 3.2, to evaluate the expression $e_1\ op\ e_2$ using lazy evaluation, we first create thunk objects v_1 and v_2 for e_1 and e_2, respectively, then create another

thunk object that represents the *op*. Formally, this is described as

$$\frac{\langle s, e_1\rangle \to \langle s', v_1\rangle \quad \langle s', e_2\rangle \to \langle s'', v_2\rangle \quad v_1 = (s', e_1') \quad v_2 = (s'', e_2') \quad v = (\sigma' \cup \sigma'', e_1'\ op\ e_2')}{\langle s, e_1\ op\ e_2\rangle \to \langle s'', v\rangle} \quad \text{[Binary (lazy)].}$$

Note that the environment for v is the union of the environments from v_1 and v_2 since we might need to look up variables stored in either of them. But the mappings stored in σ' and σ'' (from s' and s'', respectively) are identical except for the mapping of e_1' in σ' and e_2' in σ''. In the case in which e_1' equals to e_2', then the two mappings are exactly identical.

On the other hand, as discussed in Section 3.3, under lazy evaluation, query calls are evaluated by first forcing the evaluation of the thunk that corresponds to the query string, then registering the query with the query store. This is formalized as

$$\frac{\langle (Q, D, \sigma, h), e\rangle \to \langle (Q', D', \sigma, h'), (\sigma', e)\rangle \quad id \text{ is a fresh identifier} \quad \mathsf{force}(Q', D', (\sigma', e)) \to \langle Q'', D'', v\rangle \quad Q''' = Q''[id \to (v, \emptyset)]}{\langle (Q, D, \sigma, h), R(e)\rangle \to \langle (Q''', D'', \sigma, h'), ([\,], id)\rangle} \quad \text{[Read Query (lazy)].}$$

The force function is used to evaluate thunks, similar to that described in the earlier examples using Java. $\mathsf{force}(Q, D, t)$ takes in the current database D and query store Q and returns the evaluated thunk along with the modified query store and database. When force encounters an *id* in a thunk, it checks the query store to see if that *id* already has a result associated with it. If it does not, it issues as a batch all the queries in the query store that do not yet have results associated with them, then assigns those results once they arrive from the database. The full specification of force and other rules for evaluating under extended lazy semantics are included in the appendix.

5.4. Soundness of Extended Lazy Evaluation

We now show that extended lazy semantics preserves the semantics of the original program that is executed under standard evaluation, except for the limitations such as exceptions that are described in Section 3.7. We show that fact using the following theorem:

THEOREM 1 (SEMANTIC CORRESPONDENCE). *Given a program p and initial states Q_0, D_0, σ_0 and h_0. If $[\![\langle D_0, \sigma_0, h_0\rangle, p]\!] \to \langle D_S, \sigma_S, h_S\rangle$ under standard evaluation, and $[\![\langle Q_0, D_0, \sigma_0, h_0\rangle, p]\!] \to \langle Q_E, D_E, \sigma_E, h_E\rangle$ under extended lazy evaluation, then $\forall x \in \sigma_S\,.\,\sigma_S[x] = \mathsf{force}(Q_E, D_E, \sigma_E[x])$ and $\forall x \in h_S\,.\,h_S[x] = \mathsf{force}(Q_E, D_E, h_E[x])$. In other words, the program states are equivalent after we evaluate all the thunks in σ_E.*

Here, the notation $[\![s, p]\!] \to s'$ means that we evaluate program p under state s, which results in a new state s'. For presentation purposes, we use $P(S_S, S_L)$ to denote that the state $S_S = \langle D_S, \sigma_S, h_S\rangle$ under standard evaluation and the state $S_L = \langle Q_L, D_L, \sigma_L, h_L\rangle$ under extended lazy evaluation satisfy the semantic correspondence property presented earlier. To prove this, we use structural induction on the abstract language. Since most

of the proof is mechanical, we do not include the entire proof in the following. Instead, we highlight a few representative cases here.

Constants. The rules for evaluating constants do not change the state under both standard and extended lazy evaluation. Thus, the correspondence property is trivially satisfied.

Variables. Using the evaluation rules, suppose that $[\![S_S, x]\!] \rightarrow S'_S, \sigma[x]$, and $[\![S_L, x]\!] \rightarrow S'_L, ([x \rightarrow \sigma[x]], x)$. From the induction hypothesis, assume that $P(S_S, S_L)$ and $P(S'_S, S'_L)$ are true. Given that, we need to show that $P(S'_S, S''_L)$ is true, where S''_L is the state that results from evaluating the thunk $([x \rightarrow \sigma[x]], x)$. This is obvious since the extended lazy evaluation rule for variables change neither the database, environment, nor the heap. Furthermore, the definition of force shows that evaluating the thunk $([x \rightarrow \sigma[x]], x)$ returns $\sigma[x]$ as in standard evaluation, hence proving the validity of $P(S'_S, S''_L)$.

Unary Expressions. As in the earlier case, suppose that $[\![S_S, \ uop\, e]\!] \rightarrow S'_S, v_S$ under standard evaluation and $[\![S_L, uop\, e]\!] \rightarrow S'_L, (\sigma, uop\, e)$ under extended lazy evaluation. Let S''_L be the state that results from evaluating the thunk $(\sigma, uop\, e)$. From the induction hypothesis, we assume that $P(S_S, S_L)$ and $P(S'_S, S'_L)$ are true. We need to show that $P(S'_S, S''_L)$ is true. First, from the definition of force, we know that evaluating the thunk $(\sigma, uop\, e)$ results in the same value as v_S. Next, we need to show that $D''_L = D'_S$ as a result of evaluating the thunk. Note that there are three possible scenarios that can happen as a result of evaluating $(\sigma, uop\, e)$. First, if e does not contain a query, then obviously $D''_L = D'_S$. Next, if e contains a query, then it is either a write query or a read query. If it is a write query, then the query is executed immediately as in standard evaluation, thus $D''_L = D'_S$. Otherwise, it is a read query. Since read queries do not change the state of the database, $D''_L = D'_S$ as well.

Binary Expressions. Binary expressions of the form $e_1\, op\, e_2$ are similar to unary expressions, except that we need to consider the case when both expressions contain queries and the effects on the state of the database when they are evaluated (otherwise, it is the same situation as unary expressions). For binary expressions, the extended lazy evaluation rule and the definition of force prescribe that e_1 is first evaluated prior to e_2. If e_1 contains a write query, then it would already have been executed during lazy evaluation, since write queries are not deferred. e_2 will be evaluated using the database as a result of evaluating e_1, thus will be evaluated the same way as in standard evaluation. The situation is similar if e_1 contains a read query and e_2 contains a write query. On the other hand, if both e_1 and e_2 contain only read queries, then evaluating them does not change the state of the database, and the correspondence property is satisfied.

Read Queries. If the query has previously been evaluated, then the cached result is returned and no change is induced on the program state, and the property is satisfied. Otherwise, it is a new read query and a new query identifier is created as a result of extended lazy evaluation. Evaluating the query identifier using force will execute the read query against the database. Since the database state is not changed as a result of read queries, the correspondence property is preserved.

Method Calls. Calls to methods that are either external or internal with side effects are not deferred under extended lazy evaluation and are thus equivalent to standard evaluation. For the pure function calls that are deferred, the definition of force for evaluating such thunks is exactly the same as those for evaluating method calls under standard evaluation (except for the evaluation of thunk parameters), thus does not alter program semantics.

Field Accesses, Array Dereferences, and Object Allocations. Since extended lazy evaluation does not defer evaluation of these kinds of expressions, the correspondence property is satisfied.

Statements. Since evaluation of statements is not deferred under extended lazy evaluation, the correspondence property for statements is satisfied, as it follows from the proof for expressions.

6. IMPLEMENTATION

We have implemented a prototype of SLOTH. The SLOTH compiler is built on top of Polyglot [Nystrom et al. 2003]. We have implemented a query store for the thunk objects to register and retrieve query results. To issue the batched queries in a single round-trip, we extended the MySQL JDBC driver to allow executing multiple queries in one `executeQuery` call, and the query store uses the batch query driver to issue queries. Once received by the database, our extended driver executes all read queries in parallel. In addition, we have also made the following changes to the application framework to enable them to process thunk objects that are returned by the hosted application. Our extensions are not language specific and can be applied to other ORM and application hosting frameworks. Besides the extensions to JDBC driver and JPA layer, the other changes are optional and were done to further increase query batching opportunities.

JPA Extensions. We extended the Java Persistence API (JPA) [DeMichiel 2006] to allow returning thunk objects from calls that retrieve objects from the database. For example, JPA defines a method `Object find(Class, id)` that fetches the persistently stored object of type `Class` with `id` as its object identifier. We extended the API with a new method `Thunk<Object> find_thunk(Class, id)` that performs the same functionality except that it returns a thunk rather than the requested object. We implemented this method in the Hibernate ORM library. The implementation first generates an SQL query that would be issued to fetch the object from the database, registers the query with the query store, and returns a thunk object to the caller. Invoking the `_force` method on the returned thunk object forces the query to be executed, and Hibernate will then deserialize the result into an object of the requested type before returning to the caller. Similar extensions are made to other JPA methods. Note that our extensions are targeted to the JPA, not Hibernate—we implemented them within Hibernate as it is a popular open-source implementation of JPA and is also used by the applications in our experiments. The extensions were implemented using about 1000 lines of code.

Spring Extensions. We extended the Spring web application framework to allow thunk objects to be stored and returned during model construction within the MVC pattern. This is a minor change that consists of about 100 lines of code.

JSP API Extensions. We extended the JavaServer Pages (JSP) API [Roth and Pelegrí-Llopart 2003] to enable thunk operations. In particular, we allow thunk objects to be returned while evaluating JSP expressions. We also extended the `JspWriter` class from the JSP API that generates the output HTML page when a JSP is requested. The class provides methods to write different types of objects to the output stream. We extended the class with a `writeThunk` method that writes thunk objects to the output stream. `writeThunk` stores the thunk to be written in a buffer, and thunks in the buffer are not evaluated until the writer is flushed by the web server (which typically happens when the entire HTML page is generated). We have implemented our JSP API extensions in Tomcat, which is a popular open-source implementation of the JSP API. This is also a minor change that consists of about 200 lines of code.

