Safe and Timely Dynamic Updates for Multi-threaded Programs *

Technical Report

University of California, Riverside

June 2009

Iulian Neamtiu
University of California, Riverside
Riverside, CA 92521, USA
neamtiu@cs.ucr.edu

Michael Hicks
University of Maryland
College Park, MD 20742, USA
mwh@cs.umd.edu

Abstract

Many dynamic updating systems have been developed that enable a program to be patched while it runs, to fix bugs or add new features. This paper explores techniques for supporting dynamic updates to multi-threaded programs, focusing on the problem of applying an update in a timely fashion while still producing correct behavior. Past work has shown that this tension of *safety* versus *timeliness* can be balanced for single-threaded programs. For multi-threaded programs, the task is more difficult because myriad thread interactions complicate understanding the possible program states to which a patch could be applied. Our approach allows the programmer to specify a few program points (e.g., one per thread) at which a patch may be applied, which simplifies reasoning about safety. To improve timeliness, a combination of static analysis and run-time support automatically expands these few points to many more that produce behavior equivalent to the originals. Experiments with thirteen realistic updates to three multi-threaded servers show that we can safely perform a dynamic update within milliseconds when more straightforward alternatives would delay some updates indefinitely.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.4 [*Processors*]: Compilers; C.4 [*Performance of Systems*]: Reliability, availability, and serviceability

General Terms Languages, Performance, Reliability

Keywords dynamic software updating, update safety, update timeliness, multi-threading

1. Introduction

Continuous operation is a requirement of many of today's computer systems. Nonetheless, such systems must be updated to fix bugs and add new features. To permit on-line updates, many researchers have proposed variations of an approach called *dynamic software updating* (DSU). In this approach, a running program is patched with new code and data on the fly, while it runs. DSU is appealing because of its generality: in principle any program can be updated in a fine-grained way, without need for redundant hardware or special-purpose software architectures. Application state is naturally preserved between updated versions, so that current processing is not compromised or interrupted. General-purpose DSU systems have been shown to successfully support long strings of updates derived from actual releases of server applications [16, 5] and operating systems [3], while more specialized systems support smaller bug fixes or security patches [12, 1, 18].

The primary challenge in building a DSU system is balancing flexibility and safety: the system should support as many kinds of dynamic changes as possible, and it must provide means to ensure that an program update is well-timed, to avoid incorrect behavior [6]. To see why timing is important, consider that many programs change functions' type signatures as they evolve [14, 16]. If a patch changes function `f`'s type signature, applying the patch just before the program is about to call `f` would result in a type error since the caller still presumes `f`'s old signature. To avoid such problems, some DSU systems impose some automatic timing restrictions. For example, a system may require that updated functions not be running when a patch is applied [19, 3, 1, 2]. Since changing a function's type signature necessitates changing its callers, we can be sure, using this constraint, that no code will be running that presumes the old signature. Despite the reliance of some systems on purely automatic checks [19, 3, 1, 2], these are in general insufficient to ensure safety [6], so manual assistance may be needed to further constrain legal update times [8, 20].

Here lies a tension: timing constraints are needed to ensure that an update will be applied safely, but if the constraints are too strict, they may prevent the update from taking place at all. For example, if a patch may only be applied when changed functions are not active, a change to a function running an infinite loop will be delayed indefinitely. For single-threaded programs, the safety/timeliness trade-off can be managed with some care, e.g., by extracting infinite loops into separate functions that occasionally become inactive [16]. However, many long-running systems that could benefit from DSU are multi-threaded, and for them the situation is far worse. As explained further in Section 2, myriad interactions among threads complicates reasoning about safety, and may render approaches to control timing ineffective. For example, while a single thread may be sure to eventually exit an extracted loop body, there is no guarantee that all threads will eventually exit it at once to allow it to be updated [12].

This paper explores new ideas for balancing the tension between safety and timeliness when dynamically updating multi-threaded programs. We start with the idea that the programmer will specify

* Extended version of *Safe and Timely Dynamic Updates for Multi-threaded Programs* appearing in Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, June 15–20, 2009, Dublin, Ireland.

Σ, a handful of safe *update points*, perhaps one or two per thread, which are program locations at which an update may take place. In a multi-threaded setting, a dynamic patch π may be applied when all threads become *quiescent*, meaning they have each reached an update point $\ell \in \Sigma$ and that all automatic timing restrictions, such as activeness, have been satisfied. By keeping $|\Sigma|$ small, it is relatively straightforward for the programmer to reason that π will be applied correctly. We could implement this semantics by barrier-synchronizing all threads at update points ℓ, but doing so could degrade performance (while threads block, or even deadlock) and fail to achieve quiescence quickly because fewer update points implies greater delays between the times they are reached.

Therefore, to improve timeliness while retaining the same level of safety, we propose two novel concepts, described in Section 3: *induced update points* and *relaxed synchronization*. An induced update point for a given patch π is a program point $\ell_\pi \notin \Sigma$ such that applying π at ℓ_π is equivalent to having applied π at some $\ell \in \Sigma$. Thus, induced update points expand the points at which an update may take place without increasing the burden of reasoning about safety. However, expanding the number of possible update points reduces, but does not eliminate, the drawbacks of barrier synchronization. Therefore, we have also developed a technique we call *relaxed synchronization* that eliminates the need for threads to block at update points. The basic insight is the following. Suppose that a consecutive range of program statements $s_1, ..., s_n$ qualify as induced update points for Σ and π. Instead of potentially synchronizing at each s_i, a thread t "checks in" when it reaches s_1 to indicate that, until told otherwise, its execution is safe with respect to π. Thread t may then immediately proceed with executing $s_2, ...s_n$, "checking out" after executing s_n to indicate that applying π would no longer be safe. Once all threads have checked in, π can be applied safely. Check-in points often create more opportunities for updating (whole blocks of code, rather than particular statements) and allow threads to continue to execute until quiescence is reached, avoiding performance degradation and deadlock.

We have implemented these techniques as an extension to Ginseng, a freely available DSU framework for C programs (Section 4). We call our extended system STUMP (for *Safe and Timely Updates to Multi-threaded Programs*). We have used STUMP to dynamically update three open-source, multi-threaded server programs—the Icecast streaming server, the Memcached caching server, and the Space Tyrant game server—with updates that correspond to thirteen actual releases that span one year (Section 5). Experimental results in Section 6 show that induced update points and relaxed synchronization effectively balance safety and timeliness for these programs: with only a few update points (1 or 2 per thread) and a few other annotations, we could reach quiescence very quickly, typically in less than ten milliseconds. Without these mechanisms some patches failed to take effect at all, and others took hundreds of milliseconds or several seconds to apply. We also measured the impact of STUMP's update support on application performance and found that the slowdown for application-specific benchmarks is modest, less than 7% for all applications.

In summary, the main contributions of this paper are as follows:

- We introduce two techniques—induced update points and relaxed synchronization—for balancing safety and timeliness when dymamically updating multi-threaded programs.
- We implement these ideas in STUMP, a framework for dynamically updating C programs, and find that STUMP is effective in practice: all updates we considered can be successfully applied in short order, while overhead on application performance is small.

```
typedef struct event { int e_id ; ... } evt;
void process(evt* ev) {
  switch (ev→e_id) { case X: m(ev); ... case Y: n(ev );... };
  log(ev→e_id);
}
void clean(evt* ev) { ... }
void handle_thread() {
  while (1) {
    /* definite update point */
    evt* ev = get_event();
    /* candidate induced update point */
    process(ev);
    clean(ev);
} }
```

(a) Original version

```
typedef struct event { char* name; int e_id ; ... } evt;
void process(evt* ev) {
  switch (ev→e_id) { case X: m(ev); ... case Y: n(ev );... };
  /* no log() call */
}
void clean(evt* ev) { /* added: */ log(ev→e_id); ... }
/* handle_thread() as before */
```

(b) Changes in new version

Figure 1. Example program and update

2. Balancing safety and timeliness

Given run-time support for on-line program updates, we must understand how to use this support safely, and yet in such a manner that an update can be applied in a timely fashion. This section explains why balancing these tensions is difficult, particularly for multi-threaded programs.

2.1 Example

Figure 1(a) depicts a simple server in which n threads execute the `handle_thread` function to process events extracted from a global queue (via the potentially blocking call `get_event`, code not shown). Events are values of the type `evt`, which contains an `e_id` field (among others not shown) for identifying the event's type. The `process` function is called to actually handle each event, dispatching to other functions based on the `e_id` field, and logging the result when it completes (via the `log` function). The `clean` function performs further post-processing. (The comments in `handle_thread` will become clear later.)

Figure 1(b) shows a sample update to this program, which changes the definition of the `evt` type to add a new field and changes the implementations of the `process` and `clean` functions. Notice that the `log` function has been moved from the end of the `process` function in the old version to the beginning of the `clean` function in the new version.

Several existing DSU systems could support this update or one like it, including POLUS [5], Ginseng [16], DLpop [8], Jvolve [21], K42 [3, 9], and UpStare [11]. While the mechanisms differ between systems, essentially the definitions in Figure 1(b) would be gathered into a *dynamic patch* along with a *type transformer* function to convert values whose type definition has changed.[1] In our example, the dynamic patch would contain a type transformer for **struct** `event`. This function might initialize the new `name` field to `"none"` while preserving the contents of the other (unchanged) fields.

