WHOLE-PROGRAM DYNAMIC SOFTWARE UPDATING

by

Kristis Makris

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY
December 2009

WHOLE-PROGRAM DYNAMIC SOFTWARE UPDATING

by

Kristis Makris

has been approved

October 2009

Graduate Supervisory Committee:

Rida A. Bazzi, Chair

Kyung Dong Ryu

Donald Miller

Partha Dasgupta

ACCEPTED BY THE GRADUATE COLLEGE

This is a single-spaced version of the dissertation under a different style guide

ABSTRACT

Dynamic Software Updating (DSU) aims to update an old version of an application to a new version without causing downtime. This dissertation presents a new whole-program update DSU mechanism that can apply updates which cannot be applied by other DSU mechanisms. Compared to existing work, it has two main advantages that improve the updateability of applications. First, whole-program update effects updates immediately: atomically and with bounded delay. Immediate updates are not supported by existing DSU systems, but immediate updates are necessary to safely update multi-threaded and multi-process applications without service interruption. Second, whole-program update can update functions and data that are active on the stack. To update functions active on the stack, existing update mechanisms rely on the user to anticipate the future evolution of an application, or on the application to quiesce (which may never happen). To update data active on the stack, existing update mechanisms rely on data-access indirection, which can incur unacceptable overhead.

This dissertation also presents a compiler-based DSU system, called UpStare, that implements the whole-program update mechanism and offers useful safety guarantees. UpStare automatically converts applications to be updateable through source-to-source transformations. It also automatically prepares DSU patches to apply an effective mapping of the state of the old version of an application to the new state. This significantly reduces the input needed by the user in preparing an update mapping. Experience using UpStare to update real-world, multi-process server applications shows that UpStare can systematically apply safe updates with little to no user intervention for typical application use cases. However, updating an application anywhere during its execution with minimal user input needs additional analysis to verify updates are semantically safe. UpStare reports comparable or less overhead than the current state of the art and the overhead can be reduced with additional optimizations.

ACKNOWLEDGMENTS

This dissertation would not have materialized without help from many people.

My parents sacrificed a lot for me to have the opportunity to produce this dissertation. Besides having their unconditional love and support, I am fortunate for them being educators. I am also fortunate that my sister had paved the road and could offer advice.

I am indebted to my advisor Rida Bazzi for his guidance, continuous support, and encouragement while doing research. Rida was patient, helped think abstractly, and contributed many ideas, including the suggestion to experiment with stack reconstruction. He helped with organizing and presenting this work and taught captivating compiler construction courses.

I need to thank my former advisor Kyung Ryu who diligently supported me and provided me with opportunities, motivation, and an appreciation for persevering to produce results. I am grateful for Donald Miller's support and fascinating operating system internals courses without which I would have not attempted to explore DSU. I also thank Partha Dasgupta for his support and many insights into real-world systems.

This work benefited considerably from the availability of CIL and from discussions with, and feedback from, Michael Hicks and Iulian Neamtiu. I thank them for providing a KissFFT instrumented with Ginseng and for making Ginseng available. Discussions with Andrew Baumann and Jonathan Appavoo were also essential in better understanding the DSU problem.

This dissertation was supported in part by NSF Grant CSR-0849980.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1
INTRODUCTION

The ability to continuously run systems and applications is a critical business and technology need. Downtime can disrupt the service a business provides and is expensive [1, 2]. It can lead to direct loss of revenue, and during the downtime resources which have already been paid for remain unutilized. A study on business continuance in various industries conducted by The Gartner Group [3, 4] reported that the cost of one hour of downtime ranges from $1.3M in Information Technology, $1.5M in Financial Institutions, $2.8M for Credit Card Sales Authorizations, up to $7.8M for Brokerage Operations, and an average of $0.9M per industry. In another study [5] 28% of correspondents reported each hour of downtime would cost their companies between $51K-$250K, 18% said it would cost between $251K-$1M and 8% said it would cost over $1M.

Avoiding downtime of applications is an important problem that has not been adequately addressed. Downtime is still experienced in many non-stop environments, such as medical [6, 7], air-traffic control [8, 9, 10], telecom [11, 12, 13], financial exchange [14, 15], power plant [16], and spacecraft flight systems [17, 18]. A recent study reports [19] that minimizing downtime is one of the top three concerns of large enterprises (over 10,000 employees). In 60% of the cases the primary source of downtime is application changes [20], but application changes are inevitable. Feature additions and bug fixes (bugs cost the U.S. economy between $22.2B - $59.5B annually [21]) are part of the software maintenance lifecycle.

