Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction

Kristis Makris *Rida A. Bazzi*
Arizona State University
Tempe, AZ
{*makristis,bazzi*}*@asu.edu*

Abstract

We propose a new approach for dynamic software updates. This approach allows updating applications that until now could not be updated at runtime at all or could be updated but with a possibly indefinite delay between the time an update is initiated and the time the update is effected (during this period no service is provided). Unlike existing approaches, we allow arbitrary changes to functions active on the stack and without requiring the programmer to anticipate the future evolution of a program. We argue, using actual examples, that this capability is needed to dynamically update common real applications.

At the heart of our approach is a *stack reconstruction* technique that allows all functions on the call stack to be updated at the same time to guarantee that all active functions have the same version after an update. This is the first general approach that maintains both code and data representation consistency for multi-threaded applications. Our system *UpStare* was used to update the PostgreSQL database management system (more than 200,000 lines of code) and apply 5.5 years-worth of updates to the very secure FTP server vsFTPd (about 12,000 lines of code).

1 Introduction

Downtime experienced by applications due to software updates (feature additions, bug fixes, security patches) can be prohibitive for applications with high-availability requirements. Dynamic Software Update (DSU) can help minimize the downtime by allowing applications to be updated at runtime. Instead of completely stopping the application process and then executing the newer version, DSU would only momentarily pause the application while applying the change *in-memory*. A typical dynamic update would consist of: (1) pausing the execution of the old version in a given state, s; (2) applying a state mapping function $\mathcal{S}$ to s to obtain a state $\mathcal{S}(s) = s_{new}$ (loading new code segments can be a part of the mapping); and (3) resume execution of the new version using s_{new} as the initial state. In general, a state mapping needs not happen instantaneously and can be done lazily in stages. The state mapping should be safe in that the resulting state s_{new} should be a valid state of the new application (in a sense that we will make precise in Section 2). In general, a valid state mapping is not always possible, and, when it is possible, it is not necessarily possible for all states of the old application.

The dynamic software update problem consists of two components. First, DSU needs to determine the states, or execution points, of the old application for which it is possible to apply a valid update, and, for those states for which a valid update is possible, to determine the state mapping function to effect the update - this is the *update safety problem*. Second, DSU needs to effect the update through a mechanism that maps an old execution state to a new execution state - this is the *update mechanism problem*. In general, the safety problem is undecidable [6]. This implies that, in general, user help is needed to determine safe update points and to specify the state mapping function. Nevertheless, this does not mean that it is not possible to solve the problem automatically or semi-automatically without or with little user help for many practical cases of interest.

Since user help is unavoidable, it is important to provide the user with an update mechanism and safety checks that make it easier to reason about the update. Current DSU mechanisms are limited in their support of the update of *active functions and data structures* and in their support for *immediate updates*. To support the update of functions that are active on the call stack and for the update of stack-resident data structures, current DSU systems require the user to anticipate the future evolution of a program [3, 14]. Immediate updates are not supported by existing DSU systems. An update is immediate if it satisfies: (1) atomicity: before the update only

old code executes and after the update only new code executes; (2) bounded delay: if a valid mapping is known for a given state and the execution is in that state, then the mapping is applied in a bounded amount of time. Atomicity is desirable because it is sufficient to guarantee *logical consistency* [16, 13]: the execution of the application is *indistinguishable* from an execution in which the old version executes for some time, then the new version executes. While bounded wait is not necessary for logical consistency, we argue that for multithreaded applications, immediate updates are needed to provide logically consistent updates *without service interruption*; i.e. the update does not cause the service to be unavailable for an unbounded amount of time.

To address the limitations of current DSU systems, we propose a new DSU mechanism and a new DSU system for C programs. Our system *UpStare* supports the immediate update of functions that are active on the call stack as well as the update of stack-resident data structures without requiring the user to anticipate the future evolution of the program. Our system is also the first system to allow immediate update for multi-threaded as well as multi-process applications. *UpStare* applies source-to-source transformations to make applications dynamically updateable. At the heart of the mechanism is a novel *stack reconstruction* updating mechanism that allows an application to unroll the call stack when an update occurs (while saving all the stack frames) and then reconstitute the call stack by replacing old versions of functions with their updated versions (while at the same time mapping data structures in the old frames to their updated versions). Stack reconstruction guarantees that after an update is applied only new code executes. Mapping to the new state is automated with an effective heuristic: a patch generator produces data transformers for global variables and for local variables of all stack frames, and a default stack execution continuation mapping resumes execution from the new version. These mappings and transformers can be further fine-tuned by the user. *UpStare*'s immediate data mapping *eliminates the need for data wrappers* that are used by other DSU systems [5, 14, 11, 2] to allow updating datatypes. The elimination of data wrappers greatly reduces execution overhead for data intensive applications.

UpStare supports the update of applications anywhere during their execution including multithreaded applications with blocking system calls. This is achieved by inserting update points in long-lived loops and transforming blocking system calls into non-blocking calls. This guarantees that we can update threads and processes *without interrupting service indefinitely*, since we are not constrained by the need for an active function to exit before we can update it as in other DSU systems.

In summary, our immediate update mechanism guarantees the following: (1) Representation consistency; (2) Update immediacy for multi-threaded and multi-process applications; (3) High updateability; (4) No data-access indirection.

UpStare is able to update real-world applications of significant size, such as vsFTPd and PostgreSQL, with minimal manual adjustments from the user and with modest overhead. Still our current implementation has some limitations. First, it is not optimized for performance due to a limitation of existing compilers when partitioning code in hot-cold blocks given branch prediction hints. This limitation can lead to large overhead for systems with a small instruction cache and large functions. Second, it does not yet integrate support to automatically transform pointers, which we developed in previous work [10]. Third, it does not support updates of data in shared memory or in-transit data in internal buffers, but this is not a limitation of the approach; it is a limitation of the current implementation. Finally, since our emphasis in this work is on the updating mechanism, we do not provide automatic safety checks through code analysis as in other DSU systems. Adopting such checks would increase the usefulness of *UpStare*.

The rest of the paper is organized as follows. Section 2 introduces the DSU problem. Section 3 describes our DSU system. Section 4 presents our implementation. Section 5 evaluates the performance of our system and analyzes the sources of overhead. Section 6 discusses related work.

2 The Dynamic Software Update Problem

In this section, we reintroduce the dynamic software update problem (DSU), describe some common safety guarantees that are desirable for DSU systems and argue for the need for immediate updates.

2.1 Dynamic Software Update

Given (Π, s), where Π is program code and s is an execution state, updating Π to Π_{new}, where Π_{new} is a new version of Π, consists of:(1) pausing the execution Π; (2) applying a state mapping function S to s to obtain a state $S(s) = s_{new}$; and (3) resuming execution of Π_{new} from state s_{new}.

By updating an executing program, we obtain a hybrid execution that in general needs not satisfy the semantics of either the old or the new versions. In general, the desired semantics for the hybrid executions needs to be determined by the user. We say that a state s for program Π is *valid for update* from Π to Π_{new} if there is a state mapping function S that can be by applied in state s such that the resulting hybrid execution satisfies the de-

sired semantics. The dynamic software update problem has two aspects:

- Update safety: Identify a valid state s and a corresponding state mapping function.
- Update mechanism: Implement the state mapping function.

