A Study of Dynamic Software Update Quiescence for Multithreaded Programs

Christopher M. Hayden, Karla Saur, Michael Hicks, Jeffrey S. Foster
University of Maryland, College Park
{*hayden, ksaur, mwh, jfoster*}*@cs.umd.edu*

***Abstract*—Dynamic software updating (DSU) techniques show great promise in allowing vital software services to be upgraded without downtime, avoiding dropped connections and the loss of critical program state. For multithreaded programs, DSU systems must balance *correctness* and *timeliness*. To simplify reasoning that an update is correct, we could limit updates to take place only when all threads have blocked at well-defined *update points*. However, several researchers have pointed out that this approach poses the risk of delaying an update for too long, even indefinitely, and therefore have developed fairly complicated mechanisms to mitigate the risk. This paper argues that such mechanisms are unnecessary by demonstrating empirically that many multithreaded programs can be updated with minimal delay using only a small number of manually annotated update points. Our study of the time taken for all of the threads in six real-world, event-driven programs to reach their update points ranged from 0.155 to 107.558 ms, and most were below 1 ms.**

I. Introduction

As users have become increasingly dependent on software services, the cost of bringing those services down for maintenance has grown considerably. *Dynamic software updating* (DSU) aims to address this problem by allowing programs to be updated with minimal service disruption. However, although DSU systems excel at preserving the running state of programs during an update, they are often unable to avoid all disruption. In particular, at the time that an update is applied, there may be a period during which a server is briefly unavailable while waiting for all threads to reach a state in which the system can safely be updated.

Finding ways to apply multithreaded updates with minimal delay has been a fruitful topic for the research community, and a variety of solutions have been proposed. Most of this research has followed a common theme: updates should be supported at as many points during program execution as possible. However, in our opinion this goal has a serious drawback: developers must reason about the correctness of all possible update timings, and that task becomes harder to do the more update points and threads there are. Moreover, many of the proposed techniques also employ complex program analyses or other mechanisms that are difficult to use, scale poorly, and/or impose run-time overhead.

We are researching and building DSU systems [5], [3] that take the opposite point of view: they only support updates at a few, developer-identified *quiescent points*, i.e., program locations that are reached between iterations of event-processing loops and at which there is typically less in-flight state. For example, the code below shows a typical thread body to which we have added an update point.

```
1 void *thread_entry(void *arg) {
2   /* thread init code */
3   while (1) {
4     qbench_update(); /* update point */
5     /* loop body: typically handles a single program event */
6   }
7 }
```

We refer to the state in which all program threads have reached an update point as *full quiescence*. While full quiescence is attractive because it reduces the programmer's reasoning burden, one concern is that reaching full quiescence may significantly delay the application of an update, and may degrade the program's performance in the process.

In this paper, we present a small empirical study that shows that, for many programs, simple modifications allow quiescent points to be reached sufficiently often to support updates with little delay. This result suggests that rather than use mechanisms that are hard to implement and hard to reason about, we can instead ask programmers to modify programs in simple ways to make DSU more effective.

For our study, we added quiescent points to Apache httpd, icecast, memcached, suricata, iperf, and space tyrant. We chose programs covering a wide range of domains including media streaming, caching, intrusion detection, and gaming. We chose the quiescent points according to our experience updating multithreaded programs [3], [8], [11]. We linked each modified program with a library, QBench, with which we measured the time the program took to reach full quiescence under various workloads.

Reaching quiescence may be delayed by blocking calls, e.g., those that perform I/O. We found that two simple program changes could effectively overcome this delay. First, the DSU runtime can interrupt some blocked calls by sending a signal to the process; several updating systems do this "for free" since signals are used to alert a process that an update is available. Second, for the remaining blocking calls, we changed the programs to use alternative, interruptible implementations of some system functions (specifically, `pthread_cond_wait` and `sleep`) and added code to redirect control flow back to an update point when a blocking call is interrupted. All of these changes, and the implementation of QBench, are described in detail in Section II. On average,

programs needed 22 lines of code to be changed including adding update points. With these changes, the median time to quiescence with workload was 0.200ms (0.169ms w/o load), with a worst case of 107.558ms (w/o load) for icecast, and a best case of 0.078ms (w/o load) for space tyrant. These results are described in detail in Section III.