7. EXPERIMENTS

In this section, we report our experimental results. The goals of the experiments are to (i) evaluate the effectiveness of SLOTH at batching queries, (ii) quantify the change in application load times, and (iii) measure the overhead of running applications using lazy evaluation. All experiments were performed using Hibernate 3.6.5, Spring 3.0.7, and Tomcat 6 with the extensions mentioned earlier. The web server and applications were hosted on a machine with 8GB of RAM and a 2.8GHz processor. Data was stored in an unmodified MySQL 5.5 database with 47GB of RAM and twelve 2.4GHz processors. All measurements were taken from warm runs after 5 min, for which the JVM had a chance to perform JIT compilation on the loaded classes. The experiments were run using Java 1.6.[1] Results from cold runs without the warm-up period are within 15% of the numbers reported. Unless stated, there was a 0.5ms round-trip delay between the two machines (this is the latency of the group cluster machines). We used the following applications for our experiments:

—itracker version 3.1.5 [itracker 2014]: itracker is an open-source software issue management system. The system consists of a Java web application built on top of the Apache Struts framework and uses Hibernate to manage storage. The project has 10 contributors with 814 Java source files with a total of 99k lines of Java code.
—OpenMRS version 1.9.1 [OpenMRS 2014]: OpenMRS is an open-source medical record system that has been deployed in numerous countries. The system consists of a Java web application built on top of the Spring web framework and uses Hibernate to manage storage. The project has over 70 contributors. The version used consists of 1326 Java source files with a total of 226k lines of Java code. The system has been in active development since 2004 and the code illustrates various coding styles for interacting with the ORM.

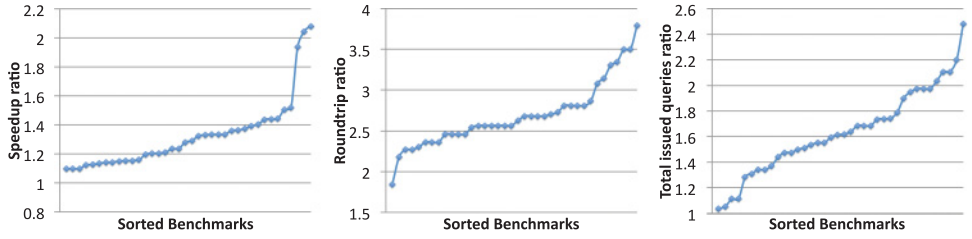
We created benchmarks from the two applications by manually examining the source code to locate all webpage files (html and jsp files). Next, we analyzed the application to find the URLs that load each of the webpages. This resulted in 38 benchmarks for itracker and 112 benchmarks for OpenMRS. Each benchmark was run by loading the extracted URL from the application server via a client that resides on the same machine as the application server.

We also tested with TPC-C and TPC-W coded in Java [Database Test Suite 2014]. Because the implementations display the query results immediately after issuing them, there is no opportunity for batching. We use them only to measure the runtime overhead of lazy evaluation.

7.1. Page Load Experiments

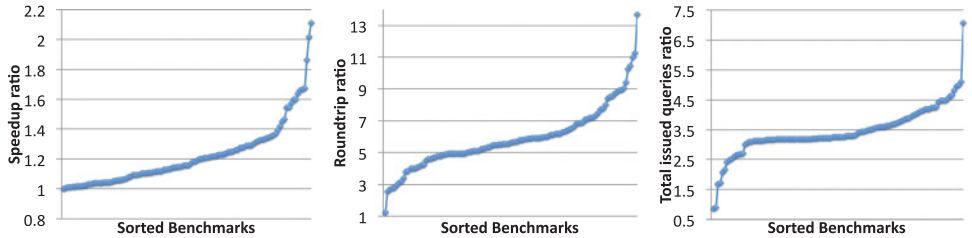
In the first set of experiments, we compared the time taken to load each benchmark from the original and the SLOTH-compiled versions of the applications. For each benchmark, we started the web and database servers and measured the time taken to load the entire page. Each measurement was the average of 5 runs. For benchmarks that require user inputs (e.g., patient ID for the patient dashboard, project ID for the list of issues to be displayed), we filled the forms automatically with valid values from the database. We restarted the database and web servers after each measurement to clear all cached objects. For OpenMRS, we used the sample database (2GB) provided by the application. For itracker, we created an artificial database (0.7GB) consisting of 10 projects and 20 users. Each project has 50 tracked issues, and none of the issues has attachments. The

[1]Java 1.8 supports lambda expressions [Oracle Corporation 2015], which might further reduce the cost of instantiating thunks. We have not experimented with that feature, as it requires nontrivial engineering efforts in changing the code generation in SLOTH.



(a) load time ratios CDF (b) round trip ratios CDF (c) queries issued ratios CDF

Fig. 5. itracker benchmark experiment results.



(a) load time ratios CDF (b) round trip ratios CDF (c) queries issued ratios CDF

Fig. 6. OpenMRS benchmark experiment results.

application allows users to associate each project with custom scripts or nonstandard components, although we did not do that for the experiments. We also created larger versions of these databases (up to 25 GB) and report their performance on selected benchmarks in Section 7.4, showing that our gains continue to be achievable with much larger database sizes.

We loaded all benchmarks with the applications hosted on the unmodified web framework and application server, and repeated with the SLOTH-compiled applications hosted on the SLOTH extended web framework using the ORM library and web server discussed in Section 6. For all benchmarks, we computed the speedup ratios as

$$\frac{\text{load time of the original application}}{\text{load time of the SLOTH compiled application}}.$$

Figures 5(a) and 6(a) show the cumulative distribution function (CDF) of the results, in which we sorted the benchmarks according to their speedups for presentation purposes (and similarly for other experiments). The detailed results are included in Appendix D.

The results show that the SLOTH-compiled applications loaded the benchmarks faster compared to the original applications, achieving up to 2.08× (median 1.27×) faster load times for itracker and 2.1× (median 1.15×) faster load times for OpenMRS. Figures 5(b) and 6(b) show the ratio of the number of round-trips to the database, computed as

$$\frac{\text{\# of database round trips in original application}}{\text{\# database round trips in SLOTH version of application}}.$$

For itracker, the minimum number of round-trip reductions was 27 (out of 59 round-trips) while the maximum reduction was 95 (out of 124 original round-trips). For OpenMRS, the minimum number of reductions was 18 (out of 100 round-trips) and the maximum number was 1082 (out of 1705 round-trips). Although these may seem like large numbers of round-trips for a single webpage, issues such as the $1 + N$ issue

```
1  if (Context.isAuthenticated()) {
2    FormService fs = Context.getFormService();
3    for (Obs o : encounter.getObsAtTopLevel(true)) {
4      FormField ff = fs.getFormField(form, o.getConcept(),..);
5      ...
6      obsMapToReturn.put(ff, list);
7    }
8  }
9
10 map.put("obsMap", obsMapToReturn);
11 return map;
```

Fig. 7. Code fragment from OpenMRS benchmark (`encounterDisplay.jsp`).

in Hibernate [StackOverflow 2014e] make it quite common for developers to write applications that issue hundreds of queries to generate a webpage in widely used ORM frameworks.

Finally, Figures 5(c) and 6(c) show the CDF of the ratio of the total number of queries issued for the applications. In OpenMRS, the SLOTH-compiled application batched as many as 68 queries into a single batch. SLOTH was able to batch multiple queries in all benchmarks, even though the original applications already make extensive use of the eager and lazy fetching strategies provided by Hibernate. This illustrates the effectiveness of applying lazy evaluation in improving performance. Examining the generated query batches, we attribute the performance speedup to the following.

Avoiding Unnecessary Queries. For all benchmarks, the SLOTH-compiled applications issued fewer total number of queries as compared to the original (ranging from 5%–10% reduction). The reduction is due to the developers' use of eager fetching to load entities in the original applications. Eager fetching incurs extra round-trips to the database to fetch the entities and increases query execution time, and is wasteful if the fetched entities are not used by the application. As noted in Section 1, it is very difficult for developers to decide when to load objects eagerly during development. Using SLOTH, on the other hand, frees the developer from making such decisions while improving application performance.

Batching Queries. The SLOTH-compiled applications batched a significant number of queries. For example, as shown in Figure 7, one of the OpenMRS benchmarks loads observations about a patient's visit. Observations include height, blood pressure, and so on; there were about 50 observations fetched for each patient. Loading is done as follows: (i) all observations are first retrieved from the database (Line 3); (ii) each observation is iterated over and its corresponding `Concept` object (i.e., the textual explanation of the observed value) is fetched and stored into a `FormField` object (Line 4). The `FormField` object is then put into the model similar to Figure 1 (Line 10). The model is returned at the end of the method and the fetched concepts are displayed in the view.

In the original application, the concept entities are lazily fetched by the ORM during view generation; each fetch incurs a round-trip to the database. It is difficult to statically analyze the code to extract the queries that would be executed in the presence of the authentication check on Line 1; techniques based on static analysis of programs [Chavan et al. 2011] will require a detailed interprocedural analysis of the loop body to ensure that the methods invoked are side-effect free in order to apply loop fission. On the other hand, since the fetched concepts are not used in the method, the SLOTH-compiled application batches all the concept queries and issues them in a

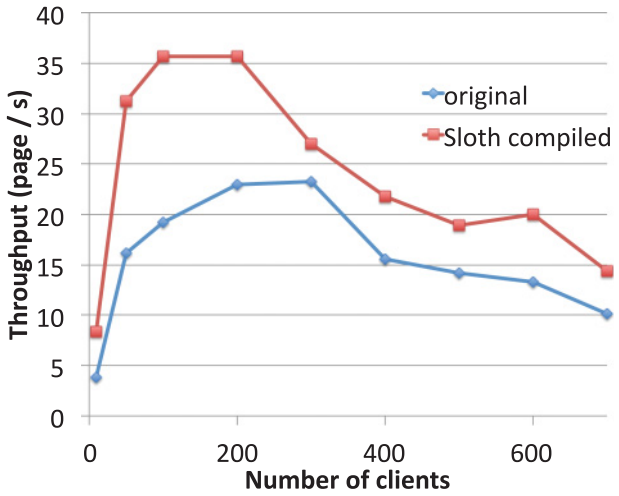


Fig. 8. Throughput experiment results.

