Performance Profiling with EndoScope, an Acquisitional Software Monitoring Framework

Alvin Cheung
MIT CSAIL
akcheung@mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

We propose EndoScope, a software monitoring framework that allows users to pose declarative queries that monitor the state and performance of running programs. Unlike most existing monitoring tools, EndoScope is *acquisitional*, meaning that it only instruments the portions of the program that need to be monitored to answer queries. The use of a high level declarative language allows EndoScope to search for efficient physical instantiations of queries by applying a suite of optimizations, including control flow graph analysis, and traditional database query optimization techniques, such as predicate pushdown and join optimization, to minimize the number of program instrumentation points and overhead to the monitored program. Furthermore, a flexible, high level language and the ability to attach to running programs enable developers to build various program analysis and monitoring applications beyond traditional software profilers with EndoScope. We describe a prototype implementation of the EndoScope framework and a simple profiler for Java programs implemented with EndoScope. We show results from using our profiler on a collection of real-world programs, including a TPC-C implementation using the Derby database and the petstore application running on top of Tomcat application server. Our results show the benefit of our optimization framework and demonstrate that our declarative, acquisitional approach can yield program instrumentation overheads that are dramatically lower than conventional profiling tools (for example, when profiling the Derby Database running TPC-C, our system's overhead ranges from 1% to about 25%, whereas the fastest existing profiler we measured imposes a minimum overhead of about 30%.)

1. INTRODUCTION

Understanding and monitoring the behavior and performance of large software systems can be a daunting task. Existing profiling tools are of some help, but typically impose a large overhead [33] or provide only limited functionality (*e.g.*, they only compute the amount of time taken by all functions in a program, or only sample CPU usage over a program's lifetime). Our goal is to build a tool that allows programmers and software administrators (of web or database servers, for example) to monitor many different aspects of running programs (memory usage, function invocations, variable values) in one framework and detect and respond to interesting or exceptional conditions inside their software in real time. For example:

1. A database administrator may wish to know when the runtime of a given query exceeds some fixed threshold, and what the system memory usage, query parameters, and other queries running in the system were when the condition occurred. Such performance guarantees for queries are often required for transaction processing systems where certain service level agreements (SLAs) on response time with customers must be met.
2. A security specialist may wish to receive a report whenever a particular function is invoked or when a variable has a certain value. For example, in a web application, if an administrative function (*e.g.*, creating a user account) is invoked when a global flag indicates that the current session has administrator privileges is unset, that may indicate a security breach or incorrect user authentication code.
3. A compiler developer may want to continuously monitor the frequency with which each function is invoked in a program. Such information would be useful in identifying hot spots in a program for dynamic recompilation.

To assist programmers and system administrators with these kinds of monitoring applications, we are building a software monitoring framework called EndoScope. EndoScope takes a declarative, query-based approach to software performance monitoring. In EndoScope, users pose queries to monitor the operation of running programs in an SQL-like language. This language exposes the state of programs—including the values of variables and the execution flow of threads—as a collection of data streams over which queries can be posed. Results of queries are themselves data streams, which can be fed into other tools or queries.

EndoScope is *acquisitional* [30] in the sense that these data streams are not actually materialized. Instead, only the subsets of streams needed to answer queries posed over the program are captured and recorded. This is essential because the cost of simply recording all of the state of a running program can be prohibitively expensive: imagine, for example, trying to record each value taken by every variable over the life of a running program—such instrumentation

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

would cause the program to run hundreds of times slower than it would during normal execution. Because EndoScope is acquisitional, different queries impose different overheads on a program: finding the longest running function will require instrumenting all functions, whereas finding instances when the response time of a function violates a particular response time goal will require instrumenting just the function in question.

Conventional profilers (such as gprof for C/C++ programs and hprof for Java) are typically non-acquisitional: they record all of the functions invoked by a program over time. This is because they are targeted at the task of finding the most costly portions of a program, which requires global instrumentation. Unfortunately, this means that existing profilers are poorly suited for fine scale monitoring of just a few functions or memory locations as they impose a severe overhead regardless of what kind of monitoring they are performing. In contrast, EndoScope's acquisitional approach and query-based interface allow it to instrument only the portions of the program that are needed to answer the user's queries. Furthermore, by using a high-level language, EndoScope's optimizer is able to search for the lowest-overhead instrumentation strategy that will satisfy the user's query. EndoScope includes a suite of optimizations, including adaptations of traditional ordering optimizations such as predicate push down and join ordering analysis that take on a different flavor in the case of program monitoring.

In summary, the key contributions of EndoScope framework are:

1. A high level, declarative language that allows users to monitor the performance of running programs either locally or remotely. Results of performance monitoring are exposed as data streams that can be fed into debugging, alerting, and other tools that make use of such data.
2. An acquisitional approach that allows it to instrument just the portions of a running program or sample the type of system resources that are needed to answer user's queries.
3. A formulation of the problem of determining the order in which multiple system resources should be monitored as a search for an optimal query plan. We then apply traditional database cost-based query optimization techniques in plan search.
4. A unified data model that abstracts all information collected during a program's runtime as data streams to expose to the end user.
5. An implementation of a prototype system that implements our proposed streaming data model, and a software profiler that makes use of EndoScope. The resulting profiler introduces less overhead than existing profiling tools in many cases. Our profiler also provides a richer interface that allows users to pose queries and triggers that are beyond those offered by existing tools.

In the remainder of this paper, we first survey related work in Section 2. We then describe the architecture and data model used by EndoScope and the mechanisms it uses for determining where to instrument a running program given a particular query in Sections 3 and 4. We present our optimization framework and instrumentation-based optimizations in Section 5. Section 6 discusses our Java-based implementation and demonstrates how EndoScope can be used to implement monitoring tools. Section 7 provides performance results showing that our implementation allows low-overhead monitoring of real systems, including the Tomcat application server and the Derby database, and Section 8 concludes with open research questions.

2. RELATED WORK

Profiling tools exist for all major programming languages and platforms. We broadly classify these tools into three categories based on the method each tool uses to capture program information.

Sampling. Sampling based tools such as gprof [26], and hprof [7] are implemented by halting program execution at pre-specified times. When halted, the profiler samples the program counter and stack. At the end of program execution, a profile is constructed by computing the number of times the profiled program invoked each function. Such tools are mainly used to determine the functions that were most time-consuming over the program's lifetime. Because they are based on sampling, they are inadequate for answering precision-related queries, such as the number of times a function is invoked, or for alerting the user when a specific condition becomes true (*e.g.*, the response time of a function exceeds a threshold.)

Hardware Counters. Hardware-counter based tools capture program information by reading special hardware counters provided by processors, as in those included with Pentium 4 and POWER4. These counters are special-purpose registers that count the number of hardware-related events in the system, such as the number of cache misses. Software tools (such as DCPI [15] and oprofile [12]) sample the values from these counters periodically. While hardware counters capture low-level system information that might be difficult to obtain via software programs, it is not an easy task to infer program behavior from hardware counters since they do not always capture software information (*e.g.*, which line of code affected the counter.)

