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

Alvin Cheung Samuel Madden Armando Solar-Lezama
MIT CSAIL
{akcheung,madden,asolar}@csail.mit.edu

ABSTRACT

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 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. In this paper, 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. We evaluated Sloth using over 100 benchmarks from two large-scale open-source applications, and achieved up to a 3× reduction in page load time by delaying computation.

1. INTRODUCTION

Most web applications are backed by database servers that are physically separated from the servers hosting the application. Even though the two machines tend to reside in close proximity, a typical page load 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 libraries automatically translate accesses to objects into SQL, often resulting in multiple queries (and round trips) to reconstruct a single object. For example, even with the application and database servers hosted in the same data center, we found that many pages spend 50% or more of their time waiting on network communication.

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 web page that takes more than three seconds to load [9]. As another example, Google reported in 2006 that an extra 0.5 second of latency reduced the overall traffic by 20% [6]. These numbers are likely worse on modern computers where pages load faster and faster, making it increasingly important to reduce the amount of time spent on database queries. 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 10's of milliseconds per object can turn into seconds of additional latency for an entire web page [12, 2, 7, 10]. Though some techniques (such as Hibernate's "eager fetching") aim to mitigate this, they are far from perfect as we discuss below. Finally, decreasing latency often increases throughput: as each request takes less time to complete, the server can process more requests simultaneously.

There are 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 et al. [23], where they 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 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 in order to reduce the number of round trips. The default strategy is usually "lazy fetching," where 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 the 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,

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2593672.

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 non-trivial 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 paper 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. The key idea is to collect queries by relying on a new technique which we call *extended lazy evaluation* (or simply "lazy evaluation" in the rest of the paper.) As the application executes, queries are batched into a query store instead of being executed right away. In addition, non-database 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 the queries that are registered with the query store are executed by the database in a single batch, and the results are then used to evaluate the outcome of the computation. The technique is conceptually related to the 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 [20]. 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 they can be issued together with other queries in the batch.

Compared to query extraction using static analysis, 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 multi-query optimization approaches that optimize batches of queries [16]; we do not merge queries to improve their performance, or 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 (which still executes the individual query statements.)

We have implemented this approach in a new system called Sloth. The system is targeted towards 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. In summary, our paper makes the following contributions:

- We devise a new mechanism to batch queries in database-backed applications based on a combination of a new "lazy-ifying" compiler and dynamic program analysis to generate the queries to be batched. 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×.

In the following, we first describe how Sloth works through a motivating example in Sec. 2. Then, we explain our compilation strategy in Sec. 3, followed by optimizations to improve generated code quality in Sec. 4. We describe our prototype implementation in Sec. 5, and report our experimental results using both real-world benchmarks in Sec. 6.

```
1  ModelAndView handleRequest(...) {
2  Map model = new HashMap<String, Object>();
3  Object o = request.getAttribute("patientId");
4  if (o != null) {
5    Integer patientId = (Integer) o;
6    if (!model.containsKey("patient")) {
7      if (hasPrivilege(VIEW_PATIENTS)) {
8        Patient p = getPatientService().getPatient(patientId);
9        model.put("patient", p);
10       ...
11       model.put("patientEncounters",
12         getEncounterService().getEncountersByPatient(p));
13       ...
14       List visits = getVisitService().getVisitsByPatient(p);
15       CollectionUtils.filter(visits, ...);
16       model.put("patientVisits", visits);
17       model.put("activeVisits", getVisitService().
18         getActiveVisitsByPatient(p));
19   ...
20   return new ModelAndView(portletPath, "model", model);
21 }
```

Figure 1: Code fragment abridged from OpenMRS

2. OVERVIEW

In this section we give an overview of Sloth using the code fragment shown in Fig. 1. The fragment is abridged from OpenMRS [8], 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, and the code fragment 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. The controller 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 which 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 `getXXX` on the data access objects, i.e., 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 18 to fetch various data associated with the patient, and adds this data to the

model as well. 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 be constructed. In fact, the results from the other queries are only stored in the model and not used until the view 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 in 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 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 is *lazy evaluation*. In most traditional programming languages, the evaluation of a statement causes that statement to execute, so 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 place-holder 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), or 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 they represented will finally be executed.

