Dynamic software updates for C applications

Sebastian Hahn
Friedrich-Alexander-Universität Erlangen-Nürnberg
sebastian.hahn@fau.de

ABSTRACT
A common inconvenience of keeping a computer system's installed applications updated is the loss of the availability of services while the respective application is updating. Typically, the user has a hard choice to make: Accept the downtime or pay the cost of a redundant system. Many regular applications are not well suited to be run in a truly redundant way, but even when a redundant system is available, updating carries the risk of total loss of service in case of a failure in the part which remains online.

This article presents a solution in the form of Dynamic Software Updates for applications written in C. Several implementations of dynamic software updates are discussed, and their applicability to some common applications is explored. The tools necessary to create patches between versions are also scrutinized, along with performance characteristics of the individual approaches and the required time to conduct an update. Special attention is given to applications employing concurrency, as the added complexity of threads which are interacting with each other needs to be addressed to guarantee the safety of an update process.

1. INTRODUCTION
Updating software is easy:

1. Stop the software by denying new requests made to it, handle all currently pending requests, then turning off
2. Replace the binary
3. Start the new binary, start handling new requests

This simple approach has two obvious problems: While pending requests are processed, all new requests are denied and the state of the application is lost. Depending on the nature of the software, both issues have different weight - but they are typically both present.

Two separate models have been developed to deal with the complexities: Either a runtime system loads a patch, updates function call sites at a *safe point* [5], and data is transformed to its new state as it gets accessed; or the whole program is updated at once, with a complete transformation of the entire state.

In this article, three separate implementations of dynamic software update systems are evaluated and their relative strengths and weaknesses are compared. The goal of all systems is to provide safe, reliable, easy-to-implement, and timely updates. The chosen methods differ in that one system is able to update single-threaded programs only, and that one chooses an atomic whole-program update approach, whereas the other two favor a lazy updating mechanism.

2. IMPLEMENTATIONS
Three implementations of dynamic software update systems are considered:

- Ginseng, a system employing type wrapping, function indirection and loop extraction to update single-threaded applications [5]
- Stump (for Safe and Timely Updates to Multi-threaded Programs), an extension of Ginseng making use of per-thread update points to facilitate multi-threaded application updates [4]
- Kitsune, a whole-program update approach which transforms applications to be updatable automatically while allowing explicit state migration to be created by the programmer [2]

2.1 Ginseng
Ginseng is an implementation of dynamic software update for single-threaded applications by Neamtiu, Hicks et al [5]. Its major improvement on previous work was the concenpt of loop extraction and a novel safety analysis to prevent updated and non-updated code to access the same data. In addition, they use function indirection and type padding to implement lazy updates. Ginseng's toolchain relies on a dynamic patch generator, which gets run before an updated version of a program is compiled (see figure 1).

2.1.1 Update techniques
To update a single-threaded application, it is almost sufficient to modify function calls to use the new versions of functions, and transform the state in such a way that types which got changed during the update get transformed into the new available types. One underlying concept is *representation consistency*, which means that all values of a certain type belong to the same version of that type [5].

This means that all function calls which happen after the updating process was initiated are using the new version of a function, and all function calls which were already part of

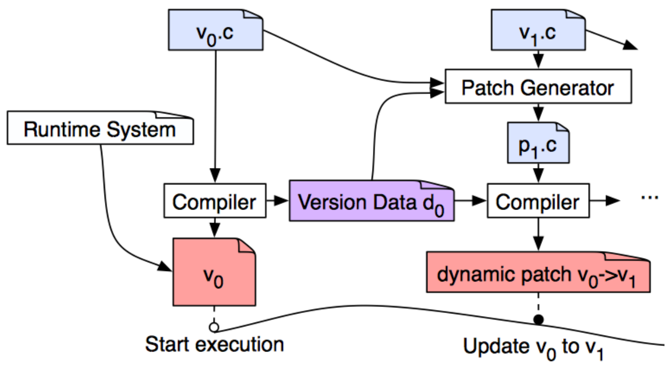

Figure 1: Overview of the Ginseng build chain [5]

the call stack at the time of the update are using the old versions. For this to be safe, it is necessary to restrict the points at which the application can be updated to places where the call stack is short and thus not many functions are active. These points are called *quiescent* points [3].