Program Instrumentation. Program instrumentation tools insert binary code ("instrumentation points") into a program. Instrumentation can be performed offline by examining source code, or online by analyzing execution binaries or bytecodes. When the program reaches an instrumentation point, it invokes a callback function in the tool, which examines the status of the running program and records relevant information. Profilers that are built using instrumentation include dtrace [21], shark [13], jip [9], and visualvm [14]. The same mechanism can also be used to implement watchpoints and breakpoints in debuggers, as in gdb [5], jdb [5], and jswat [11]. While instrumentation provides a more precise way to specify when to halt a running program, inserting instrumentation points into a program can be costly, both in terms of the up front cost in program analysis to determine places in a program to instrument, and also the extra overhead in executing the callback functions. As a result, profilers often don't use program instrumentation, because it can dramatically slow down a running program. In addition, the tool developer also needs to be careful in not introducing any side effects (*e.g.*, polluting call stacks or activation records) into the profiled program when adding or removing instrumentation points.

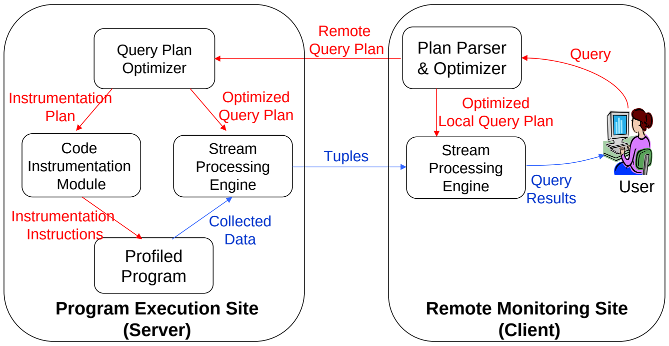

Figure 1: EndoScope Framework Overview

Our current implementation of EndoScope is primarily instrumentation-based, although it is possible to read hardware counters from an EndoScope query, and we anticipate adding sampling based methods in the future. The primary difference between EndoScope and existing instrumentation based tools is that the existing tools are all focused on achieving a specific task (*e.g.*, profiling, debugging), whereas our goal is to build a general software monitoring platform that can be used to implement a number of different applications. As such, the interface provided by EndoScope is richer (declarative query based) and more extensible (it is relatively easy to add new data streams). Furthermore, EndoScope employs a number of novel query-driven optimizations to limit the performance impact of instrumentation. It also simplifies the tool builders' task in the future by allowing them to easily and safely (without concern for corrupting program state) specify what parts of a program they would like to monitor.

Of course, a huge number of tools have been developed that make use of the collected runtime data beyond traditional profilers and debuggers. For example, the HotSpot / JIT compilers for Java observe runtime performance and dynamically recompile code as needed (see, for example, the Jalapeño JVM [17] and followup work [18], and the Sun HotSpot Compiler [8]). In addition, collecting runtime information about predicates in a program (*e.g.*, when a variable takes on a specific value) has been used in runtime bug detection [28], invariant detectors [23], race detectors [31], system behavior predictors [35], learning decision functions to classify system states [22], and latency analysis [27].

3. ARCHITECTURAL OVERVIEW

EndoScope is implemented as a two-part architecture consisting of a component that runs on the program execution site (*i.e.*, where the monitored program is being run, which we also call the server), and a component that runs on the machine from which the users issue queries (which we call the client). These two parts can reside on the same machine or on different machines to facilitate remote monitoring. Figure 1 illustrates the high-level components in the system.

The control flow in the EndoScope framework is as follows. The client (either a human user or another monitoring tool built on top of EndoScope) enters a query at the monitoring site. The query is parsed into a query plan by the query processor. The plan optimizer at the monitoring site then determines how the plan is to be implemented. The portion of the query plan to be executed at the monitoring site is given to the local stream processing engine, which then instantiates the stream operators as required by the query plan, while the rest of the query plan is sent to the plan optimizer running at the program execution site. The plan optimizer at the program execution site decides how the query plan should be implemented along with instrumentation points that need to be inserted into the running program (to observe the values of variables, entry and exit to functions, memory usage, and so on), and subsequently asks the stream engine and the code instrumentation module to carry out processing tasks (*i.e.*, instantiate stream operators and perform any instrumentation). While the query plan is executed, information collected from the running program (in the form of stream tuples) is processed by the stream engines running on both sites, and the query results are returned to the user.

Given this high-level overview, we now turn to describe the logical data model in EndoScope and explain how queries are formulated in the system.

4. LOGICAL DATA MODEL

In this section we describe the EndoScope logical data model along with the query language used to express monitoring queries. Note that only the logical data model is presented here; the actual implementation is discussed in Section 6.

4.1 Data Streams

EndoScope models data captured from monitored programs as data streams. Tuples in data streams record facts about the monitored program at a specific time. The system provides a number of *basic streams*. The basic streams are a set of predefined streams that users can utilize in posing monitoring queries. Users can also define new streams by combining the basic streams (see Section 4.4 for an example). Basic streams include:

- `function_start (thread_name, function_name, timestamp)`, and `function_end (thread_name, function_name, timestamp)`
  A tuple from this stream records the start (corr. end) time of a particular function being invoked by thread `thread_name`.

- `function_duration (thread_name, function_name, start_time, duration)`
  A tuple from this stream records the start and finish time of function invocations.

- `variable_value (var_name, var_value, timestamp)`
  A tuple from this stream represents the fact that variable `var_name` (can be either global or local) holds value `var_value` at a particular time.

- `cpu_usage (percent_busy, percent_idle, timestamp)`
  A tuple from this stream records CPU utilization at a particular time.

4.2 Stream Categories

We classify all data streams, including basic streams, in the EndoScope system into two categories based on whether or not the stream is enumerable.

4.2.1 Enumerability of Streams and Implications

We define an *enumerable* stream as one whose values are defined only at particular points in time. Such streams are the same as streams that conventional stream DBMSs provide. An example is the `function_start` stream, which is defined only when functions are invoked. Enumerable streams can be used directly in queries to the EndoScope system.

On the contrary, *non-enumerable* streams are those whose values are defined at any point in time.

In EndoScope, we do not allow non-enumerable streams to be directly used in queries because such streams have an infinite number of elements per unit time. For example, although it makes logical sense for a user to request to be notified whenever a new tuple arrives at the `function_start` stream (a discrete-valued, enumerable stream), it does not make sense for a user to be notified of every value of the `cpu_usage` stream (which is a continuously-valued non-enumerable stream), because the load on a CPU is defined at every instant in time. Instead, we require that before non-enumerable streams can be used in queries, special *quantifying* operations need to be applied, as discussed next.

4.2.2 Quantifying Operations

We currently support two different quantifying operations that allow non-enumerable streams to be used in EndoScope queries.