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 to batch queries. Because the computation has been delayed, the results of those queries are not yet needed, so 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 Fig. 2). 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 non-thunk 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 18 that fetches the patient's active visits (Q4), and the method returns.

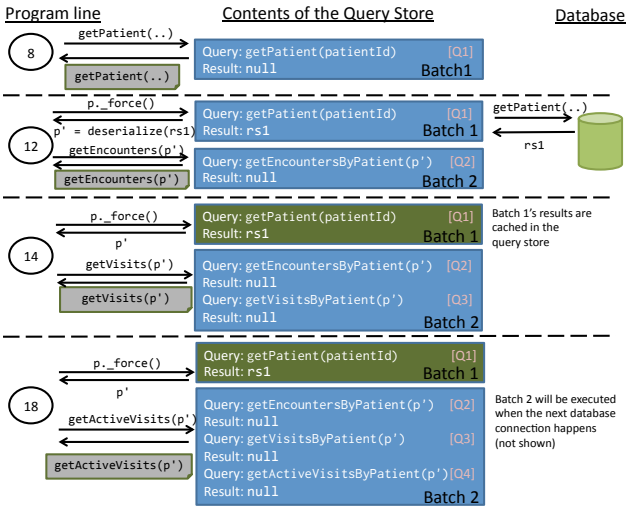


Figure 2: Operational diagram of the example code fragment

Depending on 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 [23]. Hibernate's lazy fetching mode would have to evaluate the results of the database-accessing statements such as `getVisitsByPatient(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 avoid 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 tradeoffs 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 Sec. 5). 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.

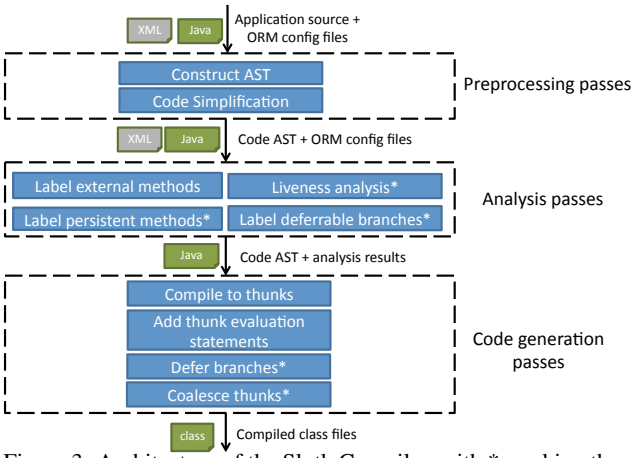


Figure 3: Architecture of the Sloth Compiler, with * marking those components used for optimization

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 next.

3.1 Code Simplification

To ease the implementation, the Sloth compiler first simplifies the input source code. All looping 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.

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 Sec. 3.8 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 following statement:

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

is compiled into lazy semantics as:

```
Thunk<int> x =
  new Thunk<int>() {
    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<int>` after compilation, 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`. Secondly, to avoid redundant evaluations, we memoize the return value of `_force` so that subsequent calls to `_force` will return the memoized value instead (the details of which are not shown).

While the compilation is relatively straightforward, the mechanism presented above 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. Sec. 4 describes several optimizations that we have devised to reduce the overhead. Sec. 6.6 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 it can be issued when the next batch is sent to the database. To facilitate this, the query store 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; 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` the current batch will be immediately sent to the database to ensure these updates are not left lingering in the query store. On the other hand, the method avoids introducing redundant queries into the batch, so if `sql` 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 Fig. 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 a SQL result set.

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 and 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). The target of the call might be a class external to the application, such as a method of the standard JDK, where the Sloth compiler does not have access to its source code (we call such methods "internal" otherwise). Because of that, method calls are compiled differently according to the type of the called method as follows. Method labeling is done as one of the analysis passes in the Sloth compiler, and the thunk conversion pass uses the method labels during compilation.

Internal methods without side effects. This is the ideal case where 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<int> r = new Thunk<int>(x, y) {
   int _force() { return this.foo(x._force(), y._force()); }
 };
```

