The API Performance Contract

How can the expected interactions between caller and implementation be guaranteed?

Robert F. Sproull and Jim Waldo

When you call functions in an API, you expect them to work correctly; sometimes this expectation is called a *contract* between the caller and the implementation. Callers also have performance expectations about these functions, and often the success of a software system depends on the API meeting these expectations. So there's a *performance contract* as well as a *correctness contract*. The performance contract is usually implicit, often vague, and sometimes breached (by caller or implementation). How can this aspect of API design and documentation be improved?

Any significant software system today depends on the work of others: sure, you write some code, but you call functions in the operating system and in a variety of software packages through APIs that cut down on the amount of code that you must write. In some cases you even outsource work to distant servers connected to you by cranky networks. You depend on these functions and services for correct operation, but you also depend on them performing well enough that the overall system performs well. There is bound to be performance variation in complex systems that involve paging, network delays, sharing resources such as disks, and so on. Even in simple setups such as a stand-alone computer with all program and data in memory, however, you can be surprised when an API or operating system doesn't meet performance expectations.

People are accustomed to speaking of the contract between an application and the implementation of an API to describe correct behavior when an API function is invoked. The caller must meet certain initial requirements, and the function is then required to perform as specified. (These dual responsibilities are like the pre- and post-conditions that appear in Floyd-Hoare logic for proving program correctness.) While today's API specifications do not make correctness criteria explicit in a way that leads to correctness proofs, the type declarations and textual documentation for an API function strive to be unambiguous about its logical behavior. If a function is described as "add the element e to the end of list l " where e and l are described by types in the calling sequence, then the caller knows what behavior to expect.

There's more to an API function than correctness, however. What resources does it consume, and how fast is it? People often make assumptions based on their own judgments of what the implementation of a function should be. Adding an element to the end of a list should be "cheap." Reversing a list might be "a linear function of list length." For many simple functions, these intuitions suffice to avoid trouble—though not always, as described later in this article. For functions of any complexity, however, such as "draw string s in font f at (x,y) in window w ," or "find the average of the values stored on a remote file," there's room for surprise if not frustration at the performance. Further, the API documentation provides no hints about which functions are cheap and which are costly.

To complicate matters, after you've tuned your application to the performance characteristics of an API, a new version of the API implementation arrives or a new remote server stores the data

file, and rather than overall performance increasing (the eternal hope for something new), it tanks. In short, the performance contracts that accompany the composition of software components in systems deserve more attention.

A PERFORMANCE TAXONOMY

Programmers begin building intuitive API performance models early in their careers (see box 1). To be useful, the model need not be very accurate. Here's a simple taxonomy:

Always cheap (examples: `toupper`, `isdigit`, `java.util.HashMap.get`). The first two functions are always cheap, usually inlined table lookups. Lookup in a properly sized hash table is expected to be fast, but a hash collision may slow down an occasional access.

Usually cheap (examples: `fgetc`, `java.util.HashMap.put`). Many functions are designed to be fast most of the time but require occasionally invoking more complex code; `fgetc` must occasionally

Forming your own model of performance

In learning to program, you very early on acquire rules of thumb about performance. The pseudocode below is a pattern for reasonably efficient processing of a modest-size file of characters:

```
fs = fopen("~/dan/weather-data.txt", "r");  //(1)
for ( i=0; i<10000; i++) {
    ch = fgetc(fs);                          //(2)
    // process character ch
}
...
```

The function call (1) is expected to take a while, but the call to get a character (2) is expected to be *cheap*. This makes intuitive sense: to process a file, a stream need be opened only once, but the “get the next character” function will be called often, perhaps thousands or millions of times.

These two stream functions are implemented by a library. The documentation for the library² clearly states what these functions do—an informal presentation of the correctness contract between the implementation and the application. There is no mention of performance, nor any hint to the programmer that the two functions differ substantially in performance. Therefore, programmers build models of performance based on experience, not specifications.⁷

Not all functions have obvious performance properties. For example:

```
fseek(fs, ptr, SEEK_SET); //(3)
```

This function might be cheap when the target file data is already in a buffer. In the general case, it will involve an operating-system call and perhaps I/O. In a wild case, it might require reeling off several thousand feet of magnetic tape. It's also possible that the library function isn't cheap even in the simple case: the implementation may simply store the pointer and set a flag that will cause the hard work to be done on the next stream call that reads or writes data, thus pushing the performance uncertainty onto an otherwise-cheap function.

read a new character buffer. Storing a new item in a hash table may make the table so full that the implementation enlarges it and rehashes all its entries.