- Joining a non-enumerable stream with an enumerable stream. Examples of such a join includes joining the `cpu_usage` stream with the `function_start` stream on the timestamp of each arriving tuple in `function_start`.
- A sampling operation that periodically records the value of a stream at fixed time intervals. Logically this is equivalent to joining the stream with a non-enumerable one with periodically-arriving tuples.

4.3 Operations on Streams

In this section, we review the stream operations EndoScope provides. Rather than providing detailed semantics, we briefly discuss the behavior of each operation and illustrate its uses through a series of examples in Section 4.4.

4.3.1 Conventional Operations

EndoScope provides the standard select, project, and windowed aggregate operations on streams as in traditional DBMSs. Users can define additional enumerable or non-enumerable streams based on existing streams using the `create stream` statement.

4.3.2 Sampling and Joins

EndoScope includes several other types of stream operations. The `SAMPLE(stream name, interval)` operator takes a non-enumerable stream and samples it at regular intervals, which, as discussed in Section 4.2.2, allows non-enumerable streams to be used in EndoScope queries.

Our data model also supports a window-based join operation similar to that in CQL [16]. The operation involves two streams and performs a join on tuples based on the specified join condition and a time window. A special `stream1 FOLLOWED BY stream2` option can be given so that a tuple arriving at `stream1` at time t_A will only be joined with tuples from `stream2` that arrived later than t_A within the same time window. We provide this option for convenience, the same results can be achieved by adding a comparison predicate on time. This option is needed to express queries such as joining the `function_start` and `cpu_usage` streams, where the user might be interested in the amount of CPU utilization after a certain function has begun execution.

4.3.3 Triggers

The EndoScope data model also allows users to specify actions to be performed when certain conditions occur, much like database triggers. Such functionality can be used, for example, to generate an exception report to the end user when a servlet in web server takes longer than expected time to execute, or to invoke dynamic recompilation routines upon identification of a code hotspot.

4.4 Query Examples

In this section, we give a few query examples to illustrate the data model in EndoScope.

```
SELECT  *
FROM    SAMPLE(cpu_usage, 100ms)
```

This query quantifies the `cpu_usage` stream by sampling it every 100ms. This query represents a function that conventional profilers provide, although such tools do not usually allow the user to specify the sampling interval.

```
SELECT   fd.function_name, AVG(fd.duration),
         percent_busy
FROM     function_duration fd
         WINDOW JOIN
         SAMPLE(cpu_usage, 100ms) as cu
         WINDOW 1min
WHERE    fd.duration > 1s
         AND cu.percent_busy > 70%
         AND fd.function_name IN (f1, f2, f3,...)
GROUP BY fd.function_name
```

This query monitors a set of functions, and return the names of those whose average runtime exceeds one second when the CPU utilization is more than 70%. This query is useful to a user who is interested in identifying functions that ran for an excessive amount of time when the system is heavily loaded.

```
CREATE STREAM long_running_functions
AS ( SELECT  fd.function_name AS function_name,
             fd.duration AS duration
     FROM    function_duration fd
     WHERE   fd.duration > 1s
     AND     fd.function_name IN (f1, f2, f3,...) )

SELECT   lf.function_name, AVG(lf.duration)
FROM     long_running_functions lf,
         WINDOW JOIN
         SAMPLE(cpu_usage, 100ms) as cu
         WINDOW 1min
WHERE    cu.percent_busy > 70%
GROUP BY lf.function_name
```

This query produces the same result as the previous query but illustrates creating new data streams from existing ones.

```
CREATE TRIGGER sample_cpu
ON       function_duration f1
         WINDOW JOIN
```

```
        function_duration f2
        WINDOW 1min
        f1 FOLLOWED BY f2
WHEN    f1.duration > 5s AND f2.duration > 5s
        AND f1.function_name = 'foo'
        AND f2.function_name = 'bar'
BEGIN   SAMPLE(cpu_usage, 100ms)
END
```

This query samples CPU utilization every 100ms whenever function foo is invoked after bar is called, and the two calls are within one minute apart, where both functions end up taking longer than five seconds to execute. This query might be generated by a user who suspects that the two functions together are responsible for loading the CPU.

5. QUERY EVALUATION TECHNIQUES

Upon receiving a query, the EndoScope query processor parses and creates a query plan. The query optimizer then decides how the plan should be implemented. The optimizer's goal is to create a query plan that satisfies the user's monitoring request while introducing the minimal amount of overhead, in terms of system resources such as CPU cycles and memory, to the program being monitored. In EndoScope, query evaluation proceeds in four steps: query rewrite, plan placement, plan implementation, and finally stream instantiation and implementation of instrumentation points. In this section, we describe each of these steps, along with a number of optimization techniques that can be used.

5.1 Query Rewrite

The first step in query evaluation is to perform a number of query rewrite operations similar to those performed in standard DBMS, such as applying selections as early as possible, and eliminating common subexpressions. The details can be found in textbooks such as [24].

5.2 Plan Placement

Traditional program monitoring tools assume that the tool is running at the same site as the monitored program. Recently developed tools such as Google perf [6] and jip [9] enable remote monitoring of programs by allowing the monitoring client to connect to the monitored program via a network port created by the monitoring tool. In such architectures, an important question to consider is where to place the query execution code. Obviously the collected data needs to be generated at the site where the monitored program is executed, but it is unclear where the rest of the processing should be done.

We illustrate the placement tradeoff with an example: consider a query that involves monitoring the execution time of all functions and computes a complex aggregate function that generates tuples at a low rate from all the tuples received. There are two alternatives for query evaluation:

1. The program execution site (server) can send all tuples to the monitoring (client) site for processing. The server then does not need to execute the aggregate function at all, but the continuous sending of generated tuples might consume a substantial amount of network bandwidth at the server, which can be detrimental to network-bound programs such as web servers.

2. The server can perform the aggregation on site, and only send the final results to the client. This greatly reduces network bandwidth, but the execution of the complex aggregate predicate might take up a substantial number of CPU cycles at the server, which could also have negative impacts on the monitored program.

Given these two alternatives, we can compute the cost for each plan that captures the tradeoff between CPU utilization and network bandwidth. We define the cost of a query plan to be the amount of extra time introduced to the monitored program, and assume that we are not concerned about resource usage on the client. For the purposes of deciding where to place the query operators, the cost of instrumentation does not need to be considered (since they are the same in both plans). Hence, we define the cost of a plan as roughly the number of tuples produced by the monitors (N) times the server-side per-tuple processing time (T_{tup}) plus the number of tuples output by the server side processing (N') times the time to send a tuple over the network (T_{net}). That is:

$$N \times T_{tup} + N' \times T_{net}$$

For plan 1, the per-tuple processing time includes processing the aggregate function, whereas for plan 2, there is no per-tuple processing, but substantially more tuples to send over the network than in plan 1. Here, we use network transmission time as a proxy for network load: if the program execution site is low on network resources, then the time needed to send one tuple to client will increase, which follows from Little's Law [29], stating that the arrival rate of tuples has an inverse relationship with the per tuple processing time. As a result, plan 1 is favored. If instead the server is short on CPU cycles, then the time needed to process one tuple will increase, favoring plan 2. An alternative to formulating both costs in terms of total time would be to frame it as a multi-resource (network and CPU) optimization problem, or to search for a processing-optimal plan subject to constraints on network bandwidth.