Internal methods with externally visible side effects. We cannot defer the execution of such method 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 where its parameters are thunk values, and the original call site 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<int> r = bar_thunk(x);
```

with the declaration of `bar_thunk` as `Thunk<int> 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 Fig. 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 (non-thunk) return value of external method calls are stored in `LiteralThunk` objects that simply returns the non-thunk value when `_force` is called, as shown in the example above.

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` fields, the compiler adds an extra field with the original type, with its value set to the evaluated result of the corresponding thunk-ified version of the field. These fields are created so that they can be accessed from external methods. Publicly accessible non-final 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 Sec. 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 the thunk fields. However, updates to heap objects are not delayed in order to ensure consistency of subsequent heap reads. In order to carry out the write, however, the receiver object needs to be evaluated if it is a thunk. Thus, the following 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, i.e., 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, where the target array and index are evaluated before the operation.

For method calls where the execution of the method body is not delayed, the target object is evaluated prior to the call if the called method is non-static. 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, there is unlikely any significant savings in delaying such 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 Sec. 4.2 below. Similarly, statements that throw exceptions, obtain

$$
\begin{array}{rcl}
c \in \text{constant} & ::= & \text{True} \mid \text{False} \mid \text{literal} \\
e_l \in \text{assignExpr} & ::= & x \mid e.f \\
e \in \text{expr} & ::= & c \mid e_l \mid \{f_i = e_i\} \mid e_1 \; op \; e_2 \mid \neg\, e \mid f(e) \mid e_a[e_i] \mid R(e) \\
c \in \text{command} & ::= & \text{skip} \mid e_l := e \mid \text{if}(e) \text{ then } c_1 \text{ else } c_2 \mid \\
 & & \text{while(True) do } c \mid W(e) \mid c_1 \; ; \; c_2 \\
op \in \text{binary op} & ::= & \wedge \mid \vee \mid > \mid < \mid =
\end{array}
$$

Figure 4: Input language

locks on objects (`synchronized`), and that spawn new threads of control are not deferred. Finally, thunk evaluations can also happen when compiling statements that issue queries, as discussed in Sec. 3.3.

3.7 Limitations

There are two 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.

3.8 Formal Semantics

We now formally define the extended lazy evaluation outlined above. For the sake of presentation, we describe the semantics in terms of the language shown in Fig. 4. 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. For lack of space, this section provides only an outline of the semantic rules and the main idea behind the soundness proof, the complete semantics are described in [15].

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 $W(e)$ a statement that issues a query that can mutate the database, such as `INSERT` or `UPDATE`.

We first define the standard execution semantics of the language. 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 below.

$$
\frac{\langle s, e_1\rangle \rightarrow \langle s', v_1\rangle \quad \langle s', e_2\rangle \rightarrow \langle s'', v_2\rangle \quad v_1 \; op \; v_2 \rightarrow v}{\langle s, e_1 \; op \; e_2\rangle \rightarrow \langle s'', v\rangle}
$$

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 and 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, and 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, so the query v must execute on the database as it is left after the evaluation of e:

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

The rest of the evaluation rules are standard and are included in [15].

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 state is implemented as fields in each generated `Thunk` class, and e is the expression in the body of the `_force` method.

As discussed in Sec. 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, and then create another thunk object that represents the op. Formally this is described as:

$$
\frac{\begin{array}{c}\langle s, e_1\rangle \rightarrow \langle s', v_1\rangle \quad \langle s', e_2\rangle \rightarrow \langle s'', v_2\rangle \\ v_1 = (s', e_1') \qquad v_2 = (s'', e_2') \\ v = (\sigma' \cup \sigma'', e_1' \; op \; e_2')\end{array}}{\langle s, e_1 \; op \; e_2\rangle \rightarrow \langle s'', v\rangle}
$$

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.

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

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

The force function above is used to evaluate thunks, similar to that described in the examples above using Java. $\text{force}(Q, D, t)$ 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, and then assigns those results once they arrive from the database.

Using the semantics outlined above, we have proven the equivalence of standard and lazy semantics by showing that if evaluating command c on program state $\langle Q, D, \sigma, h\rangle$ results in the new state $\langle D_s, \sigma_s, h_s\rangle$ under standard semantics, and $\langle Q_l, D_l, \sigma_l, h_l\rangle$ under lazy semantics, then after forcing all thunk objects in σ_l and h_l, we have $D_l = D_s$, $\sigma_l = \sigma_s$, and $h_l = h_s$, regardless of the order in which the thunks are forced. The proof is included in [15].

4. BEING EVEN LAZIER

In the previous section, we described how Sloth compiles source code into lazy semantics. However, as noted in Sec. 3.2, there

can be substantial overhead if we follow the compilation procedure naively. In this section, we describe three optimizations. The goal of these optimizations is to generate more efficient code and to further defer computation. As discussed in Sec. 2, deferring computation delays thunk evaluations, which in turn increases the chances of obtaining larger query batches during execution time. Like the previous section, we describe the optimizations using concrete Java syntax for clarity, although they can all be formalized using the language described in Fig. 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, and simply adds 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 inter-procedural, flow-insensitive dataflow analysis [21]. It first identifies the set of methods m containing statements that perform any of the above. 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 Sec. 6.5 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 Sec. 3.6. The rationale is that the outcome of the branch affects subsequent program path. We can do better, however, 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: i) calls that issue queries; or ii) thunk evaluations as discussed in Sec. 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 following 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(b, d) {
  void _force () {
    if (c._force()) a = b._force(); else a = d._force();
  } }
Thunk<int> a = tb.a();
```