The documentation for `java.util.HashMap` makes a commendable start at exposing a performance contract: “This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the “capacity” of the `HashMap`...”³

The performance of `fgetc` depends on properties of the underlying stream. If it’s a disk file, then the function will normally read from a user-memory buffer without requiring an operating-system call, but it must occasionally invoke the operating system to read a new buffer. If it’s reading input from a keyboard, then the implementation probably invokes the operating system for each character read.

Predictable (examples: `qsort`, `regex`). The performance of these functions varies with properties of their arguments (e.g., the size of an array to sort or the length of a string to search). These functions are often data-structure or common-algorithm utilities that use well-known algorithms and do not require system calls. You can usually judge performance based on expectations for the underlying algorithms (e.g., that sort will take $n \log n$ time). When using complex data structures (e.g., B-trees) or generic collections (where it may be hard to identify the underlying concrete implementation), it may be harder to estimate performance. It is important to understand that the predictability may be only probable; `regex` is generally predictable based on its input, but there are pathological expressions that will cause exponential time explosions.

Function creep

```
SetFontSize(f, 10);  
SetDrawPosition(w, 200, 20);  
DrawText(w, f, “This is a passage.”);
```

This example is from a fictitious window system: it sets a font size and a drawing position, then draws some text. You might expect all of these functions to be quite cheap, because rendering text windows on a screen should be fast. Indeed, in early window systems, such as QuickDraw on the Macintosh, you would be right.

The functionality of today’s window systems has, rightly, crept upward. Character rasters—even those rendered on a display—are computed from geometric outlines, so preparing a font of a given size may take a while. Modern window systems speed up rendering text by generous use of caching—even saving prepared rasters on disk so that they are available to multiple applications and are preserved across system restarts. Application programmers, however, have no idea whether the fonts they are requesting have been prepared and cached, whether `SetFontSize` will do the (considerable) computation for all of a font’s characters, or whether characters will be converted one at a time as needed by strings passed to `DrawText`.

Augmented functionality may also allow storing font data on network-attached file servers, so font access may take a while, or even fail after a long timeout if the server does not respond.

In many cases the details won’t matter, but if the delays are substantial or highly variable, the application might choose to prepare during its initialization all the fonts it plans to use. Most window system APIs do not have such a function.

Unknown (examples: `fopen`, `fseek`, `pthread_create`, many “initialization” functions, and any call that traverses a network). These functions are not cheap and often have high variance in their performance. They allocate resources from pools (threads, memory, disks, operating-system objects), often requiring exclusive access to shared operating-system or I/O resources. There is usually substantial initialization required. Calling across a network is always expensive (relative to local access), but the variation in the expense can be larger still, making the formation of a reasonable performance model much more difficult.

Threads libraries are easy marks for performance issues. The Posix standard took many years to settle down, and implementations are still plagued with problems.⁶ Portability of threaded applications remains dicey. Some of the reasons for difficulties with threads are: (1) the need for tight integration with operating systems, almost all of which (including Unix and Linux) were not designed originally with threads in mind; (2) interaction with other libraries, especially making functions thread-safe and dealing with performance problems thereby induced; and (3) several different design points for threads implementations, loosely characterized as lightweight and heavyweight.

SEGMENTING APIS BY PERFORMANCE

Some libraries offer more than one way of performing a function, usually because the alternatives have quite different performance.

Returning to box 1, note that most programmers are told that using a library function to fetch each character is not the fastest way (even if the code for the function is inlined to avoid the function-call overhead). The more performance-conscious will read a very large array of characters and extract each one using array or pointer operations in the programming language. In extreme cases the application can map pages of the file into memory pages to avoid copying data into an array. In return for performance, these functions put a larger burden on the caller (for example, to

When the performance contract breaks

```
SetColor(g, COLOR_RED);  
DrawLine(g, 100, 200, 200, 600);
```

Graphics library functions are usually fast. It’s reasonable to assume that you draw many lines of varied colors by setting the color for each line; if you’re worried that the `SetColor` function is not cheap, you might call it only when the color changes. At one point, the graphics hardware offering of a workstation company required clearing the (hardware) geometry pipeline to change a line’s color, so color change was a costly operation. To be fast, the application was, for example, forced to sort lines by color and draw all red lines together. This was not the intuitive way to write the program, and it forced a major change to the structure and programming of the application.