In summary, we found that, for a representative suite of benchmark programs, reaching full quiescence can be done quickly given proper run-time support and a small number of program changes. While more experience is needed to see if this result generalizes, we believe it suggests that simple mechanisms may be sufficient to properly balance the safety and timeliness of dynamic updates.

II. Achieving full quiescence

To test our hypothesis that full quiescence permits sufficiently timely dynamic updates, we modified several multi-threaded programs to include appropriate update points, and then measured the time it took to reach full quiescence. We describe our approach here, along with the QBench library we developed to implement quiescence and measure the time required to achieve it.

A. Basic approach

For each program we must add a handful of calls to qbench_update to identify legal update points. The semantics of qbench_update is simple: if no update has been requested, it is a no-op; otherwise, the calling thread blocks until all other threads have also called qbench_update. In detail, we request an update by sending a program the SIGUSR2 signal. QBench installs a signal handler that sets a flag indicating that an update has been requested. A qbench_update call blocks if the flag is set. Once all threads have blocked, the system has reached *full quiescence*. In an actual DSU system, the update would take effect at this point. For our study, QBench instead reports the *quiescence time*, which is the elapsed time from when the first thread reaches an update point (marking the start of reduced program availability) to when the last thread has. Then it simply unsets the flag and releases all of the threads to continue their execution, so we can verify the program operates as expected.

Quiescence is achieved when all threads have reached an update point, which we track by maintaining a thread count and ensuring that each thread hits an update point. To determine when all threads have reached qbench_update (and to store other thread-specific metadata described later), we replace calls to pthread_create with calls to QBench's qbench_pthread_create, which tracks the lifetime of each thread. QBench maintains a count of threads and stores the metadata for all threads in a doubly linked list. Each thread also stores a pointer to its own metadata using thread-local storage, which permits constant time access. When a thread dies, a callback is invoked to clean up thread-local data. QBench provides a cleanup function for a thread's metadata that unlinks it from the global list and decrements the global thread count.

B. Avoiding blocking

Achieving full quiescence may be delayed or thwarted by *blocking calls*. For example, a call to qbench_update may be preceded by a call that reads from a socket. If this call blocks, the thread will not reach its update point until data is available. Worse still, one thread could hold a mutex when it reaches its update point, but then another thread could block on the same mutex prior to reaching its own update point, delaying full quiescence indefinitely.

To avoid these problems, the programmer must ensure that all blocking calls that appear on any path to an update point are *interruptible*. This requirement immediately rules out the second situation above: the program is not permitted to hold any locks when it reaches an update point, because pthread_mutex_lock is not interruptible (nor would it be sensible to make it so). Fortunately, we found that no quiescent points in the programs we considered ever held a lock.

For the benchmark programs in our study, blocking calls that could inhibit quiescence fell into two categories: blocking I/O calls and calls to pthread_mutex_wait. We found that in both cases, we could interrupt the call and the program would either behave correctly with no changes, or we could make it behave correctly with a few small modifications.

1) Blocking on I/O: Mature server programs are often written to deal with interrupted blocking calls, so adding update points to such programs requires little or no change. Consider the following example.

```
1  void *thread_entry(void *arg) {
2    /* thread init code */
3    while (1) {
4      qbench_update();
5      res = accept(sockfd, addr, addrlen);
6      if (res == -1 && errno == EINTR)
7        continue;
8      /* ... handle connection */
9    }
10 }
```

Under normal circumstances an accept call will block until a connection is accepted. However, if a signal is received the call will be interrupted, returning -1 and setting the errno to EINTR.[1] In the above code snippet, the programmer has accounted for this possibility by returning control to the start of the loop so as to retry the accept. Because an operator initiates a program update by sending the process a SIGUSR2 signal, adding the update point to line 4 in the example ensures the blocked accept call will be released and will reach the update point quickly when the update is signaled.

[1] POSIX supports auto-restarting interrupted, "slow" system calls [10] (i.e., without returning EINTR), which would defeat our scheme. We disable that feature by excluding SA_RESTART from the configuration mask used when installing the signal handler.

Note that signals are normally handled by a program's main thread, so only that thread's blocking calls are interrupted. To interrupt blocking I/O calls in all threads, QBench's main signal handler sends a signal to any other thread that has not already reached its update point and is not waiting on a condition variable; how we handle the latter situation is described next.