The dominating approach of minimizing downtime is to maintain hardware and software replicas and plan outage periods [22] to switch service to the replicas. However replication has three disadvantages. First, replication still disrupts application execution. If a hardware replica of an application is available, but the application does not offer software-level replication, access to the old replica must be disabled before the new one is enabled else data may be lost. Second, replication cannot help if an application is upgraded to a new version that is not backward compatible with the old version (e.g. when communication protocols [16] or database schemas are updated). Even if software-replication is available, the incompatibility between versions can make it impossible to replicate data. Third, the cost of planned outages, along with the cost of redundant hardware and system administration, may be prohibitively high in some environments.

Dynamic Software Updating (DSU) aims to maintain running software without causing downtime. It is the ability to replace an old version of a program with a newer version during runtime without killing the application process. DSU is desirable because of its potential to apply an update with little or no impact to the service provided by the application. For example, updating an application during runtime can preserve open network connections and in-memory data. A typical dynamic update consists of: (1) pausing the execution of the old version in a given state, s; (2) applying a state mapping function S to s to obtain a state $S(s) = s_{new}$; and (3) resuming 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 will be described more precisely in Chapter 2.2). 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 DSU problem consists of two components:

- *Update safety*: Determining the states, or execution points, of the old application for which it is possible to apply a valid update. And determining for these states the state mapping function to effect the update.
- *Update mechanism*: Effecting the update through a mechanism that maps an old execution state to a new execution state.

In general, the *update safety* problem is undecidable [23]. 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. And 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.

Regarding the *update mechanism* problem, current DSU mechanisms are limited in their support for update of *active functions and data structures* [24, 25, 26, 27, 28] and in their support for *immediate updates* [29, 30, 31, 32, 33, 34]. To support the update of functions that are active on the call stack they rely on *quiescence* (functions that will be updated should not be active on the stack [24, 35, 28]) and for the user to anticipate long-lived loops and mark them for code replacement [29, 32] or to split them in multiple logical stages [33, 36]. To support the update of data structures active on the stack [32, 33, 37] they rely on *data-access indirection* to access the data structures, which can incur unacceptable overhead in computationally-bound applications. One approach pads datatypes beforehand with enough room to

accommodate future growth [32], but after many updates there may be no space left to accommodate an update, and the padding affects negatively the data cache.

Immediate updates are not supported by existing DSU systems. An update is immediate if it satisfies:

- *Atomicity*: before the update only old code executes and after the update only new code executes;
- *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* [38, 39]: 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 delay is not necessary for logical consistency, for multithreaded applications immediate updates are needed to provide logically consistent updates *without service interruption*: the update does not cause the service to be unavailable for an unbounded amount of time.

1.1 Scope of this Dissertation

This dissertation addresses the limitations of current DSU systems by focusing on the *update mechanism* problem. It does not aim to solve the *update safety* problem, although it provides useful safety guarantees that can help address it. The dissertation argues that a strong solution to the *update mechanism* problem will allow approaching the *update safety* problem from a different viewpoint.

Current DSU mechanisms are restricted in their ability to update active functions and data structures. They are not able to update the entire old state s when producing s_{new}. They are able to update only part of the state. They handle this restriction by updating from a limited range of application states for which this restriction does not impede applying a safe update. They identify old states (often these old states are quiescent points) for which the update mechanism is able to safely produce the entire new state s_{new}. The number of these states is limited (and sometimes there are no such states), and identifying a mapping to the new state s_{new} from these points is often simple and requires little user input. In summary, existing work has focused on the identification of (limited) old states that are valid for the DSU mechanism and has not been concerned with identifying complex state mappings.