The expensive `SetColor` operation came to be understood as a mistake and was fixed in subsequent hardware. The code that had been written to overcome the mistake, however, may remain, becoming an inexplicable complexity for anyone maintaining the code who doesn’t know the performance history of the hardware.

get buffer arithmetic correct and to make the implementation consistent with other library calls such as a call to `fseek` that would require adjusting buffer pointers and perhaps contents).

Programmers are always counseled to avoid premature optimization in their programs, thus delaying extreme revisions until simpler ones have proven inadequate. The only way to be sure about performance is to measure it. Programmers usually write the entire program before confronting mismatches between performance expectations (or estimates) and the reality delivered by an implementation.

PERFORMANCE VARIATION

The performance of “predictable” functions can be estimated from properties of their arguments. The “unknown” functions may also vary greatly depending on what they are asked to do. The time required to open a stream on a storage device will certainly depend on access times, and perhaps data-transfer rates, of the underlying device. Storage accessed via network protocols may be especially costly; certainly it will be variable.

Many cheap functions are cheap only *most* of the time, or they have an *expected* cheap cost. A “get a character” routine must occasionally refill a buffer using an operating-system call that always takes much longer than fetching a character from a full buffer—and might occasionally take a very long time indeed (for example, reading a file on a heavily loaded file server or from a disk that is dying and will succeed only after multiple read retries).

The long story of malloc and dynamic memory allocation

It would be nice if dynamic memory allocation with the `malloc()` function could be characterized as “usually cheap,” but that would be wrong since memory allocation—and `malloc` in particular—is one of the first suspects when programmers start hunting for performance problems. As part of their education in performance intuition, programmers learn that if they are calling `malloc` tens of thousands of times, especially to allocate small fixed-size blocks, they are better off allocating a single larger chunk of memory with `malloc`, chopping it up into the fixed-size blocks, and managing their own list of free blocks.

Implementers of `malloc` have struggled over the years to make it usually cheap in the presence of wide variations in usage patterns and in the properties of the hardware/software systems on which it runs.⁴ Systems that offer virtual memory, threading, and very large memories all present challenges to “cheap,” and `malloc`—along with its complement, `free()`—must trade off efficiency and evils of certain usage patterns such as memory fragmentation.⁸

Some software systems, such as Lisp and Java, use automatic memory allocation along with garbage collection to manage free storage. While this is a great convenience, a programmer concerned with performance must be aware of the costs. A Java programmer, for example, should be taught early about the difference between the `String` object, which can be modified only by making a new copy in new memory, and a `StringBuffer` object, which contains space to accommodate lengthening the string. As garbage-collection systems improve, they make the unpredictable pauses for garbage collection less common; this can lure the programmer into complacency, believing that the automated reclamation of memory will never be a performance problem, when in fact it is only less often a performance problem.

The functions with “unknown” performance are likely to exhibit wide performance variations for a variety of reasons. One reason is function creep (see box 2), where a generic function becomes more powerful over time. I/O streams are a good example: a call to open a stream invokes very different code in libraries and operating systems depending on the type of stream being opened (local disk file, network-served file, pipe, network stream, string in memory, etc.). As the range of I/O devices and types of files expands, the variance of the performance can only increase. The common life cycle of most APIs—to add functionality incrementally over time—inevitably increases performance variations.

A large source of variation is differences between ports of libraries to different platforms. Of course, the underlying speed of the platform—both hardware and operating system—will vary, but library ports can lead to changes in the *relative* performance of functions within an API or performance across APIs. It’s not uncommon for a quick-and-dirty initial port to have many performance problems, which are gradually fixed. Some libraries, such as those for handling threads, have notoriously wide porting-performance variation. Thread anomalies can surface as extreme behaviors—an impossibly slow application or even deadlock.

These variations are one reason why constructing precise performance contracts is difficult. It’s usually not necessary to know performance with great precision, but extreme variations from expected behavior can cause problems.