Gupta [6] showed that, even for weak requirements on the semantics of the hybrid execution, it is undecidable to determine if a given state s is valid for update from Π to Π_{new}. The problem is related to the problem of identifying semantic differences [7] between two versions of a program. Identifying semantic differences has been studied extensively and is also undecidable although safe approximations are known [8].

So, in general, assistance from the user is required to both identify valid states and guide the state mapping. Nonetheless, there are many situations in which a default state mapping can produce a new state that will satisfy the desired semantics.

2.2 Safety

Given that it is not possible in general to guarantee the safety of updates without user help, it is helpful to provide some restricted safety guarantees that are satisfied by the updated program. The goal is to make it easier for the user to establish that the default mappings result in valid updates and, if they do not, to supplement the state mapping to make it valid. Some useful guarantees are:

1. Type-safety: No old version of code Π should be executed on a newer version of a datatype representation τ' (oldcode-type-safety) and no new version of code Π' should be executed on an older version of a datatype representation τ (newcode-type-safety).

As an example, consider adding in a C struct that contains five fields a new field as the third field listed and properly constructing a new state s_{new} for a variable of this datatype. If code from the old version accessed the newer version of this datatype in s_{new} it would incorrectly access the memory area used by the new field when intending to access the fourth field, and corrupt data.

2. Transaction-safety: Some sections of code are denoted as transactions and are specified by the user to execute completely in the old version or completely in the new version.

Unlike type safety, transaction safety requires user annotations. One way to ensure transaction safety is by prohibiting updates when execution is in such a user specified section. This can be done at runtime by querying if the current state is in a forbidden region, but this is not straightforward to achieve. If a function f is called inside a transaction and in other parts of the program, then determining the execution state requires knowledge of the stack contents. Alternatively, transaction safety can be ensured at compile time by conservatively estimating update points that will not violate the transactional requirements.

More generally, a DSU system may be able to provide the user with a more flexible notation to specify that an update is not valid in a given state. For example, stating that an update is not allowed if Thread 1 is executing in (say) <functionA,lines 135-160> while Thread 2 is executing anywhere within <functionB> can be sufficient input to a DSU system to apply the update when these threads do not violate this safety constraint.

3. Representation Consistency: Both state and program representation consistency hold. An update guarantees *state representation consistency* if at no time the executing application expects different representations of state (such as global variables or the stack-frame contents). An update guarantees *program representation consistency* if following the update only Π_{new} is executed over the new state s_{new}; no part of Π is executed again. Representation consistency (state and program) makes it easier to reason about the effects of executing code on the state because Π_{new} and s_{new} in memory match the source code, but it is not an end-goal in itself. The difference between state representation consistency and type-safety is that one could provide type-safety by allowing new and old definitions of a type to be valid simultaneously. For example, one could apply forward and backward datatype transformers [5], but this makes it harder to reason about updated programs. Additionally, it may not be possible to convert a datatype for new code, then backward for old code, and then forward for new code again, since updated types often contain more information than older types and data could be lost.

4. Logical Representation Consistency: An update system provides logical consistency if the hybrid execution is indistinguishable to an outside observer from executions that are obtained with representationally consistent updates [16, 13].

5. Thread-X-safety: An update is thread-X-safe if X-safety is provided in a multithreaded applications. For example, thread-type-safety means that type-safety is provided for a multithreaded application. In general if a DSU system guarantees that X-safety is satisfied for individual threads independently, then thread-X-safety is not necessarily guaranteed.

Our update mechanism provides the user with the ability to initiate a representationally consistent update in any state of the program. The emphasis is on the mechanism though. Determining the validity of a particular state for update requires other analyses [8, 16, 13].

2.3 Immediate Updates

In this section, we introduce immediate updates and argue that they are needed to guarantee that the update of common multithreaded applications is logically consistent and can be achieved without unbounded service interruption. We first introduce the concept of update with bounded delay.

Bounded delay update: If a valid mapping is known for a valid old state s and the application is in state s, a state mapping can be applied without pausing the application for an unbounded amount of time.

An update is *immediate* if it satisfies representation consistency and bounded delay. To understand the need for immediate updates, consider a multithreaded application in which each server thread handles a client connection and threads read/write in a shared data structure after receiving client requests. In general, there might be a long delay between successive client requests.

Now, consider an update that changes the specification of the data structure and how it is accessed and assume a number of connections are active. To effect the update, there are a number of options:

- Do not allow any new connections and wait until all active connections terminate. When all connection terminate, apply the update. This is not a good option because it can result in the service being unavailable for an unbounded amount of time.

- Allow new connections, but using the old version of the code. This can result in the update being indefinitely delayed because the new version may never get to be executed.

- Allow new connections using the new version of the code while connections created with the old version are active (possibly blocked for client input). This is the more interesting case. Once the shared data structure is accessed by threads running the new version, the data representation would have to reflect the semantics of the new version. This means that on the next access by the old version we either violate logical representation consistency or we force the thread running the old version to be transformed to the new version. Since violating logical consistency is not an option, we are left with the need to immediately update the thread running the old version. Otherwise the connection will not be available for its client for an unbounded amount of time.

So, for all cases, the capability to immediately update individual threads is necessary. If multiple threads of the old version are attempting to access the shared data structures, the updated mechanism should support their collective immediate update. The update mechanism we propose is the first that can support immediate update of single-threaded as well as multi-threaded applications.

3 Dynamic Update System

We describe our proposed update model and how we apply state mappings under this model.

3.1 Update Model

We propose an update model that is more flexible than the update models of existing works in two respects. First, we consider stack frames as part of updateable program state. Stack frames include local variables, formal parameters, and return addresses. Second, we consider the Program Counter as updateable program state. Unlike existing work, we can ensure updates meet the safety guarantees of Section 2 while employing an updating model that can modify all aspects of the old program state s. This means our approach has a wider reach (more old valid states) in applying an update compared to existing work that needs to accept fewer old valid states if it is to meet these safety guarantees.

A program (Π, s) is a pair of program code Π and program state s. Program code Π is a set containing the executable code of all the functions of the program. Program state $s = (h, T_{sf}, T_{PC})$ is a tuple consisting of a set h containing all global variables on the heap, an array T_{sf} of ordered lists sf of stack frames, one for each thread of the program, and an array T_{PC} of Program Counters for each Thread. Each stack frame $f(l, p, ra)$ in sf contains a set l of local variables on the stack, the formal parameters p and the return address ra. We omit the semantics of program execution from this description.

Software updates are effected by replacing Π with Π_{new}, applying a state transformer S to s, and continuing execution from program Π_{new} in state $S(s) = s_{new}$. Dynamic updates take place at *update points*, which are a subset of possible PC locations for the program. Our compiler inserts update points automatically when compiling a program to be update-enabled, as we discuss Section 4. The update mechanism allows the state transformer to modify the entire old program state. For example, for each new stack frame $f'(l', p', ra')$ it can add new local variables to produce l', change function signatures by extending or reducing the formal parameters to obtain the new formal parameters p', or adjust the return address ra' of a stack frame to continue from a different execution point on the parent stack frame. It can insert new stack frames in $T_{sf'}$ or remove stack frames. It can also set a new Program Counter $T_{PC'}$ for all threads. For example it can set threads to "escape" from execution of a loop or a function.

