Enabling Performance Exploration and Analysis for Multi-parametric Systems*

Younghwan Go†
KAIST
yhwan@ndsl.kaist.edu

Juan A. Colmenares
Samsung Research America
juan.col@samsung.com

ABSTRACT

Tuning third-party systems is time-consuming and sometimes challenging, particularly when targeting multiple embedded platforms. Unfortunately, system integrators, application developers, and other users of third-party systems lack proper tools for conducting systematic performance analysis on those systems, and have no easy way to reproduce the systems' advertised performance and identify configurations that yield excellent, fair, or poor behavior. To fill this void we introduce SPEX, a framework aimed at making it easier to characterize third-party systems' performance in relation to configuration parameters. SPEX enables automatic performance exploration for systems with no need to access their source code. It offers the flexibility to define pluggable *policies* that steer the exploration process by varying configuration parameters of the observed system. Our results show that SPEX adds little overhead to the monitored system, and suggest that it can be effective in providing useful information to third-party system users.

CCS Concepts

•Software and its engineering → Software performance;

Keywords

Performance characterization; performance reproducibility

1. INTRODUCTION

System integrators and application developers usually include third-party software systems (*e.g.,* libraries, middleware, and databases) in their products. These individuals along with system maintainers, all referred to in this paper as *third-party system users*, are often responsible for configuring those systems to meet certain performance requirements. But, tuning third-party systems for performance is time-consuming and sometimes even challenging, particularly when targeting multiple embedded platforms. When charged with this task, these users usually try numerous configurations until acceptable performance is reached. Such non-systematic approach often results in configurations that yield far from optimal performance. Moreover, a good configuration for one platform may not be so for another, forcing users to redo the process on a per-platform basis.

In the case of an underperforming system, diagnosing the problems may become a difficult task for users as they seldom have deep knowledge about the system's internals. By contrast, the developers of the system may easily come up with plausible conjectures and directly attack the problems in the code (*e.g.,* bottlenecks).

A popular approach to troubleshooting a "black-box" system is to *profile* its performance by collecting metrics such as execution times of functions, hotspots in the code, CPU load, and memory usage [1, 12, 13, 2]. Such information, however, is of almost no help to users with little knowledge of the system's implementation. Instead, users of a third-party system need to be able to easily *characterize* the system's performance as a function of configuration parameters, as well as *reproduce* its advertised performance. For that, users require information more specific to the system in hand [19, 5, 4], including coarse- and fine-grained performance metrics (*e.g.,* throughput and latency as well as hardware event counts), *and* their relation with configuration parameters. Due to their overhead, *logging* and *monitoring* tools are often setup to only provide coarse-grained metrics, and they are not designed to delineate a system's configuration space against performance and help users understand how configuration parameters affect those metrics.

Users are then underserved for conducting performance analysis on third-party systems, and have no easy way to determine the reasons when those systems do not perform as their developers advertise. Hence, we introduce SPEX, a framework for helping users characterize the performance of unfamiliar systems and understand its relation with configuration parameters. SPEX is designed to be easy-to-use, flexible, and applicable to various platforms, and to add low overhead to the observed system. We adopt an approach in which *developers*, who have first-hand knowledge of the system, instrument the code (as they usually do) and offer users easy access to the system's monitoring infrastructure. At the other end, *users* or an automatic *performance exploration process* decides which metrics to observe and, without requiring access to the code, enables the instrumentation in the system that is only relevant to the metrics under evaluation. SPEX also controls the performance exploration of the monitored system, following a *policy* that steers the process by varying the system's configuration parameters. It allows users and developers to define different exploration policies as these policies are specific to both the target system and the performance metrics of interest.

In this paper, we report our experience in designing the SPEX framework (§3) to support the aforementioned approach. This is our initial step toward effective and efficient automatic performance exploration of third-party systems. Besides SPEX's low overhead and good thread scalability (§5.1), we also illustrate its use in characterizing the per-

*EWiLi'16, October 6th, 2016, Pittsburgh, USA. Copyright retained by the authors.

†Y. Go was an intern at Samsung Research America.

formance of SQLite,[1] a popular embedded SQL database, with a simple exploration policy (§5.2). Our results suggest that SPEX can be effective in providing useful information to third-party system users with little effort.