FAILURE PERFORMANCE

The specifications for an API include details of behavior when a call fails. Returning an error code and throwing an exception are common ways of telling the caller that the function did not succeed. As with specifications for normal behavior, however, the *performance* of the failure is not specified. Here are three important cases:

- **Fail fast.** A call fails quickly—as fast or faster than its normal behavior. Calling `sqrt(-1)` fails fast. Even when a `malloc` call fails because no more memory is available, the call should return about as fast as any `malloc` call that must request more memory from the operating system. A call to open a stream for reading a nonexistent disk file is likely to return about as fast as a successful call.
- **Fail slow.** Sometimes a call fails very slowly—so slowly that the application program might have wanted to proceed in other ways. For example, a request to open a network connection to another computer may fail only after several long time-outs expire.
- **Fail forever.** Sometimes a call simply stalls and does not allow the application program to proceed at all. For example, a call whose implementation waits on a synchronization lock that is never released may never return.

Intuition about failure performance is rarely as good as that for normal performance. One reason is simply that writing, debugging, and tuning programs provides far less experience with failure events than with normal events. Another is that a function call can fail in many, many ways, some of them fatal, and not all described in the API’s specs. Even exception mechanisms, which are intended to more precisely describe the handling of errors, do not make all possible exceptions visible. Moreover, as library functionality increases, so do the opportunities for failure. For example, APIs that wrap network services (ODBC, JDBC, UPnP,...) intrinsically subscribe to the vast array of network-failure mechanisms.

A diligent application programmer uses massive artillery to deal with unlikely failures. A common

technique is to surround rather large parts of a program with try...catch blocks that can retry whole sections that fail. Interactive programs can try to save a user's work using a giant try...catch around the entire program, the effect of which is to mitigate failure of the main program by saving, in a disk file, key logs or data structures that record the effects of the work done before the failure.

The only way to deal with stalls or deadlocks is to set up a watchdog thread, which expects a properly running application program to check in periodically with the watchdog, saying, in effect, "I'm still running properly." If too much time elapses between check-ins, the watchdog takes action—for example, saving state, aborting the main thread, and restarting the entire application. If an interactive program responds to a user's command by calling functions that may fail slowly, it may use a watchdog to abort the entire command and return to a known state that allows the user to proceed with other commands. This gives rise to a defensive programming style that plans for the possible abortion of every command.

WHY DOES THE PERFORMANCE CONTRACT MATTER?

Why must an API adhere to a performance contract? *Because major structures of an application program may depend on the API's adherence to such a contract.* A programmer chooses APIs, data structures, and overall program structures based in part on expectations of API performance. If the expectations or the performance are grossly wrong, the programmer cannot recover merely by tuning API calls but must rewrite large, possibly major, portions of the program. The defensive structure of the interactive program previously mentioned is another example.

In effect, a serious violation of the performance contract leads to a composition failure: a program written to the contract cannot be mated to (composed with) an implementation that fails to uphold the contract.

Of course, there are many programs whose structure and performance are influenced very little by library performance (scientific computations and large simulations are often in this category). However, much of today's "routine IT," especially the software that pervades Web-based services, makes extensive use of libraries whose performance is vital to overall performance.

Even small variations in performance can cause major changes in the perception of a program by its users. This is especially true in programs dealing with various kinds of media. Dropping occasional frames of a video stream might be acceptable (indeed, more acceptable than allowing the frame rate to lag other media), but humans have evolved to detect even slight dropouts in audio, so minor changes in the performance of audio media may have major impacts on the acceptability of the overall program. Such worries have led to considerable interest in notions of quality of service, which in many ways is the attempt to ensure performance at a high level.

HOW CAN CONTRACT VIOLATIONS BE AVOIDED?

Although performance contract violations are rare and rarely catastrophic, paying attention to performance when using a software library can help lead to more robust software. Here are some precautions and strategies to use:

- 1. Choose APIs and program structures carefully.** If you have the luxury of writing a program from scratch, ponder performance contract implications as you begin. If the program starts out as a prototype and then remains in service for a while, it will undoubtedly be rewritten at least once; a rewrite is an opportunity to rethink API and structure choices.

2. API implementers have an obligation to present a consistent performance contract as new versions and ports are released. Even a new experimental API will garner users who will begin to derive a performance model of the API. Thereafter, changing the performance contract will surely irritate developers and may cause them to rewrite their programs.