[1] A type transformer's application could occur when the update is applied [8, 21] or as needed during execution [16, 5, 3].

2.2 Safe patch application

Given a dynamic patch, the next question is: when during the original program's execution can it be applied safely? Most DSU systems apply a patch according to a combination of manually specified and automatically determined timing constraints.

A typical automatically determined constraint is that functions changed by a patch may not be running when the patch is applied [19, 3, 1, 2, 21]. To see why such a constraint might be needed, suppose our update was to be applied just before the call to log in the process function on line 4 of the old program (when an active function is updated, the old version continues to run and the new version takes effect on the next call [5, 8, 16]). Therefore, in the old process code there is a subtle problem: this function has been compiled to assume that e_id is the first field of evt, but now that evt has been updated, it is the second field. Therefore, this old code will mistakenly access **char** *name as if it were the **int** e_id, a type error. The activeness constraint would prevent this problem, and indeed ensures type safety so long as the new program version, compiled from scratch, is itself type-correct [23, 20].

We may be tempted to believe that "activeness" checking is sufficient to ensure an update will be applied correctly, and indeed this presumption is made by some existing systems [2, 3]. Unfortunately, Gupta [6] has shown that automatically determining a valid time at which to apply a dynamic patch is in general undecidable. Indeed, we can see why activeness is insufficient in our example. Suppose that the update were to take effect just after the call to process(ev) on line 12 of handle_thread. If there is only a single thread running handle_thread, then no changed code is active. The next call is to the *new* clean function, whose first action is to call log. But this is probably not what we wanted: just prior to the update, the old process function would have called log (on line 4) for this same event; so this update timing has precipitated a redundant log entry.

Given that automatic timing constraints are insufficient, many DSU systems allow the programmer to assist in controlling update timing. There are two basic approaches: identify a *whitelist* of program locations, termed *update points*, that are valid for an update [8, 16, 11]; or specify a *blacklist* of functions that must be inactive prior to updating [10, 6, 5].

With such manual controls in hand, prior work has explored applying updates at "quiescent points" in a program's execution [19, 3, 16]. For example, a quiescent point could be just prior to a complete iteration of an event processing loop. Intuitively, writing correct state transformation code to take effect at quiescent points is relatively easy because the program is not in the middle of some high-level activity, and invariants concerning global data structures are clear. In our example, we could imagine requiring the update to take place at the start of handle_thread's loop on line 9, since this is when it is about to begin its basic high-level activities.

2.3 Timely patch application

Unfortunately, while timing restrictions are clearly necessary for ensuring safety, they can significantly delay when a patch is actually applied. Indeed, the activeness restriction precludes patch application indefinitely if a changed function is *always* active, as is the handle_thread function in our example. We can solve this particular problem by extracting the bodies of infinite loops into separate functions which become inactive on each iteration [16]. With such support (or support for more fine-grained updates to active code [11]), a single quiescent point often suffices to assure timeliness for single-threaded programs, e.g., because events are processed relatively quickly.

Unfortunately, for multi-threaded programs, the task of choosing suitable update points is much more difficult because the state of *all* threads must be considered when choosing appropriate times. One approach would be to identify a small number of safe update points for each thread (e.g., just one or two) and apply an update only when all threads have reached safe update points. In our example, we would apply the update only when *all* threads running handle_thread have completed a loop iteration. This approach eases reasoning about patch correctness because only a few system states need to be considered. On the other hand, this approach could seriously compromise timeliness, since it may be very difficult or unlikely for all threads to reach update points at once. We could improve the chances by expanding the number of update points per thread, but doing so complicates the programmer's reasoning that a patch is correct, and may lead to problems such as those we have described. Overall, the number of system states that must be considered will be n^m where m is the number of threads, and n is the number of update points per thread.

Thus we must find some way to balance the need to easily reason about a patch's correctness while not unduly delaying the time at which it can be applied.

3. Safe and timely dynamic updates

This section describes how we can balance safety and timeliness when applying a dynamic patch. We use the following programming model. To update a program P, the programmer provides an *update specification* U, which is a pair (π, Σ) where π is a dynamic patch and Σ is a set of *update points*. The patch π is a map from variables to new or updated definitions (or the type transformer, for changed types). We write $changes(\pi)$ to denote the names of functions, variables and type definitions changed by the patch. Each update point in Σ is simply a location (i.e., line number) ℓ in the program. The intended semantics is that π should be applied when the program counter of each thread t in P has reached an update point $\ell \in \Sigma$, and any automatic safety checks (e.g., activeness) are satisfied. Our goal is to apply the patch as quickly as possible.

3.1 Simple approach: barrier synchronization

Once U becomes available, a simple approach to applying it safely is to treat each update point $\ell \in \Sigma$ as a barrier and block any thread that reaches it. Once all threads have synchronized, the patch is applied, and the threads are resumed.

While pleasingly simple, the barrier approach has two main drawbacks. First, there is the possibility of deadlock. For example, if thread t blocks at some point ℓ while holding a lock, then any other thread attempting to acquire the lock will never make progress toward reaching some $\ell \in \Sigma$. The second problem is related: even if all threads eventually synchronize, application performance may suffer in the meantime. For example, suppose thread t_1 is responsible for accepting new connections while t_2 performs per-connection event processing. If t_1 reaches point ℓ fairly quickly, but t_2 takes much longer to reach point ℓ', say because it must complete a lengthy transaction first, then the entire system will be prevented from accepting new connections, and performance will suffer. In the limit, the barrier approach may needlessly block threads that are completely unaffected by the changes in a patch π.

Our way forward is to observe that it does not matter if the update takes place when all threads are *actually* at points in Σ. Rather, we use induced update points (described in Section 3.2) and relaxed synchronization (Section 3.3) to apply the patch π if the effect of the update will be *semantically equivalent* to having applied the patch when the PC of each thread is at one of the update points $\ell \in \Sigma$. This expands the number of program points that permit a patch to take effect, improving timeliness, without increasing the reasoning burden on the programmer.

Lines	Trace 1		Trace 2		Trace 3	
9 (def-upd)			✓		×	
10	g	$\{1\}$	g	$\{1,2\}$	g	$\{1\}$
11	update g		update p,c		update g,p	
12	p	$\{1,2\}$	p	$\{2\}$	p	$\{2\}$
3	e,m	$\{1,2\}$	e,m	$\{1,2\}$	e,m	$\{1,2\}$
4	l,e	$\{1,2\}$				
13	c	$\{1,2\}$	c	$\{2\}$	c	$\{1,2\}$
6			l,e	$\{1,2\}$		
9 (def-upd)	✓				×	
	(a) Roll-forward		(b) Rollback		(c) Disallowed	

Figure 2. Examples of legal and illegal induced updates.

3.2 Induced update points

Given a specification $U = (\pi, \Sigma)$, an *induced update point* is a program point ℓ' such that if π is applied when a thread reaches ℓ', the program will behave as if the patch had been applied at some update point $\ell \in \Sigma$. Given a set of *candidate update points* ξ, our task is to determine, via static analysis, which of those $\ell \in \xi$ that meet this criterion. In our Ginseng implementation (Section 4), we require programmers to specify ξ, but here we make no assumptions about how ξ is generated.

3.2.1 Version-consistent traces

The key concept we will use is the notion of a *version-consistent execution trace*. This idea is illustrated in Figure 2. Each of the right three columns represents an execution trace of the handle_thread function from Figure 1(a), starting at line 9 and iterating once around the loop. Each element of the trace consists of an identifier followed by a set of versions. Identifiers can be functions (indicating the function was called), global variables (indicating a read or write), and type names (indicating a variable with that type was read or written). We abbreviate the longer identifiers used in Figure 1(a) with just their first letter in the trace, e.g., g stands for a call to get_event, and e stands for a read/write from an evt value. The *version set* indicates the version of an identifier at the point where that identifier is used. Accesses to variables changed by π will always have a version set containing a single element, $\{1\}$ before the patch is applied, and $\{2\}$ afterward. When a variable is *not* changed by the patch its definition is the same in both program versions. Hence its version set is $\{1, 2\}$.

A trace also contains *update events* "update $changes(\pi)$" to indicate patch π has been dynamically applied. All the specifications (π, Σ) we consider have $\Sigma = \{9\}$; that is, the patch π may only appear to take effect at line 9. We refer to this as the *definite* update point, since it defines what the programmer believes will result in correct behavior (line 9 is annotated in the first column for reference). If given $\xi = \{11\}$, then we must determine whether applying π at line 11 would have the same effect as updating at line 9.

Figure 2(a) considers the case when $changes(\pi) = \{g\}$; that is, it contains a changed version of function get_event. In this case, 11 is indeed an induced update point: the execution of the program is equivalent to having performed the update at line 9, at the end of the trace marked with a ✓. To see why, consider the version set of each function that was called. In the trace in Figure 2(a), we can see that all version sets include version 1, the old version, and thus the program will behave just as if we "rolled forward" the program to line 14 before performing the update.

Figure 2(b) considers $changes(\pi) = \{p, c\}$. In this case, 11 is still an induced update point, but now the effect of the program is equivalent to having performed the update at line 9 at the start of the trace. This is because all accesses can be attributed to version 2, the new version, just as they would be if we had rolled back the program to perform the update at line 9.

Figure 2(c) considers $changes(\pi) = \{g, p\}$. In this case there is no program version that all accesses share, so 11 is not valid.