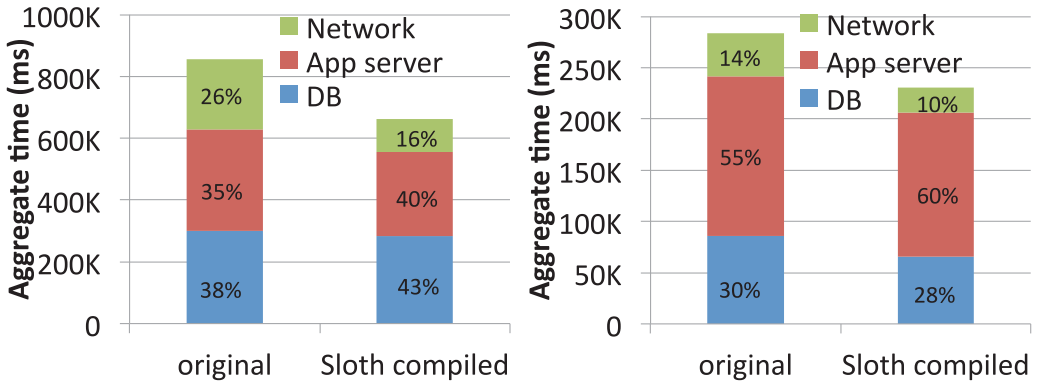
single batch along with others. This results in a dramatic reduction in the number of round-trips and an overall reduction of 1.17× in page load time.

Finally, there are a few benchmarks for which the SLOTH-compiled application issued *more* queries than the original, as shown in Figure 6(c). This is because the SLOTH-compiled application registers queries to the query store when they are encountered during execution, and all registered queries are executed when a thunk that requires data to be fetched is subsequently evaluated. However, not all fetched data are used. The original application, with its use of lazy fetching, avoided issuing those queries, which results in fewer queries executed. An interesting area for future work is to explore whether splitting the batched queries into smaller subbatches can eliminate such unnecessary queries and further improve performance. In sum, while the SLOTH-compiled application does not necessarily issue the minimal number of queries required to load each page, our results show that the benefits in reducing network round-trips outweigh the costs of executing a few extra queries.

7.2. Throughput Experiments

Next, we compared the throughput of the SLOTH-compiled application and the original. We fixed the number of browser clients; each client repeatedly loaded pages from OpenMRS for 10 minutes (clients wait until the previous load completes, then load a new page). As no standard workload was available, the pages were chosen at random from the list of benchmarks described earlier. We changed the number of clients in each run, and measured the resulting total throughput across all clients. The results (averaged across 5 runs) are shown in Figure 8.

The results show that the SLOTH-compiled application has better throughput than the original, reaching about 1.5× the peak throughput of the original application. This is expected, as the SLOTH version takes less time to load each page. Interestingly, the SLOTH version achieves its peak throughput at a lower number of clients compared to the original. This is because, given our experiment setup, both the database and the web server were underutilized when the number of clients is low, and throughput is bounded by network latency. Hence, reducing the number of round-trips improves application throughput, despite the overhead incurred on the web server from lazy evaluation. However, as the number of clients increases, the web server becomes CPU



(a) itracker benchmarks (b) OpenMRS benchmarks

Fig. 9. Time breakdown of benchmark loading experiments.

bound and throughput decreases. Since the original application does not incur any CPU overhead, it reaches the throughput at a higher number of clients, although the overall peak is lower due to network round-trips.

7.3. Time Breakdown Comparisons

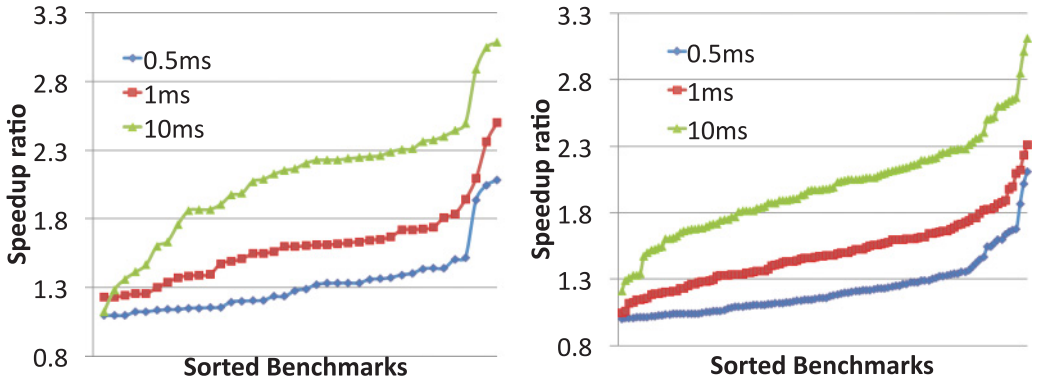
Reducing the total number of queries issued by the application reduces one source of load time. However, there are other of sources of latency. To understand the issues, we measured the amount of time spent in the different processing steps of the benchmarks: application server processing, database query execution, and network communication. We first measured the overall load time for loading the entire page. Then, we instrumented the application server to record the amount of time spent in processing, and modified our batch JDBC driver to measure the amount of time spent in query processing on the database server. We attribute the remaining time as network communication. We ran the experiment across all benchmarks and measured where time was spent while loading each benchmark, computing the sum of time spent in each phase across all benchmarks. The results for the two applications are shown in Figure 9.

For the SLOTH-compiled applications, the results show that the aggregate amount of time spent in network communication was significantly lower, reducing from 226k to 105k ms for itracker and 43k to 24k ms for OpenMRS. This is mostly due to the reduction in network round-trips. In addition, the amount of time spent in executing queries also decreased. We attribute that to the reduction in the number of queries executed, and to the parallel processing of batched queries on the database by our batch driver. However, the portion of time spent in the application server was higher for the SLOTH-compiled versions due to the overhead of lazy evaluation.

7.4. Scaling Experiments

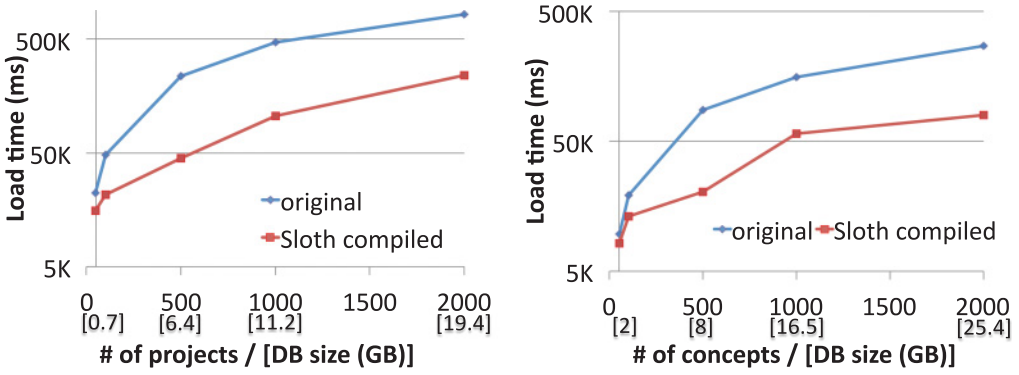
In the next set of experiments, we study the effects of round-trip reduction on page load times. We ran the same experiments as in Section 7.1, but varied the network delay from 0.5ms between the application and database servers (typical value for machines within the same data center), to 10ms (typical for machines connected via a wide area network and applications hosted on the cloud). Figure 10 shows the results for the two applications.

While the number of round-trips and queries executed remained the same as before, the results show that the amount of speedup dramatically increases as the network



(a) itracker (b) OpenMRS

Fig. 10. Network scaling experiment results.



(a) itracker (b) OpenMRS

Fig. 11. Database scaling experiment results.

round-trip time increases (more than 3× for both applications with round-trip time of 10ms). This indicates that reducing the number of network round-trips is a significant factor in reducing overall load times of the benchmarks, in addition to reducing the number of queries executed.

Next, we measured the impact of database size on benchmark load times. In this experiment, we varied the database size (up to 25GB) and measured the benchmark load times. Although the database still fits into the memory of the machine, we believe that this is representative of the way that modern transactional systems are actually deployed, since if the database working set does not fit into RAM, system performance drops rapidly as the system becomes I/O bound. We chose two benchmarks that display lists of entities retrieved from the database. For itracker, we chose a benchmark that displays the list of user projects (`list_projects.jsp`) and varied the number of projects stored in the database; for OpenMRS, we chose a benchmark that shows the observations about a patient (`encounterDisplay.jsp`), a fragment of which was discussed in Section 7.1, and varied the number of observations stored. The results are shown in Figure 11(a) and 11(b), respectively.

Application	# persistent methods	# non-persistent methods
OpenMRS	7616	2097
itracker	2031	421

Fig. 12. Number of persistent methods identified.

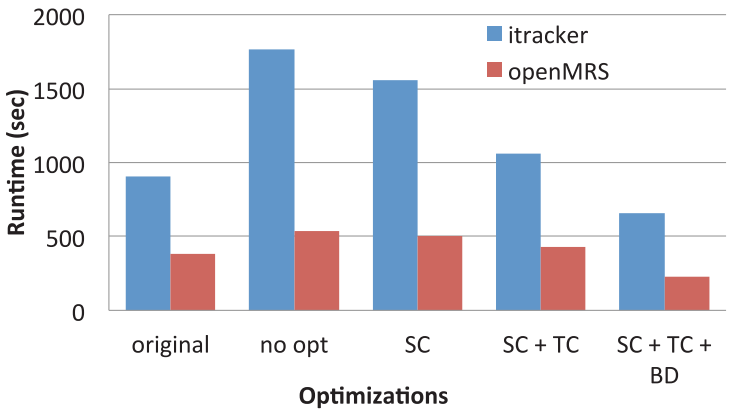


Fig. 13. Performance of SLOTH on two benchmarks as optimizations are enabled. SC = Selective computation, TC = Thunk Coalescing, BD = Branch Deferral.

The SLOTH-compiled applications achieved lower page load times in all cases, and they also scaled better as the number of entities increased. This is mostly due to query batching. For instance, the OpenMRS benchmark batched a maximum of 68, 88, 480, 980, and 1880 queries as the number of database entities increased. Examining the query logs reveals that queries were batched as discussed in Section 7.1. While the numbers of queries issued by two versions of the application are the same proportionally as the number of entities increased, the experiment shows that batching reduces the overall load time significantly, both because of the fewer round-trips to the database and the parallel processing of the batched queries. The itracker benchmark exhibits similar behavior.