An ideal update mechanism would be one that is capable of updating the entire old state. For example, update all functions on the stack, replace or remove functions from the stack, add more functions on the stack, modify function signatures, modify return addresses, provide more room for all stack-resident variables, and modify the

Program Counter of all threads. Such an update mechanism would be stronger than existing mechanisms because given any old state for which it is possible to apply a valid update, and a mapping to the new state, it would always be able to produce a new state. There would be no restriction to the identification of an old state to apply an update due to the update mechanism. Any restriction of identifying a valid old state would be attributed only to the difficulty of identifying a mapping to the new state. The challenge would be to produce such a mapping with little user input. This state mapping can be more complex because it has to consider more aspects of the state (such as stack-resident data and return addresses) than existing DSU mechanisms.

Thus a stronger update mechanism would shift the focus in solving the DSU problem from the identification of (limited) old states that are valid for the DSU mechanism (and which require simple state mappings) to minimizing user input for identification of possibly complex mappings. This shift of focus can lead to higher updateability in applying DSU.

Another aim of this dissertation is to apply *immediate* updates of multi-threaded applications. The dissertation supports that this is possible with a strong update mechanism that can *atomically* update the entire state. It should also be possible for such an update mechanism to apply the update with *bounded delay*. For example, it should not be necessary to wait indefinitely until long-lived computationally-bound loops finish, or until blocking system calls, such as I/O calls, finish. Offering bounded delay does not seem far fetched. This notion is found in the preemptible Linux 2.4 [40, 41] and K42 operating systems [42], and hard-realtime embedded systems in general.

1.2 Overview of this Dissertation

The rest of this dissertation is organized as follows:

Chapter 2 discusses the DSU problem, presents safety considerations, justifies the need to apply immediate updates, proposes an update model of *whole-program update* and compares software updating mechanisms.

Chapter 3 surveys the literature and presents the various approaches that can be used to apply software updates, including dynamic software updating.

Chapter 4 presents a DSU system implementation of the whole-program update model called UpStare and describes how the state of an application is mapped using *stack reconstruction*, how the update of multithreaded applications is supported by forcing all executing threads to block, and how *bounded delay* is provided by transforming blocking system calls into non-blocking.

Chapter 5 describes the *dynamic stack tracing* approach and how it can enforce various runtime safety checks.

Chapter 6 presents the evaluation of UpStare in updating real-world applications and in studying the sources of overhead of the implementation.

Chapter 7 discusses future work and concludes this dissertation.

Chapter 2
THE DYNAMIC SOFTWARE UPDATE PROBLEM

This chapter introduces the dynamic software update problem. It discusses safety considerations, argues for the need to apply immediate updates, and presents existing update mechanisms and their impact on updateability.

2.1 Dynamic Software Update

A DSU effects both the program code (functions) and the program state (global and local variables). Given a program (Π, s), where Π is program code and s is an execution state, a typical dynamic update of Π to Π_{new}, where Π_{new} is a new version of Π, consists of: (1) pausing the execution of 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}. 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. However, 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.

Applying a dynamic update results in a hybrid execution of the running application. In general, this hybrid execution needs not satisfy the semantics of either the old or the new versions. The desired semantics need to be determined by the user. 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 applied to state s such that the resulting hybrid execution satisfies the desired semantics. The DSU problem has two aspects:

- *Update safety*: First, determining the states s, or execution points, of the old application for which it is possible to apply a valid update. Second, determining for these states the state mapping function $S(s)$ to effect the update and produce s_{new}.
- *Update mechanism*: Effecting the update through a mechanism that maps an old execution state to a new execution state.

Gupta [23] 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 [43] between two versions of a program. Identifying semantic differences has been studied extensively and is also undecidable although safe approximations are known [44]. Identifying semantic differences dynamically [45, 46] requires large amounts of memory as it needs to maintain an execution trace history. Little work [47] has investigated how semantic differences can be dynamically validated efficiently.

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 presented.

2.2.1 Type-Safety

Type-safety guarantees 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.

2.2.2 Transaction-Safety

Transaction-safety guarantees that some sections of code that are denoted by the user as transactions 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 to prohibit 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.

2.2.3 Representation Consistency

Representation consistency involves consistency of both the state s and the program Π:

- *State representation consistency*: guarantees that at no time does the executing application expect different representations of state, such as global variables or stack-frame contents (local variables, formal parameters, return addresses).
- *Program representation consistency*: guarantees that 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.

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 [30], 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.

The concept of *version consistency* introduced in related work[39, 34] is equivalent to our definition of program representation consistency.