2) Blocking on condition variables: We observed that the threads in our benchmark programs often coordinate using condition variables, blocking on calls to pthread_cond_wait. As a matter of good style, programmers guard against spurious wake-ups of such calls by placing them in loops, as the following non-highlighted code on lines 6–7 shows:

```
1  void *thread_entry(void *arg) {
2    /* thread init code */
3    while (1) {
4      qbench_update();
5      pthread_mutex_lock(&mutex);
6      while (!input_is_ready() && !qbench_update_requested()) {
7        qbench_pthread_cond_wait(&cond, &mutex);
8      }
9      pthread_mutex_unlock(&mutex);
10     if (qbench_update_requested())
11       continue; /* reaches qbench_update */
12     /* ... handle connection */
13   }
14 }
```

To allow an update to interrupt this idiomatic use of condition variables, we first modify the condition to check whether an update has been requested, as shown in the highlighted code on line 6. We also modify the code following the condition variable loop to jump back to the start of the loop if an update is requested (lines 10-11). This ensures that an update is reached if pthread_cond_wait wakes. One straightforward way to force the pthread_cond_wait call on line 7 to wake up would be to replace it with pthread_cond_timedwait with a short timeout. But this approach incurs some unnecessary delay and potentially expensive polling overhead. Therefore, we replace the pthread_cond_wait call with a call to qbench_pthread_cond_wait, which (before calling pthread_cond_wait) notes the condition variable argument in the global list of threads so that it can later be signaled by another thread once an update has been requested.

These solutions for waking a thread blocking on I/O or condition variables require that another thread be available to signal the process or condition variable (e.g., since pthread_cond_signal cannot safely be called from a signal handler). This presents a problem if the thread that receives the initial updating signal is blocked on a condition variable and so may not wake up to signal other threads. For this reason, QBench launches one additional thread that sleeps during the vast majority of execution, but periodically wakes and checks the update-requested flag. If an update was requested, it will attempt to signal any threads that have not yet reached an update point.

While the above two circumstances cover the vast majority of blocking calls, we note that one of our benchmark programs, Suricata, required custom code to be called from a signal handler to unblock one of its threads (as we describe in Section III-A). POSIX requires that the same signal handler function be used for all threads in a process. To compensate, QBench provides a library function, qbench_thread_update_callback that allows the developer to provide a callback function to be executed for the current thread when an update signal is received.

III. Results

This section presents the results of a study in which we used QBench to measure the quiescence behavior of six multithreaded programs. We found that the changes required to support full quiescence were small (an average of 22 lines per program), and quiescence could be achieved fairly quickly (in less than 1ms in most cases).

A. Experimental setup

The first three columns of Table I describe the size and thread structure of our subject programs. This subsection describes each program briefly, the workload that we used to test it, and how we needed to modify it to achieve full quiescence rapidly.

Apache httpd: Apache httpd is a widely-used web server. We configured httpd to use thread-based concurrency with 3 worker threads. To achieve full quiescence quickly, we first needed to make the standard changes described in Section II and summarized in columns 4–6 of Table I. We report the number of update points and lines of code changed for each program and keep a separate count of the changes that include calls to our library. In the remainder of this section, we describe only the changes given in the Manual Changes column, i.e., those that tweak existing program code beyond adding/substituting calls to QBench.

For httpd, the only such change was modifying a loop written to immediately retry an interrupted poll operation to break out of the loop if an update is requested.

For our experiments with httpd, we used a workload of downloading a large file from the server.

Icecast: Icecast is a streaming audio server that is popular for hosting Internet radio stations. In its standard configuration, it runs with 6 threads, all of which quiesce without modification. Several of the threads use sleep operations to reduce polling; it turns out these sleep times are the dominant component of the time to reach full quiescence.

For icecast, we selected a workload that corresponds to receiving an audio stream from an outside source and forwards it to connected clients. We used the Ezstream command-line tool to generate a source mp3 stream, connected 5 mplayer clients, and requested an update mid-stream.