2.1.2 *Function indirection*

Function indirection is the easiest updating concept to understand. Instead of performing a normal call of the implemented function, a new function pointer for that function is introduced, which points to the function. This wraps all calls to any function in another layer, facilitating updates - only the function pointer needs to be updated to the new version of the function, and nothing else needs to be touched. From the point where the function pointer was updated on the new function will be called. Additionally, to be able to handle function pointers which get passed around inside the application, instead of pointing to the function directly the function pointer is made to point to a wrapper function, such that the pointer which the application regularly uses does not need to be updated [5].

2.1.3 *Type wrapping*

To ensure the aforementioned consistency of values of certain types, the runtime must ensure that when a type is modified in a program update, all values of that type which are active at the time of update are modified, as well. This process simply means applying a conversion function, which does the necessary initialization of new parts of the type and conversion of old parts.

Many different ways to update values of a type are possible, but a simple lazy approach works well: The conversion function gets called when the value of the old type is accessed, and it converts just that instance of the type. As the other instances of the same old type get accessed, they are updated as well. This allows a mostly automatic update of types, without the need to actually keep track of all instances of a type (similarly to how garbage collection works in other languages). To achieve this, each user-defined type that gets wrapped has its version stored alongside with it, so that a *coercion function* is able to return the original definition when called on a wrapped type. This coercion function is always called when a field of any struct is accessed - now

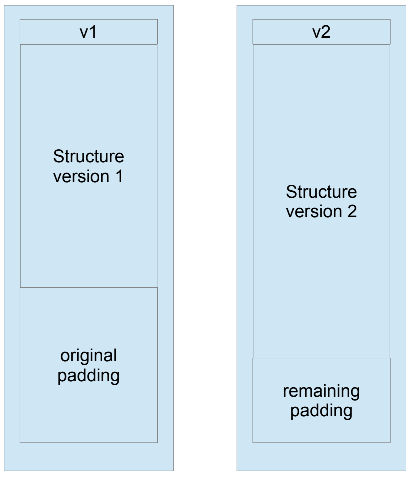

Figure 2: Type wrapping with a growing struct

the struct's version compared to the latest version for that struct, and if necessary the conversion function is applied.

There are some drawbacks of this approach. For one, the conversion functions must be written in a way that allows them to be called at arbitrary points in time, because the runtime determines these points automatically. Also, access to values of the type is slower when a conversion function needs to be executed before the value can be accessed. The biggest drawback, however, is that there must be some mechanism to allow for values of the type on the stack to be updated. This is generally not possible to do without rebuilding the entire stack up to that point, which would defeat the goal of updating in individual steps to speed up updating.

There are a few possible solutions: Padding (see figure 2) can be used to decrease the likelihood of insufficient space being available to perform the type update; at the expense of increased memory footprint of the un-updated application. Alternatively, type access can be hidden behind an indirection level using pointers (which of course have constant size), with the drawback of increasing access times for all such types, even if no change in type size happened during the upgrade. A hybrid approach is also feasible, where small padding is added and type updates which are too large for the padding to accomodate starting to use indirection. This means less overhead than adding more padding, but also adds all the required complexity to handle the indirection approach.

2.1.4 *Shadow Data Structures*

To avoid the general overhead of padding, it is possible to only store the newly added parts of a type in a *shadow data structure* [1]. It is only referenced if new data is added, so the additional cost of indirection needs to be paid for new fields which do not fit into the old structure anymore. This approach is not used by Ginseng due to the added complexity.

2.1.5 *Update Points*

Actually performing the update after the user requested it typically means reaching a quiescent point in the program's execution. These points need to be identified by the programmer manually, and the update framework must be made aware of them via an annotation or similar mechanism. This allows the tool to statically infer some safety properties, which the runtime might check to ensure safety of an update. Finding quiescent points, or points where an update is safe in general, is surpringly easy in typical server applications [5].

The main program loop that gets executed for the whole lifetime of the server process is typically adequate. A call to the update framework runtime gets inserted into the loop. At those call sites, *updateability analysis* adds annotations for all types which may not be updated at this point. Such types are those which might get accessed by old code after being updated to their respective new versions, which is of course unsafe. For this analysis, the control flow is followed from a usage of a type backwards to the control point. Future versions of a program must not violate this analysis, which the runtime ensures when an update point is reached [6].