2.2.4 Logical Representation Consistency

Logical representation consistency guarantees that a hybrid execution of both old code Π and new code Π_{new} is indistinguishable to an outside observer from executions that are obtained with representationally consistent updates [38, 39].

2.2.5 Thread Safety

Thread-safety guarantees that type-safety, transaction-safety, and logical representation consistency are provided in multithreaded applications.

In general if a DSU system ensures that a particular safety guarantee (e.g. type-safety) is satisfied for individual threads independently, then that safety guarantee is not necessarily satisfied when all threads execute together. This is further discussed in Chapter 2.3.

2.3 The Need for Immediacy

This dissertation argues that immediate updates are needed to guarantee that the update of common multithreaded applications is logically consistent and can be achieved without unbounded service interruption. Before justifying the need for immediate updates, it is necessary to 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 logical 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. Wait until all connections that use the old version terminate. 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 either logical representation consistency is violated or the thread running the old version is forced to be transformed to the new version. Since violating logical consistency is not an option, the only option left is 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.

2.4 Update Mechanism

This section introduces a general model for the mechanism of applying an update which is termed the *whole-program update* mechanism. As already discussed, the update mechanism itself does not consider the validity of the update. For simplicity the model does not consider program state stored in files, which has also not been considered by existing work so far.

The model of the update mechanism is more detailed than existing work in two major respects. First, it considers stack frames of all threads as updateable program state. Stack frames include local variables, formal parameters, and return addresses. Second, it considers the Program Counter (PC) of all threads as updateable program state. Considering the program state s in more detail allows us to better understand and compare existing mechanisms as restricted forms of the whole-program update mechanism.

This section compares the update mechanisms of *interrupt-update-restart*, *binary instrumentation*, *function-pointer indirection*, *logical-stage extraction*, and *data-access indirection*. It outlines the restrictions of each mechanism and concludes that the *whole-program update* mechanism is the least restricted update mechanism. Table I summarizes the comparison of the update mechanisms that are presented next.

	Produce Π_{new}	Preserve *os*	Update h	Update T_{PC}	Update $T_{sf(l,p,ra)}$
Interrupt-Update-Restart	Yes	No	Yes	Only reset $T_{PC} = \{main_{PC}\}$	Only set $T_{sf} = \{\}$
Binary Instrumentation	Yes	Yes	Cannot enlarge variables	Yes	For any T_{sf} update current ra Cannot enlarge variables in l or p or add new elements
Function-Pointer Indirection	Not all of Π_{new}	Yes	No	No	No
Logical-Stage Extraction	Not all of Π_{new}	Yes	No	No	No
Data-Access Indirection	No	Yes	Yes	No	Only l and p of any T_{sf}
Whole-Program Update	Yes	Yes	Yes	Yes	Yes

TABLE I
COMPARISON OF UPDATE MECHANISMS.

2.4.1 Whole-Program Update

Given a program (Π, s), where Π is program code and s is an execution state, a dynamic update mechanism produces Π_{new} and applies a state mapping function S to s to produce a new state s_{new}.

Program code Π is a set containing the executable code of all functions of the old program.

Program code Π_{new} is a set containing the executable code of all functions of the new program.

A program state $s = (os, h, T_{sf}, T_{PC})$ of program Π is a tuple consisting of a set os containing all state maintained by the operating system related to the program (e.g. open network connections and file descriptors), 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, a set p of the formal parameters, and the return address ra.

The model restricts its attention to updates that aim to preserve the os state. While in general it is possible to update the os state, it is expected that for some applications the os state only needs to be preserved.

Program state s_{new} is a program state of Π_{new} produced by applying the state mapping function S to the program state s of Π.

The *whole-program update* mechanism can produce Π_{new} and all aspects of s_{new}.

2.4.2 Interrupt-Update-Restart

The *interrupt-update-restart* mechanism kills an application process, updates the program code and restarts the application.

This mechanism can produce Π_{new} and it can update h. However it is unable to preserve os. Its update to T_{PC} is limited to setting T_{PC} to have only one element: the PC of the starting thread, which is reset to the starting state of the PC. The update to T_{sf} is limited to setting $T_{sf} = \{\}$.

2.4.3 Binary Instrumentation

The *binary-instrumentation* mechanism modifies a program and its state by operating directly on its memory image.