In general, of course, there are many choices of where and how to "split" an arbitrary program between client and server. In addition, many applications, especially long-running ones, will tend to have different resource usage profiles at different times, which argues for an adaptive approach that periodically re-evaluates the resource profile of the monitored application and reassigns different parts of the query evaluation operations to different sites. These topics have been widely addressed in the literature [32, 19, 34]; the major difference in EndoScope is that the general optimization problem is focused on minimizing performance impact on the profiled application rather than on load balancing, and on choosing where to place code instrumentation (as we discuss in the next section.)

This is another example of where a database perspective on performance monitoring is beneficial: every other remote monitoring tool that we are aware of treats the monitoring job as a monolithic task, which cannot be split between the two sites.

5.3 Query Plan Implementation

After the query plan is divided between the program execution and remote monitoring sites, the next step is to order the operators and choose the physical operator instantiations, including how and when to insert monitoring points that acquire data from the running program. We illustrate this process through another example, in this case the second query from Section 4.4. This query consists of a join between the function_duration and cpu_usage streams on execution time and CPU utilization. Here we assume cpu_usage is discretized at frequency f_{usage}. There are at least three different strategies the join can be executed, where the only difference among the three is the ordering of the two operators that constitutes the join:

1. Monitor the runtime of the set of specified functions and sample CPU utilization from the start, checking predicates on both streams.

2. Start monitoring the specified functions and check the time taken for each invocation. If any of the monitored functions exceeds the threshold of one second in execution time, then immediately start sampling CPU utilization and check the results, and stop sampling when none of the specified functions exceeds the execution time threshold.

3. Start sampling the CPU, and each time utilization crosses the 70% boundary, immediately begin the monitoring of the specified functions, and stop monitoring if CPU utilization drops.

The first strategy, which is what a conventional streaming DBMS would do (because most streaming databases aren't acquisitional in nature), introduces the most overhead to the monitored program, as it incurs the costs of both monitoring all functions and CPU sampling. It is unclear, however, which of the other two strategies is preferable. For instance, if the specified set of functions is invoked rarely and each of them takes a short time to execute, then Strategy 2 appears optimal. On the other hand, if the set of functions is invoked very frequently with short execution times, then Strategy 3 appears preferable. Section 7 provides empirical data to support these observations.

Here, again, there is a tradeoff between the overhead introduced by monitoring all executed functions and the overhead introduced by CPU sampling, aggregated over the lifetime of the query. The overhead introduced by function monitoring with the CPU and function invocation stream can be factored into three components:

1. The set of functions F to be monitored

2. The frequencies with which each currently monitored function $i \in F$ is invoked (f_i) as well as the frequency with which the CPU is set to be sampled (f_{cpu}). Clearly, f_i depends on the actual rate at which function i is called (we call this f_i') but also on the fraction of time during which function i is monitored by EndoScope. Similarly, f_{cpu} depends on f_{usage}, but also on the fraction of time CPU utilization is monitored.

3. The cost of sampling the CPU (C_{cpu}) and of monitoring a single invocation of a function (C_{fun}). We discuss the measurement of these costs in Section 7.

We can now devise a simple cost model for a query that monitors a collection of functions and the CPU as follows:

$$\sum_{i \in F}(f_i \times C_{fun}) + f_{cpu} \times C_{cpu}$$

Clearly, if we add other streams (*e.g.*, that monitor the value of variables in the running program), there would be additional terms in this expression.

The three plans differ in the frequency with which each function and the CPU are sampled. For plan 1:

$$\begin{aligned} f_{cpu} = &\quad f_{usage} \\ f_i = &\quad f_i', \forall i \in F \end{aligned}$$

That is, we must sample the CPU at the rate specified in the query *and* capture each call of each function in F.

Suppose that the fraction of time during which any function in F runs for longer than one second over the program's lifetime is T_{long}. Then the cost of plan 2 is:

$$\begin{aligned} f_{cpu} = &\quad f_{usage} \times T_{long} \\ f_i = &\quad f_i', \forall i \in F, \end{aligned}$$

Here we only have to sample the CPU when a function exceeds the threshold.

Finally, suppose that the fraction of time during which the CPU utilization exceeds 70% is T_{busy}. Then, the cost of plan 3 is:

$$\begin{aligned} f_{cpu} = &\quad f_{usage} \\ f_i = &\quad f_i' \times T_{busy}, \forall i \in F \end{aligned}$$

Here, we always sample the CPU, but only have to sample functions when the CPU utilization is above 70%.

Comparing the three plans, we see that plans 2 and 3 are clearly preferred over plan 1 (since T_{long} and T_{busy} are both ≤ 1). However, to select between plans 2 and 3 we need to be able to accurately compute T_{long} and T_{busy}, as well as the f_i's, all of which change throughout the lifetime of the monitored program. This problem is quite tricky, as these values may vary significantly over the program lifetime and may change rapidly from one extreme to another. This again suggests that an adaptive approach to plan selection may be in order.

5.4 Program Instrumentation

The last stage in query plan execution is to instantiate the streams and instrumentation points that are needed. Because of EndoScope's acquisitional nature, all streams, including the basic streams, are instantiated only when a query arrives that make use of the stream. In this context stream instantiation refers to the creation of the necessary data structures such as buffers for the stream to begin receiving and forwarding of tuples, and registering the stream with the streams management engine. The same is true for instrumentation points, where no parts of the monitored program are instrumented or sampled prior to the arrival of queries that request such actions be performed.

We have identified a number of optimization possibilities in stream implementation. For example, streams can be implemented using different data structures based on their type (enumerable versus non-enumerable), and tuples and query operators can also be prioritized in terms of their processing order as in ACQP [30].

On the other hand, instrumentation points can also be implemented in different ways. To record the amount of time a function takes to execute, the most direct manner would be to insert callback invocations at the entry and exit points within each monitored functions. However, prior work in the software engineering community [20] has discussed a number of techniques that can be used to reduce the number of instrumentation points needed, including using the function call graph to infer the relationships among the monitored functions, which might be useful to further reduce overhead to the monitored program.

6. IMPLEMENTATION

We have implemented a prototype of the EndoScope framework to monitor Java programs. We chose the Java language mostly due to the dynamic class redefinition functionalities introduced in Java 1.5. We envision that our implementation can also be extended to native (*e.g.*, C or C++) programs in the future. As mentioned in Section 3, the EndoScope architecture is divided between the monitoring client and the site where the monitored program executes. The two parts can potentially run on two different JVMs, each with its own copy of the EndoScope class library. Each part consists of threads that carry out different functions by using a number of modules as described below. The current system is implemented in about 8K lines of code.