7.5. Optimization Experiments

In this experiment, we measured the effects of the optimizations presented in Section 4. First, we study the effectiveness of selective compilation. Figure 12 shows the number of methods that are identified as persistent in the two applications. As discussed in Section 4.1, nonpersistent methods are not compiled to lazy semantics.

Next, we quantify the effects of the optimizations by comparing the amount of time taken to load the benchmarks. We first measured the time taken to load all benchmarks from the SLOTH-compiled applications with no optimizations. Next, we turned each of the optimizations on one at a time: selective compilation (SC), thunk coalescing (TC), and branch deferral (BD), in that order. We recompiled each time; Figure 13 shows the resulting load time for all benchmarks as each optimization was turned on.

In both applications, branch deferral is the most effective in improving performance. This makes sense, as both applications have few statements with externally visible side effects, which increases the applicability of the technique. In addition, as discussed in Section 4.2, deferring control flow statements further delays the evaluation of thunks, which allows more query batching to take place.

Transaction type	Original time (s)	SLOTH time (s)	Overhead
TPC-C			
New order	930	955	15.8%
Order status	752	836	11.2%
Stock level	420	459	9.4%
Payment	789	869	10.2%
Delivery	626	665	6.2%
TPC-W			
Browsing mix	1075	1138	5.9%
Shopping mix	1223	1326	8.5%
Ordering mix	1423	1600	12.4%

Fig. 14. Overhead experiment results.

Overall, there was more than a 2× difference in load time between having none and all the optimizations for both applications. Without the optimizations, we would have lost all the benefits from round-trip reductions, that is, the actual load times of the SLOTH-compiled applications would have been slower than the original.

7.6. Overhead Experiments

In the final experiment, we measured the overhead of lazy evaluation. We use TPC-C and TPC-W for this purpose. We chose implementations that use JDBC directly for database operations and do not cache query results. The TPC-W implementation is a stand-alone web application hosted on Tomcat. Since each transaction has very few queries, and the query results are used almost immediately after they are issued (e.g., printed out on the console in the case of TPC-C and converted to HTML in the case of TPC-W), there are essentially no opportunities for SLOTH to improve performance, making these experiments a pure measure of overhead of executing under lazy semantics.

We used 20 warehouses for TPC-C (initial size of the database is 23GB). We used 10 clients, with each client executing 10k transactions, and measured the time taken to finish all transactions. For TPC-W, the database contained 10,000 items (about 1GB on disk), and the implementation omitted the think time. We used 10 emulated browsers executing 10k transactions each. The experiments were executed on the same machines as in the previous experiments, with optimizations turned on. Figure 14 show the results.

As expected, the SLOTH-compiled versions were 5% to 15% slower than the original, due to lazy semantics. However, given that the Java virtual machine is not designed for lazy evaluation, we believe that these overheads are reasonable, especially given the significant performance gains observed in real applications. While query batching does not improve performance of these benchmarks, some of the benchmarks might be amenable to query prefetching [Ibrahim and Cook 2006; Ramachandra and Sudarshan 2012]. However, given that these benchmarks do not perform intensive computation, that some of the query parameters depend on the results from previously executed queries (e.g., TPC-C new order and payment transactions), and that some of these benchmarks do not issue read queries at all (e.g., a webpage that updates patient records), it is unclear how much performance improvement will be achieved with prefetching.

7.7. Discussion

Our experiments show that SLOTH can batch queries and improve performance across different benchmarks. While SLOTH does not execute the batched queries until any of

their results are needed by the application, other execution strategies are possible. For instance, each batch can be executed asynchronously as it reaches a certain size, or periodically based on current load on the database. Choosing the optimal strategy would be interesting future work.

8. RELATED WORK

In this section, we survey work related to the techniques proposed in SLOTH. We first review how queries are specified in database applications. Then, we discuss different means that have been proposed to optimize queries given how they are specified. Finally, we review how lazy evaluation has been deployed in other contexts.

8.1. Specifying Queries in Database Applications

Traditionally, database management systems (DBMSs) provide a very narrow API for applications to issue queries: applications invoke an `executeQuery` function call with the SQL string passed in as one of the parameters, and subsequently invoke `get-Results` to retrieve resulting tuples after the query has been executed by the DBMS. Such interfaces are usually standardized [Andersen 2014; Microsoft 2015], and are implemented by the driver library provided by different DBMSs [MySQL 2015; PostgreSQL 2015].

Embedding SQL text directly in the application code makes the application difficult to maintain, as application source code is usually written in a general-purpose programming language (such as Java or PHP) rather than SQL. Moreover, application developers who use such interfaces to interact with the DBMS will need to handle issues such as type mismatches between the application program and the DBMS, as embodied by the infamous "impedance mismatch" problem [Copeland and Maier 1984].

ORM libraries [Django Project 2014; Ruby on Rails Project 2014; Hibernate 2014a; Microsoft 2014] present an alternative to using traditional DBMS interfaces. Such libraries abstract query operations (e.g., object retrieval, filtering, aggregation) into a number of functions (e.g., Java Persistent API [DeMichiel 2006] and active record [Fowler 2003]) that the application can invoke. Such functions are usually written in the same language as the application program. To use such libraries, application developers first inform the library (such as using XML configuration files) about classes to be persistently stored, and how they should be stored (essentially the physical design). After that, developers can access persistently stored objects by writing SQL queries (and handling serialization in the application) or invoking functions provided by the ORM. ORM libraries usually implement their APIs by translating the call into the corresponding SQL query to be executed by the DBMS, and they often include additional functionalities such as maintaining object caches in the application program heap.

While using ORM libraries avoids some of the issues described earlier, the APIs exposed by ORMs are often not as expressive as compared to SQL. For instance, writing an "arg max" query that retrieves the tuple with the maximum value requires creating various auxiliary objects to express the criteria [StackOverflow 2014a], although it can be written as a subquery in SQL. As a result, integrating application and database query languages has been an active research area. Such languages range from a more expressive SQL similar to languages for object-oriented DBMSs (e.g., HQL [Hibernate 2015]) in which developers can express queries using an object-oriented syntax, to extensions to general-purpose languages (e.g., JQS [Iu et al. 2010] and Jaba [Cook and Wiedermann 2011] on top of Java, LINQ [Meijer et al. 2006] on top of C#), to completely new languages for writing database applications (e.g., Kleisli [Wong 2000], Links [Cooper et al. 2006], the functional language proposed by Cooper [2009],

Ferry [Grust et al. 2009], UniQL [Shi et al. 2014], and DBPL [Schmidt and Matthes 1994]). Implementations of these languages range from a translation layer from the extended language constructs to the base language to a new compiler toolchain. While many of these languages are specifically designed to address the issues associated with traditional DBMS interfaces and ORMs, they often require developers to learn new programming paradigms and rewrite existing applications.

8.2. Query Optimization for Database Applications

The traditional interfaces that applications use to interact with the DBMS completely isolate the application from the DBMS. Because of that, classical compiler optimization considers embedded queries as external "black box" function calls that cannot be optimized. Meanwhile, with no knowledge of how queries are generated by the application or how query results are to be used, DBMSs have instead focused optimization efforts on speeding up query processing. Classical query optimization techniques that can be applied to database applications include multiquery optimization [Sellis 1988], sharing of intermediate results across multiple queries [Zukowski et al. 2007; Harizopoulos et al. 2005], and sharing of query plans [Giannikis et al. 2012], among others.

ORM libraries allow users to indicate when persistent data is to be fetched using "eager" annotations. As discussed in Section 1, such annotations are often difficult to use, as the developer needs to anticipate how persistent objects will be used by other parts of the application.

Rather than relying on developers to provide hints for application optimization, recent advances in program analysis have sparked new research in co-optimization of embedded queries and database application code, with focus on converting application code fragments into queries, in order to leverage specialized implementations of relational operators in DBMSs, and analyzing application code to prefetch query results [Smith 1978] and batch queries. On the one hand, techniques have been developed based on static analysis of programs to recognize certain code idioms in the application program (e.g., iterations over persistent collections), with the goal to convert the recognized idioms into SQL queries [Iu et al. 2010; Iu and Zwaenepoel 2010; Wiedermann et al. 2008; Cheung et al. 2013, 2014b; Ramachandra and Guravannavar 2014]. However, these techniques are often limited to the code idioms that are embedded in the tool itself, and are not able to convert code fragments into queries even if they are semantically equivalent to the code idioms embedded in the tool. Although QBS [Cheung et al. 2013, 2014b] does not have that restriction, it is still limited to queries that are expressible using the theory of ordered relations, as discussed in the article. However, in contrast to Sloth, such techniques do not incur any runtime overhead. Static analysis has also been applied to query batching [Guravannavar and Sudarshan 2008; Chavan et al. 2011] and remote procedure calls [Yeung and Kelly 2003]. However, as in the case of query conversion, their batching ability is limited due to the imprecision of static analysis.

Rather than analyzing programs statically, there has also been work done on using runtime analysis and application monitoring to optimize application-embedded queries. Such work focuses on prefetching query results and batching queries. For instance, quantum databases [Roy et al. 2013] reorder transactions using developer annotations, while the homeostasis protocol [Roy et al. 2015] avoids network communication among multinode DBMSs (thus speeds up the application) by using data statistics during application execution. They aim to combine queries issued by multiple concurrent clients, whereas Sloth batches queries that are issued by the same client over time, although Sloth can make use of such techniques to merge Sloth-generated query batches from multiple clients for further performance improvement.

There has been work done on query batching by predicting future queries [Bowman and Salem 2004] and prefetching data [Apache Cayenne 2014; Hibernate 2014b; PHP 2014], although they all require programmer annotations to indicate what and when to prefetch, unlike SLOTH. Recent works have proposed prefetching queries automatically by analyzing application code. AutoFetch [Ibrahim and Cook 2006] uses application profiling to predict the queries that will be issued by the application. Unlike SLOTH, the benefits provided by this tool will depend on how accurate the collected profile reflects the actual workload. On the other hand, the work proposed by Ramachandra and Sudarshan [2012] changes the application such that queries are asynchronously ahead of time (compared to the original code) as soon as their parameter values (if any) are available. The amount of performance gain is limited by the amount of computation that lies between when the query is issued and when the results are used. SLOTH does not have such limitations.

Finally, there is work on moving application code to execute in the database as stored procedures to reduce the number of round-trips [Cheung et al. 2012], which is similar to our goals. In comparison, SLOTH does not require the program state to be distributed. While such dynamic analysis techniques do not suffer from precision issues, it incurs some runtime overhead. Thus, it would be interesting to combine both techniques to achieve a low-overhead, yet high-precision, system.