This mechanism produces Π_{new} by modifying Π in-place. It can preserve os, and update the values of existing variables in h but it cannot extend the size of variables in h. It can modify T_{PC}. It can update the ra of the current sf for each thread. It can also update the values of existing variables in l and p but it cannot extend their size add cannot add new elements in these sets.

2.4.4 Function-Pointer Indirection

The *function-pointer indirection* mechanism changes the address of the function that will be invoked next.

This mechanism can produce only parts of Π_{new}. For some old code Π executing that needs to be updated (e.g. main()), this mechanism cannot produce this code in Π_{new}. It can preserve *os*, but it cannot update h. It also cannot modify any aspect of T_{PC} or T_{sf}.

2.4.5 Logical-Stage Extraction

The *logical-stage extraction* mechanism allows separating function bodies into logical stages and updating them individually, either through function-pointer indirection or code inlining.

This mechanism can produce only parts of Π_{new}. It cannot produce Π_{new} for code in Π that crosses logical-stage boundaries. It can preserve *os*, but it cannot update h. It also cannot modify any aspect of T_{PC} or T_{sf}.

2.4.6 Data-Access Indirection

The *data-access indirection* mechanism allows accessing data either through dereferencing a pointer or using the help of a page or trap handler.

This mechanism cannot produce Π_{new}; it can only update the state s. It can preserve *os*, update h, and update only l and p of any T_{sf}. It cannot update the *ra* of any T_{sf} and cannot update T_{PC}.

2.5 Updateability

DSU systems range in their ability to update from as many old states as possible (coverage), to what extent they can update program code and state (complexity), in their ability to preserve the service provided by an application to other applications (no service interruption), and the input required by the user. These capabilities are loosely referred to as the *updateability* provided by a DSU system. Carefully balancing these capabilities is the major challenge in designing a DSU system. On one hand, limiting coverage can decrease the complexity of updates that can be applied and the input required by the user. On the other hand, increasing coverage also increases the complexity of updates that can be applied but requires more input from the user. Each of these capabilities is described next.

2.5.1 Coverage

The ability to use an update mechanism that can update from many old states is valuable for two reasons. First, it allows inspecting many old states to determine if they are safe for update: states that are valid for update and for which a state mapping function can be identified. Second, it allows an application to be updated very quickly from the time an update is requested. In contrast, if the update mechanism can update from very few old states, the number of states that can be determined to be valid for update becomes limited. Given an update mechanism that is considerably limited in the old states it can support, some updates may not be applied at all.

The concept of *timeliness* has been introduced in related work to describe the appropriate time at which applying an update will be safe. It is important to clarify that existing discussions on timeliness in the literature are in fact discussions on coverage. An update may occur over a transitional period of time in which the state mapping function $S(s)$ is applied. It is possible to incrementally produce parts of s_{new} while parts of Π and parts of Π_{new} are both executing at the same time, as long as logical representation consistency is not violated. To ensure logical representation consistency under this hybrid execution mode, some parts of s_{new} need to be produced at a time that would not endanger safety, such as type-safety and thread-safety. This "safe time" is a set of old states: states where no *ra* of any T_{sf} and no T_{PC} are set so that pending execution of code for a stack frame would violate safety or logical representation consistency.

2.5.2 Complexity

Update mechanisms can be compared according to the complexity of the updates they can support. For example an update mechanism that is unable to update program code applies updates of lower complexity compared to a mechanism that can update program code (with everything else being equal). In another example, an update mechanism that can update only global variables on the heap but not local variables on the stack applies updates of lower complexity compared to an update mechanism that can update both global and local variables. Generally, update mechanisms that can apply updates of high complexity require more user input in producing a valid state mapping.

Mechanism m is *more complex* than mechanism m' if every state mapping that can be done with m' can also be done with m. Comparing update mechanisms according to the complexity of updates they support defines an order relation on the update mechanisms, but this order relation is not a total order.

2.5.3 Service Interruption

Some updates can affect how the application interacts with its environment, but such updates would require an update of the environment as reflected in the changed interaction. This dissertation concentrates on updates that do not affect this interaction. For such updates it is important to preserve the interactions as they existed prior to the update. Update mechanisms are classified according to their ability to preserve *os* because this state captures the state of interactions with the outside world. Losing *os* can interrupt the service provided to other applications.