2. APPROACH

The salient aspects of our approach to designing a framework for performance exploration and analysis are as follows.

Ease of profiling enabled by developers: Detailed performance characterization of a real-life system usually requires human involvement and good understanding of its architecture and source code – knowledge firstly held by the system's developers. Thus, our approach exploits a natural division of responsibilities where: 1) *developers*, leveraging their knowledge, instrument the system's code to make its key performance metrics observable to users, and 2) *users* decide which of the available metrics to observe and, without accessing the source code, enable the relevant instrumentation in the system to evaluate its performance.

We believe this approach imposes a small burden on developers. First, they usually instrument their code as it is a pervasive practice in software development. Second, SPEX provides a set of probes (see Table 1) that greatly simplify the task. Nevertheless, it is ultimately in the developers' best interest that users are able to easily reproduce the system's advertised performance.

Low profiling overhead: Performance characterization of a system is meaningless if the overhead of measuring and data collection masks the monitored performance. SPEX minimizes such overhead by adopting a widely used trace-based approach [9, 3] in which instrumentation probes *make no calculations* of performance metrics and just create *compact binary trace records* with the values needed to compute those metrics outside the monitored system. Besides applying several optimizations discussed in §3.1, SPEX allows users to adjust the instrumentation overhead by collecting traces only from the probes they enable.

Flexible performance exploration: SPEX separates the *mechanisms* for collecting performance metrics from the *policies* that guide system performance exploration. By doing so, it offers developers and users enough flexibility to define different policies. This feature is important because policies are not only specific to the target system, but also to the system's performance characteristics developers and users are interested in (*e.g.,* response latency, write throughput, thread scalability). Thus, developers can release their systems along with exploration policies for users to easily evaluate the performance, very much like unit tests as well as functionality and performance tests are shipped with systems today. Users may also implement their own exploration policies to understand the systems' behavior under conditions not anticipated by the developers.

Furthermore, such flexibility enables innovation in devising policies that efficiently and effectively explore system performance in *large configuration spaces*. This is significant because systems often have numerous configuration parameters that produce a very large number of possible configurations (*e.g.,* SQLite has ∼80 parameters and ∼150 compile flags, with some overlap between them). Note that we leave the design of exploration policies for large configuration spaces as a future work, and we focus on the SPEX framework, which facilitates implementing such policies.

[1]https://www.sqlite.org

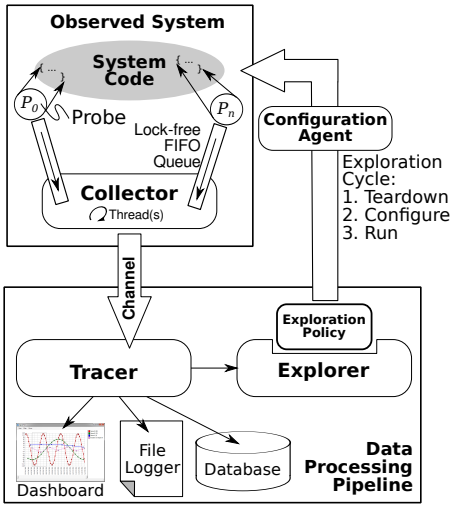


Figure 1: General architecture of SPEX.

3. THE SPEX FRAMEWORK

Figure 1 shows SPEX's architecture, which includes the Collector, Tracer, Explorer, and Configuration Agent components. The observed system is instrumented with *probes* to collect system-specific, OS-level, and architectural metrics. Probes are introduced into the system mainly by its developers, but also possibly by users with access to the source code. The observed system goes through multiple trial runs with different configurations (*teardown-configure-run cycles*) until the performance exploration completes, and during this process the system is configured automatically through its Configuration Agent.

The Collector and Tracer, along with the probes enabled in the observed system, form SPEX's *data collection subsystem*. This subsystem gathers performance traces and passes them to the Explorer for analysis. The Explorer orchestrates the performance exploration process following a given *policy*, which is a pluggable module that determines when to stop the current run and what configuration to use in the next run, as well as when the entire exploration should terminate. Upon receiving a trace record, the Explorer asks the policy how to proceed.