3.2.2 Version consistency via static analysis

To predict whether a patch π applied at some $\ell \in \xi$ will induce a version-consistent trace, we can statically analyze the program to approximate all relevant behavior that could occur in an execution including ℓ. In particular, we wish to find a static approximation of the execution behavior from any preceding definite update point to ℓ, which we shall call the *prior effect* α_ℓ, and an approximation of the execution from ℓ to the first occurrence of a definite update point, which we call the *future effect* ω_ℓ [15]. For determining version consistency, we are interested in those events we considered in our traces above: calls to functions, reads/writes from/to global variables, and accesses to values of named type. We are not interested in the number or order of these events, but only in the names of the functions, variables, or types involved. Thus a prior/future effect at ℓ can be characterized as the set of all definition names involved in events that could occur before/after ℓ.

For $\ell = 11$ in our example program we have $\alpha_\ell = \{g\}$ and $\omega_\ell = \{p, e, m, n, l, c\}$. The reason for the former is easy to see: before reaching line 11 on an execution starting from line 9, the only interesting event is the call to get_event. For the latter, we can see that before again reaching line 9, the program will execute process and clean, and these functions may themselves access values of evt type and call functions log, m, or n. On the other hand g is not included because get_event will not be called before line 9 is reached. We will discuss how these effects are computed in the next subsection.

We can determine whether applying a patch π at some $\ell \in \xi$ will result in a version-consistent trace using contextual effects. For each ℓ, we compute its prior and future effects α_ℓ and ω_ℓ, respectively. If $changes(\pi) \cap \omega_\ell = \emptyset$ we know that all definitions possibly accessed after ℓ up to the next definite update point can be attributed to the old version since they are unaffected by π. Thus ℓ can be considered a roll-forward induced update point. This condition holds for $changes(\pi) = \{g\}$ in our example, as shown in Figure 2(a). On the other hand, if $changes(\pi) \cap \alpha_\ell = \emptyset$, then the patch π has not modified any variables possibly accessed since the last definite update point. In this case, these accesses can also be attributed to the new version, and thus ℓ can be considered a rollback induced update point. This condition holds for $changes(\pi) = \{p, c\}$ in our example, as shown in Figure 2(b). Neither condition holds for $changes(\pi) = \{g, p\}$, so for this particular patch ℓ is not valid, as shown in Figure 2(c).

3.2.3 Contextual effects

Now we explain how to compute prior and future effects for the purpose of determining induced update points. Our approach is to use a generalization of *effect inference* [22] called *contextual effect inference* [15].

In a traditional effect system, the *effect* ε of some program expression e characterizes an aspect of e's non-functional behavior, for example the names of locks e allocates [17], or the abstract names of memory locations e dereferences [22]. For enforcing version consistency our notion of effect is as we just described: a set containing the names of functions that are called, global variables that are read or written (and likewise, static names for dereferenced pointers, acquired from a points-to analysis), and types whose instances—whether local or global—are read or written.

The contextual effect Φ of an expression e consists of a triple $[\alpha; \varepsilon; \omega]$, where ε is the normal effect of e; α is the *prior effect*, which characterizes the computation since the last definite update point up to (but not including) e; and ω is the *future effect*, which characterizes the computation following e, up until the next definite update point. We compute contextual effects using a constraint-based analysis that we express as a series of inference rules. In our prior paper [15], we proved that, for single-threaded programs, valid induced update points produce version-consistent executions when prior and future effects are computed with our contextual effects analysis. This basic proof is extended to multi-threaded programs in the first author's dissertation [13]. We give a flavor of the analysis here, and refer the interested reader to our prior work for details [15].

Suppose we have two statements in sequence $s_1; s_2$ which have contextual effects Φ_1 and Φ_2, respectively. Then the contextual effect Φ of the two statements in sequence is according to the judgment $\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi$ defined as follows:

$$\text{XFlow-Ctxt}\ \frac{\begin{array}{c}\Phi_1 = [\alpha_1; \varepsilon_1; (\varepsilon_2 \cup \omega_2)] \\ \Phi_2 = [(\varepsilon_1 \cup \alpha_1); \varepsilon_2; \omega_2] \\ \Phi = [\alpha_1; (\varepsilon_1 \cup \varepsilon_2); \omega_2]\end{array}}{\Phi_1 \rhd \Phi_2 \hookrightarrow \Phi}$$

There are three key elements of this rule. First, if ε_1 is the normal effect of s_1, then because s_1 precedes s_2, ε_1 must be included in the prior effect of Φ_2. Conversely, if ε_2 is the normal effect of s_2, then because s_2 follows s_1, ε_2 must be included in the future effect of Φ_1. Finally, the contextual effect of the sequence $s_1; s_2$ has the prior effect of Φ_1, the future effect of Φ_2, and its normal effect is the union of the normal effects of the two statements.

The Xflow-Ctxt rule is essentially constraining the form of prior and future effects according to the normal effects of each statement. Thus we can use standard techniques to generate per-statement effects ε and generate additional constraints for the prior and future effects α and ω [15].

The effects α and ω are also constrained by the placement of definite update points. In the general case, there could be many definite update points reachable from a given statement s. Likewise, several definite update points could reach s, and these could be in the same function as s or in its function's callers or callees. While implementing contextual effects for this general case is possible, doing so is unnecessarily complicated and computationally expensive. Therefore, we employ a simpler model, described next.

As an alternative to Σ, a set of definite update points, we define $\hat{\Sigma}$ to be a set of *pairs* of definite update points. When computing the prior effect α at ℓ, instead of finding all possible update points in Σ that could reach ℓ, we only consider a single point ℓ_1 such that $(\ell_1, \ell_2) \in \hat{\Sigma}$ and (ℓ_1, ℓ_2) defines a lexical scope that encloses ℓ. By "enclose," we mean that ℓ could literally occur in between ℓ_1 and ℓ_2 in the program text, or it could be in a function called, directly or transitively, by a statement between ℓ_1 and ℓ_2. The future effect ω of ℓ is analogously computed with respect to ℓ_2. Since pairs (ℓ_1, ℓ_2) define a lexical scope, we call them *update scopes.* To be sound, there can be at most one update scope that encloses a given ℓ, meaning that update scopes may neither nest nor overlap. Moreover, only candidate update points $\ell \in \xi$ enclosed within scopes $(\ell_1, \ell_2) \in \hat{\Sigma}$ are permitted. With these restrictions, we compute contextual effects for the whole program as follows:

- For each block of statements $\ell_1 : s_1; ...; \ell_2 : s_2$ where $(\ell_1, \ell_2) \in \hat{\Sigma}$ we compute the contextual effects of each statement in that scope, constraining the prior effect α_1 of s_1 and the future effect ω_n of s_n to be the empty set, thus delimiting the extent of prior and future effects computed within the scope. We will also compute the contextual effects of functions called by $s_1 ... s_n$.

```
 1  void process(evt* ev) {              α   ;    ε    ;    ω
 2    switch (ev→e_id) {              {g}          {e}  {e,m,n,l,c}
 3      case X: m(ev); ...            {g,e}      {m,e}      {e,l,c}
 4      case Y: n(ev); ...            {g,e}       {n,e}      {e,l,c}
 5    }; log(ev→e_id);                {g,e,m,n} {l,e}            {c}
 6  }
 7  void handle_thread() {
 8    while (1) {
 9      /* ℓ1 */
10      evt* ev = get_event();        {}    {g}       {p,e,m,n,l,c}
11      /* ind. upd. pt. */           {g}   {}        {p,e,m,n,l,c}
12      process(ev);                  {g}   {p,e,m,n,l}          {c}
13      clean(ev);                    {g,p,e,m,n,l} {c}           {}
14      /* ℓ2 */ }
15  }
```

Figure 3. Contextual effects for example in Fig. 1(a).

- We cannot allow induced update points in functions, such as those in libraries, that could be called from both within and outside an update scope. Therefore we add the additional constraint that the prior and future effects of each thread function (i.e., main or functions passed to pthread_create) include the set of all possible events, written $\top$. Thus any candidate update point in a function called from outside an update scope will have prior and future effect $\top$, effectively precluding an induced update point. On the other hand, functions called only from within update scopes will not be so restricted, since the effects of scopes are computed independently of the functions they reside in. Note there is no problem with a function called from within multiple (non-nested) update scopes—the prior and future effects in this function will naturally approximate the limits imposed by all enclosing scopes.
- Finally, we rule out nested update scopes during inference by not bounding the prior and future effects of update scopes with the empty set, as stated above, but rather with a marker set $\{K\}$. Then we check that the statements immediately outside each update scope do not have K in their prior or future effects.

Figure 3 shows the contextual effects computed for process and handle_thread from our example program (Figure 1(a)), modified to use an update scope (ℓ_1, ℓ_2) shown in comments. This scope is essentially the same as the single definite update point we used before, and the prior and future effects of the candidate update point at line 11 are as indicated in the previous subsection. There are a few other things to notice. First, notice that line 12's normal effect includes the call to process and the effect of executing its body. Second, notice that the prior and future effects within process properly include the events that take place in its caller, handle_thread. Finally, notice the handling of the **switch** statement in process: the prior/future effects of the code in each of the cases assumes that the other branches are not executed (e.g., n is not in the prior/future effect at line 3, and m is not in the prior/future effect at line 4), but the prior/future effect of the code that precedes or follows the **switch** as a whole conservatively presumes that all branches were executed (and hence both m and n are mentioned in the effects).