Due to C's extremely flexible pointer system without strong typing being enforced, the ability to use the address-of operator *&* and allowing arbitrary casts (which might of course be unsafe) it is hard to know that an alias to a wrapped type does not escape and cause an *abstraction violation*. To deal with this, *abstraction violating alias analysis* is performed. Again, if it is determined that a type might be unsafely aliased at an update point, updating that type at the specific update point is forbidden. Manual inspection might be required to resolve such a situation, because sometimes safe casts are misidentified as unsafe. This applies especially to any inline assembly, which does not get treated automatically at all currently [7].

2.1.6 Long-running loops

If only function calls are replaced with new versions, code that is running in an infinite loop (like an event loop that continuously accepts new tasks and hands them off to processing functions) would never get updated. This is problematic, because typically dynamic updating is most desirable for applications that exhibit exactly such a behaviour.

To solve this, *loop extraction* [5] is employed. The body of any long-running loop is taken and extracted into its own function, such that the new body of the loop is merely the function call. To give the former loop body access to the state and to allow transformations of said state, the extracted function gets passed the state as a parameter. This ensures that changed code which got run before the loop was run for the first time can make use of the normal type transformations (discussed below) to update, and changes in the loop body get executed as the body of the loop calls the extracted function for the next time. This also allows other transformations or initilizations to take place which might be required to make new code in the loop body or further down the call stack perform correctly (figure 3).

This extraction requires manual intervention from the programmer, but only insofar as that the programmer must

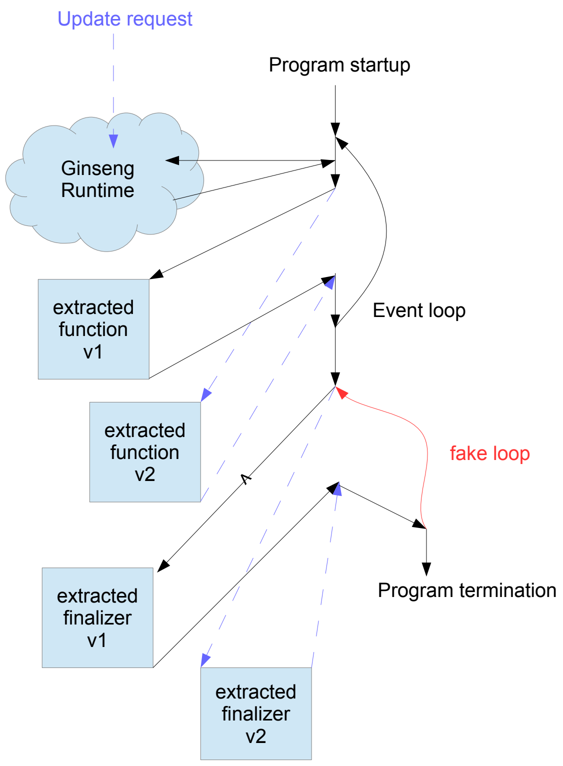

Figure 3: Loop extraction including a fake loop included to update a finalizer

indicate that the loop's body should be extracted. The actual transformation can easily be automated, with return values of the extracted function indicating whether the loop would have triggered a *return*, *break* or *continue* statement or none of those.

If it is possible for the loop to terminate, the old code following the loop might not be able to deal with the changed state. This means that not just the loop body, but the code after the loop must be extracted into a separate function. To achieve this, it is enough to simply wrap the statements which are to be executed after the loop within an infinite loops of their own and a *break* statement afterwards, and mark them for extraction. That way, when that code gets updated, the old version of the code is never executed and the new version is used instead [5]. Of course, this also applies to functions higher up the call stack such that code to be executed after the method containing the long-running loop has finished can be updated and an orderly shutdown can be guaranteed.

Similarly, when the structure of the application changes more radically and a loop gets removed completely, the extracted loop body can be used to orderly clean up and exit the loop, and the extracted code after the loop's body can be used to set up whatever new strategy gets employed.

2.1.7 Abstraction Violating Aliases

One caveat of loop extraction is that it does not deal with *goto* statements or inline assembly which violates the structure of the C program by jumping to pieces of code outside of the loop rather than using *return* or *break* from inside C. Similarly, some cases of unsafe pointer accesses cannot be handled by the runtime. The static analysis component of Ginseng is able to inform the programmer of such issues, so they can be fixed manually. The underlying analysis method is conservative, and the programmer is given an option to override the static check if manual analysis deems the conversion safe.