where the evaluation of `c` is further delayed. The `ThunkBlock` class is similar to the `Thunk` class except that it defines methods (not shown above) 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 delay thunk evaluations, which in turn might increase query batches 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 thunk 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 above. 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 if all statements inside the loop body can be deferred.

4.3 Coalescing Thunks

The basic compilation described in Sec. 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 (and 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 Sec. 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<int> foo (Thunk<int> a, b, c, d) {
2    Thunk<int> e = new Thunk0(a, b) { ... }
3    Thunk<int> f = new Thunk1(e, c) { ... }
4    Thunk<int> g = new Thunk2(f, d) { ... }
5    return g; }
```

Notice that three thunk objects are created, with the additions in 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, i.e., they are no longer *live*, after Line 4. Thus we can combine the first three statements into a single thunk, resulting in the following:

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

The optimized version reduces the number of object allocations from 3 to 2: one allocation for `ThunkBlock3` and another one 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 (and 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 of `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 and thus improves the efficiency of the generated code as shown in Sec. 6.5.

5. IMPLEMENTATION

We have implemented a prototype of Sloth. The Sloth compiler is built on top of Polyglot [22]. 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 process thunk objects that are returned by the hosted application. Our extensions are not language specific and can be applied to other ORM and app 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) [5] to allow returning thunk objects from calls that retrieves 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 a 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, and the extensions were implemented using about 1000 lines of code.

Spring Extensions. We extended the Spring web application framework to allow thunk objects 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 [4] 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 write 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.

6. EXPERIMENTS

In this section we report our experiment 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 above. The web server and applications were hosted on a machine with 8GB of RAM and 2.8GHz processor, and data was stored in an unmodified MySQL 5.5 database with 47GB of RAM and 12 2.4GHz processors. 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 [3]: 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 [8]: 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 web page files (html and jsp files). Next, we analyzed the application to find the URLs that load each of the web pages. 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 [11]. Because the implementations display the query results immediately after issuing them, there is no opportunity for batching. We only use them to measure the runtime overhead of lazy evaluation.

6.1 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 in size) provided by the application. For itracker, we created an artificial database (0.7GB in size) consisting of 10 projects and 20 users. Each project has 50 tracked issues, and none of the issues has attachments. We did not define custom scripts or components for the projects that we created. We also created larger versions of these databases (up to 25 GB) and report their performance on selected benchmarks in Section 6.4, 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 Sec. 5. For all benchmarks, we computed the speedup ratios as:

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

Fig. 5(a) and Fig. 6(a) show the CDF of the results (result details are described in [15]), where we sorted the benchmarks according to their speedups for presentation purposes (and similarly for other experiments). 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. Figure 5(b) and Fig. 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 web page, issues such as the $1+N$ issue in Hibernate [12] make it quite common for developers to write apps that issue hundreds of queries to generate a web page in widely used ORM frameworks.

Finally, Fig. 5(c) and Fig. 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 Sec. 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, one of the OpenMRS benchmarks (`encounterDisplay.jsp`) loads observations about a patient's visit. Observations include height, blood pressure, etc, and 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 Fig. 1 (Line 8). The model is returned at the end of the method and the fetched concepts are displayed in the view.

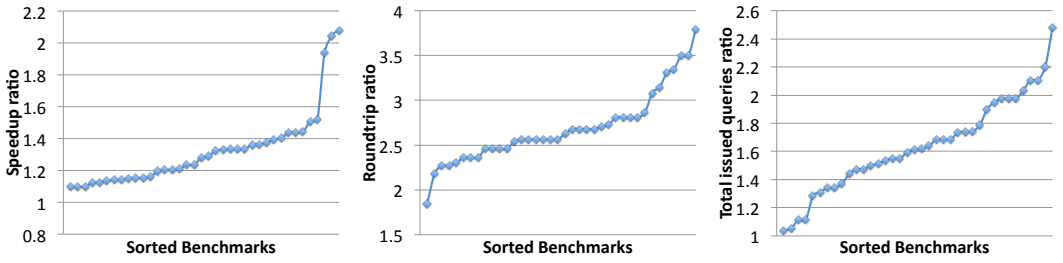
```
1 if (Context.isAuthenticated()) {
2   FormService fs = Context.getFormService();
3   for (Obs o : encounter.getObsAtTopLevel(true)) {
4     FormField ff = fs.getFormField(form, o.getConcept(),..);
5     ...
6     obsMapToReturn.put(ff, list);
7   }}
8 map.put("obsMap", obsMapToReturn);
9 return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