Table I
THREAD INFORMATION

Program	LoC Total	# of Threads	Upd Points	Changed LoC (†)	Required Manual Chgs	w/Load (ms)		w/o Load (ms)	
						All Chgs	Upd only	All Chgs	Upd only
httpd-2.2.22	232651	$2 + c^*, \ c = 3$	5	7 (5)	3 (Cond. Var. Loop)	0.185	0.230	0.123	0.150
icecast-2.3.2	17038	6	12	3 (3)	1 (Thread Sleeps)	105.152	954.32	107.558	986.265
iperf-2.0.5	3996	$3 + n^\circ, \ n = 1$	5	8 (3)	1 (Cond. Var. Loop)	0.193	DNQ	0.169	DNQ
memcached-1.4.13	9404	$2 + c^*, \ c = 4$	4	27 (4)	2 (libevent changes)	0.166	DNQ	0.155	DNQ
space-tyrant-0.354	8721	$3 + 2n^\circ, n = 5$	6	8 (6)	1 (Thread Sleeps)	0.426	20.583	0.078	20.304
suricata-1.2.1	260344	$8 + c^*, \ c = 3$	7	11 (6)	1 (libpcap break)	0.503	68.098	0.378	DNQ

*Configurable: c workers °Varies by n connected clients †Calls to QBench excluding update DNQ = Does not quiesce.

Iperf: Iperf is a program that measures the network performance (e.g., bandwidth, delay jitter, and datagram loss) between two machines. Although the same executable is used for both client and server modes, we only modified the server code to reach update points during execution. Iperf has 3 threads at startup and an additional thread for each connected client. The main thread has a conditional wait in a while loop. We added an additional update-request–flag check to jump back to the update point when needed. We measured iperf quiescence times while a client (running on the same machine) performed a network measurement.

Memcached: Memcached is a high-performance, distributed caching server that uses libevent to drive its main thread and a configurable number of worker threads. We added an additional libevent handler for the main-thread event loop to respond to SIGUSR2 and break out of libevent. Upon return, the main thread sends a byte on the notification sockets for each worker and then reaches an update point. The notifications cause each worker thread to enter its event handler, where it sees that an update was requested ands returns from libevent to reach an update point. Effectively, these update points were placed at quiescent points, although the style is different from the other programs.

Our test requests an update to a Memcached instance under load from the memslap Memcached benchmark.

Space Tyrant: Space Tyrant is a server for a text-based, multiplayer space strategy game. At startup, Space Tyrant has 3 threads and creates 2 more for each connected player. Space Tyrant implements long sleep operations using loops that check for server-shutdown events between shorter sleeps. We added an additional update-request-flag to jump back to the update point when needed. Space Tyrant's threads required no additional modification. We updated Space Tyrant with 5 concurrent telnet client connections.

Suricata: Suricata is a network intrusion detector that monitors the packets that pass through a network interface. By default, Suricata is configured to use 11 threads. One thread required special treatment: It calls into libpcap's blocking pcap_dispatch function to process packets. libpcap provides a function pcap_breakloop that can be called from a signal handler to interrupt pcap_dispatch. We install a thread-specific handler function (cf. Section II-B2) to break out of the loop when an update signal is received.

In our tests we ran Suricata with a default set of 7,946 packet analysis rules. We requested an update as Suricata processed the packets produced by a constant stream of 10 concurrent http requests and one large file download.

B. Quiescence times

The four rightmost columns of Table I report the median quiescence times of 11 benchmark runs. All tests were run on a machine with an Intel Core 2 Duo T5550 processor with 2GB of memory. For each program, we measured the time taken to reach full quiescence under two workloads: while the server was idle (i.e., no connected clients) and while performing the (program-dependent) work described in the previous section. The idle workloads were used to reveal problematic cases where threads block indefinitely waiting for input. We also measured the quiescence times when using only update points and no other QBench calls.

The table shows we were able to reach full quiescence quickly for both workloads when using QBench; limiting ourselves only to update points would fail to quiesce some programs. Without using our library with Suricata, the quiescence time is variable depending on the rate of traffic filling the input buffers. Nearly all programs quiesced in under 1ms (for both workloads); Icecast's longer times are due to sleep operations inserted by the programmers.

C. Threats to validity

For this study, we did not actually apply dynamic updates to the benchmark programs, so we cannot be sure that the quiescent points inserted are the ones that would be used in practice. However, the choice of quiescent points is largely dictated by the structure of the code (usually at the beginning of each thread's event loop), so it is unlikely that we might find a preferred point that would be reached less often.