3.3 Relaxed synchronization

While induced update points increase a thread's opportunities for applying a dynamic update, they still force the thread to block, and thus do not eliminate the potential for degraded performance and deadlock. Therefore we have developed a second technique we call *relaxed synchronization* that avoids the need to block at update points while still greatly improving the chances that an update will be applied quickly.

We now present the idea behind relaxed synchronization. Suppose we have a sequence of statements $s_1 ... s_n$ at locations $\ell_1 ... \ell_n$, respectively, that execute within some update scope. Further suppose each of the ℓ_i in $\ell_1 ... \ell_n$ is a valid induced update point for patch π, which implies that π could be safely applied while a thread is executing any of $s_1 ... s_n$. As an optimization, then, there is no need to block when a thread first reaches ℓ_1. Instead, the thread can "check in" with the run-time system to indicate it has reached a point at which it is safe to apply π. Then it may continue its execution. When the thread reaches ℓ_n it must inform the run-time system it is no longer safe to perform the update; thus it "checks out," and again resumes execution. When all threads have checked in, the update may commence. If the update is still taking place when a thread reaches a check-out point, it simply waits for the update to complete.

We can implement this idea as follows. For each $\ell \in \xi$ enclosed in some update scope $(\ell_1, \ell_2) \in \hat{\Sigma}$, we will determine whether ℓ is a legal check-in point (otherwise, it is a check-out point). Let L denote the set of all $\ell' \in \xi$ reachable from an execution starting at ℓ. Let L_S denote the set of locations ℓ_i of all statements s_i that, starting from ℓ, could be executed prior to reaching some $\ell' \in L \cup \{\ell_2\}$. Then ℓ is a valid check-in point if all $\ell_i \in L_S$ are valid induced update points. Otherwise, ℓ must be considered a check-out point.

In the prior subsection we argued that it would be difficult to compute contextual effects if we had to determine all definite update points that could be reached after executing a given definite update point. Similarly, it would be difficult to determine the sets L and L_S, above. We simplify the problem in the same way: rather than specify individual candidate update points $\ell \in \xi$, we specify candidate update *scopes* $(\ell_(, \ell_)) \in \hat{\xi}$, with the interpretation that $\ell_($ is a potential check-in point, and $\ell_)$ is a check-out point. Since the two form a lexical scope, the statements $s_1; ...; s_n$ between them are readily apparent. Moreover, to check that each of these statements is a valid induced update point, if ε is the normal effect of the block $s_1; ...; s_n$ (which we already compute as a matter of course), then it suffices to compute $\bigcup_{1 \le i \le n} \alpha_i \equiv \alpha_1 \cup \varepsilon$ and $\bigcup_{1 \le i \le n} \omega_i \equiv \omega_n \cup \varepsilon \equiv \omega_1$. We call the former the *check-in prior effect* of $\ell_($ and the latter the *check-in future effect* of $\ell_($. For each $(\ell_(, \ell_))$ pair for which our validity check using prior and future check-in effects is satisfied, at run-time the thread will check in when reaching $\ell_($ and check out at $\ell_)$. Those pairs for which this check is not satisfied will have no run-time effect.

Returning to our example (Figure 3), consider (11,14) as a candidate scope. The normal effect ε of the statements process(ev); clean(ev) in this scope is $\{p, e, m, n, l, c\}$. Thus the prior check-in effect is $\{g\} \cup \{p, e, m, n, l, c\} = \{g, p, e, m, n, l, c\}$ and the future check-in effect is $\{\} \cup \{p, e, m, n, l, c\} = \{p, e, m, n, l, c\}$. So 11 would be a valid check-in point for π where $changes(\pi) = \{g\}$ since this does not conflict with the future check-in effect, but would not be considered valid for π where $changes(\pi) = \{p\}$ since p appears in both the prior and future check-in effects. Note the clear tradeoff here. The larger the check-in scope, the larger the check-in effects and the less likely a check-in point will be valid. On the other hand, the smaller the check-in scope, the less likely that all threads will be simultaneously executing in valid scopes when an update is available. As we show with our experimental results in Section 6, we find a few well-chosen check-ins and relaxed synchronization to be largely beneficial.

A more complicated variation of relaxed synchronization is proved sound in the first author's dissertation [13]; we believe it would be straightforward to extend the argument there to the present system.

4. Implementation

We have implemented induced update points and relaxed synchronization as extensions to Ginseng v1.2.2,[2] a DSU compiler for single-threaded C programs [16]. We call our extended version STUMP (for *Safe and Timely Updates to Multi-threaded Programs*). We believe our techniques could be implemented for other DSU systems as well.

4.1 Background: DSU in Ginseng

In Ginseng, a dynamic patch π consists of definitions—functions, types, or global variables—that have been added or changed since the last (deployed) program version. The dynamic patch is compiled into a shared object file and then loaded into the specially compiled running program to take effect. In an updatable program, all direct function calls are compiled to be indirected through function pointers, and after the patch is loaded, the targets of the function pointers are redirected to the newly loaded versions.

To support changes to type definitions, Ginseng compiles all accesses to typed values to be through *concretization functions*. For example, if p has type **struct** T∗, then Ginseng will compile the source-program expression p→x to be __con_struct_T(p)→x instead. This function will examine the contents of its argument to see whether it has been updated to the new version of type **struct** T, and if so, it executes a user-defined *type transformer* function to bring it up to date. Ginseng uses a simple compilation strategy to make this check, and the transformation, possible: all values of a named type that are updatable are given an extra version field and padded to permit future growth.

The programmer may also define a *state transformer* function ST. Function ST() is called at the time the update is applied and contains code needed to set up the state of the new program, e.g., to initialize the contents of newly created global variables, or to make system calls that normally occur in the new version of main when the program is started from scratch.

Controlling update timing. To specify the time at which a dynamic patch is applied, Ginseng requires the programmer to insert explicit calls to the function DSU_update at those program points at which a dynamic patch π may take effect. Thus, these calls indicate definite update points Σ, though Ginseng in effect forces the programmer to define Σ at deployment time, before the next patch to P is known. We further discuss this requirement in Section 4.3.

When the program calls DSU_update and a patch is available, some safety checks are performed before the patch is applied. Prior to deployment, a static *updatability analysis* [20] of the program determines, for each DSU_update call, a set Δ of variables and type names that may *not* be changed by a future π if applied at that point, to ensure type-safety. Then the DSU_update call is compiled to pass a representation of Δ to the run-time system, which will enforce $\Delta \cap changes(\pi) = \emptyset$ before applying the update.[3] This check essentially takes the place of the "activeness check" described in Section 2.2, but admits some updates to active code.

Code extraction. Ginseng's compilation strategy ensures that an updated function will be used the next time it is called. This creates a problem for functions that run indefinitely, as described in Section 2.3. To cope with this, Ginseng provides a *code extraction* mechanism that excises a programmer-indicated code block into its own function. The boundaries of extracted functions in the old version effectively designate points at which an update could take place *within a function*, and the boundaries of the same extracted

[2] Available at http://www.cs.umd.edu/projects/PL/dsu/.

[3] The implementation uses *set id*s as arguments, rather than sets, to keep this operation fast regardless of set size.

functions indicate the corresponding code to execute (i.e., to where to map the PC) in the new version.

4.2 STUMP extensions

We made several extensions to Ginseng to support induced update points and relaxed synchronization, and also extended parts of its run-time system and compiler to ensure thread-safety. As with definite update points in Ginseng, update scopes ($\hat{\Sigma}$) and candidate update points/scopes ($\xi/\hat{\xi}$) are specified when a program is deployed, before a particular patch π is known.

Definite update points and update scopes. To designate the sequence $s_1; ...; s_n$ as an update scope $(\ell_1, \ell_n) \in \hat{\Sigma}$, the programmer explicitly labels the sequence as UPDATE_SCOPE:$\{s_1; ...; s_n\}$. Thus, update scopes are specified at deployment time, rather than patch time, replacing analogous calls to DSU_update in standard Ginseng. The compiler uses such scopes when computing contextual effects, and then inserts calls to DSU_update at the beginning and end of the scope. As usual, these calls are passed the set Δ generated by the standard updatability analysis.

Induced update points and barrier synchronization. The programmer specifies candidate induced update points ξ by including explicit calls to function DSU_induced_update in the deployed program. A contextual effects analysis infers α and ω, and the updatability analysis infers Δ, for these program points; the compiler modifies the calls to pass a representation of these sets. Once a dynamic patch π is available and thread i calls DSU_induced_update the run-time system will check whether $(\underline{changes}(\pi) \cap \alpha_i = \emptyset \vee \underline{changes}(\pi) \cap \omega_i = \emptyset) \wedge (changes(\pi) \cap \Delta_i = \emptyset)$. (We discuss $\underline{changes}$ below.) If this check succeeds, the current update point is compatible with the update and the thread is blocked. Calls to DSU_update will block so long as $changes(\pi) \cap \Delta_i = \emptyset$. Once all threads have blocked, the update may proceed.

We define $\underline{changes}(\pi) = changes(\pi) \cup writes(\mathsf{ST})$, where $writes(\mathsf{ST})$ is the set of locations (determined by points-to analysis) that could be read or written by executing state transformer ST. This effect must be included in the safety check to ensure version consistency. For example, if not accounted for, the code prior to an induced update point could read global variable g, function ST could write to it, and then subsequently executed code in the update scope could read g again, thus seeing the new value, violating version consistency. A similar problem could occur with the execution of ST if code in the update scope prior to and following ST's execution could write to a variable read by ST, unmasking the illusion that ST is only executed at the start or end of an update scope.