8.3. Lazy Evaluation

Lazy evaluation was first introduced for lambda calculus [Henderson and Morris 1976], with one of the goals to increase the expressiveness of the language by allowing programmers to define infinite data structures and custom control flow constructs. For instance, an infinitely long list of integers can be specified by splitting the list into a head and a tail expression, where the head represents the next element to be retrieved and the tail represents the remainder of the list. The head expression is evaluated only when an element is to be retrieved, likewise for the tail expression. Some general-purpose programming languages (such as Haskell [Haskell wiki 2015] and Miranda [Turner 1986]) evaluate expressions lazily by default. Meanwhile, other languages (such as OCaml [OCaml Tutorial 2015] and Scheme [Takafumi 2015]) support lazy evaluation by providing additional constructs for programmers to denote expressions to be evaluated lazily.

Lazy evaluation is often implemented using thunks in languages that do not readily support it [Ingerman 1961; Warth 2007]. In contrast, the extended lazy evaluation proposed in this article is fundamentally different: rather than its traditional uses, SLOTH uses lazy evaluation to improve application performance by batching queries; SLOTH is the first system to do so, to our knowledge. As our techniques are not specific to Java, they can be implemented in other languages as well, including those that already support lazy evaluation, by extending the language runtime with query-batching capabilities.

9. CONCLUSION

In this article, we presented SLOTH, a new compiler and runtime that speeds up database applications by eliminating round-trips between the application and database servers. By delaying computation using lazy semantics, our system reduces round-trips to the database substantially by batching multiple queries and issuing them in a single batch. Along with a number of optimization techniques, we evaluated SLOTH on different real-world applications. Our results show that SLOTH outperforms existing approaches in query batching, and delivers substantial reduction (up to 3×) in application execution time with modest worst-case runtime overheads.

APPENDIX

In this appendix, we describe how to evaluate each of the language constructs shown in Figure 4 under standard and extended lazy evaluation. We furthermore describe formally the semantics of the force function used in extended lazy evaluation. Following that, we list the details of the benchmarks used in the experiments.

A. STANDARD EVALUATION SEMANTICS

The semantics of standard evaluation for each of the language constructs are shown here, where the $\emptyset$ symbol represents the null value. The domains of each of the variables involved are defined in Section 5.2.

Semantics of expressions:

$$\frac{}{[\![\langle D, \sigma, h\rangle, x]\!] \rightarrow \langle D, \sigma, h\rangle, \sigma[x]} \quad \text{[Variable]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \rightarrow \langle D', \sigma, h'\rangle, v}{[\![\langle D, \sigma, h\rangle, e.f]\!] \rightarrow \langle D', \sigma, h'\rangle, h'[v, f]} \quad \text{[Field dereference]}$$

$$\frac{}{[\![\langle D, \sigma, h\rangle, c]\!] \rightarrow \langle D, \sigma, h\rangle, c} \quad \text{[Constant]}$$

$$\frac{h' = h[v \rightarrow \{f_i = \emptyset\}],\ v \text{ is a fresh location}}{[\![\langle D, \sigma, h\rangle, \{f_i = e_i\}]\!] \rightarrow \langle D, \sigma, h'\rangle, v} \quad \text{[Object allocation]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e_1]\!] \rightarrow \langle D', \sigma, h'\rangle, v_1 \quad [\![\langle D', \sigma, h\rangle, e_2]\!] \rightarrow \langle D'', \sigma, h''\rangle, v_2 \quad v_1\ op\ v_2 = v}{[\![\langle D, \sigma, h\rangle, e_1\ op\ e_2]\!] \rightarrow \langle D'', \sigma, h''\rangle, v} \quad \text{[Binary]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \rightarrow \langle D', \sigma, h'\rangle, v_1 \quad uop\ v_1 = v}{[\![\langle D, \sigma, h\rangle,\ uop\ e]\!] \rightarrow \langle D', \sigma, h'\rangle, v} \quad \text{[Unary]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \rightarrow \langle D', \sigma, h'\rangle, v \quad [\![\langle D', [x \rightarrow v], h'\rangle, s]\!] \rightarrow \langle D'', \sigma'', h''\rangle \quad s \text{ is the body of } f(x)}{[\![\langle D, \sigma, h\rangle, f(e)]\!] \rightarrow \langle D'', \sigma'', h''\rangle, \sigma[@]} \quad \text{[Method]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e_i]\!] \rightarrow \langle D', \sigma, h'\rangle, v_i \quad [\![\langle D', \sigma, h'\rangle, e_a]\!] \rightarrow \langle D'', \sigma, h''\rangle, v_a}{[\![\langle D, \sigma, h\rangle, e_a[e_i]]\!] \rightarrow \langle D'', \sigma, h''\rangle, h''[v_a, v_i]} \quad \text{[Array deference]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \rightarrow \langle D', \sigma, h'\rangle, v}{[\![\langle D, \sigma, h\rangle, R(e)]\!] \rightarrow \langle D', \sigma, h'\rangle, D'[v]} \quad \text{[Read query]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \rightarrow \langle D', \sigma, h'\rangle, v \quad \text{update}(D', v) = D''}{[\![\langle D, \sigma, h\rangle, W(e)]\!] \rightarrow \langle D'', \sigma, h'\rangle} \quad \text{[Write query]}$$

Semantics of statements:

$$\frac{}{[\![\langle D, \sigma, h\rangle, \text{skip}]\!] \rightarrow \langle D, \sigma, h\rangle} \quad \text{[Skip]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h'\rangle, v \quad [\![\langle D', \sigma, h'\rangle, e_l]\!] \to \langle D'', \sigma, h''\rangle, v_l}{[\![\langle D, \sigma, h\rangle, e_l := e]\!] \to \langle D'', \sigma[v_l \to v], h''\rangle} \quad \text{[Assignment]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h', \text{True}\rangle \quad [\![\langle D', \sigma, h'\rangle, s_1]\!] \to \langle D'', \sigma', h''\rangle}{[\![\langle D, \sigma, h\rangle, \text{ if}(e)\text{ then } s_1 \text{ else } s_2]\!] \to \langle D'', \sigma', h''\rangle} \quad \text{[Conditional–true]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, e]\!] \to \langle D', \sigma, h', \text{ False}\rangle \quad [\![\langle D', \sigma, h'\rangle, s_2]\!] \to \langle D'', \sigma', h''\rangle}{[\![\langle D, \sigma, h\rangle, \text{ if}(e)\text{ then } s_1 \text{ else } s_2]\!] \to \langle D'', \sigma', h''\rangle} \quad \text{[Conditional–false]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, s]\!] \to \langle D', \sigma', h'\rangle}{[\![\langle D, \sigma, h\rangle, \text{ while}(\text{True})\text{ do } s]\!] \to \langle D', \sigma', h'\rangle} \quad \text{[Loop]}$$

$$\frac{[\![\langle D, \sigma, h\rangle, s_1]\!] \to \langle D', \sigma', h'\rangle \quad [\![\langle D', \sigma', h'\rangle, s_2]\!] \to \langle D'', \sigma'', h''\rangle}{[\![\langle D, \sigma, h\rangle, s_1 ; s_2]\!] \to \langle D'', \sigma'', h''\rangle} \quad \text{[Sequence]}$$

As discussed in Section 5, each of the evaluation rules describes the result of evaluating a program construct in the language (shown below the line), given the intermediate steps that are taken during evaluation (shown above the line). The rules for standard evaluation are typical of imperative languages. In general, each of the rules for expressions returns a (possibly changed) state, along with the result of the evaluation.

As an example, to evaluate a variable x, we simply look up its value using the environment σ and return the corresponding value. To evaluate a field deference expression $e.f$, we first evaluate the expression e to value v, then return the expression stored in the heap at location v with offset f. Finally, to evaluate a read query $R(e)$, we first evaluate the query expression e to a value v (i.e., the SQL query), then look up the value stored in the database.

On the other hand, the rules for evaluating statements return only a new program state and no values. Note that, to evaluate a write query $W(e)$, we use the update function to perform the change on the database after evaluating the query expression. The changed database is then included as part of the modified state.

B. EXTENDED LAZY EVALUATION SEMANTICS

We now define the semantics of extended lazy evaluation. As described in Section 3.2, we model thunks using a pair (σ, e), which represents the expression e that is delayed, along with the environment σ that is used to look up expression values when the delayed expression is evaluated. Furthermore, we define a function force $: Q \times D \times t \to Q \times D \times e$ that takes in a query store Q, a database D, and a thunk t. The function returns a (possibly modified) query store and a database along with the result of thunk evaluation (a value v). We show the evaluation rules here.

Semantics of expressions:

$$\frac{}{[\![\langle Q, D, \sigma, h\rangle, x]\!] \to \langle Q, D, \sigma, h\rangle, ([x \to \sigma[x]], x)} \quad \text{[Variable]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad \text{force}(Q, D, (\sigma', e)) \to v}{[\![\langle Q, D, \sigma, h\rangle, e.f]\!] \to \langle Q', D', \sigma, h'\rangle, h'[v, f]} \quad \text{[Field deference]}$$

$$\frac{}{[\![\langle Q, D, \sigma, h\rangle, c]\!] \to \langle Q, D, \sigma, h\rangle, ([\,], c)} \quad \text{[Constant]}$$

$$\frac{h' = h[v \to \{f_i = \emptyset\}],\ v \text{ is a fresh location}}{[\![\langle Q, D, \sigma, h\rangle, \{f_i = e_i\}]\!] \to \langle Q, D, \sigma, h'\rangle, ([\,], v)} \quad \text{[Object allocation]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e_1]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e_1) \quad [\![\langle Q', D', \sigma, h'\rangle, e_2]\!] \to \langle Q'', D'', \sigma, h''\rangle, (\sigma'', e_2)}{[\![\langle Q, D, \sigma, h\rangle, e_1\ op\ e_2]\!] \to \langle Q'', D'', \sigma, h''\rangle, (\sigma' \cup \sigma'', e_1\ op\ e_2)} \quad \text{[Binary]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma, e)}{[\![\langle Q, D, \sigma, h\rangle, uop\ e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma, uop\ e)} \quad \text{[Unary]}$$