In the original application, the concept entities are lazily fetched by the ORM during view generation, and 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 presence of the authentication check on Line 1, and techniques such as [13] will require a detailed inter-procedural 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 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 where the Sloth-compiled application issued *more* queries than the original, as shown in Fig. 6(c) This is because the Sloth-compiled application registers queries to the query store whenever 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 and that results in fewer queries executed. 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 database round trips outweigh the costs of executing a few extra queries.

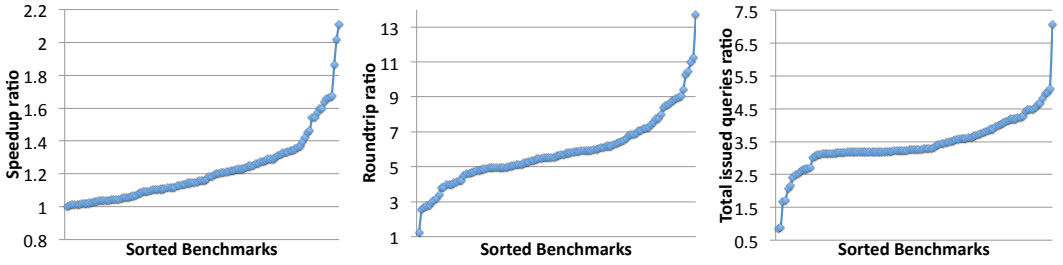
6.2 Throughput Experiments

Next, we compared the throughput of Sloth-compiled application and the original. We fixed the number of browser clients, and each client repeatedly loaded pages from OpenMRS for 10 minutes (clients wait until the previous load completes, and 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 7.



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

Figure 5: itracker benchmark experiment results



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

Figure 6: OpenMRS benchmark experiment results

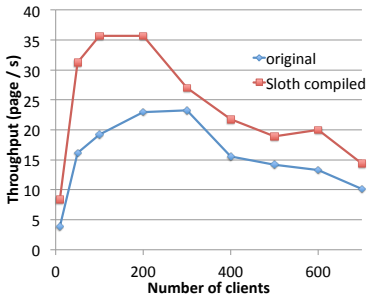
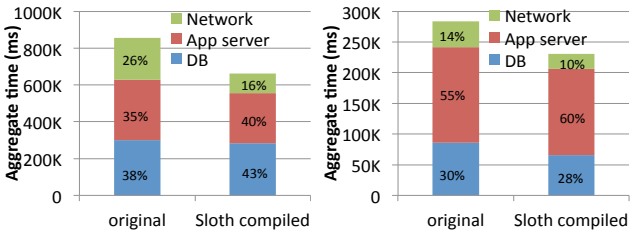


Figure 7: Throughput experiment results

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 under-utilized 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-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.