Service can also be interrupted if the update takes a long time. So, *update duration* is also of interest when considering service interruption. An update can take a long time if it requires updating a large program state (in the order of gigabytes), or data from disk (which this dissertation does not consider). If the state of the application is large and an immediate update requires a long amount of time, one might consider two other alternatives. In one alternative, the update can be postponed until the state of the application that requires update becomes small enough to be updated quickly. This might happen if some data structures are freed on the stack or the heap for example. In another alternative, the update can be applied incrementally so that only portions of the state (and code) are updated at time. This is not always possible, to do safely, but, if it is possible, it might be an attractive option. If neither of the two alternatives are possible, then service would be interrupted due to the state mapping. Such an interruption is independent of the ability of the update mechanism to preserve the internal *os* state. Even if the state of interactions with the outside world is preserved (e.g. existing network connections remain open), a long update duration may require the application to be unresponsive for a period of time. This may violate timing constraints of the interactions of the application with its environment and be equivalent to service interruption. This dissertation does not address such timing constraints of the interactions of an application.

2.5.4 User Input

In general, the state mapping function S cannot be automatically validated [23]. Availability of user input in validating the state mapping and supplementing it if necessary improves coverage. An increase in update complexity can increase the amount of user input needed for validation. For example, applying an update when multiple threads are running, and with multiple stack frames active for each thread, increases the amount of user input needed to ensure the update is valid compared to a single-threaded application with few active stack frames.

2.6 The Use of Update Mechanisms for DSU

DSU systems need to address both the *update mechanism* and the *update safety* problems to be successful.

Existing systems have employed combinations of the update mechanisms presented in this chapter. As already shown, the choice of update mechanism affects the updateability provided by a DSU system. This section evaluates the most commonly and successfully used combinations of update mechanisms in their ability to provide updates. Table II summarizes this evaluation.

	Coverage	Complexity	Service Interruption	User Input
Whole-Program Update	High	High	No	High
Interrupt-Update-Restart	Application-dependent	Low	Yes	Low
Binary Instrumentation	Medium	Medium	No	Medium
Indirection and Extraction	Medium	Medium	No	Medium

TABLE II
IMPACT OF UPDATE MECHANISMS ON UPDATEABILITY.

2.6.1 Interrupt-Update-Restart

Using the *interrupt-update-restart* mechanism, applications [48] or operating systems [49, 50] are stopped (killed, losing volatile state), have their persistent (on-disk) state updated, and restarted.

A serious limitation of this update mechanism is that it interrupts the service provided. Moreover, the update coverage depends on the state information maintained by the application. If the application does not maintain state information or if the information maintained can be discarded with no effects on application consistency (for example, if the state is saved to permanent storage and its representation does not change), then a restart can be harmless. If the state of the application cannot be discarded and the application is signalled to stop, then it might stop leaving its state logically inconsistent. So, in general, the update would have to be acted upon when the application is in a state (if such a state exists) that can be discarded. For applications that maintain network connections, this might require an indefinite wait for clients to close connections so that the application reaches a state in which an update can be applied.

If the persistent state of an application needs to be mapped between versions, this state is often easy to map for two reasons. First, the state does not include in-memory data of multiple threads when the mapping is applied. Second, users often develop mappings from the old persistent state to the new persistent state. One example are the pg_dump/pg_restore data migration utilities of the PostgreSQL DBMS. But very often the persistent state of the new version is backward compatible with the old version and no mapping is needed at all. Because the persistent state is often backward compatible and no volatile state is preserved this mechanism often needs no semantic safety checking.

2.6.2 Binary Instrumentation

Using the *binary instrumentation* mechanism, applications [51] or operating systems [52, 53] are instrumented to apply updates of basic blocks the next time the basic block is called.

Binary instrumentation is a powerful update mechanism. It is one of the few update mechanisms that are able to modify the return address ra of the current stack frame of each thread and the set of program counters T_{PC}. However in practice it has not been used to update these parts of the state and it is typically used to apply updates of low complexity. The reason is that it requires safety analyses to update most parts of the state and such analyses can be inconclusive for complex updates. For example, modifying stack frames to add, enlarge, or move around, local variables or formal parameters requires a corresponding modification in Π_{new} to refer

to the local variables and formal parameters with the appropriate offsets. Although often effective, the binary code analyses used to apply such modifications can be inconclusive because of the possibility that Π incorporates self-modifying code, data-in-code and code-in-data [54, 55, 56, 33]. Similar analysis is needed if the return address *ra* of the current stack frame of each thread or the set of program counters T_{PC} are modified.