3.2 Default State Mapping

Default state mappings are needed to reduce the effort required from the user. In general we would hope that the default mapping is what the user desires, but there are no guarantees for that. The user is always given the capability to override default mappings.

Our approach involves an effective heuristic that relies on verification of its validity by the user. We apply data transformers of global variables on the heap h and local variables l of every stack frame T_{sf}, and re-issue function parameters p. Additionally we map execution continuation of return addresses ra and Program Counters T_{CP}. Transformers and mappings are automatically generated, can be overridden by the user, and, for the cases we have tested, they are effective enough and require minimal user involvement.

Datatype updates. When an update is requested, stack frames T_{sf} and program counters T_{PC} of all running threads are saved and the stack is unrolled up to the thread entry-point function. At this point, the entire old state s at the time the update was initiated is available (having just been saved) to systematically produce the new state s_{new}. For every global variable whose datatype τ has changed, a new global variable of the new datatype τ' is allocated in h'. If the datatype is a struct or union and it has been extended, a transformer copies the old fields (only new fields must be initialized by the user). If the datatype is reduced, the remaining fields are copied with no user assistance. If the variable is an array, a transformer is applied on all array elements. If the datatype change simply extends an array with more elements (e.g. parseconf_uint_array in vsFTPd offers more configuration options), a new array with more room is allocated and the values of all old elements are copied.

Stack frames $f'(l', p', ra')$ are reconstructed with a default automatic mapping by copying the old stack frame $f(l, p, ra)$. Local variables l' are grouped into a struct and automatically copied from l. Variable additions are treated as new field additions in a struct and can be initialized to a default value by a user-supplied stack transformer. Datatype changes of local variables l are mapped in a way similar to global variables h and formal parameters p' are automatically copied from p or further extended by the user.

Execution continuations. Return addresses ra and Program Counters T_{PC} are automatically preserved, and they correspond to continuation points. Continuation points are all points prior to function calls and all update points. That's how execution control flow can descend to reconstruct a callee, or resume a program after an update, respectively. We take the simple approach of assigning unique numeric ids to continuation points in the order they appear in each function body. By default, we map continuation points with the same enumerator in Π and Π_{new}. If the call graph of the application did not change and the loop structure did not change, this mapping is very effective for actual updates. By adjusting a continuation point a user can define how control flow should continue upon returning to a parent stack frame. We have not found it necessary to insert additional continuation points (e.g. one in every basic block).

```
struct vsf_transfer_ret
vsf_ftpdataio_transfer_file(
struct vsf_session* p_sess, int remote_fd,
int file_fd, int is_recv, int is_ascii)
{
  // Continuation point 1
  if (!is_recv) {
    if (is_ascii) {
      // Continuation point 2
      return
        do_file_send_ascii(p_sess, remote_fd, file_fd);
    } else {
      // Continuation point 3
      return
        do_file_send_binary(p_sess, remote_fd, file_fd);
    }
  } else {
    // Continuation point 4
    return do_file_recv(p_sess, remote_fd,
                        file_fd, is_ascii);
  }
}
```

(a) vsFTPd v1.2.2

```
struct vsf_transfer_ret
vsf_ftpdataio_transfer_file(
struct vsf_session* p_sess, int remote_fd,
int file_fd, int is_recv, int is_ascii)
{
  filesize_t curr_offset;
  filesize_t num_send;

  // Continuation point 1
  if (!is_recv) {
    if (is_ascii || p_sess->data_use_ssl) {
      // Continuation point 2
      return do_file_send_rwloop(p_sess, file_fd,
                                 is_ascii);
    } else {
      // Continuation point 3
      curr_offset =
        vsf_sysutil_get_file_offset(file_fd);
      // Continuation point 4
      num_send = calc_num_send(file_fd, curr_offset);
      // Continuation point 5
      return do_file_send_sendfile(p_sess, remote_fd,
                     file_fd, curr_offset, num_send);
    }
  } else {
    // Continuation point 6
    return do_file_recv(p_sess, file_fd, is_ascii);
  }
}
```

(b) vsFTPd v2.0.0

Figure 1: Continuation points in vsFTPd.

Figure 1 shows an example of mapping the continuation of do_file_send_binary in an update of vsFTPd from v1.2.2 to v2.0.0. Updating this function requires mapping the ra to its parent stack frame vsf_ftpdataio_transfer_file. It requires mapping con-

```
upstare_mapping_t mappings_v200[] = {
  { "vsf_ftpdataio_transfer_file",
    "vsf_ftpdataio_transfer_file",
    2, // 2 continuation points are mapped
    { { 3, 5 },
      { 4, 6 }
    }
  },
  { "do_file_send_binary",
    "do_file_send_sendfile",
    5,  // 5 continuation points are mapped
    { { 6, 2 },
      { 7, 3 },
      { 8, 4 },
      { 9, 5 },
      { 10, 6 }
    }
  }
};
```

Figure 2: Relevant continuation mapping for an update of do_file_send_binary in vsFTPd v1.2.2 to v2.0.0.

tinuation point 3 from v1.2.2 to continuation point 5 in v2.0.0, including supplying the new parameters curr_offset and num_send (initialized in the stack transformer) to the new version do_file_send_sendfile. Without this mapping an update would incorrectly resume from ra=3 in v2.0.0, which would load vsf_sysutil_get_file_offset on the stack, and the old state T_{sf} of callee stack frames of do_file_send_binary would not be restored.

Figure 2 shows the relevant declaration of the variable (source code in C) used to express the continuation mapping to update to vsFTPd 2.0.0. There are two mapping points for vsf_ftpdataio_transfer_file: 3 maps to 5, and 4 maps to 6. 1 and 2 use the default mapping: they map to their old values of 1 and 2. Also do_file_send_binary is replaced with do_file_send_sendfile and execution continues from the replaced function at an offset continuation of -4, which means some code from the beginning of do_file_send_binary was removed.

Mapping pointers. Mapping pointers of datatypes known at compile-time is straightforward. However, void* pointers are cast at runtime to generic datatypes and are harder to map. Support for tracking pointer types at runtime is needed to invoke the appropriate datatype transforms. We have developed this support in previous work [10] and it has low overhead (1-7%), but we do not yet integrate it with UpStare.