6.3 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, and computed the sum of time spent in each phase across all benchmarks. The results for the two applications are shown in Fig. 8.



(a) itracker benchmarks (b) OpenMRS benchmarks

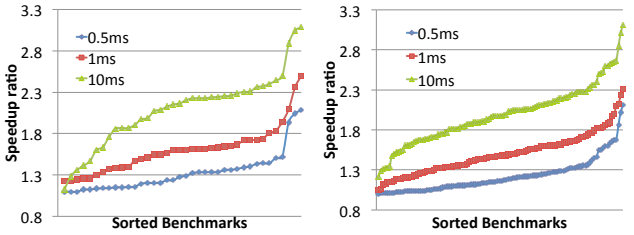
Figure 8: Time breakdown of benchmark loading experiments

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.

6.4 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 Sec. 6.1, but varied the amount 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 9 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 round trip time increases (more than 3× for both applications with round trip time of 10ms). This

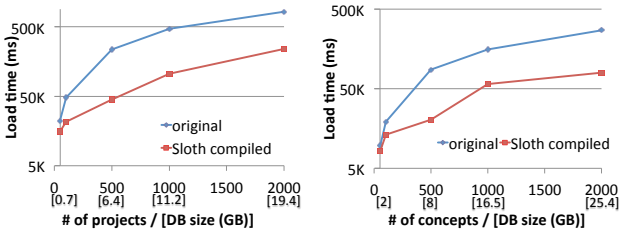


(a) itracker (b) OpenMRS

Figure 9: Network scaling experiment results

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 25 GB) and measured the benchmark load times. Although the database still fits into the memory of the machine, we believe 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 Sec. 6.1, and varied the number of observations stored. The results are shown in Fig. 10(a) and (b) respectively.



(a) itracker (b) OpenMRS

Figure 10: Database scaling experiment results

The Sloth-compiled applications achieved lower page load times in all cases, and they also scaled better as the number of entities increases. 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 Sec. 6.1. While the numbers of queries issued by two versions of the application are the same proportionally as the number of entities increases, 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.

6.5 Optimization Experiments

In this experiment we measured the effects of the optimizations presented in Sec. 4. First, we study the effectiveness of selective compilation. Figure 11 shows the number of methods that are identified as persistent in the two applications. As discussed Sec. 4.1, non-persistent 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

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

Figure 11: Number of persistent methods identified

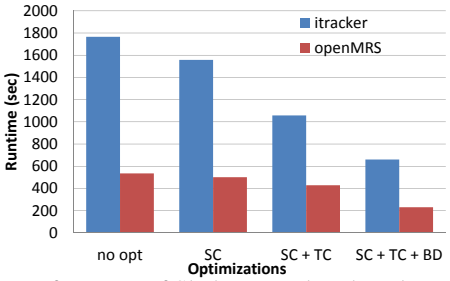


Figure 12: Performance of Sloth on two benchmarks as optimizations are enabled. SC=Selective computation, TC=Thunk Coalescing, BD=Branch Deferral.

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 and Fig. 12 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 Sec. 4.2, deferring control flow statements further delays the evaluation of thunks, which allows more query batching to take place.

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, i.e., the actual load times of the Sloth-compiled applications would have been slower than the original.

6.6 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 standalone 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 1 GB 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 13 show the results.

As expected, the Sloth compiled versions were 5-15% slower than the original, due to lazy semantics. However, given that the Java virtual machine is not designed for lazy evaluation, we believe these overheads are reasonable, especially given the significant performance gains observed in real applications.

6.7 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

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%

Figure 13: Overhead experiment results

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 an interesting future work.

7. RELATED WORK

Lazy evaluation was first introduced for lambda calculus [18], 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. Lazy evaluation is often implemented using thunks in languages that do not readily support it [19, 27]. In contrast, the extended lazy evaluation proposed in this paper is fundamentally different: rather than its traditional uses, Sloth uses lazy evaluation to improve application performance by batching queries, and 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.