2.1.8 Evaluation

Loop extraction provides a flexible tool to allow updates both to infinite loops as well as cleanup functions, which was previously typically not possible. The lazy approach to state conversion ensures a relatively quick update process, while type wrapping incurs significant performance overhead both due to coercion functions as well as increased memory footprint of types and the related consequences - reduced cache locality, wasted memory as types are deleted, and general performance degradation during longer upgrade streaks.

So far, the discussion focused on update frameworks which allow incremental updates by updating values of types as they are accessed, and only some global state transformation functions for state which was created higher up in the call stack. The alternative approach of whole-program update is discussed in the next section, because it naturally fits multi-threaded program update as well. This does not, however, limit its applicability to single-threaded applications.

The authors tested Ginseng with three single-threaded open source servers: vsftpd (the Very Secure FTP Daemon), sshd (the Secure SHell Daemon implementation of the OpenSSH project), and zebra, GNU Zebra's routing software package. They found that they were able to meet their goals of not having to extensively change the application and not needing to restrict the form of their updates too much.

The measured overhead was less than 30% in all considered applications, which the authors deem acceptable [5].

2.2 STUMP

Ginseng's inability to deal with multithreading led to the exploration of the idea of keeping the lazy updating approach with a new tool based on Ginseng, but extended to support timely updates for multi-threaded programs. The resulting work, STUMP (Safe and Timely Updates to Multi-threaded Programs) does exactly that. It was created by Neamtiu and Hicks, who already collaborated on Ginseng. Most concepts still apply, but additional challenges need to be resolved. Their contribution is the introduction of induced update points, which are automatically generated from the programmer's specified regular update points (of which one to a few are needed per thread). Instead of stopping at an update point to apply the update, there is a sliding window with threads checking in and out to get updated (relaxed synchronization). This mechanism is handled by STUMP's runtime.

2.2.1 Multi-threaded Programs

While all challenges discussed so far also apply to multi-threaded applications, they present a lot of extra difficulties to consider. The number of possible system states increases dramatically [4] and thread interactions mean that both an updated and a non-updated function might access the same data simultaneously. Sleeping threads or those uninterruptibly blocking on some resource make update timing predictions hard or impossible.

2.2.2 Update points

Earlier the concept of update points was already discussed. To apply it to multi-threaded applications, additional safety constraints come into play: When a particular thread reaches an update point, other threads can be in arbitrary states. One obvious solution would be to block until all threads have reached an update point (barrier synchronization). Once that has happened, the update can be performed and execution can resume. As long as only a few update points exist globally, the programmer can reason about their relative interactions and conclude the safety of the update process [4].

In practice, this approach does not work very well. Arguing about update points is complicated and taxing for the programmer, so only a few points will exist per thread. Due to the barrier synchronization, all threads are unable to make progress once they have reached their update point, regardless of whether they could have performed more work safely while the other threads reach update points. Also, deadlocks are a prominent concern, and the programmer needs to take care to evaluate the interdependency of update points if they add more to alleviate the first concern mentioned here.

2.2.3 Induced update points

For multi-threaded applications, *induced update points* [4] add another aspect to the concept of update points to work around the aforementioned issues. For any programmer-inserted update point, static analysis generates so-called induced update points, which have the same safety guarantees as the selected update point. This analysis is conservative, but effective, in that it allows a per-thread update window (see figure 4) where updates are safely possible. Now, with the concern of too few update points alleviated, it would be possible to use all these update points and just continue with the old Ginseng implementation. Unfortunately, this does not address the concern that threads are prevented from making further progress as they block immediately as soon as they reach any update point.

2.2.4 Relaxed Synchronization

The other novel technique that STUMP introduces is *relaxed synchronization* [4]. Instead of requiring a multi-threaded program to block each thread at an update point, waiting for all threads to have reached such a state, and then proceeding with the update while all threads are blocked, releasing them only when the update process is completed, STUMP only requires that *update windows* overlap (see figure 5).

An update window is a series of induced update points which are equivalent according to the safety analysis. Threads check in when they reach the first statement belonging to an update windows, and check out when they have executed the last such statement. The update framework's runtime takes care of the necessary bookkeeping to know which thread is currently inside such an update window.