$$\frac{[\![\langle Q, D, \ \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma', h'\rangle, (\sigma', e)}{[\![\langle Q, D, \ \sigma, h\rangle, f(e)]\!] \to \langle Q', D', \sigma, h'\rangle, ([x \to (\sigma', e)], f(x))} \quad \text{[Method–internal and pure]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \\ [\![\langle Q', D', [x \to (\sigma', e)], h'\rangle, s]\!] \to \langle Q'', D'', \sigma'', h''\rangle \\ s \text{ is the body of } f(x)\end{array}}{[\![\langle Q, D, \sigma, h\rangle, f(e)]\!] \to \langle Q'', D'', \sigma'', h''\rangle, \sigma''[@]} \quad \text{[Method–internal and impure]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \\ [\![\langle Q', D', [x \to (\sigma', e)], h'\rangle, s]\!] \to \langle Q''', D''', \sigma'', h''\rangle \\ \text{force}\,(Q', D', (\sigma', e)) \to Q'', D'', v \\ s \text{ is the body of } f(x)\end{array}}{[\![\langle Q, D, \sigma, h\rangle, f(e)]\!] \to \langle Q''', D''', \sigma'', h''\rangle, \sigma''[@]} \quad \text{[Method–external]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e_i]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e_i) \\ [\![\langle Q', D', \sigma, h'\rangle, e_a]\!] \to \langle Q'', D'', \sigma, h''\rangle, (\sigma'', e_a) \\ \text{force}\,(Q'', D'', (\sigma', e_i)) \to Q''', D''', v_i \\ \text{force}\,(Q''', D''', (\sigma'', e_a)) \to Q'''', D'''', v_a\end{array}}{[\![\langle Q, D, \sigma, h\rangle, e_a[e_i]]\!] \to \langle Q'''', D'''', \sigma, h''\rangle, h''[v_a, v_i]} \quad \text{[Array deference]}$$

$$\frac{\begin{array}{cc}[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) & Q''' = Q''[id \to (v, \emptyset)] \\ \text{force}\,(Q', D', (\sigma', e)) \to Q', D'', v & id \text{ is a fresh identifier}\end{array}}{[\![\langle Q, D, \sigma, h\rangle, R(e)]\!] \to \langle Q''', D'', \sigma, h'\rangle, ([\,], id)} \quad \text{[Read query]}$$

$$\frac{\begin{array}{c}[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \\ \text{force}\,(Q', D', (\sigma', e)) \to Q', D'', v \\ \text{update}(D'', v) \to D''' \\ \forall id \in Q''\,.\, Q'''[id] = \begin{cases} D'''[Q''[id].s] & \text{if } Q''[id].rs = \emptyset \\ Q''[id].rs & \text{otherwise} \end{cases}\end{array}}{[\![\langle Q, D, \sigma, h\rangle, W(e)]\!] \to \langle Q''', D''', \sigma, h'\rangle} \quad \text{[Write query]}$$

Semantics of statements:

$$\frac{}{[\![\langle Q, D, \sigma, h\rangle, \text{skip}]\!] \to \langle Q, D, \sigma, h\rangle} \quad \text{[Skip]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad [\![\langle Q', D', \sigma, h'\rangle, e_l]\!] \to \langle Q'', D'', \sigma, h''\rangle, v_l}{[\![\langle Q, D, \sigma, h\rangle, e_l := e]\!] \to \langle Q'', D'', \sigma[v_l \to (\sigma', e)], h''\rangle} \quad \text{[Assignment]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad \text{force}(Q', D', (\sigma', e)) \to Q'', D'', \text{True} \quad [\![\langle Q'', D'', \sigma, h'\rangle, s_1]\!] \to \langle Q''', D'', \sigma', h''\rangle}{[\![\langle Q, D, \sigma, h\rangle, \text{if}(e)\ \text{then}\ s_1\ \text{else}\ s_2]\!] \to \langle Q''', D''', \sigma', h''\rangle} \quad \text{[Conditional–true]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, e]\!] \to \langle Q', D', \sigma, h'\rangle, (\sigma', e) \quad \text{force}(Q', D', (\sigma', e)) \to Q'', D'', \text{False} \quad [\![\langle Q'', D'', \sigma, h'\rangle, s_2]\!] \to \langle Q''', D'', \sigma', h''\rangle}{[\![\langle Q, D, \sigma, h\rangle, \text{if}(e)\ \text{then}\ s_1\ \text{else}\ s_2]\!] \to \langle Q''', D''', \sigma', h''\rangle} \quad \text{[Conditional–false]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, s]\!] \to \langle Q', D', \sigma', h'\rangle}{[\![\langle Q, D, \sigma, h\rangle, \text{while(True)}\ \text{do}\ s]\!] \to \langle Q', D', \sigma', h'\rangle} \quad \text{[Loop]}$$

$$\frac{[\![\langle Q, D, \sigma, h\rangle, s_1]\!] \to \langle Q', D', \sigma', h'\rangle \quad [\![\langle Q, D', \sigma', h'\rangle, s_2]\!] \to \langle Q'', D'', \sigma'', h''\rangle}{[\![\langle Q, D, \sigma, h\rangle, s_1\ ; s_2]\!] \to \langle Q'', D'', \sigma'', h''\rangle} \quad \text{[Sequence]}$$

The evaluation rules for extended lazy semantics are similar to those for standard evaluation, except for creation of thunk objects and using force to evaluate thunks. For instance, to evaluate a variable x, we create a new thunk with the delayed expression being x itself. In addition, the thunk contains a new environment that maps x to its value under the current environment, in order for the appropriate value to be returned when the thunk is evaluated. On the other hand, to evaluate a binary expression, we first create thunks for the operands. Then, we create another thunk for the binary operation itself, with the environment being the combination of the environments from the two thunks from the operands.

As discussed in Section 3.4, method calls are evaluated using three different rules based on the kind of method that is invoked. For methods that are internal and pure, we first evaluate the actual parameters of the call (by creating thunks to wrap around each of the actuals), then create another thunk with the delayed expression being the call itself, as shown in the rule [Method–internal and pure]. However, since we cannot delay calls to external methods or methods that have side effects, we evaluate such calls by first evaluating the parameters, then either calling the specialized method that takes thunk parameters (if the method is internal, as shown in the rules [Method–internal and impure]), or evaluating the parameter thunks and calling the external method directly, as shown in the rule [Method–external].

Finally, the evaluation rules for statements are also similar to those under standard evaluation. The most complex one is that for write queries, for which we first evaluate the query expression thunk. Then, before executing the actual write query, we first execute all the read queries that have been delayed and not executed in the query store due to query batching. This is shown in the evaluation rule by issuing queries to the database for all the query identifiers that have not been executed, and changing the contents of the query store as a result. After that, we execute the write query on the database, and use the update function to change to the database contents, as in standard evaluation.

The formalism used to present the evaluation rules for basic extended lazy evaluation can also formalize the optimizations discussed in Section 4; we omit the details here.

C. SEMANTICS OF THUNK EVALUATIONS

Next, we define the full semantics of the force function that is used to evaluate thunks, as mentioned earlier in Section 5.3.

$$\frac{}{\text{force}(Q, D, ([x \to \sigma[x]], x)) \to Q, D, \sigma[x]} \quad \text{[Variable]}$$

$$\frac{}{\text{force}(Q, D, (\sigma, c)) \to Q, D, c} \quad \text{[Constant]}$$

$$\frac{\begin{array}{c}\text{force}(Q, D, (\sigma, e_1)) \to Q', D', v_1 \qquad \text{force}(Q', D', (\sigma, e_2)) \to Q'', D'', v_2 \\ v_1 \quad op \quad v_2 \quad = \quad v\end{array}}{\text{force}(Q, D, (\sigma, e_1\ op\ e_2)) \to Q'', D'', v} \quad \text{[Binary]}$$

$$\frac{\text{force}(Q, D, (\sigma, e)) \to Q', D', v_1 \quad uop\, v_1 \to v}{\text{force}(Q, D, (\sigma, uop\, e)) \to Q', D', v} \quad \text{[Unary]}$$

$$\frac{\begin{array}{l}\text{force}(Q, D, (\sigma, e)) \quad \to Q', D', v \\ [\![\langle Q', D', [x \to v], h\rangle, s]\!] \to \langle Q'', D'', \sigma', h\rangle \\ s \text{ is the body of } f(x)\end{array}}{\text{force}(Q, D, ([x \to (\sigma, e)], f(x)) \to Q'', D'', \sigma'[@]} \quad \text{[Method–internal and pure]}$$

$$\frac{Q[id].rs \neq \emptyset}{\text{force}(Q, D, (\sigma, id)) \to Q, D, Q[id].rs} \quad \text{[Issued query]}$$

$$\frac{Q[id].rs = \emptyset \quad \forall id \in Q.\, Q'[id] = \begin{cases} D[Q[id].s] & \text{if } Q[id].rs = \emptyset \\ Q[id].rs & \text{otherwise} \end{cases}}{force(Q, D, ([\,], id)) \to Q', D, Q'[id].rs} \quad \text{[Unissued query]}$$

These evaluation rules show what happens when thunks are evaluated. The rules are defined based on the type of expression that is delayed. For instance, to evaluate a thunk with a variable expression, we simply look up the value of the variable from the environment that is embedded in the thunk. For thunks that contain method calls, we first evaluate each of the parameters by calling force , then we evaluate the method body itself, as in standard evaluation. Note that since we create thunks only for pure method calls, the heap remains unchanged after the method returns. Finally, for read queries, force either returns the result set that is stored in the query store if the corresponding query has already been executed, as described in the rule [Issued query], or issues all the unissued queries in the store as a batch before returning the results, as described in the rule [Unissued query].

D. EXPERIMENT DETAILS

In the following, we list the details of experiments as discussed in Section 7.1. For the original application, one query is issued per network round-trip.