As shown in Figure 1, the *data processing pipeline*, encompassing the Tracer and Explorer, and the observed system run as separate processes either on the same or different machines, and communicate via inter-process communication (IPC) or remote procedure call (RPC) mechanisms. The data processing pipeline is configurable and may contain additional components that store, display, and perform statistical analyses on the data from the Tracer. SPEX includes a database, a file logger, and a dashboard, while few other auxiliary components are currently being developed.

Probe API: Probes produce trace records with the format shown in Listing 1, and insert them into lock-free queues, described in §3.1. Each record is a compact binary log entry, and this simple format is able to accommodate the information necessary to monitor and analyze different system performance metrics.

SPEX currently provides six predefined types of enclosing

Predefined Probes	Description
`CNT_PROBE_BEGIN(id,sc)`	Number of times the enclosed code has executed.
`LAT_PROBE_BEGIN(id,sc)`	Number of CPU cycles the enclosed code takes to execute (for latency calculation).
`TPT_PROBE_BEGIN(id,sc)`	Number of times the enclosed code has executed and the CPU cycles it has taken (for throughput calculation).
`LLC_PROBE_BEGIN(id,sc)`	Number of last-level cache references and misses.
`FLT_PROBE_BEGIN(id,sc)`	Number of memory page faults.
`CTXSW_PROBE_BEGIN(id,sc)`	Number of context switches.

Flexible Probes	Description
`PERFCNTR_PROBE_BEGIN(id,sc,pcid)`	Value of a performance counter setup via `SET_PERFCNTR(pcid,event,mask)`.
`SNAPSHOT_PROBE(id,sc,`$v_0,\ldots,v_{N-1}$`)`	Developer-defined values. $N = 6$ by default.

Table 1: SPEX Probe API. Note that the `X_PROBE_END(id)` statements are not shown in the table.

Listing 1: Trace record format.

```
#define N (6) // Default maximum number of fields.
struct TraceRecord {
  // Probe's idendifier (value set by developer).
  uint16_t probe_id;
  // Probe type (bitmask).
  uint32_t probe_type;
  // Processor that captured the record.
  uint16_t cpu_id;
  // Thread that captured the record.
  uint32_t thread_id;
  // Timestamp counter's value
  // when storing the record.
  uint64_t timestamp;
  // Number of fields in the record.
  // Valid values: [0,(N-1)].
  uint8_t field_count;
  // Array of value fields.
  uint64_t fields[N];
};
```

probes listed in Table 1, and more can be easily added. Each type of enclosing probes assigns values to specific fields of the trace records. The probes are given *unique identifiers* (`id`) in order to be distinguishable to the users. Moreover, developers can adjust the system's instrumentation overhead by setting each probe's *subsampling counter* (`sc`). This counter indicates the number of times the probe needs to execute in order to produce a trace record.

For flexibility, SPEX also offers the `PERFCNTR_PROBE`, which is an enclosing probe type that developers can use to read hardware performance counters of their choice. A developer can setup a performance counter by specifing a unique identifier (`pcid`) along with the `event` and `mask` values (*e.g.,* `event=0x2E` and `mask=0x41` for LLC misses on Intel processors). Lastly, SPEX offers the single-statement `SNAPSHOT_PROBE(...)` that allows developers to store values of their choice in the trace records.

Exploration setup: To run a performance exploration, the user needs to set several Explorer's options. The user must indicate the XML file with the configuration parameters needed by the observed system, and can also specify non-default values for generic exploration parameters, such as `trials_per_config`. In addition, the user may tune the parameters of the exploration policy in use (*e.g.,* minimum and maximum number of threads enabled in the system).

3.1 Performance Data Collection

The Collector gathers trace records from the lock-free queues and passes them to the Tracer. At initialization, SPEX creates communication channels between the Collector and Tracer: one or more *data channels* for trace records, and a *control channel* for commands. The number of threads the Collector runs is configurable, and each thread is assigned a disjoint set of queues to read records from. The Collector's threads continuously scan their queues in round-robin, consume upto a configurable number of records from each queue, and send the record batches to the Tracer via the data channels. By enforcing a maximum record count a collector thread can fetch from each queue, no queue can monopolize data collection. The Collector can also simulate workloads colocated with the observed system by creating additional load on CPUs and storage devices, and allocating extra memory.