4 Implementation

UpStare consists of a compiler to generate updateable programs, a runtime environment for dynamically applying updates, a patch generator, and a dynamic updating tool, as shown in Figure 3. This architecture is similar to those of existing updating systems. The compiler applies high-level, source-to-source transformations that make

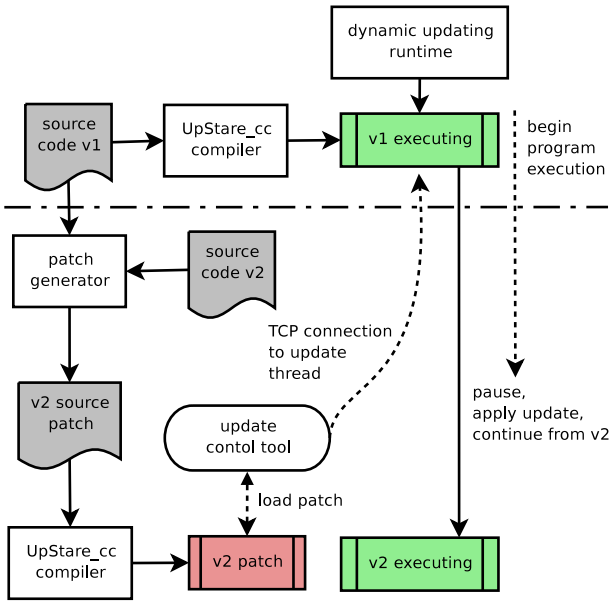

Figure 3: UpStare system architecture.

programs dynamically updateable. It is written in OCaml using the CIL framework[15] v1.3.6 and is architecture and operating system independent. Users replace in their build process (e.g. Makefiles) calls to an existing compiler like gcc with calls to the compiler of our system (hcucc.pl). No source code modifications by a user are required in existing programs. Programs are transformed as needed to coordinate application of updates with the dynamic updating runtime (written in C; 64KB memory footprint). Updates are initiated by the user with a separate dynamic updating control tool that connects using TCP to a thread waiting for update requests. Updates are loaded in memory using dlopen and applied under the guidance of the runtime.

Given the source code of the old and updated programs, a patch generator automatically produces the source code for a dynamic update patch. The patch includes the newer versions of functions, and the old and updated datatype definitions of modified variables, either global or declared on the stack. It also includes automatically generated datatype and stack transformers, and, optionally, user-defined execution continuation mappings that override the default ones to produce the new state.

4.1 Stack Reconstruction

Stack reconstruction consists of two major steps. It saves the existing stack state when unrolling and restores the updated state when reconstructing. To reduce the size of active instrumented code, wrapper functions that efficiently save and restore stack frames are produced away

```
functionA()
{
   char a;
   int param;

   ...
   functionB(param);
}
```

(a) Non-Instrumented

```
typedef struct {
  char a;
  int param;
} stack_functionA_v1_t;

(*functionB_ptr) (int) =
   &functionB_transformed;
functionA_transformed()
{
  stack_functionA_v1_t locals;

  ...
  functionB_6_before:
  functionB_ptr(locals.param);
  if (may_reconstruct && must_reconstruct()) {
    if (must_unroll_up(''functionA'')) {
       save_frame__functionA(&locals, 6);
       return;
    }
    goto functionB_6_before;
  }
}
```

(b) Instrumented

Figure 4: Transformation of function calls for stack reconstruction (functionB_ptr just returned).

```
functionA()
{
   char a;
   int param;

   ...
   functionB(param);
L1:...
}
```

(a) Non-Instrumented

```
functionA_transformed()
{
   stack_functionA_v1_t locals;

   if (may_reconstruct && must_reconstruct()) {
      restore_frame__functionA(&locals);
      switch (next_continuation_point()) {
         ...
         case 3:
           goto try_to_update_3_after;
         ...
         case 6:
           goto functionB_6_before;
         ...
      }
   }
   ...
   functionB_6_before:
   functionB_ptr(locals.param);
   if (may_reconstruct && must_reconstruct()) {
      if (must_unroll_up(''functionA'')) {
         save_frame__functionA(&locals, 6);
         return;
      }
      goto functionB_6_before;
   }
L1:...
}
```

(b) Instrumented

Figure 5: Transformation of function entrypoints for stack reconstruction (entering functionA_transformed).

from the text segment in a separate memory area of cold code executed only during reconstruction.

Figure 4 shows how stack frames are saved. functionA is transformed to check upon returning from the callee functionB whether the stack should be reconstructed. Note that may_reconstruct is a global flag raised only in reconstruction mode to improve performance. If must_reconstruct is true (this thread should participate in reconstruction) and execution should be unrolled (must_unroll_up is true: the topmost frame, by default, has not been reached yet, but the user can specify that unrolling stops at a different frame), the stack frame and continuation point 6 are saved and functionA returns to its caller. Returning to callers continues until the start of the program is reached: the main function in single-threaded applications or the start routine passed to a pthread_create call for multi-threaded applications. Otherwise unrolling should stop (must_unroll_up is false). A goto statement resumes execution from functionB_6_before and descends in functionB for reconstruction.

Figure 5 shows how execution is resumed from functionA. If on function entry the stack should be reconstructed downwards, the stack frame is restored. A switch statement maps the continuation point 6 to continuation label functionB_6_before using a goto statement. Execution flow continues by calling functionB. When the update is complete (may_reconstruct is false: we are no longer in reconstruction mode) and functionB finishes, execution continues normally (from L1).

Thread entry-points. If the main function or the start routine passed to a pthread_create attempt to return during reconstruction they will terminate permanently. To allow the update of main or thread entry points, calls to such functions are initiated from a wrapper function. To accurately discover thread entry-points (and signal-handlers, discussed next) we use the points-to alias analysis provided by CIL.

Signal handlers. The address of signal handlers, defined with sigaction and signal, is stored inside the operating system. To avoid resetting signal handlers when they are updated calls to them are initiated from a wrapper function. Additionally, signal handlers return execution to the kernel and are incompatible with stack reconstruction. They are instrumented to raise a flag on entry and reset the flag before exiting. Requests to update are rejected when a program is executing a signal handler.

```
functionA()
{
   char a;
   int param;

   while(condition)
   {
      ...
   }
}
```

(a) Non-Instrumented

```
functionA_transformed()
{
   stack_functionA_v1_t locals;

   ...
   while(condition)
   {
      if (must_update) {
         coordinate_update_top(&locals, 3);
         return;
       try_to_update_3_after:
         coordinate_update_bottom();
      }
      ...
   }
}
```

(b) Instrumented

Figure 6: Insertion of an update point at the beginning of a loop.

They are immediately satisfied when the program continues in normal execution mode, and can update signal handlers at that point. Signal handlers are discovered using points-to alias analysis provided by CIL.

Redirecting function calls. Function calls are executed using pointer indirection. For each function f_v1, a global variable f_ptr is created that points to &f_v1 and calls to f_v1 are transformed to calls to *f_ptr. For each function pointer *g_v1, wrapper functions are created that call it.

Inserting update points. Update points are automatically inserted at the beginning of each function and each loop so they can be encountered often to allow immediate updates. Figure 6 shows an example update point inserted at the beginning of a loop. When the must_update flag is raised, the current thread participates in synchronization to block all threads. The current continuation point 3 and the stack frame of functionA are saved, and execution returns to the function's caller. When the stack is reconstructed and functionA is called again (see Figure 5b), execution flow resumes from try_to_update_3_after.

Our current implementation is restricted to a coarse-activation of update points using a single must_update flag. However, it is straightforward to support more fine-grain selective activation by dynamically disengaging update points. For example, the user could specify when requesting an update that (say) all update points except 250-259 should effect the update.

Exporting local variables. The dlopen library call will successfully load a dynamic update patch only if the patch references global variables. References to variables that were declared local in the original version (using the static keyword) are not accessible after dynamic loading, leading to system exceptions when executing state transformers. Our compiler removes the static keyword from all local variables and exports them to global.