OpenMRS benchmarks

Benchmark name	Original application		SLOTH compiled application			
	Time (ms)	# of network round trips	Time (ms)	# of network round trips	Max # of batched queries	Total # of queries issued
dictionary/conceptForm.jsp	2811	183	2439	36	3	38
dictionary/conceptStatsForm.jsp	5418	100	5213	82	16	112
dictionary/concept.jsp	1778	92	1626	36	3	38
optionsForm.jsp	1882	93	1196	19	5	30
help.jsp	1326	67	1089	21	2	27
admin/provider/providerAttributeTypeList.jsp	1988	102	1792	15	6	28
admin/provider/providerAttributeTypeForm.jsp	2000	88	1826	14	6	27
admin/provider/index.jsp	1845	99	1523	17	6	27
admin/provider/providerForm.jsp	1925	124	1893	11	5	25
admin/concepts/conceptSetDerivedForm.jsp	2055	89	1958	13	3	28
admin/concepts/conceptClassForm.jsp	2038	89	1724	12	4	28
admin/concepts/conceptReferenceTermForm.jsp	2252	120	2202	26	4	33
admin/concepts/conceptDatatypeList.jsp	2109	91	2071	24	3	29
admin/concepts/conceptMapTypeList.jsp	1997	119	1918	21	3	28
admin/concepts/conceptDatatypeForm.jsp	2178	148	1930	26	2	32
admin/concepts/conceptIndexForm.jsp	2072	119	1867	14	3	28
admin/concepts/conceptProposalList.jsp	2033	115	1920	16	3	29
admin/concepts/conceptDrugList.jsp	1933	102	1823	21	3	29
admin/concepts/proposeConceptForm.jsp	2406	89	1940	18	3	28
admin/concepts/conceptClassList.jsp	2072	91	1860	17	4	29
admin/concepts/conceptDrugForm.jsp	2535	133	2056	21	4	36
admin/concepts/conceptStopWordForm.jsp	1989	89	1803	18	5	28
admin/concepts/conceptProposalForm.jsp	2651	89	2103	18	6	28
admin/concepts/conceptSourceList.jsp	1897	94	1838	17	5	30
admin/concepts/conceptSourceForm.jsp	2215	92	2103	15	5	29
admin/concepts/conceptReferenceTerms.jsp	2565	143	2030	16	6	28
admin/concepts/conceptStopWordList.jsp	2560	92	1939	19	6	29
admin/visits/visitTypeList.jsp	2220	89	1356	18	6	28
admin/visits/visitAttributeTypeForm.jsp	1865	88	1125	22	5	27
admin/visits/visitTypeForm.jsp	2304	88	1141	22	5	27
admin/visits/configureVisits.jsp	2214	100	1716	21	5	25
admin/visits/visitForm.jsp	2043	140	1805	18	6	28
admin/visits/visitAttributeTypeList.jsp	2125	109	1523	15	6	28
admin/patients/shortPatientForm.jsp	2552	136	1742	33	9	54
admin/patients/patientForm.jsp	4402	222	2641	34	10	54
admin/patients/mergePatientsForm.jsp	2845	149	2457	21	20	47
admin/patients/patientIdentifierTypeForm.jsp	2269	118	1859	21	11	33
admin/patients/patientIdentifierTypeList.jsp	1847	91	1773	18	8	29
admin/modules/modulePropertiesForm.jsp	2033	89	1550	18	6	28
admin/modules/moduleList.jsp	2488	87	1819	15	5	27
admin/hl7/hl7SourceList.jsp	2599	89	1681	15	5	27
admin/hl7/hl7OnHoldList.jsp	2485	101	1829	14	5	27
admin/hl7/hl7InQueueList.jsp	2365	93	1779	15	5	27
admin/hl7/hl7InArchiveList.jsp	2273	93	1761	15	5	27
admin/hl7/hl7SourceForm.jsp	2053	87	1697	15	4	27
admin/hl7/hl7InArchiveMigration.jsp	2250	90	1698	15	5	25
admin/hl7/hl7InErrorList.jsp	2272	100	1894	17	4	27
admin/forms/addFormResource.jsp	1386	50	1065	18	8	58
admin/forms/formList.jsp	2167	89	1761	16	6	28
admin/forms/formResources.jsp	1320	50	1300	17	7	29
admin/forms/formEditForm.jsp	2966	219	1855	16	7	31
admin/forms/fieldTypeList.jsp	2082	89	1743	15	6	28
admin/forms/fieldTypeForm.jsp	1978	87	1894	17	7	27
admin/forms/fieldForm.jsp	2495	115	1845	18	10	33
admin/index.jsp	2782	91	2429	19	4	29
admin/orders/orderForm.jsp	2578	89	1919	21	4	28
admin/orders/orderList.jsp	2246	99	1966	21	4	33
admin/orders/orderTypeList.jsp	2077	89	1995	16	5	28
admin/orders/orderDrugList.jsp	1970	116	1233	15	8	34
admin/orders/orderTypeForm.jsp	1962	87	1774	17	6	27
admin/orders/orderDrugForm.jsp	2713	124	2296	21	6	30
admin/programs/programList.jsp	2013	89	1977	15	5	28
admin/programs/programForm.jsp	2248	121	1757	20	7	27

Fig. 15. OpenMRS load time detail results.

Benchmark name	Original application		SLOTH compiled application			
	Time (ms)	# of network round trips	Time (ms)	# of network round trips	Max # of batched queries	Total # of queries issued
admin/programs/conversionForm.jsp	2318	89	1817	21	7	29
admin/programs/conversionList.jsp	1786	89	1689	13	11	28
admin/encounters/encounterRoleList.jsp	2034	97	2013	15	8	25
admin/encounters/encounterForm.jsp	2449	172	1587	19	30	52
admin/encounters/encounterTypeForm.jsp	2128	89	1855	16	5	27
admin/encounters/encounterTypeList.jsp	1954	107	1803	20	5	28
admin/encounters/encounterRoleForm.jsp	2076	87	1811	17	8	25
admin/observations/obsForm.jsp	2399	131	2397	28	11	49
admin/observations/personObsForm.jsp	3911	230	3126	86	25	137
admin/locations/hierarchy.jsp	2319	172	2121	43	11	64
admin/locations/locationAttributeType.jsp	2344	88	1758	16	6	27
admin/locations/locationAttributeTypes.jsp	1821	90	1766	18	6	28
admin/locations/addressTemplate.jsp	1939	90	1672	15	5	28
admin/locations/locationForm.jsp	2423	138	2129	36	5	52
admin/locations/locationTagEdit.jsp	2497	166	2274	41	5	51
admin/locations/locationList.jsp	2170	127	1924	41	6	47
admin/locations/locationTag.jsp	2152	139	1726	21	5	31
admin/scheduler/schedulerForm.jsp	1902	95	1710	18	5	27
admin/scheduler/schedulerList.jsp	2224	106	1989	18	6	28
admin/maintenance/implementationIdForm.jsp	1979	124	1912	18	4	28
admin/maintenance/serverLog.jsp	1884	104	1628	17	4	27
admin/maintenance/localesAndThemes.jsp	2085	93	2033	17	4	30
admin/maintenance/currentUsers.jsp	1990	87	1799	16	5	27
admin/maintenance/settings.jsp	2179	92	2161	20	5	29
admin/maintenance/systemInfo.jsp	1902	90	1762	20	5	28
admin/maintenance/quickReport.jsp	2109	101	2021	17	3	28
admin/maintenance/globalPropsForm.jsp	3406	89	3042	17	4	28
admin/maintenance/databaseChangesInfo.jsp	12555	88	6732	15	8	27
admin/person/addPerson.jsp	1870	89	1843	18	5	28
admin/person/relationshipTypeList.jsp	2170	89	1806	10	5	28
admin/person/relationshipTypeForm.jsp	2222	121	1808	11	5	27
admin/person/relationshipTypeViewForm.jsp	1953	113	1825	11	7	28
admin/person/personForm.jsp	3154	149	1883	21	8	32
admin/person/personAttributeTypeForm.jsp	1854	89	1834	18	6	27
admin/person/personAttributeTypeList.jsp	2149	104	2021	19	6	33
admin/users/roleList.jsp	1892	113	1567	20	7	27
admin/users/privilegeList.jsp	2974	89	2051	18	8	27
admin/users/userForm.jsp	2443	126	1821	15	12	31
admin/users/users.jsp	2003	113	1921	15	11	27
admin/users/roleForm.jsp	2523	115	2060	11	13	27
admin/users/changePasswordForm.jsp	1481	105	1325	17	6	31
admin/users/alertForm.jsp	2595	113	1826	12	12	27
admin/users/privilegeForm.jsp	1905	87	1824	10	5	27
admin/users/alertList.jsp	12413	1705	9621	623	16	824
patientDashboardForm.jsp	7617	494	3610	95	15	190
encounters/encounterDisplay.jsp	9648	878	8242	260	68	406
forgotPasswordForm.jsp	1439	96	1128	12	12	27
feedback.jsp	1399	97	1121	11	5	27
personDashboardForm.jsp	2390	145	1965	17	3	32

Fig. 15. Continued

iTrackerbenchmarks

Benchmark name	Original application		SLOTH compiled application			
	Time (ms)	# of network round trips	Time (ms)	# of network round trips	Max # of batched queries	Total # of queries issued
module-reports/list_reports.jsp	22199	74	15929	24	9	38
self_register.jsp	22776	59	16741	23	5	41
portalhome.jsp	22435	59	14888	25	10	56
module-searchissues/search_issues_form.jsp	22608	59	16938	21	6	44
forgot_password.jsp	22968	59	15927	24	6	33
error.jsp	21492	59	15752	22	5	40
unauthorized.jsp	21266	59	17588	21	8	28
module-projects/move_issue.jsp	21655	59	18994	32	5	57
module-projects/list_projects.jsp	22390	59	15588	26	4	38
module-projects/view_issue_activity.jsp	23001	76	17240	33	11	47
module-projects/view_issue.jsp	22676	85	10892	39	20	62
module-projects/edit_issue.jsp	22868	129	11181	34	5	52
module-projects/create_issue.jsp	22606	76	18898	23	14	40
module-projects/list_issues.jsp	22424	59	19492	25	4	40
module-admin/admin_report/list_reports.jsp	21880	59	19481	23	5	35
module-admin/admin_report/edit_report.jsp	21178	59	18354	22	6	46
module-admin/admin_configuration/import_data_verify.jsp	22150	59	17163	22	4	38
module-admin/admin_configuration/edit_configuration.jsp	22242	59	18486	24	3	28
module-admin/admin_configuration/import_data.jsp	21719	59	16977	23	15	37
module-admin/admin_configuration/list_configuration.jsp	21807	59	19100	25	16	53
module-admin/admin_workflow/list_workflow.jsp	22140	59	19510	23	4	34
module-admin/admin_workflow/edit_workflowscript.jsp	22221	59	18453	22	3	29
module-admin/admin_user/edit_user.jsp	25297	59	23043	24	4	36
module-admin/admin_user/list_users.jsp	23181	59	15245	24	10	53
module-admin/unauthorized.jsp	22619	59	16964	23	5	39
module-admin/admin_project/edit_project.jsp	23934	59	17057	21	5	44
module-admin/admin_project/edit_projectscript.jsp	22467	59	20480	26	6	35
module-admin/admin_project/edit_component.jsp	21966	59	16596	21	7	45
module-admin/admin_project/edit_version.jsp	22577	59	15666	23	6	35
module-admin/admin_project/list_projects.jsp	23222	63	17445	22	4	39
module-admin/admin_attachment/list_attachments.jsp	22370	63	18109	24	8	41
module-admin/admin_scheduler/list_tasks.jsp	22282	61	16231	24	6	35
module-admin/adminhome.jsp	22138	73	11423	27	4	42
module-admin/admin_language/list_languages.jsp	22585	77	19597	22	6	39
module-admin/admin_language/create_language_key.jsp	22217	77	19174	23	5	39
module-admin/admin_language/edit_language.jsp	23347	66	20747	21	5	30
module-preferences/edit_preferences.jsp	22891	77	20845	22	4	39
module-help/show_help.jsp	22985	60	18598	22	6	40

Fig. 16. itracker load time detail results.