The Tracer comprises one or more threads in charge of processing records carried in messages from the Collector. When a message arrives, a Tracer processing thread deserializes it, calculates the performance metrics, and passes the metric values or trace records to the Explorer or auxiliary downstream components.

Lock-free queues: Threads running in the observed system call the (enabled) probes to produce and insert trace records into in-memory queues, which are later consumed by the Collector's threads (see Figure 1). Each queue allows multiple producer and consumer threads to exchange records in a lock-free manner through a fixed-size circular buffer. When a queue is full, we choose to sacrifice the oldest records in order to: 1) prevent probes from blocking the system's execution, and 2) simplify pausing and resuming data collection.

Reducing collection cost: Our probe implementation incorporates several techniques to minimize the performance cost. First, trace records are compact binary log entries, and arithmetic operations on them are performed outside the observed system. Second, probes insert the records into in-memory queues with a fast, non-blocking operation. Third, by default SPEX creates a queue per CPU and per probe type. Thus, a thread, calling a probe, can look up the probe's backing queue in constant time ($O(1)$) because the queues are indexed by CPU ID and probe type. The net result is that probes are able to produce trace records at low cost and with good thread scalability (see §5.1).

SPEX dedicates a fixed-size memory space for data collection in the observed system. But it gives users the ability to adjust such memory space as trace record queues and communication channels are of configurable sizes.

3.2 Performance Exploration

Exploring the performance of a target system involves capturing its behavior under different configurations. In SPEX, an *exploration policy* is the key module responsible for guiding such process. The policy decides: 1) the configuration the observed system should use in the next run, 2) the probes to collect traces from, and 3) when to stop the current run

Pseudocode 1 EXPLORER's main loop.

```
    ▷ pol: The performance exploration policy.
 1: while (pol.has_next_config() = true) do
 2:     conf ← pol.next_config()
 3:     duration ← pol.max_duration(conf)
 4:     probe_set ← pol.probes_to_enable(conf)
 5:     configure_observed_system(conf)
 6:     start_observed_system()
 7:     setup_tracer(EXPLORATION_MODE, probe_set, ...)
 8:     start_tracer_collection()
 9:     Wait until (duration expires or TRACER finishes collection)
10:     stop_tracer_collection()
11:     teardown_tracer()
12:     stop_observed_system()
13: pol.process_results()
```

Pseudocode 2 TRACER's collection loop.

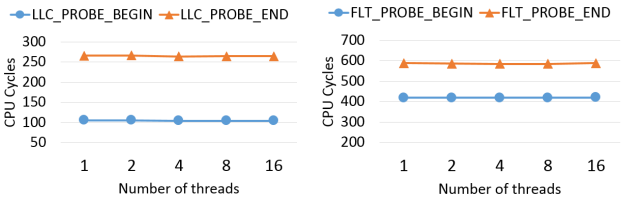
```
    ▷ channel: A data channel between COLLECTOR and TRACER.
1: while (exit = false) do
2:     record ← channel.get_record()
       ...
3:     explorer.process_record(record)
4:     if (explorer.stop_run() = true) then
5:         break
       ...
```

and the entire exploration. Exploration policies adhere to a *common interface*, which makes them easily interchangeable and allows the EXPLORER to use a single and simple logic to explore various performance characteristics of the target system in different ways. Currently, the policies are implemented as C++ classes and compiled into dynamically linked libraries. Using a policy is thus a simple matter of indicating its library's location to the EXPLORER.

During performance exploration, the observed system goes through multiple runs with different configurations. These teardown-configure-run cycles are carried out by the EXPLORER as shown in Pseudocode 1, with error handling omitted for clarity. The cycles continue while the policy offers configurations to explore (line 1). Once the EXPLORER obtains a configuration (lines 2-4), it tells the CONFIGURATION AGENT to configure the system via a control message with the configuration in XML, and starts the system. In line 7, the TRACER is set up to run in *exploration mode*, and starts collecting trace records from enabled probes until duration expires or the TRACER finishes data collection.