Once an API is mature, it is essential that the performance contract not change. In fact, the most universal APIs (e.g., `libc`) arguably have become so in part because their performance contracts have been stable during the evolution of the API. The same goes for ports of APIs.

One could hope that API implementers might routinely test new versions to verify that they have introduced no performance quirks. Unfortunately, such testing is rarely done. This doesn't mean, however, that you can't have your own tests for the parts of an API that you depend on. With a profiler, a program can often be found to rely on a small number of APIs. Writing a performance test suite that will compare new versions of a library against the recorded performance of earlier versions can give programmers an early warning that the performance of their own code is going to change with the release of the new library.

Many programmers expect computers and their software to get faster with time—*uniformly*. That is, they expect each new release of a library or a computer system to scale up performance of all API functions equally, especially the “cheap” ones. This is in fact very difficult for vendors to guarantee, but it so closely describes actual practice that customers believe it. Many workstation customers expected newer versions of graphics libraries, drivers, and hardware to increase the performance of *all* graphics applications, but they were equally keen on functionality improvements of many kinds, which usually reduce the performance of older functions, even if only slightly.

One could also hope that API specifications would make the performance contract explicit, so that people using, modifying, or porting the code would honor the contract. Note that a function's use of dynamic memory allocation, whether implicit or automatic, should be part of this documentation.

3. Defensive programming can help. A programmer can use special care when invoking API functions with unknown or highly variable performance; this is especially true for considerations of failure performance. You can move initializations outside performance-critical areas and try to warm up any cached data an API may use (e.g., fonts). APIs that exhibit large performance variance or have a lot of internally cached data could help by providing functions for passing hints from the application to the API about how to allocate or initialize these structures. Occasional pings to servers that a program is known to contact can establish a list of those that might be unavailable, allowing some long failure pauses to be avoided.

One technique sometimes used in graphics applications is to do a dry run, rendering a window of graphics into an off-screen (not visible) window, merely to warm up caches of fonts and other graphics data structures.

4. Tune parameters exposed by the API. Some libraries offer explicit ways of influencing performance (e.g., controlling the size of buffers allocated for files, the initial sizes of tables, or the size of a cache). Operating systems also provide tuning options. Adjusting these parameters can improve performance *within* the confines of a performance contract; tuning does not remedy gross problems but can mitigate otherwise fixed choices embedded in libraries that heavily influence performance.

Some libraries provide alternative implementations of functions with identical semantics, usually in the form of concrete implementations of generic APIs. Tuning by choosing the best concrete implementation is usually very easy. The Java Collections package is a good example of this structure.

Increasingly, APIs are designed to adapt to usage on the fly, freeing the programmer from choosing the best parameter settings. If a hash table gets too full, it is automatically expanded and rehashed (a virtue balanced by, alas, the performance hit of the occasional expansions). If a file is being read sequentially, then more buffers can be allocated so that it is read in larger chunks.

5. Measure performance to verify assumptions. Common advice to programmers is to instrument key data structures to determine whether each structure is being used correctly. For example, you might measure how full a hash table is or how often hash collisions occur. Or you might verify that a structure designed to be fast for reading at the expense of write performance is actually being read more than written.

Adding sufficient instrumentation to measure the performance of many API calls accurately is difficult, a large amount of work, and probably not worth the information it yields. Adding instrumentation on those API calls that are central to the performance of an application (assuming that you can identify them and that your identification is correct), however, can save a lot of time when problems arise. Note that much of this code can be reused as part of the performance monitor for the next release of the library, as mentioned earlier.

None of this is meant to discourage dreamers from working on tools that would automate such instrumentation and measurement, or on ways to specify performance contracts so that performance measurements can establish compliance with the contract. These are not easy goals, and the return may not be huge.