6.1 Stream Processing

We first describe the components related to stream processing that are common to both the client and program execution sites.

6.1.1 Tuple Representation

Each tuple in EndoScope is implemented as a set of fields, with each field representing an attribute. Currently the system supports fields of type integer, long, double, and character strings.

6.1.2 Stream Operators

Each of the stream operations discussed in Section 4 is implemented as a stream operator object. Each operator consists of an input buffer for incoming tuples, along with other structures as needed to carry out its operation (*e.g.*, predicates for selection, aggregation objects that compute the value of the aggregate, *etc*). Each operator is also aware of the downstream operators that are connected to it.

Each stream operator supports two functions: `addTuple` and `processTuple`. The `addTuple` function is called by tuple generators (such as system monitors) or other operators when they want to push new tuples into the input buffer of an operator. The `processTuple` function is called by the stream processing engine (discussed in Section 6.1.4) to process the tuples currently residing in the input buffer(s).

To implement triggering, operators can be annotated with trigger rules that are fired whenever a tuple arrives on one of an operators inputs. A trigger is implemented as a predicate-action pair, where the predicate is a condition to be checked on the arriving tuples, and the action represents the task(s) to be performed when the predicate evaluates to true. The current supported actions include adding or removing instrumentation points and resource monitoring. A trigger can be declared as a one-time trigger, *i.e.*, activated forever once the predicate evaluates to true, or continuous, *i.e.*, the associated action is performed each time when an incoming tuple satisfies the predicate and is undone when an incoming tuple does not satisfy the predicate.

The EndoScope system currently provides an implementation of the following query operators:

1. A stream scan operator that simply passes all tuples it receives in the input buffer and pushes them to the downstream operators connected to it.

   Each basic stream is implemented as a stream scan operator coupled with a tuple generator, such as a code instrumentation callback function, or a system resource monitor. The tuple generator invokes the `addTuple` function on the basic stream operator after a new tuple is created (say as a result of sampling). We have currently implemented the `function_start`, `function_end`, `function_duration`, `cpu_usage`, and `variable_value` (for local variables) basic streams.
2. Selection, projection, and aggregation operators implemented on top of the stream scan operator. For instance, a selection operator is a stream scan operator coupled with a predicate. When `processTuple` is called on a selection operator, each tuple in the buffer is checked against the predicate, and is pushed to operators downstream if the predicate evaluates to true. Projection and aggregation are implemented in a similar manner.
3. Two special operators, the network send and network receive operators, are implemented to transmit tuples between the program execution site and the client site. To reduce network overhead, tuples are sent in batches from the program execution site to the client site. The protocol works as follows: when the `processTuple` function is invoked on the network send operator, the operator sends a control message to the client via a network socket telling it the number of tuples in the batch to be sent, along with the query id and query operator on the client side for which the tuples are intended. Upon receiving the batch of tuples, the communication module on the client side puts the batch of tuples in the input buffer of the network receive operator. When the `processTuple` function is called on the network receive operator, it finds the query operator(s) that the tuples were intended for, and puts the tuples in their corresponding input buffer(s).
4. For debugging purposes, we have implemented output operators that print the received tuples to standard output or to a file.

6.1.3 Query Plans

A query plan is simply a collection of interconnected query operators. In order to support plan optimization, we also maintain in each query plan runtime statistics such as the number of tuples received by each operator, how long the current plan has been executing, and so on.

6.1.4 Stream Processing Engine

The stream processing engine is a thread that processes tuples in the system. Tuples enter the system initially as they are created by a monitor (such as callback functions or system resource samplers), and are pushed into the input buffer of one of the basic stream operators by invoking the `addTuple` function. When enough tuples have arrived at the basic stream operator, it informs the stream processing engine that it is ready to process tuples by placing itself in the stream processing engine's operator queue. The processing engine periodically dequeues an operator from the operator queue and invokes its `processTuple` function, which processes an incoming tuple and produces new tuples as needed. The operator then pushes the new tuples to operators that are connected to it downstream by invoking their `addTuple` function, and the same process repeats until the tuple is dropped by an operator or reaches the top-most operator in the query plan.

Obviously, operator fairness is an important issue here. In our earlier implementation, we had a tuple queue rather than an operator queue in the processing engine, where tuples instead of operators to be processed are enqueued. We found that implementation to be more fair in terms of the amount of processing time allocated to each operator over time, but the overall overhead introduced was higher, due a larger number of enqueue and dequeue invocations. A future direction would be to devise a mechanism that balances fairness and efficiency.

6.2 EndoScope Client

The EndoScope client provides the following functionalities that are built on top of the stream processing modules described above.

6.2.1 Query Parser

A front-end query parser processes incoming monitor queries into equivalent query plans made up of stream operators as explained in Section 6.1.3.

6.2.2 Plan Optimizer

Although we have not implemented the plan optimizer in the current system, in the full implementation the optimizer will take the query plan created by the query parser and determine the division of the query plan between the client and the program execution site, along with deciding how each of the stream operators in the client portion of the query plan should be implemented, based on the cost model to be formulated from the discussion in Section 5. For evaluation purposes, we hand-optimized each of the incoming query plans, and manually divided the plan up into a local and remote portion.

6.3 Program Execution Site Modules

EndoScope operates a number of modules within the same JVM as that of the monitored program. To enable monitoring by EndoScope, the user adds an `-agentlib` command line option to the JVM pointing to the EndoScope jar file, and gives all other options and parameters to the JVM as usual. No change to the JVM is needed.

6.3.1 Listener Thread

Once started, EndoScope runs a listener thread inside the executed program that listens on a specified port, waiting for incoming connections from clients. When a client connection is established, the client sends the query plan to be executed to the program execution site, which is then passed to the plan optimizer running there.

6.3.2 Plan Optimizer

Like the plan optimizer on the client side, we envision that a similar module would exist at the program execution site, except that at the program execution site the decisions to be made for each incoming query plan consist of choosing the implementation of stream operators along with instrumentation points. This process is manually performed in the current system.

6.3.3 Monitoring and Code Instrumentation

To sample system resource usage, EndoScope invokes a native library that reads the `/proc` filesystem for CPU and memory utilization at fixed time intervals as requested in the query plan.

For code instrumentation, EndoScope makes use of the Instrumentation classes that Java provides. This set of classes allows a program to be transformed by user-specified bytecode transformation routines, and also to intercept the normal class loading procedure for a new class so that a runtime-modified version of class with different bytecode can be loaded. In EndoScope, none of the methods or system resources are monitored initially. When a new query plan arrives that requests code instrumentation to be performed on specific classes or methods (monitoring of all classes or all methods within a class is also supported), the EndoScope instrumentation module first checks to see if the requested classes have been loaded. If so, it asks the JVM to transform the loaded classes. Otherwise, the module records the names of the classes to be instrumented and performs the transformation when the requested classes are loaded. Code instrumentation is done using the ASM [4] bytecode analyzer, with EndoScope providing the routines that perform the addition and removal of callback functions in the bytecode stream. We have also implemented a call graph analyzer that constructs a call graph of the newly loaded class. After code instrumentation and call graph generation, the instrumented classes are returned to the JVM to continue the loading process. We currently do not instrument classes that are provided by the Sun JVM or classes from the EndoScope package to avoid circularity.