4.2 Multi-Threaded Updates

Updating a multi-threaded or multi-process application requires all threads to be blocked. If some threads are not blocked the possibility of thread-safety violations remains open.

We adapted an algorithm that blocks all threads in heterogeneous checkpointing for multi-threaded applications[9] to dynamic updates. The idea is to force all but one thread to block when the application must update. The one thread that is not blocked will be the coordinator of the update. It polls the status of the remaining threads until it can tell for sure that all threads are blocked, as defined below.

When a thread reaches an update point and the application must update, it raises a flag indicating that it is *willing to cooperate* on the update and then attempts to acquire a *coordination lock*. The first thread to acquire the coordination lock is the *coordinator* of the update. The coordinator can tell that some threads are blocked if their cooperation flags are raised. But this does not cover all threads. Some threads might be blocked waiting on an application lock owned by a thread that is already willing to cooperate and that is blocked on the *coordination lock*. To that end, the system needs to keep track of the blocking status of various threads. Calls to pthread_mutex_lock and pthread_mutex_unlock are replaced with wrapper calls to keep track of the blocking status of threads. When a thread attempts to acquire a lock, it adds the lock to a WANT list. When the lock is acquired, the lock is removed from the WANT list and placed on a HAVE list. When the thread releases the lock, the lock is removed from the HAVE list.

The coordinator determines that a thread is *really blocked* if:

1. The thread is willing to update;
2. The thread is blocked waiting on a lock owned by another thread that is *really blocked*.

The coordinator keeps on checking the status of the other threads until it can determine that all other threads are *really blocked*, at which time the coordinator initiates the actual update: the stack of each thread is unrolled

and the threads block; all datatypes are transformed; the stacks are reconstructed and the threads block; and, the threads resume executing the updated version.

The algorithm outlined above has been extended to support blocking threads that use semaphores[9], but our current implementation does not yet integrate that capability with the dynamic update system.

Multi-process updates. We extend multi-threaded updates in multi-process applications. fork calls are replaced with wrapper calls that maintain a hierarchy of children. This information is used by the parent process, which acts as a central coordinator of the individual update steps, to apply an atomic update among all children: it waits for all threads of all children to block; all stack frames to be unrolled; transforms datatypes; reconstructs stacks; and, releases all children after all their threads are ready to resume execution. wait and waitpid are also intercepted to cleanup the children hierarchy.

4.3 Blocking System Calls

To enable the runtime to regain execution when an update is initiated, we transform blocking I/O calls into non-blocking calls and we segment write calls into writes of smaller chunks.

Calls to sendfile, which is used in vsFTPd for file transfer, are segmented into 256KB chunks. We do not yet implement segmentation for send but it should be straightforward to do so. read, recv, accept, and select calls are wrapped to check if the file descriptor is set to blocking mode. If it is, the file descriptor is converted to non-blocking mode, the operation is issued, and execution is voluntarily blocked in a manner that allows unblocking: we issue a select that includes in its read set the file descriptor of a pipe created by the runtime. If an update must be applied, hence we need to unblock, we write to the pipe to force select to return and encounter an update point. A bottom handler executed after the update point resets the file descriptor to blocking mode. To allow state transformation while a blocking system call is issued without corrupting the data buffer of read or recv, these calls are issued with a buffer allocated on the heap. When the operations complete, the data are copied back to the original buffer. A possible optimization, which we have not yet implemented, is to transform programs to always allocate I/O data buffers on the heap instead of the stack, to avoid copying data back to the buffer when such operations complete.

A more general approach to handling any blocking system call, not just I/O calls, is to always issue the call in a separate thread. This allows the runtime to remain in control and initiate reconstruction even if the system call has not returned yet. Our original implementation of blocking I/O calls followed this approach but was not as efficient as the self-pipe select solution, due to the cost of pthread_create (we did not try worker threads).

5 Evaluation

We demonstrate the working of UpStare on three applications. The data-intensive KissFFT, the vsFTPd server, and the PostgreSQL database. We give a detailed analysis of the sources of overhead, such as runtime overhead, memory footprint, and network overhead.

5.1 KissFFT

We compiled (at -O3) the KissFFT[1] v1.2.0 Fast Fourier Transform library (1936 lines of code) using float datatypes to be dynamically updateable and performed 100,000 iterations on 20,000 points. This is an application with heavy data access and for which source code instrumented with Ginseng was made available to us. We did not update this application, but we compiled it to be updateable. We used this application to get a better understanding of the sources of overhead introduced by our instrumentation. We ran experiments that selectively omitted parts of the code that UpStare introduces in an application. We measured the time to run this application: (1) using the original compiler, (2) using CIL, (3) when only wrapper functions to save/restore stack frames are produced, (4) when functions were called directly without pointer indirection, (5) when if-statements without a body are inserted for update points (Figure 6), the switch statement prologue (Figure 5b), or upwards stack unrolling (Figure 4b); here we aim to measure the overhead of branch checks when the must_update and must_reconstruct flags are not raised, and (6) after adding the body of these if-statements.

Figure 7 shows the impact of the presence of reconstruction-aware code in the program. To compare the results we identify the best compiler to use with a non-instrumented KissFFT and the best compiler to use under instrumentation. Given an non-instrumented KissFFT, gcc 4.1 (GNU C Compiler) is the best compiler and given an instrumented KissFFT the best compilers are icc 10.1 (Intel C Compiler) for Ginseng and gcc 3.4 for UpStare, all on a Pentium M. Under this comparison, the best performing Ginseng reports overhead of 149.8% (87.1% for UpStare) and the best performing UpStare reports overhead of 38.2% (179.3% for Ginseng). The overhead of Ginseng stems from accessing data through a versioned pointer indirection instead of accessing them directly. In comparison, the overhead of UpStare is rooted at the increase of function size that

[1] http://sourceforge.net/projects/kissfft

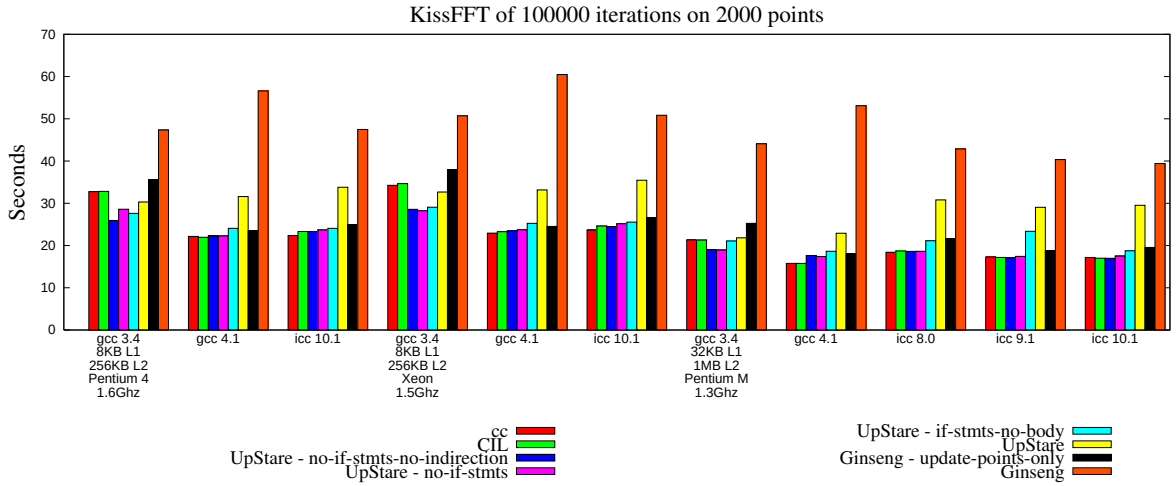

Figure 7: KissFFT: Impact of reconstruction code on running time.

overexerts the ITLB and branch predict unit of the processor.