It is often possible to make performance measurements without having instrumented the software in advance (e.g., by using profilers or tools such as DTrace⁵). These have the advantage of not requiring any work until there's a problem to track down. They can also help diagnose problems that creep in when modifications to your code or to libraries upset performance. As *Programming Pearls* author Jon Bentley recommends, "Profile routinely; measure performance drift from a trusted base."¹

6. Use logs: detect and record anomalies. Increasingly, violations of performance contracts appear in the field when distributed services are composed to form a complex system. (Note that major services offered via network interfaces sometimes have SLAs [service-level agreements] that specify acceptable performance. In many configurations, a measurement process occasionally issues service requests to check that the SLA is met.) Since these services are invoked over a network connection using methods akin to API function calls (e.g., remote procedure call or its variants such as XML-RPC, SOAP, or REST), the expectation of a performance contract applies. Applications detect failure of these services and often adapt gracefully. Slow response, however, especially when there are dozens of such services that depend on each other, can destroy system performance very quickly. Professionally managed services environments, such as those that provide Web services at major Internet sites, have elaborate instrumentation and tooling to monitor Web-service performance and to tackle problems that arise. The far more modest computer collections in homes also depend on such services—many that are part of the operating system on each laptop computer and some that are embedded in appliances on the network (e.g., a printer, network file system, or file backup service)—but there are no aids to help detect and deal with performance problems.

It would be helpful if clients of these services made note of the performance they expect and produced log entries that help diagnose problems (this is what syslog is for). When your file backup seems unreasonably slow (one hour to back up 200 MB), is it slower than yesterday? Slower than it was before the latest operating-system software update? Given that several computers may be sharing

the backup device, is it slower than you should expect? Or is there some logical explanation (e.g., the backup system finds a damaged data structure and embarks on a long procedure to rebuild it)?

Diagnosing performance problems in compositions of opaque software (where no source code is available, and there are no details of the modules and APIs that form the composition) requires that the software play a role in reporting performance and detecting problems. While you cannot address performance problems within the software itself (it's opaque), you can make adjustments or repairs to operating systems and the network. If the backup device is slow because its disk is almost full, then you can surely add more disk space. Good logs and associated tools would help; sadly, logs are an undervalued and neglected area of computer system evolution.

MAKING COMPOSITION WORK

Today's software systems depend on composing independently developed components in such a way that they work—meaning they perform the desired computations at acceptable speed. The dream of statically checking a composition to guarantee that the composition will be correct (“correct by composition”) remains elusive. Instead, software-engineering practice has developed methods for testing components and compositions that work pretty well. Every time an application binds to a dynamic library or launches atop an operating-system interface, the correctness of the composition is required.

Despite its importance, the performance of a composition—whether the client and the provider of an interface adhere to the performance contract between them—has been given short shrift. True, this contract is not as important as the correctness contract, but the full power of composition depends on it.

ACKNOWLEDGMENTS

The ideas in this paper are not new; casts of thousands have preceded us. Special mention is due to Butler Lampson, who coined the term *cheap* to describe functions whose performance is fast enough to dispel all attempts to optimize them away; it has a wonderful ring of “economical yet serviceable.” Thanks also to Eric Allman, who made helpful comments on a rather too glib early draft, and Jon Bentley, an inveterate performance debugger.

REFERENCES

1. Bentley, J. Personal communication.
2. GNU C Library; http://www.gnu.org/software/libc/manual/html_node/index.html.
3. Java Platform, Standard Edition 7. API Specification; <http://docs.oracle.com/javase/7/docs/api/index.html>.
4. Korn, D. G., Vo, K.-P. 1985. In search of a better malloc. In *Proceedings of the Summer '85 Usenix Conference*: 489-506.
5. Oracle. Solaris Dynamic Tracing Guide; <http://docs.oracle.com/cd/E19253-01/817-6223/>.
6. Pthreads(7) manual page; <http://man7.org/linux/man-pages/man7/pthreads.7.html>.
7. Saltzer, J. H., Kaashoek, M. F. 2009. Principle of least astonishment. In *Principles of Computer System Design*. Morgan Kaufmann: 85.
8. Vo, K.-P. 1996. Vmalloc: a general and efficient memory allocator. *Software Practice and Experience* 26(3): 357-374; <http://www2.research.att.com/~astopen/download/ref/vmalloc/vmalloc-spe.pdf>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ROBERT F. SPROULL is an adjunct faculty member of the computer science department at the University of Massachusetts (Amherst), a position he assumed after retiring as director of Sun Microsystems Laboratories.

JIM WALDO is a professor of the practice of computer science at Harvard University, where he is also the chief technology officer, a position he assumed after leaving Sun Microsystems Laboratories.

© 2013 ACM 1542-7730/14/0100 \$10.00