While we properly account for such changes to the heap, we do not account for interactions with the environment outside the process. For example, if ST writes to the file system and code in an update scope could read from the same file location before and after an induced update point, then an update at that point could violate version consistency. While we could imagine tracking I/O effects to avoid this situation, in practice we have never needed to write a state transformer that changes the external environment in a manner visible to existing code.

Check-ins and relaxed synchronization. When using relaxed synchronization, we designate check-in scopes $(\ell_(, \ell_))$ by labeling a code block as CHECKIN:$\{s_1; ...; s_n\}$. The compiler inserts a call to the function DSU_checkin at the beginning of a check-in block, and the computed check-in effect representations (and a similarly adjusted-for-checkin Δ set) are passed as arguments. In particular, if the normal effect of $s_1; ...; s_n$ is ε, and contextual/updatability effects for s_1 and s_n are $(\alpha_1, \omega_1, \Delta_1)$ and $(\alpha_n, \omega_n, \Delta_n)$, respectively, the call inserted by the compiler will be DSU_checkin$(\alpha_1 \cup \varepsilon, \omega_n \cup \varepsilon, \Delta_1)$.

Generally speaking we should insert a call to check out at the end of a check-in scope, but we avoid doing this in our experiments by making check-in blocks back-to-back and non-nested, so that the code range covered by one is immediately followed by another. Otherwise we would need a stack of triples per thread, where the topmost element represents the thread's current restriction, a check-in pushes the given triple on the stack, and a check-out pops it off; the stack is initialized to $(\top, \top, \top)$. With relaxed synchronization, inserted DSU_update calls no longer have run-time effect—rather than modify them to include check-in effects we simply ignore them and in practice place a CHECKIN annotation on the first block within the update scope.

The pseudocode for the relaxed synchronization protocol is shown in Figure 4. The restriction array (line 6) is indexed by a (normalized) thread identifier, and contains the arguments passed to the most recent DSU_checkin call, i.e., a triple of set IDs for the prior and future check-in effects α and ω, and the capability Δ. When no update is in progress, threads may change restriction in parallel because restriction_mutex is acquired in reader mode. This is safe because each thread will only write to its own portion of the array. When a patch π becomes available, the flag update_requested (line 10) is set to 1, and the sets $changes(\pi)$ and $\underline{changes}(\pi)$ are populated. The next thread that checks in and acquires the update_mutex will attempt to apply the update (lines 35–36). This updating thread will then acquire the restriction_mutex in writer mode (line 37), and thus other threads will block at their next check-in points (line 31). Next, the updating thread checks whether the update contents conflict with the current per-thread restrictions; the call to conflicts checks that for each thread i, $(\underline{changes}(\pi) \cap \alpha_i = \emptyset \vee \underline{changes}(\pi) \cap \omega_i = \emptyset) \wedge (changes(\pi) \cap \Delta_i = \emptyset)$, where restriction [i] $= \{\alpha_i, \omega_i, \Delta_i\}$. If this check succeeds, apply_update proceeds with the update, redirecting function pointers, installing type transformers, and calling ST, as described above. Either way, the updating thread releases the restriction_mutex to unblock any checked-in threads, and releases the update_mutex to enable future updates (or retries).

Supporting concurrency. Ginseng's type wrapping changes the representation of updatable named types, so we must be careful not to introduce races. Since con functions can potentially call the type transformer to update a value to the current version, a race-free read–read access can become a racing write–write access. To avoid this problem, we changed con functions to use per-type locks to ensure atomic type transformation, and used double-checked locking to speed up the version check. However, introducing locks creates the potential for deadlock; e.g., a type transformer could call a function that tries to acquire an application lock while another thread holding that lock invokes the same type transformer. The problem can be avoided by writing type transformers that never call functions (including other type transformers) that could acquire locks; adhering to this restriction was easy for all our test applications. (Another way to avoid this problem is to convert data eagerly, at update time, an approach we might consider in future work.)

4.3 Discussion

Ginseng requires the programmer to choose update points Σ at deployment time, and STUMP likewise requires a deployment-time choice of update scopes $\hat{\Sigma}$ and check-in scopes $\hat{\xi}$.[4] In our experience, the choice of Σ (for single-threaded applications [16]) or $\hat{\Sigma}$ (for multi-threaded applications) is relatively clear and applies for all versions. As such, the deployment-time restriction imposes no practical limitation. On the other hand, delaying the choice of $\hat{\xi}$

[4] We do not specifically consider induced update points ξ in this discussion; they are analogous to check-in scopes.

```
typedef struct {
  set α; set ω; set Δ;
} thread_restr ;


 thread_restr   restriction []; // per−thread check−ins
 rwlock restriction_mutex ;


 volatile bool update_requested=0; // patch available
set $changes$; // elements changed by the patch
set $\underline{changes}$; // $changes \cup writes(ST)$
 mutex update_mutex; // synchronizes patch application


set update_contents; // elements changed by the update

bool conflicts ( rst [], u)
{
  bool res = false;
  for (i = 0; i < n_threads; i++)
    if ( rst [ i ]. Δ ∩ u ≠ ∅ ||
         ( rst [ i ]. α ∩ u ≠ ∅ &&
          rst [ i ]. ω ∩ u ≠ ∅))
    { res = true; break; }

  return res;
}
void DSU_checkin(α, ω, Δ)
{
  read_lock( restriction_mutex );
  restriction [ thread_self ()] = {α, ω, Δ};
 unlock( restriction_mutex );

  if (update_requested) {
    if ( trylock (update_mutex) == OK) {
      write_lock ( restriction_mutex );
      if (! conflicts ( restriction , $changes$, $\underline{changes}$)) {
        apply_update ();
        update_requested = 0;
      } // else constraint unsatisfied ; defer
      unlock( restriction_mutex );
      unlock(update_mutex);
    }
} }
```

Figure 4. Run-time system implementation for check-in based relaxed synchronization.

until the patch π is known could reduce update-related delays. In particular, we could maximize the extent and number of check-in scopes in $\hat{\xi}$ by computing contextual effects for every statement in the program, and designating update scopes for each sequence of statements that are safe with respect to π. The problem is that an update-time choice of $\hat{\xi}$ is difficult to implement efficiently. Compiling the program to insert check-in/check-out stubs, where some subset of them is enabled when the patch is known, would add significant bloat. Shepherding each thread's execution, e.g., using *ptrace*, also seems fairly heavyweight.

On the other hand, deploying with just definite update scopes and a few check-in blocks opens the possibility of deploying more check-ins later, if needed. The programmer could compute the maximal set $\hat{\xi}$ of check-in scopes relative to the given patch, adjust the original program source to use this set, and then update the deployed program with the new check-ins. Since the only difference in the two programs is the presence of check-ins, which have no impact on the normal semantics of the program, we could apply this patch piecemeal (e.g., one function at a time) to reduce possible conflicts. Once the check-in patch is deployed, the actual patch π can be applied with maximum effectiveness. A similar trick can be played to accommodate further code extractions, if necessary.

5. Experience

We used STUMP to dynamically update three open-source multi-threaded programs: the Icecast streaming media server, Memcached, a high-performance, distributed-memory object caching system, and the Space Tyrant multi-player gaming server. We chose these programs because they are long-running, maintain soft state that could be usefully preserved across patches, employ a variety of multi-threaded programming patterns, and spawn a non-trivial number of threads. In the remainder of this section, we briefly present each program and its threading model, then we describe the evolution of these programs during the period we considered, and finally discuss changes we made to prepare the programs for compilation with STUMP.

5.1 Application overview

Icecast is a streaming media server—a popular solution for building Internet radio stations. Updating Icecast on the fly would enable media content providers to keep their streams live 24/7, yet be protected with the latest security fixes, or supporting the newest features. Icecast employs an event-based server model with a fixed number of threads, each performing separate duties: accepting a connection, handling incoming connections, reading from a media source, keeping statistics, etc.

Memcached is a high-performance, distributed-memory object caching system used on high-traffic sites such as YouTube, Facebook, and Wikipedia to store and deliver pre-rendered Web content, avoiding slow per-client database accesses and on-demand rendering. Updating Memcached on the fly would help maintain high web server throughput; taking Memcached down to install the next version would discard the in-memory cache and cause degraded operation while the restarted program's cache refills. Memcached uses a homogeneous threading model, where all application threads (a user-configurable number) perform the same fixed task.

Space Tyrant is a multi-threaded gaming server. It uses a mixed threading model: three fixed threads (for managing the game state, accepting new connections and performing backups) and two threads for each client, one dealing with user input, one dealing with output from the server to the client. On-the-fly updates to Space Tyrant would enable continuous game server operation, without having to disconnect clients for each update.

Evolution history. Table 1 summarizes the release information and shows some of the ways the programs changed over time. The first two groups of columns describe the first and last release we

Program	**Updates**	**First release**			**Last release**			**Function changes**		**Type changes**	**Global var. changes**	
		Ver.	Date	Size (LOC)	Ver.	Date	Size (LOC)	Proto	Body		Init	Type
Icecast	4	2.2.0	12/2004	25,349	2.3.1	11/2005	29,079	10	292	25	0	1
Memcached	3	1.2.2	05/2007	5,743	1.2.5	03/2008	6,345	14	118	6	1	5
Space Tyrant	6	0.307	10/2006	18,738	0.351	10/2007	20,223	0	107	11	2	3

Table 1. Application update information (all versions).

considered for each program. The last three groups of columns contain the cumulative number of changes that occurred to the software over that span. "Type changes" refers to **struct**s, **union**s and **typedef**s. We can see that programs have changed significantly during the period we considered. For example, Icecast added nearly 4,000 lines of code; there were 25 changes to types, 10 changes to function prototypes, and 292 changes to function bodies.