7. EXPERIMENTS

The goal of this section is to study the performance of the EndoScope framework, including the effects of the acquisitional approach and how it scales as the amount of instrumentation grows, and the plausibility and effectiveness of our optimizations. To offer a point of comparison with other tools, we implemented a simple Java program profiler on top of EndoScope and compared its performance with other Java profilers, since we could not find Java tools that perform some of the more sophisticated monitoring tasks described in the introduction.

7.1 General Experimental Setup

The experiments were run on a desktop machine with a dual core 3.2 GHz Pentium D processor and 4 GB of main memory. We used JDK 1.6.0 release 1 and created a JVM with a 2 GB heap for each run under Linux (RedHat Fedora Core 7).

In the experiments we chose the following publicly available Java program profilers for comparison:

- visualvm [14] Beta version. Visualvm is an instrumentation-based profiler with a graphical interface. The tool provides monitoring of heap size and status of threads while the program executes. We attempted to use the CPU and memory profiling functionalities in our experiments, but doing so crashes the tool. Thus we report the overhead from using only heap size and thread status monitoring.

- jip [9] version 1.1.1. jip is a sampling-based profiler similar to hprof.

- jrat [10] version 1 beta 1. jrat is an instrumentation-based profiler that monitors all classes and memory while a program executes, and provides a graphical tool to view the profiled results afterwards.

- hprof [7] is sampling-based profiler written in C. It is Sun's reference implementation that illustrates the use of the Java JVM Tool Interface. The tool provides methods and memory profiling, and is part of the JDK release.

We then compared the performance of the suite of profilers on the following reference programs:

- SimpleApp, a test program that comes with Apache Derby [1] version 10.3.2.1. Derby is a fully featured Java SQL database originally developed as CloudScape and then released as open source. SimpleApp opens a JDBC connection, creates a database and table, and runs a few insert, select, and update statements, followed by a transaction commit.

- A implementation of a TPC-C-like benchmark in Java. The following parameters were used to set up TPC-C data:

  - Total number of warehouses: 3
  - Number of districts per warehouse: 10
  - Number of customers per district: 3000
  - Number of orders per district: 3000
  - Maximum number of items per transaction: 15

  We performed two set of experiments on this data. The first consisted of a single thread executing 250 transactions, and the second 20 threads each executing 10 transactions. The transactions were created at random, and the Apache Derby database was used.

- The petstore reference application [2] using the Apache Derby database. The application was hosted on Apache Tomcat version 6.0.14 [3]. In each run, we created a workload of 200 random http requests to the petstore from a remote machine. To simulate customers browsing the store, each request asks for a valid webpage that is hosted on the application.

Note that in each scenario we are able to profile not only the user application but also the database and web server that hosted the application, as they are all implemented in Java.

7.2 Runtime Overhead Experiments

In the first set of experiments, we measured the overhead of EndoScope by progressively varying the proportion of methods monitored. First, we monitored all methods and computed the average execution time of each across a number of target program runs. We then created a profile by ranking the methods according to the number of times each of them was invoked over the program's lifetime.

We then performed a second set of runs where we varied the proportion of methods monitored, starting with monitoring the 10% of methods that were called the fewest number of times according to our previously created profile, then monitoring the 20% of all methods, and so on, until we reach 100%. This methodology was used as a means to gradually increase the total number of instrumentation points we introduce into the system. We measured the total program runtime under each run, and compared it with the base (*i.e.*, no profiling), along with profiling using other tools, which do not support varying the number of monitored methods.

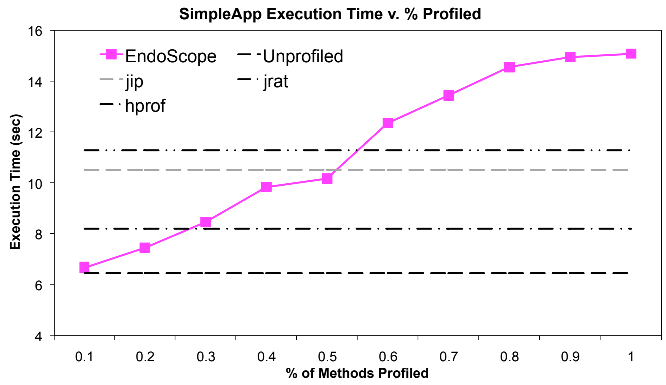

Figure 2: Time to execute SimpleApp using different profilers

Figures 2 and 3 show the execution time of the SimpleApp and TPC-C applications as a function of the percentage of methods profiled (labeled as "Method Mon." in Figure 3). Also shown are the corresponding execution times when other profilers are used to profile all methods. The numbers presented are the average over 3 runs. Note that since SimpleApp is a relatively simple program, thus the execution time obtained using different profilers do not differ significantly from each other (with a difference of only 9 seconds between the fastest and slowest). We obtained similar CPU overheads when running on the petstore applications, with CPU overheads generally lower than SimpleApp and slightly higher than TPC-C. In all cases, the number of tuples generated scales approximately linearly with the percentage of program monitored.

We also implemented method-local variable monitoring (*i.e.*, data watchpoints) by compiling the source with debug symbols, and monitoring all Java bytecode store instructions at runtime, checking each store instruction to see if it writes to a monitored local variable (debug information is used to extract variable names.) As in the method monitoring experiment, we varied the fraction of local variables profiled and measured the overhead. The results are also shown in Figure 3 ("Var. Mon."), demonstrating a similar trend as the method monitoring experiment. We were not able to compare with other profilers as none of them provide a similar feature.

Figures 4 shows the number of tuples generated by TPC-C in the method monitoring and local variable monitoring experiments; note that both experiments scale nearly linearly with the fraction of the program that is profiled. Other applications show a similar trend.