In our study, we ensured that blocking operations that occur at the beginning of event handling, when it is safe to immediately jump back to the beginning of the loop, are interruptible. A blocking I/O operation in the middle of event-handling could delay full quiescence if clients are extremely slow or stalled. Our current experiments do not attempt to force this situation to occur, so we do not know whether this is a problem in practice for these programs.

It is also possible that our observations for this particular selection of programs do not generalize to most other programs. To avoid this risk, we have attempted to consider

at a wide variety of program types. It may also be useful to specifically look for programs that are implemented in such a way that they would not work well in this approach.

IV. Prior work

Here we summarize prior work on multithreaded program updates, focusing on how that work controls update timing. We find that while some prior work is insufficiently flexible, much prior work is perhaps overly concerned with minimizing update times. The results of our study suggest that such concerns may not be warranted.

Several systems [1], [6], [11] forbid updates to any code that is actively running. Some synchronization is needed to ensure that all threads satisfy this condition. Unfortunately, as we have observed in prior work [4], this safety condition is insufficient to ensure update safety, and it provides no guarantee that an update is applied in a timely manner. For example, if a program's main function is modified by an update, the update will be delayed indefinitely because main is always running.

STUMP [8] lifts the restriction against updates of active code: instead, any update may take effect when all threads have reached programmer-identified *update points*. To potentially reduce delay at update time, STUMP implements a relaxed synchronization protocol that permits an update whenever it *appears* as if the update took effect at legal update points. A static analysis determines which program points are equivalent to update points [9] and incorporates this information into the synchronization protocol. Unfortunately, in the worst case, there is no guarantee that meaningful opportunities for updating will be created. Moreover, it may be difficult for a developer to understand the results of the analyses, e.g., to understand why it did not permit more update points. Finally, the static analysis itself is fairly intricate, and may not scale to large programs. The reported update times for STUMP for the same programs used in our study (icecast, space tyrant, and memcached) are higher—1,068ms, 6ms, and 1ms, respectively—though the experimental setup is different.

UpStare [7] supports *immediate* updates, with no synchronization, by allowing threads to update at any point during program execution. To provide this support, UpStare requires the developer to create a mapping between each program point in the old version of a changed function and the corresponding point in its new version; such a mapping could require a significant manual effort, depending on the size and complexity of the change. UpStare prevents blocking library calls from delaying an update by substituting versions that include special handling when an update has been requested; we use a similar, but simpler, approach in this paper. The UpStare paper does not report update times for any multithreaded programs.

POLUS [2] supports immediate updates by permitting contemporaneous threads to execute code from different program versions. When a thread accesses a piece of shared state, POLUS uses developer-provided, bidirectional transformation functions to ensure that each thread sees the representation of state that it expects. With this approach, however, the developer must additionally puzzle out the possible multi-version executions and reason that thread interactions via bi-directional transformations will make sense. POLUS was applied to one multithreaded program, Apache httpd. The authors report (for a different hardware configuration) update times on the order of 15ms, but these also include time to transform any in-flight state.

V. Conclusions

In this study, we found that, for a diverse set of benchmark programs, explicit update points at quiescent program points were able to support multithreaded updates quickly and with little implementation complexity. This finding suggests that DSU systems that do not rely on complex program transformation or analysis (i.e., those most likely to see real-world adoption) may be sufficient to bring runtime updates to a non-trivial set of programs. We plan to continue this line of research to better understand the limits of this technique.

This research was supported by the partnership between UMIACS and the Laboratory for Telecommunications Sciences and NSF grant CCF-0910530.

References

[1] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. EuroSys*, 2009.

[2] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *IEEE TSE*, 37(5), September 2011.

[3] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. Technical Report UMD CS-TR-5008, 2012.

[4] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Evaluating dynamic software update safety using efficient systematic testing. *IEEE TSE*, 99(PrePrints), September 2011.

[5] Christopher M. Hayden, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime upgrades. In *Proc. HotSWUp*, 2011.

[6] The K42 Project. http://www.research.ibm.com/K42/.

[7] Kristis Makris and Rida Bazzi. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction. In *USENIX ATC*, 2009.

[8] Iulian Neamtiu and Michael Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI*, 2009.

[9] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proc. POPL*, 2008.

[10] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[11] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. In *Proc. PLDI*, 2009.