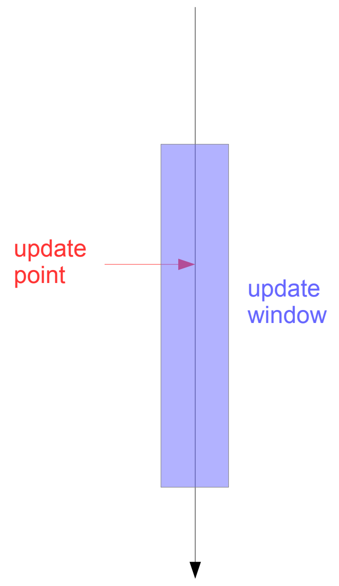

Figure 4: Stretching an update point to an update window; the arrow marks linear program flow

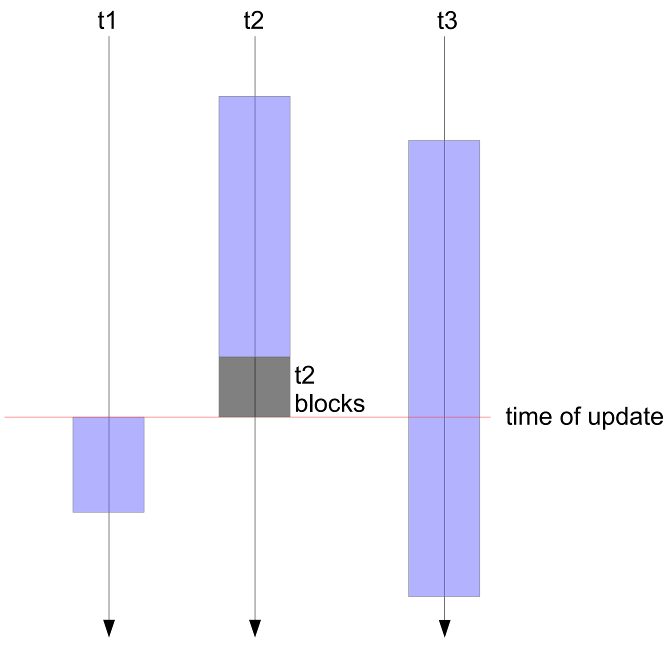

Figure 5: Relaxed synchronization: Threads t1, t2, and t3 progress towards an update. Only t2 has to block until t1 reaches its window.

This is functionally equivalent to forcing the programmer to manually specify update windows for each thread, except that there is a static safety analysis checker preventing mistakes. If necessary, additional input allows the window to be extended if the automatic analysis turns out to be too conservative, but is generally not necessary as the evaluation of STUMP will show.

2.2.5 *Evaluation*

The time necessary for all threads to reach an update window is of practical concern, as the evaluation of an implementation of relaxed synchronization will show compared to an implementation which does not make use of the concept.

The authors tested their implementation on three different multi-threaded server programs: Icecast (a multimedia streaming service), Memcached (a high-performance memory caching server) and the Space Tyrant game server with updates that spanned the time period of one year. Their results indicate that with a few added update points per thread and some annotations, quiescence was typically reached within less than 10 milliseconds. They also tried updating these programs without the additional techniques developed for STUMP, but updates either took very long (up to several seconds) to apply or did not apply at all due to deadlock.

Their overhead measurement for the tested applications showed that there was at most a 7% slowdown induced by STUMP. Their framework was able to be used to update all applications, after manual annotations were applied [4].

2.3 Kitsune

Rather than updating a program lazily one piece of data at a time, it might also be suspended and then updated as a whole, similarly to how updating traditionally works - except that program state is preserved and transformed, and updating is fast enough to not cause interruptions in the usage of the program. This is the approach chosen by Kitsune [2], a dynamic software update system which is inspired by Ginseng and UpStare, [3], the latter being another whole-program dynamic software update tool. For Kitsune, Hayden et al. combine the approaches of the two previous systems to enable whole-program multithreaded application updates at specified update points, while getting rid of some of the drawbacks. The whole program update approach appears more complicated than a lazy approach with on-demand updating, as the entire state of the whole program needs to be converted, including global, local and heap data as well as program text. A study of the amount of code writing necessariy to enable a program to be updated by the Kitsune update system shows that this is not a practical concern, however, as a lot of tools are available to smooth this process.