CIL. CIL transforms source code in simpler terms and should not alter performance. It generally doesn't, but it reported up to 4.2% overhead (Pentium 4: icc 10.1) and up to 1.0% improvement (Pentium M: icc 10.1).

Wrapper save/restore functions. Compared to CIL, producing wrapper functions to save/restore stack frames should not report overhead because these functions are stored outside the text segment. However, on a Pentium M it reported 11.8% overhead with gcc 4.1 and 11.0% improvement with gcc 3.4. Intel compilers report no overhead, suggesting a problem with gcc.

Function indirection. Functions called via pointer indirection should incur constant overhead. They report overhead up to 3.0% on a Pentium M (icc 10.1), 1.2% on a Xeon (gcc 4.1), and 10.3% on a Pentium 4 (gcc 3.4).

If-statements. On a Pentium M, inserting if-statements adds an overhead of 7.2% for icc 10.1, 7.2% for gcc 4.1 and 11.3% for gcc 3.4. This suggests branch prediction can be a significant factor in final performance. Still, update points in Ginseng and UpStare incur comparable overhead.

Increased function size. In comparison to the total overhead of if-statements without a body (Pentium M: 18.0% for gcc 4.1; 9.2% for icc 10.1), an increased function image size adds an overhead of 23.0% and 57.4% respectively, and is responsible for most of the system overhead. We used OProfile to collect performance statistics on the Pentium 4 with gcc 4.1 (overhead 31.3%) and further investigate this issue. We observed a 15% increase in the number of ITLB translations and an 11% increase in the number of instruction fetch requests from the branch predict unit. Other events like ITLB misses, retired mispredicted branches and page walks showed no significant deviation.

We attempted to use inline assembly to place the body of if-statements outside the text segment. Inline assembly convention prohibits using branch instructions since their presence is not available to high-level optimizations. The compiler would produce intermediate assembly code for the stack unrolling code that would fail to link (inline code supplying linking directives in unreachable basic blocks would not be produced). We also attempted to partition code in hot and cold blocks with -freorder-blocks-and-partition using both gcc 4.1 and icc 10.1 but the compilers moved the cold blocks only to the end of the function image without reducing the overhead. Placing the cold blocks to the end of the process image instead may reduce the final overhead.

Memory footprint. We measured the resident set size at the various stages of instrumentation. CIL does not increase the working set. Wrapper code that saves/restores stack frames is responsible for most of the memory increase, up to 236KB (48.7%) using gcc 4.1 on a Pentium M. If-statements marginally increase memory by 4-8KB (0.9-1.7%). The best performing UpStare in respect to running time (Pentium M: gcc 3.4), increased memory by a total of 260KB (53.7%), while Ginseng (Pentium M: icc 10.1) increased memory by 76KB (13.3%). Ginseng increases memory by type wrapping struct datatypes, while UpStare adds updateable code inside functions and wrapper functions to save/restore stack frames.

5.2 The Very Secure FTP Daemon

vsFTPd is a fast, secure, widely used FTP application that forks connection handlers that do not communicate

Ver.	Date	LoC[2]	Types					Variables					Functions				
			Tot.	Same	Add.	Del.	Upd.	Tot.	Same	Add.	Del.	Upd.	Tot.	Same	Add.	Del.	Upd.
1.1.0	2002-07-31	8,389	628	-	-	-	-	158	-	-	-	-	436	-	-	-	-
1.1.1	2002-10-07	8,468	628	628	0	0	0	161	156	3	0	2	436	420	0	0	16
1.1.2	2002-10-16	8,731	639	626	11	0	2	165	159	4	0	2	447	428	11	0	8
1.1.3	2002-11-09	8,839	646	638	7	0	1	167	164	2	0	1	449	439	2	0	8
1.2.0	2003-05-29	10,011	659	641	16	3	2	201	163	35	1	3	481	378	39	7	64
1.2.1	2003-11-13	10,506	664	655	7	0	2	205	196	7	3	2	486	447	6	1	33
1.2.2	2004-04-26	10,547	664	664	0	0	0	204	202	1	2	1	487	476	1	0	10
2.0.0	2004-07-01	11,527	998	649	342	8	7	218	200	16	2	2	513	421	35	9	57
2.0.1	2004-07-02	11,543	687	674	8	319	5	219	218	1	0	0	513	506	0	0	7
2.0.2	2005-03-03	11,612	688	687	1	0	0	219	219	0	0	0	513	489	1	1	23
2.0.3	2005-03-19	11,743	688	688	0	0	0	226	216	8	1	2	516	481	5	2	30
2.0.4	2006-01-09	11,857	694	687	6	0	1	229	225	3	0	1	519	499	4	1	16
2.0.5	2006-07-03	11,923	694	693	0	0	1	234	228	5	0	1	519	494	0	0	25
2.0.6	2008-02-13	12,202	701	691	7	0	3	239	231	5	0	3	523	497	4	0	22

Table 1: vsFTPd: Source code evolution.

with each other or their parent. We applied 13 updates spanning 5.5 years of application evolution, compiled with gcc 4.1 on a 2.4Ghz Xeon. The updates were prepared automatically using the patch generator. They required a total of 11 user-defined continuation mappings for the two use cases we tested. Additional mappings will probably be needed to update from other update points. They also involved some manual initialization of new variables and struct fields.

Table 1 studies the source code evolution of vsFTPd. New datatypes are more often added than modified. Variable additions are common, and there are few datatype changes or variable deletions. Functions are updated very often and are less likely to be deleted. We also note that a large collection of functions and variables are added in major revisions of the program, such as from v1.1.3 to v1.2.0 and from v1.2.2 to v2.0.0. The large number of types added in v2.0.0 is due to including header files from GnuTLS (for secure communication) while in v2.0.1 (released one day later) the OpenSSL header files were removed. We applied updates to vsFTPd under two use cases:

- **Idle client**. A client connected to the server, authenticated correctly, and was waiting idle for user input on the command line. An update was applied.
- **File transfer**. A client connected to the server, authenticated correctly, and requested to retrieve a large file. The file begun being transmitted to the client but has not finished transmission. An update was applied.

Our goal was to determine if vsFTPd required updates of functions on the stack under these use cases, which are typical for this type of application. In 7 out of 13 updates the vsf_session struct variable allocated in main was extended with new fields and needed to be updated. For an idle client, in 7 out of 13 updates functions on the stack needed to be updated. 5 of those 7 updates were of forward control flow that had not been executed yet and was pending on the stack. For a file transfer, in 9 out of 13 updates functions on the stack needed to be updated and 6 of those 9 updates were of forward control flow. Additionally, we observed a case where an update applied during a large file transfer possibly needed to escape a loop. During the update from v1.1.2 to 1.1.3 the new code in do_sendfile should be executed only if a new global flag is on. If the update requires the initial state of this flag to be off, execution should break out of the loop and stop transferring the file.