While TRACER's collection proceeds (Pseudocode 2), each of its threads gets a record from a data channel and asks the EXPLORER to process the record (line 3) and whether or not it should stop the run (line 4). The EXPLORER delegates the execution of the functions `process_record(...)` and `stop_run()` to the policy . Then, if the policy decides to stop the run, the TRACER terminates and sets a flag indicating that data collection has finished.

Central to performing teardown-configure-run cycles is the ability of the EXPLORER and the CONFIGURATION AGENT, which is system-specific, to configure the observed system. Configurations often contain diverse and detailed information, such as number of threads, file paths, names of remote nodes, and other input parameters. We have adopted an XML format for configurations. Developers as well as users can create XML configuration files that match the intent of their performance exploration policies (*e.g.,* yield peak performance, or analyze system's sensitivity to the allocation of certain resources). Policies use those XML files as templates to generate configurations in function `next_config()`, which



(a) LLC refs and misses. (b) Memory page faults.

Figure 2: CPU cycles consumed by probes.

are interpreted and applied onto the observed system by the CONFIGURATION AGENT.

4. IMPLEMENTATION

SPEX is written in ∼6K lines of C++11 code. The TRACER and EXPLORER run atop a data processing pipeline, comprising actors and components. *Actors* are memory-protected processes that exchange messages. The messages are typed with Google Protobuf[2] and exchanged over local shared memory channels or the network via ZeroMQ.[3] Actors encapsulate *components*, which are in-process loadable modules that provide typed interfaces and receptacles that can be dynamically inspected and bound (much like Microsoft's COM architecture [15]). In addition, CONFIGURATION AGENTS use TinyXML[4] to parse XML configurations for the observed system.

5. EVALUATION

In this section, we show that SPEX's instrumentation probes incur low overhead and exhibit good thread scalability. We also exemplify how SPEX can be used to explore the performance of a third-party system with SQLite3 and a simple policy. Our test platform is a desktop running Ubuntu 14.04.4 LTS (Linux kernel 3.13.0-79) and equipped with a 3.2-GHz Intel Xeon processor (6 cores, 12 hyper-threads), 8 GB of RAM, and a 500-GB HDD.

5.1 Probe Overhead and Scalability

We measured the average CPU cycles that the different probe types in Table 1 take with increasing number of threads. We evaluated each `PROBE_BEGIN`/`PROBE_END` pair separately. Due to space limitations, we only report results for two probes (`LLC_PROBE` and `FLT_PROBE`), but we verified that the other probes incur less or similar overhead. Figure 2(a) shows that `LLC_PROBE`s spend ∼360 CPU cycles in total: `LLC_PROBE_BEGIN` takes ∼100 cycles to retrieve the initial value from the hardware performance counter, and `LLC_PROBE_END` spends ∼160 cycles extra to create a trace record and push it into the queue. Similarly, Figure 2(b) shows that `FLT_PROBE_BEGIN` and `FLT_PROBE_END` take about 400 and 600 cycles, respectively, for a total of ∼1,000 cycles. `FLT_PROBE`'s high cost comes mostly from `getrusage` system call, but its impact can be easily reduced by increasing the probe's sub-sampling counter.

We also observed that the CPU cycles consumed by the probes stay almost constant as the thread count increases. Such nice scalability comes from our design choice of having a trace record queue per CPU and per probe type.

[2]https://developers.google.com/protocol-buffers/
[3]http://www.zeromq.org
[4]http://www.grinninglizard.com/tinyxml/

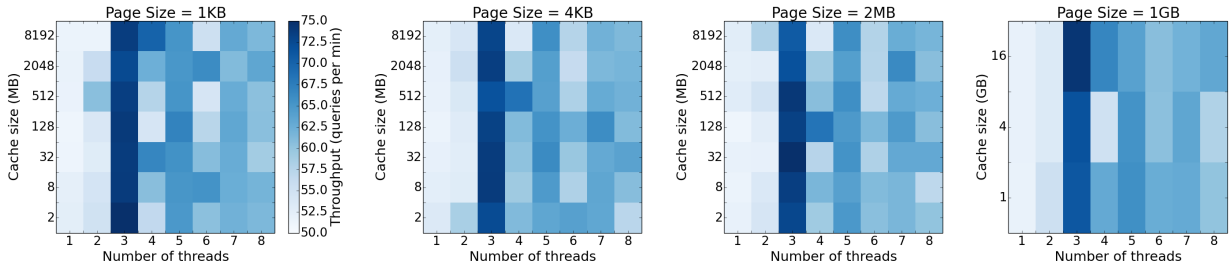


Figure 3: SQLite query throughput (in queries per minute) for different configurations. The throughput color scale, where darker is higher, is at the right side of the left-most plot and is the same across the plots. Note that throughput is highest with three threads in all plots.