2.3.1 *Update Runtime*

The entire update system is powered by a very small (109 lines of C code) runtime system. An application gets linked against the update framework as a shared library. The update framework's runtime is thus the only part of the resulting application which cannot get updated. Due to its small size and relatively low complexity, it does not have to change with newer versions of the application.

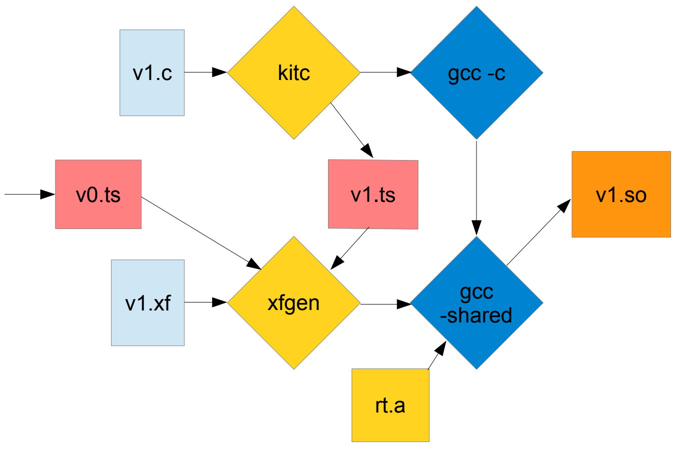

Figure 6: Kitsune's build toolchain

When an update is triggered, the runtime checks for the availability of a newer version of the software, also linked as a shared library (see figure 6). Modern operating systems allow loading of shared libraries at runtime, so nothing special has to happen for this step to work. This does not constitute the whole updating process however, as the program's state (stack and heap-allocated data, open file handles, open sockets, etc) all are not updated yet. The following sections explore how state conversion works in Ginseng, and show how it differs from Upstare.

2.3.2 *Converting Stack Data*

Converting data saved in the stack segment is tricky, as the data here is part of the call stack which might look completely different in the updated program. One method of handling this is *stack reconstruction* as implemented in UpStare [3]. The entire stack of the application has to be recreated by first being unwound all the way up the call stack to main, and then rewound to a new point of the application, as specified by the programmer. This approach lends great flexibility, as updates are possible from any program state, but the runtime costs are very high. All function calls get annotated to allow UpStare's runtime to correctly follow the stack, and the amount of correct follow up states for any given update time is too large to handle efficiently. The programmer, however, is relieved from the duty to specify an explicit state conversion function for converting stack data - the unwind/rewind code inside UpStare handles this automatically.

Also, in spite of the flexibility afforded by this appraoch and the high amount of additional code which gets generated, this does not mean that the programmer only needs to add a little bit of code for the transformation; instead, very many stack transitions have to be carefully evaluated by the programmer. If this mental burden on the programmer is to be reduced by only allowing a few update points (this is where Kitsune borrows from Ginseng), a manual approach can be chosen:

Instead of relying on the programmer to specify where to rewind the stack to, it is also possible to ask the programmer to provide update point alongside a conversion function for this specific update point. x Writing these functions is often easy, because the programmer already wrote the patch to the software, so they know what they changed and which part of the old application maps to the new application. A typicaly number for the amount of update points per thread is one or two. Adding more update points might make the update commence a bit more quickly, but the added burden on the programmer makes this not worthwhile: The state conversion functions need to be specified per update point, and careful evaluation of their safety needs to be performed.

The reward for taking the extra pains of specifying update points and conversion functions is a very speedy update process with no inherent runtime cost while the update is not being run.

2.3.3 *Heap Object Replacement*

In addition to the stack, memory allocated on the heap needs to be converted just the same. Kitsune's approach here differs from Ginseng dramatically: Instead of wrapping types, it explicitly transforms the entire state of the application, allocating new storage as necessary. This works similarly to a tracing garbage collector, where the heap is traversed. For global objects, this is not necessary of course: Their symbols are directly known. If more space is needed, additional memory is allocated, with the general structure of the heap being conserved. All no-longer needed old data is freed upon completion of the conversion process. This means that no overhead is added while the updating code is not being run, so application performance should be relatively unaffected compared to a regular instance of the program running without dynamic update support.