REFERENCES

Akamai. 2010. PhoCusWright/Akamai Study on Travel Site Performance. Retrieved April 21, 2016 from http://www.akamai.com/html/about/press/releases/2010/press_061410.html.

Lance Andersen. 2014. JSR 221: JDBC 4.0 API Specification. http://jcp.org/en/jsr/detail?id=221. (2014).

Apache Cayenne. 2014. Apache Cayenne ORM documentation. Retrieved April 21, 2016 from http://cayenne.apache.org/docs/3.0/prefetching.html. (2014).

Ivan T. Bowman and Kenneth Salem. 2004. Optimization of query streams using semantic prefetching. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'04)*. 179–190.

Mahendra Chavan, Ravindra Guravannavar, Karthik Ramachandra, and S. Sudarshan. 2011. Program transformations for asynchronous query submission. In *Proceedings of IEEE International Conference on Data Engineering (ICDE'11)*. 375–386.

Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C. Myers. 2012. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment (PVLDB 12)* 5, 11, 1471–1482.

Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. 2014a. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'14)*. 931–942.

Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C. Myers. 2014b. Using program analysis to improve database applications. *IEEE Data Engineering Bulletin* 37, 1, 48–59.

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. 3–14.

William R. Cook and Ben Wiedermann. 2011. Remote batch invocation for SQL databases. In *Proceedings of Database Programming Languages*.

Ezra Cooper. 2009. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *Proceedings of the International Symposium on Database Programming Languages (DBPL'09)*. 36–51.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web programming without tiers. In *Proceedings of International Symposium on Formal Methods for Components and Objects*. 266–296.

George Copeland and David Maier. 1984. Making smalltalk a database system. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'84)*. 316–325.

Database Test Suite. 2014. TPC-C and TPC-W reference implementations. Retrieved April 21, 2016 from http://sourceforge.net/apps/mediawiki/osdldbt. (2014).

Linda DeMichiel. 2006. JSR 220: Enterprise JavaBeans 3.0 specification (persistence). Retrieved April 21, 2016 from http://jcp.org/aboutJava/communityprocess/final/jsr220.

Django Project. 2014. Django web framework. Retrieved April 21, 2016 from http://www.djangoproject.com.

M. Fowler. 2003. *Patterns of Enterprise Application Architecture*. Addison-Wesley, New York, NY.

Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2012. SharedDB: Killing one thousand queries with one stone. *Proceedings of the VLDB Endowment (PVLDB'12)* 5, 6, 526–537.

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. 2009. FERRY: Database-supported program execution. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'09)*. 1063–1066.

Ravindra Guravannavar and S. Sudarshan. 2008. Rewriting procedures for batched bindings. *Proceedings of the VLDB Endowment (PVLDB'08)* 1, 1, 1107–1123.

Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A simultaneously pipelined relational query engine. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'95)*. 383–394.

Haskell wiki. 2015. Lazy evaluation in Haskell. Retrieved April 21, 2016 from http://wiki.haskell.org/Lazy_evaluation.

Peter Henderson and James H. Morris, Jr. 1976. A lazy evaluator. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'76)*. 95–103.

Hibernate. 2014a. Hibernate – relational persistence for idiomatic Java. Retrieved April 21, 2016 from http://hibernate.org/ hibernate.html.

Hibernate. 2014b. Hibernate fetching strategies. Retrieved April 21, 2016 from http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch20.html.

Hibernate. 2015. HQL: The Hibernate Query Language. Retrieved April 21, 2016 from http://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch16.html. (2015).

Ali Ibrahim and William R. Cook. 2006. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*. 50–73.

Peter Zilahy Ingerman. 1961. Thunks: A way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM* 4, 1, 55–58.

itracker. 2014. itracker issue management system. Retrieved April 21, 2016 from http://itracker. sourceforge.net.

Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. 2010. JReq: Database queries in imperative languages. In *Proceedings of International Conference on Compiler Construction (CC'10)*. 84–103.

Ming-Yee Iu and Willy Zwaenepoel. 2010. HadoopToSQL: A mapReduce query optimizer. In *Proceedings of the European Conference on Computer Systems (EuroSys'10)*. 251–264.

Simon L. Peyton Jones and André L. M. Santos. 1998. A transformation-based optimiser for Haskell. *Science of Computer Programming* 32, 1–3, 3–47.

Gary A. Kildall. 1973. A unified approach to global program optimization. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'73)*. 194–206.

Greg Linden. 2006. Marissa Mayer at Web 2.0. Retrieved April 21, 2016 from http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html. (2006).

E. Meijer, B. Beckman, and G. Bierman. 2006. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'06)*. 706–706.

Microsoft. 2014. Microsoft Entity Framework. Retrieved April 21, 2016 from http://msdn.microsoft.com/en-us/data/ef.aspx.

Microsoft. 2015. ODBC Programmer's Reference. Retrieved April 21, 2016 from http://msdn.microsoft.com/en-us/library/windows/desktop/ms714177(v=vs.85).aspx.

MySQL. 2015. MySQL ODBC Connector. Retrieved April 21, 2016 from http://dev.mysql.com/downloads/connector/odbc.

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An extensible compiler framework for Java. In *Proceedings of International Conference on Compiler Construction (CC'03)*. 138–152.

OCaml Tutorial. 2015. Functional Programming in OCaml. Retrieved April 21, 2016 from http://ocaml.org/learn/tutorials/functional_programming.html.

OpenMRS. 2014. OpenMRS medical record system. Retrieved April 21, 2016 from http://www.openmrs.org.

Oracle Corporation. 2015. Java 1.8 Lambda Expressions. Retrieved April 21, 2016 from http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

PHP. 2014. PHP query prefetch strategies. Retrieved April 21, 2016 from http://php.net/manual/en/function.oci-set-prefetch.php.

PostgreSQL. 2015. PostgreSQL ODBC driver. Retrieved April 21, 2016 from http://odbc.postgresql.org.

Karthik Ramachandra and Ravindra Guravannavar. 2014. Database-aware program optimization via static analysis. *IEEE Data Engineering Bulletin* 37, 1, 60–69.

Karthik Ramachandra and S. Sudarshan. 2012. Holistic optimization by prefetching query results. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'12)*. 133–144.

Mark Roth and Eduardo Pelegrí-Llopart. 2003. JSR 152: JavaServer Pages 2.0 specification. Retrieved April 21, 2016 from http://jcp.org/aboutJava/communityprocess/final/jsr152.

Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'15)*. 1311–1326.

Sudip Roy, Lucja Kot, and Christoph Koch. 2013. Quantum databases. In *Proceedings of Conference on Innovative Data Systems Research (CIDR'13)*.

Ruby on Rails Project. 2014. Ruby on Rails. Retrieved April 21, 2016 from http://rubyonrails.org.

Joachim W. Schmidt and Florian Matthes. 1994. The DBPL project: Advances in modular database programming. *Information Systems* 19, 2, 121–140.

Timos K. Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems* 13, 1, 23–52.

Xiaogang Shi, Bin Cui, Gillian Dobbie, and Beng Chin Ooi. 2014. Towards unified ad-hoc data processing. In *Proceedings of ACM SIGMOD/PODS Conference (SIGMOD'14)*. 1263–1274.

Alan Jay Smith. 1978. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems* 3, 3, 223–247.

StackOverflow. 2014a. Get record with max id, using Hibernate Criteria. Retrieved April 21, 2016 from http://stackoverflow.com/ questions/3900105/get-record-with-max-id-using-hibernate-criteria.

StackOverflow. 2014b. Hibernate performance issue. Retrieved April 21, 2016 from http://stackoverflow.com/questions/5155718/hibernate-performance.

StackOverflow. 2014c. Network latency under Hibernate/c3p0/MySQL. Retrieved April 21, 2016 from http://stackoverflow.com/ questions/3623188/network-latency-under-hibern ate-c3p0-mysql.

StackOverflow. 2014d. Round trip/network latency issue with the query generated by hibernate. Retrieved April 21, 2016 from http://stackoverflow.com/questions/13789901/round-trip-network-latency-issue-with-the-query-generated-by-hibernate.

StackOverflow. 2014e. What is the n+1 selects issue? Retrieved April 21, 2016 from http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue.

Shido Takafumi. 2015. Lazy evaluation in Scheme. Retrieved April 21, 2016 from http://www.shido.info/lisp/scheme_lazy_e.html.

David Turner. 1986. An overview of Miranda. *SIGPLAN Notices* 21, 12, 158–166.

Alessandro Warth. 2007. LazyJ: Seamless lazy evaluation in Java. *Proceedings of the FOOL/WOOD Workshop.*

Ben Wiedermann, Ali Ibrahim, and William R. Cook. 2008. Interprocedural query extraction for transparent persistence. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'08)*. 19–36.

Limsoon Wong. 2000. Kleisli, a functional query system. *Journal of Functional Programming* 10, 1, 19–56.

Kwok Cheung Yeung and Paul H. J. Kelly. 2003. Optimising Java RMI programs by communication restructuring. In *Middleware*, Lecture Notes in Computer Science, Vol. 2672. 324–343.

Fubo Zhang and Erik H. D'Hollander. 2004. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering* 30, 4, 231–245.

Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. 2007. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'07)*. 723–734.

Received February 2015; revised October 2015; accepted February 2016