5.2 Exploring Query Performance of SQLite

Next, we show how SPEX can help an application developer explore the query performance of SQLite v3.8.7.4, an embedded SQL database engine. We chose SQLite as our target system because it is very popular and well documented. However, application developers often need to experiment with its configuration parameters to find the right values that meet their performance goals.

Setup: We wrote a simple application that continuously queries a SQLite database. The application includes a `TPT_PROBE` pair that encloses the function making the queries. The subsampling counter for the probe was set to 1. We loaded the NYC Taxi Trips dataset[5] (169M records) into the database, occupying 35GB of storage. We use a range query that computes the mean passenger count in trips that start in a 100×100Km^2 area (4 constraints); it accesses about 200K records. Our test application also acts as the CONFIGURATION AGENT; it receives XML configurations from the EXPLORER and applies them to SQLite via PRAGMA statements.

Configuration space: From the parameters that may affect query throughput, we select *cache size*, which is the amount of memory in bytes SQLite uses for caching data. It results from the multiplication of two SQLite's PRAGMA parameters: `page_size` (in bytes) and `cache_size` (in number of pages). Also, we are interested in how the throughput varies with the *number of threads* issuing queries.

Thus, the explored configurations are as follows. We choose four widely used `page_size` values: 1KB (SQLite's default), 4KB, 2MB, and 1GB. We then select values for the PRAGMA `cache_size` parameter (in number of pages) so that the calculated *cache memory* values are multiples of 4, starting from 2MB up to 8,192MB. In the case of `page_size` = 1GB, we start from `page_size` = 1 page up to 16. The rest of SQLite's parameters use their default values. Finally, *thread counts* simply go from 1 to 8, and each thread opens a read-only connection to the database under multi-thread and shared-cache modes.

Exploration policy: In this example, we use a simple, brute-force policy (∼40 LOC) that evaluates all the configurations for each `page_size` value (24 for 1GB, and 56 for each of the other values). The policy stops each run after receiving 20 trace records from each query thread.

Results: Figure 3 shows the query throughput for the different configurations in color scale (darker is higher). We can see that the throughput consistently reaches the maximum with three threads and starts decreasing beyond that. The scalability collapse [8] in SQLite at such a small number of threads *only doing reads* may get an application developer by surprise. Without sufficient knowledge of SQLite's implementation and access to relevant benchmark results, the developer would have not known this, and may simply decide to use 4 or more query threads for a (mostly) read-only workload, obtaining far from optimal performance.

We also observe that the query throughput does not necessarily increase with the cache size. The reason is that for our test query and dataset, SQLite did not require much memory to cache the results. However, with different data and queries, the throughput could very well vary with the cache size. Developers of cache-sensitive applications can explore their performance with SPEX and find a cache size that yields acceptable throughput.

This example suggests that SPEX can be effective and useful. It is encouraging that an application developer could obtain meaningful information about the performance of a third-party system with relatively little effort (*i.e.,* an exploration policy with tens of lines of code and few probes). We have had similar experience characterizing the performance of libraries (*e.g.,* FFTW[6]) and internal research prototypes (*e.g.,* a time-series datastore [17]), but the details are beyond the scope of this paper.

6. RELATED WORK

From a large body of work, this section summarizes the most relevant efforts compared to our SPEX framework.

Performance profiling: Profilers are tools commonly used to understand systems' performance characteristics. System statistics are obtained through performance events and counters [1, 12], memory checkers [2], and call-graphs [13]. Profilers can be extended to monitor detailed system operations by inserting probes or progress points into the source code [9, 3, 10], but many off-the-shelf systems keep their code private, and even for those with accessible source code, users unaware of the implementation details would not know where to put the trace probes.