2.3.4 *Converting Other State*

There is more data to be converted, strings which are saved in the text segment need to have their locations updated. This is because after the update finished, the shared library with the old code gets unlinked from the application, such that the new program state is just the same as if the update never had taken place and the newer version had been started directly.

2.3.5 *xfgen*

To assist the programmer in writing type transforming code, Kitsune provides *xfgen*. It makes use of a domain-specific language which allows access to the complete old program state as well as all the new state which has been converted already. The syntax closely resembles C with a few additions, and the tool generates pure C conversion functions from the update specification. xfgen is called as a regular part of Kitsune's toolchain (see figure 6).

2.3.6 *Example*

The following code shows a simple program, and what changes when it gets updated. Initially, config variables are managed in an array, but in the updated version of the software, config options are stored in a linked list. The original C code (listing 1) and the xfgen update specification (listing 2) is shown.

The update point in the C code is marked by the call to *kitsune_update*, with its argument specifying an identifier for xfgen to identify the update point. The $kitsune_do_automigrate$ call allows all automatic type converters to run, whereas the part with the $kitsune_is_updating$ call is used to implement the programmer-specified transition from one state of the stack to another.

In the xfgen code, $\$out$ stores the new value of the updated type, $\$oldsym$ allows access to the original values of the entire state, and $\$newtype$ ensures the updated type is used when a type is specified by name. The other code is just C code to do the actual conversion from an array to a linked list.

Listing 1: C code implementing some kind of config argument storing in an arary. The code which will get updated is marked blue, the kitsune specific hook function calls are marked red

```
int c foo , c bar , c_size ; // config
int *mapping; // array of config options
int main() __attribute__((kitsune_note_locals)) {
  int main_sock, client_sock;
  kitsune_do_automigrate();
  if (!kitsune_is_updating()) {
      load_config();
      mapping = malloc(c_size * 4); }
  if (!MIGRATE_LOCAL(main_sock))
    main_sock = setup_connection();
  while(1) {
    kitsune_update("main"); //call runtime
    client_sock = get_connection(main_sock);
    client_loop(client_sock);
  }
}
```

Listing 2: xfgen code to transform the type of the variable mapping from an array to a linked list. xfgen specific syntax is marked red, updated types are marked blue.

```
struct list {
  int key; int val; struct list *next;
} *mapping;

mapping -> mapping: {
  int key;
  $out = NULL;
  for (key = 0; key < $oldsym(c_size);
        key++) {
    if ($in[key] != 0) {
      $newtype(struct list) *cur =
          malloc(sizeof($newtype(
                            struct list)));
      cur->key = key;
      cur->val = $in[key];
      cur->next = $out;
      $out = cur;
} } }
```

2.3.7 Evaluation

The authors used Kitsune to update 5 programs in total, three of them single-threaded and two multi-threaded. vsftpd (the Very Secure FTP Daemon), redis (a key-value store), and Tor (an implementation of an anonymity network) belong to the former category, while Memcached (a high-performance memory caching server) and icecast (a multimedia streaming server) make up the latter.

The needed code changes were relatively small, with a maximum of 159 lines of code added for the Tor software. This is due to an effect discussed below. Typically, even smaller changes were necessary.

The authors were unable to measure a runtime penalty for the tested applications, but measured up to 18.4% performance degradation when using Ginseng and up to 41.6% degradation when using UpStare to dynamically update the same applications. The update time itself is typically less than 40ms, with the exception of icecast, discussed below.

Kitsune uses POSIX reliable signals to notify a running application of an available update. It adds a signal handler to trap the USR2 signal, and trigger its runtime to begin an update once the next update point is reached.

Tor was a special case, in that it already made use of the USR2 signal which Kitsune uses to indicate that an update is to be applied. To support this, they extended Tor's control port interface to enable application updates. This is actually a very realistic scenario, as the control port is used to generally reconfigure and manage a running Tor relay. By adding a special update command which hooked into the Kitsune runtime, they were able to add a mechanism which Tor controller software could easily make use of to dynamically update to a newer version.

Icecast took longer to update because it has many threads which execute sleep cycles which last for up to one second. The authors note that this does not cause an issue in practice, and was the reason they did not change the sleep mechanism, which would have been an alternative had shorter update times been a requirement.