The graphs show that our profiler is comparable to other profilers in terms of maximum overhead introduced, and that both the overhead and number of tuples created scale linearly when the number of instrumentation points increases. In fact, when the percentage of monitored methods is small, we outperform other profilers by a significant amount except when running trivial programs like SimpleApp. This illustrates the power of our acquisitional approach: by only instrumenting the code that is needed, it is possible to satisfy

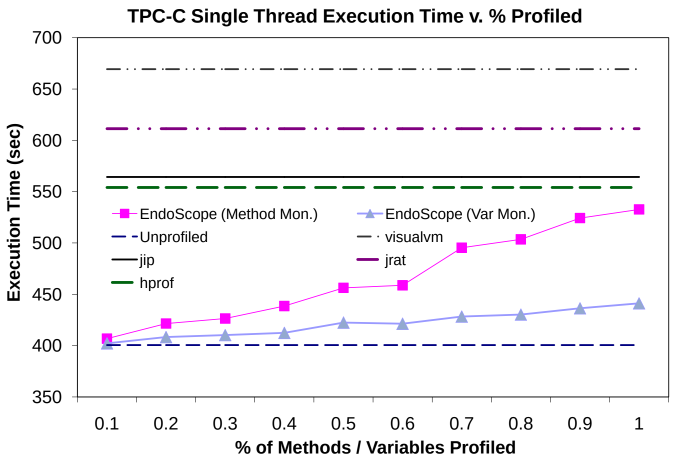

Figure 3: Time to execute TPC-C single threaded run using different profilers

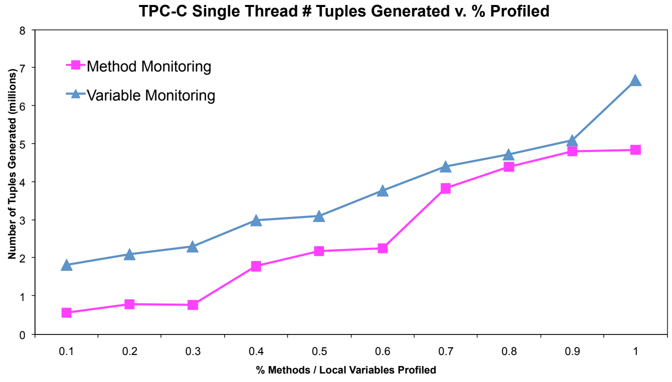

Figure 4: Number of tuples generated from TPC-C single threaded run

the user's request for program monitoring while introducing little overhead on the monitored program.

7.3 Join Operator Ordering Experiments

Next, we investigated the effect of join operator reordering by implementing the three query plans from Section 5.3. The plans are implemented using standard query operators supplied by EndoScope, and trigger actions are included which start CPU sampling when the execution time of any function exceeds the specified threshold (for plan 2), and starts function monitoring when CPU utilization crosses the specified boundary (for plan 3). We vary the percentage of functions monitored in the same manner as in the experiment described in Section 7.2. We used the TPC-C single-threaded application as base (the petstore application shows similar trends), and Figure 5 shows our results.

As a side note, from our experiments we found that the time needed to execute the callback functions from an instrumentation point is around 0.1-0.2ms, while the time needed to take a CPU sample from the `/proc` filesystem together with the post-processing time is around 0.3-0.5ms. Note that the graph demonstrates the tradeoff between the cost of function monitoring and CPU sampling. It is clear that when the set of functions to be monitored is large, then plan 3 is better. It is not as clear, however, that plan 2 is the winner when the set of functions to be monitored is small. It turns out that is due to the fact that in the cases where the proportion of methods that are monitored are small in our experiment, the methods that we chose to monitor did not actually exceed the pre-specified threshold. Thus CPU sampling was never triggered. Figure 6 show the results when we ran the experiments again, but ensured that the first methods to be profiled were infrequently invoked (making their profiling overhead low) but had execution times that were high. In this case, plan 2 is indeed preferable up to about 20% of the program being instrumented.

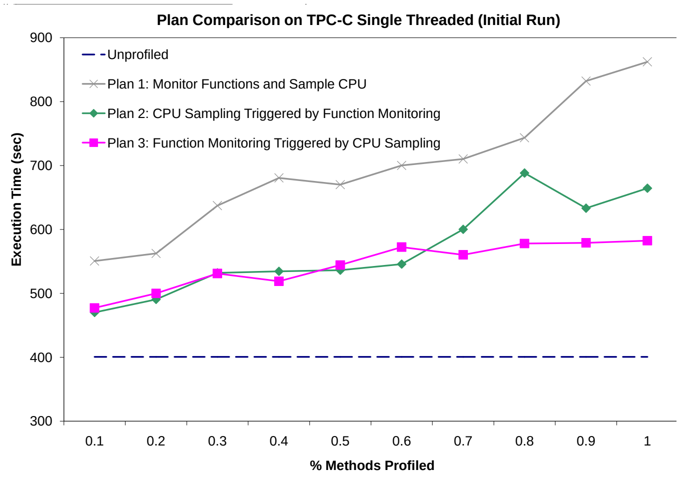

Figure 5: Execution time of three different TPC-C instrumentation plans.

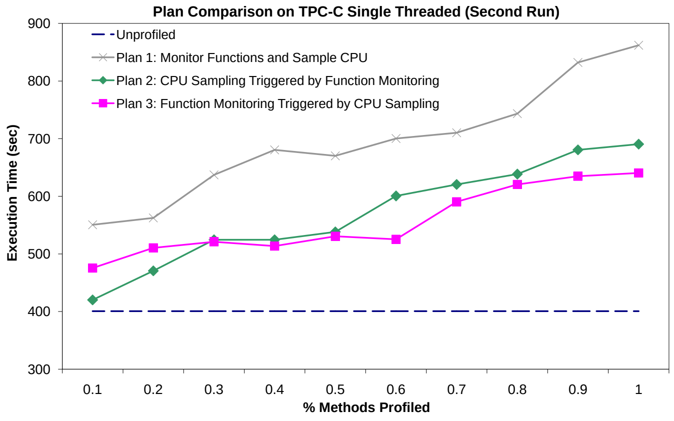

Figure 6: Execution time of three different TPC-C instrumentation plans, profiling long running but rarely called functions first.

This experiment illustrates the difficulty in constructing a good plan cost estimator. The cost estimator needs to be aware of not only the overhead in CPU sampling and function monitoring, but also how often the triggers are executed, as discussed in Section 5.3.

7.4 Plan Placement Experiments

We next performed two experiments to validate the query optimization approaches proposed in Sections 5.2 and 5.3.

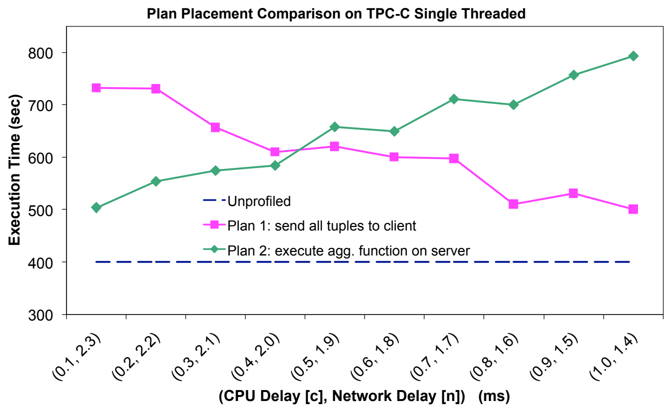

Figure 7: Query plan placement comparison for TPC-C.

For the plan placement experiment, we monitored all method calls. We then created two special query operators to study the effects of CPU and network load. The first special operator is created to mimic a complex aggregate function. When its `processTuple` method is invoked, the operator scans all tuples in the incoming buffer, then pauses for c milliseconds on each tuple to simulate computation. After that, 10% of the incoming tuples are forwarded to downstream operators. We also created a special network send operator. For each batch of tuples in its incoming buffer, the operator delays for n milliseconds prior to sending the batch out. We intend n to model the queuing time of tuples in a real system, which is affected by buffering and congestion delays.