Researchers have proposed to profile systems by other means. Aguilera *et al.* [4] monitor message traces to infer causal paths between nodes in a distributed system, while SNAP [19] collects TCP statistics and socket-call logs of shared network resources to find correlations between TCP connections. However, these solutions still require full knowledge of the system's topology and can only identify bottlenecked nodes. Some works try to remove this require-

[5]http://www.andresmh.com/nyctaxitrips/

[6]http://www.fftw.org

ment by performing instruction-level analysis via dynamic compilation [14] or taint tracking of binaries [5]. But, they still offer limited usability as they require additional system instrumentation and incur non-negligible performance overhead due to logging and dynamic analysis.

Performance exploration: Another approach to improve system performance is to tune configurations through training and exploration. Flex [7] tunes a database by exploring different workloads, configurations, data, and resources. Contrary to SPEX that uses C/C++, Flex requires users to learn a new language, *Slang*, to define the exploration environments. Bodik *et al.* [6] train a statistical performance model of a system to maintain optimal performance by predicting future workload based on change point detection. They require a fixed policy, whereas SPEX lets user to decide what policy to use.

Autonomous adaptation: Extensive work exists on self-adaptive systems (*e.g.,* [16, 11]) and auto-tuning (*e.g.,* [18]). In this case, configuration parameters (knobs) are automatically adjusted to improve the underlying system's runtime performance. Our work is complementary since it facilitates characterizing systems' performance, a essential step in designing adaptation and tuning strategies.

7. CONCLUSIONS AND FUTURE WORK

In response to the challenges that users face in tuning third-party systems, we have presented SPEX, a framework that allows users to characterize the performance of such systems by investing a modest effort. We exploit the natural separation of responsibilities by 1) having developers instrument their system with probes, and 2) allowing the users to reuse the same instrumentation and enable only the probes needed to explore certain performance characteristics of the system. Moreover, SPEX offers great flexibility by permitting users to not only choose exploration policies shipped with the system, but also implement their own policies.

Our evaluation showed that SPEX's instrumentation probes add low overhead (within few hundred CPU cycles) to the observed system thanks to our data collection optimizations, particularly the lock-free queues. Furthermore, our experience exploring SQLite's performance showed that SPEX can provide useful information with relatively little effort, pinpointing unfavorable configurations with simple exploration policies of tens of lines of code.

As part of our future work, we plan to design and evaluate more sophisticated exploration policies (*e.g.,* using adaptive sampling) for various systems. The goal is to identify high-performing configurations in reduced times when compared with simple, brute-force policies. To ease SPEX's adoption, we also plan to support binary instrumentation as it eliminates the need to modify the source code of existing systems. Currently, SPEX is able to explore single-node systems, but we want to extend it to support performance exploration of multi-node systems. We are also looking into running SPEX with other languages such as Java, Scala, and Python.

8. REFERENCES

[1] https://perf.wiki.kernel.org.

[2] http://www.valgrind.org.

[3] Xentrace. http://xenbits.xen.org.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[5] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[6] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *ACDC*, 2009.

[7] N. Borisov and S. Babu. Rapid experimentation for testing and tuning a production database deployment. In *EDBT*, 2013.

[8] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *OSDI*, 2008.

[9] B. Cantrill, A. Leventhal, M. Shapiro, and B. Gregg. Dtrace. http://dtrace.org.

[10] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *SOSP*, 2015.

[11] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *ICSE*, 2014.

[12] S. L. Graham, P. B. Kessler, and M. K. Mckusick. gprof: a call graph execution profiler. In *SIGPLAN*, 1982.

[13] J. Levon and P. Elie. Oprofile: A system profiler for linux. http://oprofile.sourceforge.net, 2004.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[15] D. Rogerson. *COM*. Microsoft programming series. Microsoft Press, 1997.

[16] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. D. Santambrogio. Metronome: operating system level performance management via self-adaptive computing. In *DAC*, 2012.

[17] D. G. Waddington and C. Lin. A fast lightweight time-series store for iot data. *CoRR*, 2016.

[18] S. Williams, L. Oliker, J. Carter, and J. Shalf. Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning. In *SC*, 2011.

[19] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.