Batching query plans and sharing query results are well-known query optimization techniques [16, 25], and there is also work on re-ordering transactions using developer annotations [24]. However, they aim to combine queries issued by multiple concurrent clients, whereas Sloth batches queries that are issued by the same client over time, although we can make use of such techniques to merge Sloth-generated query batches from multiple clients for further performance improvement. There is work on using static analysis to expose batching opportunities for queries [17, 13] and remote procedure calls [28]. However, their batching ability is limited due to the imprecision of static analysis. While Sloth does 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.

As discussed in Sec. 1, data prefetching is another means to reduce database round trips. Prefetching has been studied theoretically [26] and implemented in open source systems [1], although they all require programmer annotations to indicate what and when to prefetch. Finally, there is work on moving application code to execute in the database as stored procedures to reduce the number of round trips [14], which is similar to our goals. In comparison, Sloth does not require program state to be distributed.

8. ACKNOWLEDGMENT

We thank the anonymous reviewers for the constructive feedback. This research was supported by the Intel Science and Technology Center for Big Data, and NSF grants SHF-1116362 and SHF-1139056.

9. CONCLUSIONS

In this paper 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.

10. REFERENCES

[1] Hibernate fetching strategies. http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch20.html.
[2] Hibernate performance issue. http://stackoverflow.com/questions/5155718/hibernate-performance.
[3] iTracker issue management system. http://itracker.sourceforge.net.
[4] JSR 152: JavaServer Pages 2.0 specification. http://jcp.org/aboutJava/communityprocess/final/jsr152.
[5] JSR 220: Enterprise JavaBeans 3.0 specification (persistence). http://jcp.org/aboutJava/communityprocess/final/jsr220.
[6] Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html.
[7] Network latency under Hibernate/c3p0/MySQL. http://stackoverflow.com/questions/3623188/network-latency-under-hibernate-c3p0-mysql.
[8] OpenMRS medical record system. http://www.openmrs.org.
[9] PhoCusWright/Akamai Study on Travel Site Performance. http://connect.phocuswright.com/2010/06/phocuswrightakamai-study-on-travel-site-performance/.
[10] Round trip / network latency issue with the query generated by hibernate. http://stackoverflow.com/questions/13789901/round-trip-network-latency-issue-with-the-query-\generated-by-hibernate.
[11] TPC-C and TPC-W reference implementations. http://sourceforge.net/apps/mediawiki/osdldbt.
[12] What is the n+1 selects issue? http://stackoverflow.com/questions/97197/what-is-the-n1-selects-issue.
[13] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. Program transformations for asynchronous query submission. In *Proc. ICDE*, pages 375–386, 2011.
[14] A. Cheung, O. Arden, S. Madden, and A. C. Myers. Automatic partitioning of database applications. *PVLDB*, 5(11):1471–1482, 2012.
[15] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *CSAIL Tech Report MIT-CSAIL-TR-2014-006*.
[16] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. *PVLDB*, 5(6):526–537, 2012.
[17] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. *PVLDB*, 1(1):1107–1123, 2008.
[18] P. Henderson and J. H. Morris, Jr. A lazy evaluator. In *Proc. POPL*, pages 95–103, 1976.
[19] P. Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Comm. ACM*, 4(1):55–58, 1961.
[20] S. L. P. Jones and A. L. M. Santos. A transformation-based optimiser for haskell. *Sci. Comput. Program.*, 32(1-3):3–47, 1998.
[21] G. A. Kildall. A unified approach to global program optimization. In *Proc. POPL*, pages 194–206, 1973.
[22] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. CC*, pages 138–152, 2003.
[23] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proc. SIGMOD*, pages 133–144, 2012.
[24] S. Roy, L. Kot, and C. Koch. Quantum databases. In *Proc. CIDR*, 2013.
[25] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.
[26] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, 1978.
[27] A. Warth. LazyJ: Seamless lazy evaluation in java. *Proc. FOOL/WOOD Workshop*, 2007.
[28] K. C. Yeung and P. H. J. Kelly. Optimising java RMI programs by communication restructuring. In *Middleware*, volume 2672 of *LNCS*, pages 324–343, 2003.