Table 2 shows release and update information for each program. Columns 2–4 show the version number, release date and program size for each release. Column 5 contains the nature of individual releases.[5] Column 6 shows the number of type transformers for that specific update, while column 7 presents the size of the state transformer (in lines of code); '-' means no type or state transformers were needed for a particular release.

5.2 Source code changes

When building updatable applications with STUMP, the programmer may need to intervene at two phases: when preparing the source code for compilation with STUMP, and when creating dynamic patches. We present details on the strategy we followed, and programmer effort (annotations or lines of code) for each of these phases, in turn, for our three test programs.

Stage extraction. As mentioned in Section 4.1, Ginseng supports code extraction as a solution for updating long-running code. In single-threaded Ginseng programs, long-running loop bodies are often extracted into separate functions where a DSU_update call is placed just after the extracted call. This approach ensures the next loop iteration will be to the new version, and reduces the Δ restrictions (types that are not allowed to change) at the update point, since the extracted function is inactive there. However, just extracting the loop body turns out to be insufficient in some cases for our multi-threaded programs, because many threads may be running the same loop, and blocking synchronization may prevent them from all exiting the loop at once. For example, producer/consumer-style threads may result in one thread blocked in the first part of a loop while the second thread does work in the second part [12]. To address this problem, we must further extract logical "stages" of some loops into separate functions that can be updated separately.

Multi-threading annotations. Table 3 presents the number of annotations we added to our test programs to prepare them for compilation with STUMP. Identifying long-running loops, update scopes, and stages was relatively straightforward, as we explain below. The second column shows the number of update scopes. The multi-threaded servers we have considered perform a few high-level operations whose boundaries suggest natural definite update points. Examples of such operations are processing one event, accepting and dispatching a client connection, etc. We enclosed each thread loop body for Icecast and Space Tyrant into an update scope. In the case of Memcached, the update scope delimits the processing of one event. The third column shows the number of blocks marked as check-ins; we placed check-in annotations around stages. To use the same source code for both barrier and relaxed approaches, we directed STUMP to compile the beginning of check-in blocks to DSU_induced_update for the barrier approach, and to DSU_checkin for the relaxed approach (see Section 4.2).

[5] As described at `http://freshmeat.net/` in the case of Icecast and Memcached; for Space Tyrant we provide our own characterization.

Program	**Update scopes**	**Check-ins**	**Extractions**		**Other changes**
			loop	stage	
Icecast	11	17	11	17	42
Memcached	1	1	0	0	23
Space Tyrant	5	16	7	16	19

Table 3. Source code annotations.

The last two columns show the number of times we used loop or stage extraction. Identifying long-running loops was easy, as each long-running thread essentially executes a loop. We identified 11 loops in Icecast and 7 in Space Tyrant; these numbers are higher than the number of update scopes because some loops are nested. Loop extraction was not necessary for Memcached because looping occurs in an separate event-handling library. Finally, the last column shows the number of stages designated for extraction. These often coincide with the check-in blocks, but for Memcached no stage extraction was necessary.

Other changes to source code. In addition to designating update scopes, check-ins and stages, we had to make several small changes to application source code to cope with Ginseng's conservative safety analysis [16]. The number of lines added or changed (presented in column 6 of Table 3) consisted of:

- Directives to the compiler to override the conservativity of the safety analysis. STUMP's analysis does not model existentially quantified types, so even though they are safely used, STUMP reports a possible safety violation. We had to add two such directives in Icecast, three in Memcached and two in Space Tyrant.
- Changing four low-level, type unsafe, field access macros in Memcached into function calls, so STUMP's compiler and safety analysis can reason about them.

Adjusting auto-generated patches. Ginseng automatically generates candidate type transformer functions for type definitions that have changed. We inspected (and completed, where necessary) the generated type transformers, and wrote state transformers when needed; across all patches, we had to write 80 lines of code for Icecast, 12 for Memcached and 81 lines for Space Tyrant.

6. Experiments

We performed two sets of experiments. First, we measured how quickly an update can take effect in STUMP as compared to various alternatives (Section 6.1). Second, we measured the overhead that STUMP's update support imposes on application performance compared to stock versions of the programs, and the original Ginseng without our added support (Section 6.2).

We conducted our experiments using a client-server setup, where the updatable applications ran on a quad-core Xeon 2.66GHz

Program	Release	Date	Size (LOC)	Description	Type xform (count)	State xform (LOC)
Icecast	2.2.0	12/2004	25,349			
	2.3.0rc1	08/2005	28,593	Major feature enhanc.	23	5
	2.3.0rc2	09/2005	28,788	Other	4	1
	2.3.0rc3	09/2005	28,796	Other	-	-
	2.3.1	11/2005	29,079	Other	7	1
Memcached	1.2.2	05/2007	5,743			
	1.2.3	07/2007	5,732	Other	1	-
	1.2.4	02/2008	6,144	Major bugfixes	3	2
	1.2.5	03/2008	6,345	Major bugfixes	2	-
Space Tyrant	0.307	10/2006	18,738			
	0.316	10/2006	19,077	Minor feature enhanc.	11	-
	0.319	10/2006	19,399	Minor feature enhanc.	2	-
	0.331	04/2007	19,526	Minor bugfixes	-	-
	0.335	05/2007	19,753	Minor feature enhanc.	-	6
	0.347	08/2007	19,979	Minor bugfixes/enhanc.	1	2
	0.351	10/2007	20,223	Minor feature enhanc.	2	1

Table 2. Application releases.

server with 4GB of RAM running Red Hat Enterprise Linux AS release 4, kernel version 2.6.9. The clients ran on a two-way SMP Xeon 2.8GHz machine with 3.6GB of RAM running Red Hat Enterprise Linux WS release 3, kernel version 2.4.21. The client and server systems were connected by a 100Mbps network. All C code (generated by STUMP or otherwise), was compiled with `gcc` 3.4.6 at optimization level `-O2`.

6.1 Update timeliness

To measure the effectiveness of induced update points and relaxed synchronization in STUMP, we measure update timeliness, for all 13 patches that we developed, using gradual refinements (called *update protocols*) to our approach for reaching safe update points. We start with a straightforward extension to Ginseng, and show that this performs poorly in practice. We then add induced update points, and find that they improve update timeliness, though sometimes we fail to reach safe update points. Finally, we add relaxed synchronization and show that it is very effective at reaching safe update points, fast. We first describe the experimental setup, then proceed to describing the protocols and the results.

We ran experiments for each update and measured the time it took the system from the moment the update was signaled to the moment it could safely be applied; results (in milliseconds) are presented for each protocol in Table 4. The first column shows the program, while the second column shows the update sequence number, e.g., according to Table 2, entry 0 for Icecast corresponds to the update 2.2.0 → 2.3.0rc1. The subsequent columns show the results for each protocol.

For each protocol, we tested two configurations, 4 and 16 concurrent clients. The number of server-side threads varied, depending on the application and number of clients. Icecast has a fixed number of threads; in our configuration this number was 6 in Icecast 2.2.0, and 7 in later versions. Memcached has a thread pool with a configurable number of handler threads, independent of the number of clients. We present results for two configurations, one with four server threads (Memc-4), and one with 16 server threads (Memc-16). Space Tyrant uses two threads per connected client, plus three fixed threads that perform housekeeping, so the number of Space Tyrant server-side threads were 11 (8 client handlers + 3 fixed) for the 4-client configuration and 35 (32 client handlers + 3 fixed) for the 16-client configuration.

We are interested in measuring update timeliness while the server is under load (since thread activity is likely to obstruct updates from taking effect), and most importantly whether an update may take place at all. The methodology for each program was to start the server, connect 4 (or 16) clients that are constantly asking for data, and send an update request while the server is performing work. We then measured the time from the moment an update was requested to the moment it could be safely applied, or timed out after 15 minutes. We performed each experiment 11 times; we report the median time to reach a safe update point for terminating runs. An X entry means that for that specific configuration, none of the 11 runs could reach such a point within 15 minutes. We also measured update loading times, i.e., time to load a dynamic patch after reaching a safe point. Loading times are proportional to patch size, and in all cases were less than 3 ms; we omit details due to space constraints. We now present our protocols and findings.

P-OPNOIND models a straightforward extension to Ginseng, and does not use induced update points (`DSU_induced_update` calls are treated as no-ops) or relaxed synchronization. It naïvely tries to coax all threads to quiesce by yielding (via `sched_yield`) when a definite update point is reached. If the last thread discovers that all threads are at definite update points, and that all these points are compatible with the update, the update is applied. The results are in columns 3 and 4 of Table 4. We can see that this protocol fails to reach a safe update point for almost all scenarios, and when it does, it can take several minutes to apply an update.