613 update points where automatically inserted in vsFTPd v2.0.5. Updating during a large file transfer occurred at stack depth 11 (maximum depth is 16, average 8.9) and took 59.7ms: 50.2ms to block all processes; 0.4ms to unroll the stack; 0.95ms to unroll the stack of children processes; 0.45ms to reconstruct; 1ms to reconstruct the stack of children processes. In comparison, Ginseng applies a vsFTPd update in under 5ms [14].

While Ginseng can support the update of vsf_session struct, it achieves that with data padding whose limitations we have already discussed.

We setup a client-server configuration connected with a cross-over cable to eliminate network fluctuations. We found this setup necessary to accurately measure performance: in preliminary measurements our system reported performance improvement, which was counter-intuitive. We installed vsFTPd to serve files both from a hard-disk and from an in-memory filesystem to eliminate performance perturbation of hard-disk accesses and identify the worst-case overhead. We measured the latency of establishing a connection and retrieving a 32-byte file 1000 times and the throughput of retrieving a 300MB file. Table 2 reports the median of 11 runs and shows comparable performance for files served either from a hard-disk or from memory. Stack reconstruction slows down an updateable vsFTPd v2.0.5 by $\sim$0.37-0.50ms (4.9-5.3%), multi-process support by $\sim$0.65-0.70ms (6.8-7.4%), and support for blocking system calls

[2]Generated using David A. Wheeler's 'SLOCCount'.

vsFTPd Configuration	Connection Latency(ms) 32-byte file	
	Hard-disk	Memory
v2.0.5 - NonInstrumented	9.61	9.49
v2.0.5 - CIL	9.64 (0.3%)	9.54 (0.5%)
v2.0.5 - Reconstruction	10.08 (4.9%)	9.99 (5.3%)
v2.0.5 - MultiProcess	10.26 (6.8%)	10.19 (7.4%)
v2.0.5 - BlockingCalls	9.97 (3.8%)	9.76 (2.9%)
v2.0.5 - UpStare-FULL	11.15 (16.0%)	11.06 (16.5%)
v2.0.6 - NonInstrumented	9.62	9.52
v2.0.6 - CIL	9.63 (0.1%)	9.54 (0.2%)
v2.0.6 - UpStare-FULL	11.16 (16.0%)	11.09 (16.5%)
v2.0.5 - update to v2.0.6	11.22 (16.6%)	11.12 (16.8%)

Table 2: vsFTPd: Impact of instrumentation on latency.

PostgreSQL Configuration	pgbench throughput (t/s) 100,000 transactions	
	Hard-disk	Memory
v7.4.16 - NonInstrumented	175.6	319.7
v7.4.16 - CIL	169.7 (3.4%)	319.0 (0.2%)
v7.4.16 - Reconstruction	133.0 (24.3%)	199.2 (37.7%)
v7.4.16 - MultiProcess	170.5 (2.9%)	312.9 (2.1%)
v7.4.16 - BlockingCalls	161.1 (8.3%)	293.4 (8.2%)
v7.4.16 - UpStare-FULL	130.7 (25.6%)	189.7 (40.7%)
v7.4.17 - NonInstrumented	174.3	317.8
v7.4.17 - CIL	171.3 (1.7%)	316.6 (0.4%)
v7.4.17 - UpStare-FULL	128.0 (26.6%)	189.8 (40.3%)
v7.4.16 - update to v7.4.17	131.8 (24.4%)	188.8 (40.6%)

Table 3: PostgreSQL: Impact of instrumentation on throughput.

by ∼0.27-0.36ms (2.9-3.8%). The worst-case overhead is from memory: 1.57ms (16.5%), and 1.63ms (16.8%) when updated to v2.0.6. Ginseng reported overhead of 3% for an updateable and 5% for an updated vsFTPd, but did not report if it eliminated hard-disk accesses or the network from the experiment. In terms of throughput, an updateable v2.0.5 and an update to v2.0.6 reported zero overhead, like Ginseng.

The numbers for latency are presented as a worst-case scenario because, in a practical situation, transferring a file remotely would incur a latency that is considerably larger than the latency of retrieving a 32-byte file. For transferring files, throughput is more relevant and for that measure our system reports zero overhead.

5.3 PostgreSQL Database

PostgreSQL is an advanced DBMS that forks connection handlers that communicate with each other through shared memory. It is a large application of 369K lines of code, with the postmaster process consuming 225K lines of code (source code from src/backend/). Using the patch generator, we automatically prepared an update from v7.4.16 to v7.4.17 compiled with gcc 4.1 on a 2.4Ghz Xeon. v7.4.17 updated 64 functions and added one variable. The update was applied dynamically without any user-specified continuation mappings when a client was waiting idle for user input. User-specified mappings will probably be needed to update from other update points (9931 update points where automatically inserted in v7.4.16). The update occurred at stack depth 10 (maximum depth is 35, average 15) and took 60ms: 53.7ms to block all processes; 0.2ms to unroll the stack; 0.45ms to unroll the stack of children processes; 0.3ms to reconstruct the stack; 0.4ms to reconstruct the stack of children processes.

The instrumented v7.4.16 and the update to v7.4.17 passed 85 (out of 93) tests of the PostgreSQL testsuite, both in serial and parallel execution. For the remaining 8 testcases we verified with MPatrol and Valgrind that a non-instrumented PostgreSQL was causing buffer overflows, illegal memory accesses, and uses of uninitialized data. While these access errors seem to produce no problems for an non-instrumented PostgreSQL, they were contributing to failures of other testcases or crashes of a PostgreSQL instrumented with stack reconstruction. Since the memory corruption bugs of PostgreSQL can produce unpredictable results we cannot guarantee our implementation will work in the presence of such bugs.

We measured over a cross-over cable the overhead of an updateable v7.4.16 compared to a non-instrumented v7.4.16 using the PostgreSQL pgbench tool that runs a "TPC-B like" benchmark: five SELECT, UPDATE, and INSERT commands per transaction. We measured the time to run 100,000 transactions after a ramp-up time of 40,000 transactions. Table 3 measures the throughput when the database is loaded both on hard-disk and in memory. Stack reconstruction reports 37.7% overhead in memory but this is a worst-case scenario because a database needs stable storage to be durable (24.3% on hard-disk). Although only one client connection was established overall, multi-process support reported overhead 2.1%-2.9% and blocking system calls 8.3%. An updateable v7.4.16 was 40.7% slower in memory and 25.6% slower on hard-disk. For these cases, the transactions were all executed over the same connection. The numbers show that each transaction consumes 5.7ms and 7.7ms for the non-instrumented and updateable v7.4.16 cases respectively. This translates into a latency overhead of 34.4% for each transaction on average. This latency is for transactions over the same connection.