2.4 Comparison

Ginseng only works for single-threaded applications, whereas both STUMP and Kitsune are able to deal with dynamic updates to multiple threads. Therefore, only the single-threaded features of each system are compared first, and an analysis of the additional differences between STUMP and Kitsune follows.

All three tools offer the programmer support in writing the code which is necessary to transform state correctly, either via annotations or via xfgen instructions. Both approaches are more conservative than a carefully operating programmer would have to be, and in some cases manual intervention is required. This is always true when new fields are introduced to structures, because the tools cannot know what kind of initialization needs to be run before execution of the updated program can proceed.

Ginseng values fast update times over runtime overhead, which means it can generally apply updates very quickly. This is implemented using calls into the Ginseng runtime system at programmer-specified points, which means the programmer needs to ensure they specify enough update points that one will always be reached quickly. The programmer, of course, has to take care that it is also a suitable update point - meaning that they have to ensure code higher up the call stack is sufficiently loop-extracted to ensure the updatability of all parts of the program. This means that adding a new update point can be tricky, because it might be necessary to add additional loop extractions for code looking nothing like an infinite loop.

STUMP is based on Ginseng, and thus is very similar in architecture and work requirements for the programmer. Because it uses induced update points, which it infers automatically from an update point provided by the programmer, updates can often be applied more quickly. This does not mean, however, that the burden of the programmer to ensure picking good update points is lifted.

Both systems add quite a bit of overhead even to a not-yet-updated program, because they need calls into their runtime at update points (for Ginseng) and to check in and out (for Kitsune). Additionally, type padding does not only mean more memory is used for each value of a type, but also that properties such as cache locality can degrade heavily. If, however, the costs of these issues are manageable, both provide the ability to execute an update very quickly and lazily, meaning there might be a short period of time where application performance degrades somewhat, but the application does not need to stop to execute the whole update.

Additionally, after an update is applied, some old data will remain. This means that a program's memory usage can grow over time as updates are applied.

Kitsune, on the other hand, tries not to degrade runtime performance at the cost of taking a bit longer for an update. Its approach is that of whole-program update at some specified update points. This means it does not improve on the burden of the programmer to specify these update points. What it does do away with, however, is loop extraction. Because the whole program is updated at once, with all state transformed immediately, even code that would have been executed after a long-running loop terminated does not need to be handled specially. Unfortunately, some cases remain where the programmer needs to write more code, but Kitsune's goal is to make that code easier to understand.

Ginseng/STUMP and Kitsune have a philosophical difference with regard to programmer interaction with them. Kitsune wants dynamic software update "a first-class program feature" [2], whereas Ginseng/STUMP care more about making fewer changes to a (probably pre-existing) software base, at the expense of those changes being harder to reason about.

3. CONCLUSION

Several effective approaches exist to allow dynamic software updates of general-purpose software written in the C programming language. They have been tested with many open source server applications, and the results are satisfactory. Depending on the non-functional requirements for a given piece of software, and the relative maturity of it, allows choosing one over the other.

Ensuring the safety of such updates is supported by tools that aid the programmer in the automatic generation of patches, with only some manual intervention necessary. Even for multi-threaded applications with long-living stateful interacting threads, the concept of update points allows safe, state-preserving updates which execute quickly.

For the software developer, enabling their software to be updated dynamically is not as easy as simply linking in a library, but it also typically is not a task that would require re-designing the entire application. The benefits to the end user can very well be worth the work to enable dynamic software updating.

While both tools make different safety/performance trade-offs, they are not too far away from each other in terms of performance, so using the framework which the programmer finds more comfortable seems like the best way forward.

Kitsune's approach seems to be more appealing to me, as its goals include easy integration into the regular application development process; with easier to reason about safety requirements. If the maximum time allowed for an upgrade is more important than general application performance, Ginseng/STUMP can apply their strengths.

4. REFERENCES

[1] J. Arnolds. Ksplice: automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198, April 2009.

[2] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for c. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 249–264, October 2012.

[3] K. Makris. *Whole-program dynamic software updating.* PhD thesis, Arizona State University Tempe, AZ, USA, 2009.

[4] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 13–24, June 2009.

[5] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 72–83, June 2006.

[6] G. Stoyle, M. Hicks, G. Biermann, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 29(4):22, August 2007.

[7] G. P. Stoyle. *A Theory of Dynamic Software Updates.* PhD thesis, University of Cambridge, 2006.