P-BARRIERNOIND attempts to improve P-OPNOIND. When a thread reaches a definite update point, and the effects at that point do not conflict with the update, it blocks. When the last thread discovers that all threads are blocked, the update is applied. We can see that this strategy (columns 5 and 6) is more successful than P-OPNOIND, but we still cannot reach a safe update point for the Icecast updates and some Memcached updates. We discuss the reasons below.

P-OPWAIT follows the optimistic approach of P-OPNOIND, but calls `sched_yield` at induced update points as well. We can see (columns 7 and 8) that the extra points help—P-OPWAIT reaches safe update points more quickly than P-OPNOIND, and in more cases. Likewise, P-BARRIER is the same as P-BARRIERNOIND but synchronizes at induced update points, and again (columns 9 and

		Protocol											
		P-OpNoInd		P-BarrierNoInd		P-OpWait		P-Barrier		P-Relaxed		P-PostRelaxed	
# Clients →		4	16	4	16	4	16	4	16	4	16	4	16
Update id ↓													
Icecast	0	X	X	X	X	X	X	1,062	945	1,750	1,068	17,037	10,243
	1	X	X	X	X	X	X	976	942	2.1	2.1	805	1,233
	2	X	X	X	X	X	X	478	937	0.7	0.7	603	1,000
	3	X	X	X	X	X	X	484	941	2.3	2.3	691	1,303
Memc-4	0	X	144,901	851	710	150	21	0.9	1	0.6	0.7	1.1	1.2
	1	27,493	134,958	847	706	373	29	1.7	1.8	1.2	1.2	1.5	1.5
	2	148,003	129,580	853	725	275	28	1	1	0.8	0.8	1.3	1.4
Memc-16	0	X	X	X	760	X	1,163	X	1.7	1	1.1	X	2.3
	1	X	X	X	729	X	1,255	X	4.5	2.8	2.8	X	3.6
	2	X	X	X	727	X	3,465	X	2	1.4	1.4	X	2.9
Space Tyrant	0	X	X	5,184	5,093	X	X	3,505	3,439	3.4	5.2	3,513	3,492
	1	X	X	5,193	5,147	X	X	3,506	3,456	4.7	9.4	3,524	3,511
	2	X	X	5,186	5,082	X	X	3,526	3,514	1	3.4	3,545	3,477
	3	X	X	5,151	5,054	X	X	3,524	3,445	3.9	4.2	3,621	3,461
	4	X	X	5,110	5,101	X	X	3,508	3,459	1.4	1.5	3,553	3,469
	5	X	X	5,146	5,111	X	X	3,504	3,426	4.4	3.7	3,504	3,482

Table 4. Time to reach a safe update point using various update protocols (in milliseconds).

		Completion time (sec)									
		4 clients					16 clients				
	Compilation	Stock	Ginseng		Stump		Stock	Ginseng		Stump	
Remote	Icecast	11.09	11.08	(-0.09%)	11.09	(0.00%)	40.50	40.58	(0.20%)	40.84	(0.84%)
	Memcached	31.64	31.30	(-1.07%)	31.58	(-0.19%)	86.32	87.08	(0.88%)	87.66	(1.55%)
	SpaceT	44.54	44.48	(-0.13%)	44.49	(-0.11%)	44.62	44.51	(-0.25%)	44.60	(-0.04%)
Local	Icecast	1.64	1.66	(1.22%)	1.75	(6.71%)	2.65	2.63	(-0.75%)	2.70	(1.89%)
	Memcached	7.58	7.89	(4.09%)	7.92	(4.49%)	31.37	32.13	(2.42%)	32.55	(3.76%)
	SpaceT	34.60	34.62	(0.06%)	34.61	(0.03%)	44.52	44.65	(0.29%)	44.62	(0.22%)

Table 5. Benchmark completion times (elapsed times, in seconds, and in % relative to the stock server).

10) the result is improved performance in both update successes and update times.

P-Relaxed employs Stump's support for check-ins and relaxed synchronization. As shown in columns 11 and 12, P-Relaxed is the only protocol able to reach a safe point for all updates, and it does so quickly, sometimes orders of magnitude faster than the other protocols. The only update for which P-Relaxed is not the fastest is #0 for Icecast (shown in the first row). P-Relaxed takes 1.75 seconds with 4 clients and 1.06 seconds with 16 clients to reach a safe update point, compared to 1.06 and 0.94 seconds, respectively, for P-Barrier. The added slowdown is due to the more conservative safety check for P-Relaxed. As explained in Section 4.2, in the relaxed approach, a check-in scope $s_1; ...; s_n$ with normal effect ε calls $\mathsf{DSU\_checkin}(\alpha_1 \cup \varepsilon, \omega_1, \Delta_1)$ (recall $\omega_1 = \omega_n \cup \varepsilon$). This effectively prevents anything in ε from being updated while $s_1; ...; s_n$ is being executed. By contrast, a DSU_induced_update call just prior to s_1 would pass in $\alpha_1, \omega_1, \Delta_1$, reducing the chance of a conflict with the prior effect. The Icecast update #0 contains a particularly large number of changes, and the safety check fails for many check-in scopes where it succeeds for induced update points at the same positions.

On the other hand, the barrier-based protocols P-Barrier and P-BarrierNoInd must wait for all threads to reach an (induced or definite) update point before applying the update, which results in some updates failing to take place. For P-Relaxed, the update is applied as soon as it becomes available if allowed by the current restriction. In all the failing cases for Icecast and Memcached one or more threads are suspended due to a blocking call (on I/O or a condition variable) before reaching an update point, and will not proceed before new clients connect. For example, in the Memc-16 scenario, we have 16 server threads, and activity from only 4 clients prevents each of the 16 threads from being scheduled within the 15 minute time-out window.

While the barrier protocols could reach safe points eventually under different workloads (e.g., without a fixed set of clients), a more robust solution might be to treat blocking calls as a kind of induced update point, wrapping them with code to register α, ω, Δ with the run-time system just prior to the actual call, and adding code to synchronize the thread upon returning from the call if an update has since become available. An update may be applied once currently active threads reach safe points as long as registered threads are safe. While this solution should work for these programs, we feel barrier synchronization is still generally undesirable for two reasons: (1) the server's performance will be degraded because the existing client threads will block at update points and not perform any work until the patch is eventually applied, and (2) there is still the possibility of deadlock (despite not observing it in our example applications), and this possibility must be addressed.

The final protocol shown in Table 4 is P-PostRelaxed, which is like P-Relaxed except that check-ins are treated as no-ops until an update has been requested—once all threads have checked in their restrictions at least once, the update protocol is the same as P-Relaxed. This protocol potentially reduces overhead during normal operation at the cost of slower patch application times.

The performance here (columns 13 and 14) is roughly similar to P-BARRIER. The failure cases are for the same reason: when the patch becomes available, some threads are blocked, and will only check in once awakened. The successful cases have similar times, though Icecast update #0 is slower due to the more conservative safety check, as described above. Wrapping blocking calls as described above should also help P-POSTRELAXED. However, our performance experiments in Section 6.2 show that, in practice, the cost of always doing check-ins is modest, so P-RELAXED provides a good balance between overhead and timeliness.

6.2 Performance overhead

We evaluated the impact of dynamic update support on application performance by running application-specific benchmarks, and measuring memory footprint.

For each application, we measured the performance of its most recent version under three configurations. The *stock* configuration forms our base for benchmarking, and consists of the application compiled normally, without support for updating and without involving STUMP. The *Ginseng* configuration is the application compiled with a normal Ginseng, which generates type transformer code assuming single-threading, and treats check-ins as no-ops. The STUMP configuration is the application compiled with STUMP, which assumes multi-threading and implements check-ins calls as registering check-in effects as described earlier. Comparing the Ginseng and STUMP configurations shows the additional overhead STUMP imposes on applications, i.e., double-checked locking when calling concretization functions, and effect registration at check-ins.

For each application and configuration, we ran a specific benchmark and measured the completion time and memory footprint (at the completion of the benchmark). We ran each benchmark in two setups. The first setup, **remote**, shows the results of running the clients and server on separate machines, a scenario that models how the updatable servers would be used in practice. The second setup, **local**, shows the results of running the clients and server on the same machine (we used the quad-core machine mentioned above). The local configuration factors out network latency and bandwidth issues (while reducing parallelism of the server). Similar to the update protocol experiments in Section 6.1, we report figures for 4 and 16 clients, respectively.

Application performance. For Icecast, we measured the time it took the streaming server to serve eight mp3 files to a `wget` client. Each file has size 1, 2, 3, . . . 8 MB. To eliminate jitter due to disk I/O, we directed `wget` to send both its output and the downloaded file to `/dev/null`. For Memcached, we ran a "slap" test that ships with the server. The test program spawns 4 or 16 clients in parallel (the same number of clients as the number of server threads), each client inserting key/value pairs into Memcached's hash table. We measured the time it took the test program to complete insertion of 50,000 key/value pairs. For Space Tyrant, we created a scenario file that directs a client to perform 500 random moves across the universe, and spawned concurrent clients running this scenario. We measured the time it took the server to process all the clients.

In Table 5 we report the median benchmark completion time across 11 runs. In the remote, more realistic, setup, for Icecast and Space Tyrant, the completion time is similar to the stock server. Memcached is however slower in the 16-thread configuration, with the multi-threaded updatable version 1.6% slower. In the local setup, impact of update support on completion time is higher than in the remote setting; this is because, as expected, update support (e.g., check-ins, function and type indirection) slows down the application and the slow-down cannot be masked by network latency. However, even in this scenario the slowdown is small, less than 7% in all cases. Finally, by comparing the Ginseng and STUMP columns we can quantify the additional cost of supporting multi-threading on top of a DSU compiler. For example, by looking at the second-to-last row, we can see that for Memcached, the cost of multi-threading is an extra 0.4% (4.09% vs. 4.49%) in the 4-client configuration, and 1.34% (2.42% vs. 3.76%) in the 16-client configuration, respectively.