We simulated CPU load by varying c, with the idea that if the program execution site is running CPU intensive jobs, the amount of time needed for the complex aggregate function to process each tuple, c, should increase. Otherwise it should be fairly constant. The same idea applies to n as well.

We study the strategies for plan placement by implementing the two query plans as proposed in Section 5.2. In the first plan, the monitoring site forwards all tuples from the function call monitor to the special network send operator which, after delaying for n seconds, sends each batch of tuples to the monitoring client that hosts the complex aggregate function operator (the runtime of the aggregate function operator is not of our concern in this case since it runs on the client site). In the second plan, the aggregate function is executed at the monitoring site with delay c on each tuple, after which outputs of the aggregate operator are sent to the client using the network send operator with $n = 0$, *i.e.*, we assume that the network is under normal conditions and that sending a small amount of aggregate data does not introduce any extra overhead on top of normal network delay.

We ran the TPC-C single threaded application and measured the amount of time needed for program completion in the two plans. The results are shown in Figure 7. Note that both c and n vary with the X axis in this figure; c increases from left to right while n decreases. We performed the same experiment on the petstore application with similar results.

The results confirm our intuition about plan placement. When the program execution site is CPU loaded, pushing the complex aggregate function to the client is optimal. On the other hand, if the program execution site is running network intensive jobs, it is more preferable to reduce the need for network bandwidth by executing the aggregate function on site. The goal of the optimizer, then, is to be aware of the load on the CPU and network, and be able to switch between the two plans in an adaptive manner.

8. CONCLUSIONS AND FUTURE WORK

In this paper we introduced EndoScope, a flexible, low overhead, query-driven and acquisitional software monitoring framework. We demonstrated the framework by using it to implement a simple Java profiler, and showed that its performance in real world software monitoring is substantially lower in overhead than existing tools. It is also amenable to a number of database style optimizations and more flexible than many existing profiling tools. Below we highlight a few problems for future research.

Data Stream Implementations: Section 4.2.1 discussed a classification scheme for data streams. An interesting question is to further investigate properties of each type of stream, similar to the analysis regarding which types of streams can be used in queries. A related question is to investigate data structures for implementing the different stream types, and correlate that with query plan optimization.

Cost Model: The discussion on query plan optimizations showed the need for new cost models in searching for optimal query plans, and demonstrated that query plans will need to be adaptive over the lifetime of the query.

Approximate Query Answers: Our current system focuses on getting exact answers for queries, but there are situations where approximate answers are already sufficient (in sampling based profilers, for example). An interesting direction would be to consider query answer approximation as an optimization strategy, and design additional language features that enable formulating approximate queries.

An API for System Monitoring: Many monitoring tools allow users to interactively define new probes and monitors (*e.g.*, visualvm [14]) or pass in command line options to instruct the monitor what data to collect (*e.g.*, gprof [26]). On the other hand, users typically interact with data stream processors by issuing SQL like queries on a console or via a “driver” like API such as JDBC. However, since the goal of EndoScope is to provide an infrastructure where tools that make use of runtime information can be built, it would be more beneficial to the tool developer if the infrastructure exposes a library-like API in the spirit of the D Language [21] or WaveScript [25].

Monitoring Distributed Applications: It would be useful to extend EndoScope’s two-part architecture by having the ability to monitor programs that run on distributed systems (such as monitoring Internet mail servers or distributed databases) and allow multiple clients to connect to the monitoring system simultaneously. Correlating events that occur on different machines and presenting them in a coherent fashion to the end user is an interesting challenge.

9. REFERENCES

[1] Apache derby. URL `http://db.apache.org/derby`.

[2] Apache derby petstore demo. URL `http://db.apache.org/derby/integrate/DerbyTomcat5512JPetStor.html`.

[3] Apache tomcat. URL `http://tomcat.apache.org`.

[4] Asm java bytecode manipulation framework. URL `http://asm.objectweb.org`.

[5] gdb. URL `http://sourceware.org/gdb`.

[6] Google performance tools. URL `http://goog-perftools.sourceforge.net`.

[7] hprof. URL `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`.

[8] The java hotspot virtual machine, v.1.4.1. URL `http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html`.

[9] Jip - the java interactive profiler. URL `http://jiprof.sourceforge.net`.

[10] Jrat the java runtime analysis toolkit. URL `http://jrat.sourceforge.net`.

[11] jswat graphical java debugger. URL `http://www.bluemarsh.com/java/jswat`.

[12] oprofile. URL `http://oprofile.sourceforge.net`.

[13] Shark 4. URL `http://developer.apple.com/tools/sharkoptimize.html`.

[14] Visualvm java profiler. URL `https://visualvm.dev.java.net`.

[15] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Hezinger, S.-T. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems*, 15(4):357 – 390, November 1997.

[16] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In *VLDB Journal*. 2005.

[17] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proc. of OOPSLA*. October 2000.

[18] M. Arnold, M. Hind, and B. Ryder. Online Feedback-Directed Optimization of Java. In *Proc. of OOPSLA*. November 2002.

[19] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proc. USENIX NSDI*. March 2004.

[20] T. Ball and J. Larus. Optimally Profiling and Tracing Programs. In *19th ACM Symposium on Principles of Programming Languages*, pp. 59 – 70. January 1992.

[21] B. Cantrill, S. Michael, and A. Leventhal. Dynamic Instrumentation of Production Systems. In *Proc. of USENIX*. 2004.

[22] I. Cohen, J. Chase, and T. Kelly. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proc. of OSDI*. 2004.

[23] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69:35 – 45, 2007.

[24] H. Garcia-Molina, J. Widom, and J. Ullman. *Database Systems, the Complete Book*. Prentice Hall, 2001.

[25] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. The case for a Signal-Oriented Data Stream Management System. In *Proc. of CIDR*. January 2007.

[26] S. Graham, P. Kessler, and M. McKusick. gprof: a Call Graph Execution Profiler. In *Proc. of the ACM SIGPLAN Symposium on Compiler Construction*, pp. 120 – 126. 1982.

[27] N. Joukov, A. Traeger, R. Iyer, C. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proc. of OSDI*. November 2006.

[28] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable Statistical Bug Isolation. In *Proc. of PLDI*. June 2005.

[29] J. Little. A Proof of the Queueing Formula L=λW. *Operations Research*, 9:383–387, 1961.

[30] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *ACM SIGMOD*. 2003.

[31] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[32] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proc. ICDE*. 2003.

[33] A. Wilcox. JIP – The Java Interactive Profiler, whitepaper. 2005. URL `http://jiprof.sourceforge.net`.

[34] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. In *Proc. ICDE*. 2005.

[35] W. Xu, Bodík, and D. Patterson. A Flexible Architecture for Statistical Learning and Data Mining from System Log Streams. In *Proc. of ICDM*. November 2004.