To measure a worst-case scenario, we measured latency for establishing a connection and running only one transaction over the connection. We measure the latency by running a transaction 1000 times (1000 connections

PostgreSQL Configuration	pgbench latency (ms) Average of 1000 transactions	
	Hard-disk	Memory
v7.4.16 - NonInstrumented	25.62	23.56
v7.4.16 - CIL	25.70 (0.3%)	23.77 (0.9%)
v7.4.16 - Reconstruction	34.98 (36.5%)	33.03 (40.2%)
v7.4.16 - MultiProcess	27.33 (6.7%)	25.44 (8.0%)
v7.4.16 - BlockingCalls	26.94 (5.2%)	25.45 (8.0%)
v7.4.16 - UpStare-FULL	48.09 (87.7%)	45.97 (95.1%)
v7.4.17 - NonInstrumented	25.56	23.53
v7.4.17 - CIL	25.73 (0.7%)	23.64 (0.5%)
v7.4.17 - UpStare-FULL	48.34 (89.1%)	45.85 (94.9%)
v7.4.16 - update to v7.4.17	48.36 (89.2%)	46.21 (96.4%)

Table 4: PostgreSQL: Impact of instrumentation on latency.

were established and torn down). Table 4 reports that the combination of stack reconstruction, multi-process support and blocking system calls support have a severe impact on latency. When isolated, these features report a total overhead of 48.4-56.2%. However, when combined an updateable v7.4.16 is 22.41-22.47ms slower (87.7-95.1%), and 89.2-96.4% slower when updated to v7.4.17. We speculate this is due to the limited size of the processor cache and we intend to run more experiments to better understand the results. Note that the overhead due to reconstruction is comparable to that of KissFFT. We speculate that is due to the nature of the application (data-intensive). We could not obtain a number for Ginseng because it could not compile PostgreSQL but we would expect that the data accesses through pointer indirection in Ginseng would result in high overhead.

6 Related Work

Table 5 compares existing DSU systems with *UpStare*. It first compares kernel updating systems, and then application updating systems.

DynAMOS [11] demonstrates transaction safety through user-supplied adaptation handlers. However it may need to wait indefinitely for a safe update point. Its newcode-type-safety relies on pointer indirection through "shadow data structures", which incurs overhead, to access the new fields of updated datatypes. But it cannot guarantee oldcode-type-safety if old types change their semantics, like other binary instrumentation systems [17, 2, 1].

K42 [3] is an OS that is particularly crafted to be updateable and its approach cannot be generally applied to existing systems without significant re-engineering. By design it requires all kernel-threads to be short-lived and non-blocking to guarantee *quiescence*: no to-be-updated functions should be active on the stack.

POLUS [4] accomplishes type-safety of global variables by trapping all data accesses for the duration of an update and synchronizing the state of the old and new types. But it cannot update data on the stack, and does not address representation consistency or the thread safety issues of DSU.

Ginseng [14] pads datatypes with enough space to accommodate future growth. Retrieving the appropriate version of padded datatypes during runtime requires indirection for data access. This leads to considerable overhead in data-intensive applications and after many updates there may be no space left to accommodate the update. Ginseng does not provide state and program representation consistency but it offers logical consistency through static analyses [16, 13] which improve safety and updateability. Since its state mapping is restricted, because of its updating mechanism, these conservative analyses may not always find safe update points for that mapping. Still, Ginseng can update multi-threaded applications [12], although continuation may not be immediate. Additionally, Ginseng requires users to anticipate long-lived loops and mark them for "loop extraction" of the loop body into a separate function to update them before the next iteration begins.

Generally, existing systems have difficulty in updating functions [11, 2, 3, 1, 4, 14] and datatypes [1, 3, 4] that are already active on the stack, or function return addresses [17, 11, 2, 3, 1, 4, 14]. They mostly allow functions to be updated the next time they are called [11, 2, 1, 3, 4, 14]. This is due to their restrictive updating mechanism that opens the possibility for executing part old code, part new code, and part old code again, which can be undesirable. Some systems eliminate the possibility of executing mixed code by requiring quiescence before they update[1, 3, 4] but this limits updateability in practice [1, 4]. In Table 5 the overall ability to update from as many old states as possible is coarsely captured in the *updateability* parameter.

UpStare offers high updateability because of its updating mechanism. It can modify all aspects of the old program state (stack-resident functions, datatypes, and return addresses), which allows updating from a wider range of old valid states. Although it provides useful safety guarantees, it requires some involvement from the user in validating semantic safety of updates. UpStare has the potential to provide transaction-safety by dynamically disengaging update points, although this is not implemented yet. The transaction safety analysis [13] offered by Ginseng could be used by UpStare to reduce user input in validating state transformers.

Acknowledgements. We would like to thank our shepherd George Candea, the anonymous reviewers, and Michael Hicks for their feedback. This work was supported in part by NSF Grant CSR-0849980.

	DynAMOS [11]	K42 [3]	POLUS [4]	Ginseng [14]	UpStare
Domain	Kernel	Kernel	Applications	Applications	Applications
Preparation	Binary	Source	Binary	Source	Source
No program anticipation by user	√	X	√	X	√
Datatype access	Direct	Direct	Direct	Indirect	Direct
Updated datatype access	Part-indirect	Direct	Trap+Sync	Indirect	Direct
User involvement for update	High	Medium	Low	Low	Medium
Oldcode type-safety	X	√	Globals only	Static Analysis	√
Newcode type-safety	√	√	Globals only	Static Analysis	√
Transaction safety	Adaptive	Quiescence	Quiescence	Static Analysis	*Possible*
Representation consistency	X	√	X	X	√
Logical representation consistency	X	√	X	√	√
Thread safety	X	√	X	√	√
Immediate continuation	X	√	X	X	√
Updateability	Medium	High	Low	Medium	High

Table 5: Comparison of existing DSU systems.

References

[1] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *14th USENIX Security Symposium*, pages 287–302, July 2005.

[2] Jeff Arnold and M. Frans Kaashoek. KSplice: Automatic Rebootless Kernel Updates. In *EuroSys 2009*, April 2009.

[3] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, and Robert W. Wisniewski. Providing Dynamic Update in an Operating System. In *USENIX Symposium on Operating Systems Design and Implementation*, April 2005.

[4] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[5] Dominic Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[6] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *Software Engineering*, 22(2):120–131, 1996.

[7] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 234–245, White Plains, NY, June 1990.

[8] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *In Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, 1992.

[9] Feras Karablieh and Rida A. Bazzi. Heterogeneous Checkpointing for Multithreaded Applications. In *21st Symposium on Reliable Distributed Systems (SRDS)*, October 2002.

[10] Feras Karablieh, Rida A. Bazzi, and Margaret Hicks. Compiler-Assisted Heterogenous Checkpointing. In *20th IEEE Symposium on Reliable Distributed Systems (SRDS)*, October 2001.

[11] Kristis Makris and Kyung Dong Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *EuroSys 2007*, March 2007.

[12] Iulian Neamtiu. *Practical Dynamic Software Updating*. PhD thesis, University of Maryland, August 2008.

[13] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, pages 37–50, January 2008.

[14] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2006.

[15] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of Conference on Compilier Construction*, 2002.

[16] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006.

[17] Ariel Tamches and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Third Symposium on Operating System Design and Implementation*, February 1999.