Program	**Size (LOC)**	**Build time (sec)**	
		gcc	STUMP
Icecast	25,349	2.3	27
Memcached	5,743	0.7	4.2
Space Tyrant	18,738	2.6	20.1

Table 7. Time to build (compile and link) the test programs.

Memory footprint. Memory footprint overhead is presented in columns 3–8 of Table 6. As expected, local or remote setups exhibit the same memory footprint. Update support (function and type indirections, and the STUMP runtime) increases the memory footprint of application compiled with STUMP. To quantify this impact, we measured the virtual memory footprint in the stock and updatable configurations using `pmap`. For Memcached the difference is almost imperceptible. For Icecast, the memory footprint increases by up to 4.8% compared to the stock server. For Space Tyrant, the increase is at most 46.7%; the reason why the increase is so large has to do with Space Tyrant and STUMP interaction. The median memory footprint for the stock version is around 63 MB, while the median memory footprint for STUMP-compiled versions is around 92 MB. Space Tyrant uses a pre-allocated global array that keeps the game map, divided into 100,000 sectors, and the size of this array is 7.6 MB. The STUMP compilation scheme allows room for growth in each structure [16]; by default, room for growth is equal to the initial structure size. Because the large array resides within two nested structures, its size becomes 31.2 MB, hence the growth to 92 MB and the 46% increase. This problem could be solved by keeping updatable data by reference or by allocating sector data on demand. Since in our experiments we used at most 8 concurrent clients, each patrolling at most 500 sectors, the actual memory overhead would be at most 11.7% for the 4-threads case, and 13.3% for the 16-threads case.

In almost all cases, the multi-threaded setup (STUMP) has a slightly higher overhead on the application than the single-threaded setup (Ginseng): around 3% more for Icecast and 4% more for Space Tyrant. This increase is due to extra work and extra memory requirements, e.g., check-ins and locks for `con` functions (Section 4.2).

Compilation time. To provide a sense of STUMP's compilation overhead, in Table 7 we present the time to compile and link each test program in two configurations. The first, normal configuration, without update support, using `gcc` (which in turn calls `ld`) is presented in column 3. The second configuration shows build time for updateable applications using STUMP; it consists of compiling C code into updateable C code using STUMP, then followed by `gcc` and linking together with the STUMP runtime system. The build times for this case are presented in column 4; the bulk of the time is spent in the safety analyses [15], and we imagine the figures could be improved with some more engineering.

7. Related work

Several existing systems can dynamically update multi-threaded programs, but to our knowledge our work is the first to identify the safety/timeliness trade-off explicitly, and to consider solutions for it in any depth. Existing systems do provide restrictions on update

Config.		Memory footprint (MB)					
	# Clients →	4			16		
	Compilation →	Stock	Ginseng	STUMP	Stock	Ginseng	STUMP
	Application ↓						
Remote	Icecast	69.83	70.08 (0.36)	72.32 (3.56)	70.36	70.09 (-0.39)	72.84 (3.52)
	Memcached	99.94	100.21 (0.27)	99.93 (-0.00)	223.74	223.42 (-0.14)	223.91 (0.08)
	SpaceT	62.75	90.15 (43.66)	92.04 (46.68)	63.42	90.18 (42.19)	91.81 (44.76)
Local	Icecast	69.71	70.33 (0.89)	72.23 (3.62)	69.86	70.03 (0.24)	73.18 (4.75)
	Memcached	99.39	99.14 (-0.25)	99.67 (0.28)	222.67	223.23 (0.25)	223.05 (0.17)
	SpaceT	63.73	89.85 (40.99)	91.96 (44.29)	62.97	89.44 (42.03)	91.73 (45.68)

Table 6. Memory footprint (absolute footprint, in MB, and in % relative to the stock server).

timing, both manual and automatic, but fully automatic approaches are insufficient to solve the problem while little or no guidance is given in how to use manual mechanisms effectively.

Many systems require that updates not be performed on currently running functions, with some relying on this "activeness check" as the sole means for ensuring safety; examples include the K42 operating system [19, 3], OPUS [1], and Ksplice [2]. Relying on the activeness check alone negatively affects safety and timeliness. As shown in Sections 2.2 and 2.3, activeness does not preclude some invalid update times, and may indefinitely delay an update if it changes functions that contain infinite loops. Some systems [2, 1] attempt to avoid safety problems by limiting the form of updates to code only, and not state. However, even this restriction is not sufficient to avoid all problems. Notice that if our example update in Figure 1 only consisted in moving the call to log from process to clean then it could still exhibit the problematic execution while changing only code.

Several systems—including LUCOS [4], POLUS [5], and UpStare [11]—provide fine-grained control over when to apply a patch so that it can be applied quickly. LUCOS and POLUS permit updates to active code where active functions continue to execute at the old version; by default, subsequent calls target the most recent version, but the programmer can override a particular call to be fixed at one version. UpStare uses *stack reconstruction* to allow an actively running function to transition to a corresponding point in the new version of the same function when an update is applied. This technique has the same effect as Ginseng's code extraction, but is more flexible, as transition points can be specified at patch time, not deployment time. (The need for this support was motivated by earlier experience writing small updates to long-running functions in the Linux kernel [12].) While these mechanisms are useful, they do not mitigate the problem of reasoning about the effects of a patch in the large state space of a multi-threaded program. By contrast, our approach reduces the programmer's burden of reasoning about safety to a few definite update points, while induced update points and relaxed synchronization ensure timeliness. We imagine our techniques could be applied to these existing systems.

Our previous work [15] introduced the idea of version-consistency and the use of contextual effects to enforce it. In that work we considered single-threaded programs whereas for the present work we have a full implementation for multi-threading, STUMP, which we have evaluated on several realistic programs. STUMP incorporates the new ideas of check-in effects and relaxed synchronization. In our prior work, we proposed enforcing version-consistency within programmer-specified *update transactions*. We initially expected this idea to transfer directly to multi-threaded programs, with programmers using update transactions frequently and at a fine granularity just as they might use modern software transactions (a.k.a. *atomic blocks*) [7]. However, we found it much easier to reason about dynamic updates occurring at a small number of definite update points/scopes at the top-level, as advocated by the present work, rather anywhere within a web of nested transactions.

8. Conclusion

In this paper, we presented an approach for updating multi-threaded programs while they run, and show how we have implemented this approach in STUMP. Updating multi-threaded programs is more difficult than updating single-threaded programs because myriad thread interactions complicate reasoning how an update will interact with the many states of the system, while timing restrictions on an update's application that would reduce this burden may unduly delay the update from taking effect. We address this tension using the novel concepts of induced update points and relaxed synchronization, which can be used to ensure updates are performed promptly while easing the programmer burden of reasoning about patch application correctness. We evaluated our approach on three realistic multi-threaded servers. We found that programmer effort for building updatable versions of these applications was modest, and experiments show that update support does not significantly impact application performance.

Acknowledgments This research was supported in part by NSF grants CCF-0541036 and CNS-0346989, and the partnership between UMIACS and the Laboratory for Telecommunication Sciences. We thank Ray (Otis) Eargin for providing us earlier versions of Space Tyrant. We also thank Gavin Bierman, Gianfranco Ciardo, Jeff Foster, Eric Hardisty, Chris Hayden, Scott Owens, Polyvios Pratikakis, Peter Sewell, Nikhil Swamy, and the anonymous referees for their helpful comments on drafts of this paper.

References

[1] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. Opus: online patches and updates for security. In *USENIX Security*, 2005.

[2] J. Arnold and F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *EuroSys*, 2009.

[3] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, et al. Reboots are for hardware: challenges and solutions to updating an operating system on the fly. In *USENIX ATC*, 2007.

[4] H. Chen, R. Chen, et al. Live updating operating systems using virtualization. In *VEE*, 2006.

[5] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A powerful live updating system. In *ICSE*, 2007.

[6] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE TSE*, 22(2), 1996.

[7] T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In *PPOPP*, 2005.

[8] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6), 2005.

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

[10] I. Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Dept. of Computer Science, University of Wisconsin, Madison, April 1983.

[11] K. Makris and R. Bazzi. Multi-threaded dynamic software updates using stack reconstruction. In *USENIX ATC*, 2009.

[12] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, 2007.

[13] I. Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, College Park, August 2008.

[14] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In *MSR*, 2005.

[15] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL*, Jan. 2008.

[16] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[17] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *PLDI*, 2006.

[18] S. Sidiroglou, S. Ioannidis, and A. D. Keromytis. Band-aid patching. In *HotDep*, 2007.

[19] C. Soules, J. Appavoo, K. Hui, et al. System support for online reconfiguration. In *USENIX ATC*, 2003.

[20] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *TOPLAS*, 29(4), Aug. 2007.

[21] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates for Java: A VM-centric approach. In *PLDI*, June 2009.

[22] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *JFP*, 2, 1992.

[23] C. Walton. *Abstract Machines for Dynamic Computation*. PhD thesis, University of Edinburgh, 2001. ECS-LFCS-01-425.