This update mechanism has been generally used to interpose additional code in basic blocks, and hence apply small and isolated updates. It extends the semantics of an updated system by adding performance profiling, debugging, and optimization capabilities. The safety that must provided by a DSU system using this mechanism is often limited to preserving the existing program semantics, which can be achieved at a fine-grain through control-flow graph and live register analysis [52, 53]. It is also possible to apply updates of higher complexity, which will require user input to validate, by updating complete functions [57, 58, 33]. The advantage of using binary instrumentation to apply updates of low complexity is that it offers high coverage.

2.6.3 Indirection and Extraction

Using the *function-pointer indirection* mechanism applications [24, 57, 58, 27, 59, 32, 28] or operating systems [35, 33, 37] are instrumented to apply updates of functions the next time the function is called. This mechanism has been combined with *logical-stage extraction* (in applications [29, 32, 36] and OSs [60, 33]) and *data-access indirection* (in applications [32, 28] and OSs [61, 33, 37]) to be more effective. *Combined, these are currently the dominant update mechanisms used by DSU systems.* These update mechanisms are evaluated in their ability to update single-threaded and multi-threaded applications.

Single-threaded applications. In single-threaded applications, these mechanisms often offer high updateability. They have important restrictions on coverage because they cannot update all parts of the state, such as the T_{PC} or the *ra* of the current stack frame. But often an application reaches a *quiescent* point where the application state is known to be logically consistent and most functions that will be updated are inactive. After this point the new version of functions that will be updated can be entered again using function indirection. Few active functions cannot be updated (such as the main() function), and in practice this often does not prohibit applying complex updates. If infinite, or long-lived, loops need to be updated, the loop body can be treated as a function that can be updated on entry with some user help [29, 32].

It is possible to further improve the updateability provided by these mechanisms. Instead of applying an update only at the quiescent point, it is possible to update some functions and data eagerly. Under such an eager update, new versions of functions

are still entered using function indirection and they are activated before the quiescent point is reached. This increases the occurrence of code paths executing old code, new code and then old code again. If an eager update is applied, it must be first guaranteed that such an update does not violate type-safety, transaction-safety, or logical representation consistency [38, 39].

Guaranteeing a safe eager update is not straight-forward. It requires either static (at compile-time) or dynamic (at run-time) safety checking. For example, guaranteeing type-safe eager updates statically involves computing *universally* type-safe update points, which is bound to be a conservative approximation that limits the number of old valid states the update can be applied from. Guaranteeing type-safe eager updates dynamically involves evaluating type-safety based on the execution context, which requires instrumentation for dynamically tracing the call stack of the program and incurs overhead, although this approach is more accurate.

Data can be updated in various ways. First, data can be padded [32] beforehand with enough room to accommodate datatype growth. But the padding can affect negatively the data cache and after many updates there may be no space left to accommodate a datatype update. Second, the appropriate version of a datatype can be retrieved during runtime using data-access indirection [32]. This indirection occurs for every data access and leads to considerable overhead in data-intensive applications. Third, semantically non-conflicting new field additions can be maintained separately in *shadow data structures* [33, 37] and accessed through pointer indirection. This is more efficient than general data-access indirection because indirection is applied only for new field additions. Fourth, instances of old and new data can coexist [59] and can be maintained through forward and backward data transformers [30, 28], but this may not be possible if the datatype semantics are no longer compatible.

Multi-threaded applications. In multi-threaded applications, these update mechanisms offer low updateability because it is not possible to apply an update for a large number of old states without violating logical representation consistency. As discussed in Chapter 2.3, this difficulty is due to the possibility of unbounded delay in the execution of some threads and the inability of these update mechanisms to effect the update in its entirety (atomically). Splitting the body of long-lived loops into multiple *logical stages* [33, 36] and coordinating the update in multiple phases [33] have been suggested as approaches to increase the number of old valid states, hence improve overall updateability. These approaches are effective only if the update does not violate logical representation consistency. They are also unable to update in the presence of unbounded delay, thus they can interrupt the service provided by an application indefinitely.

2.6.4 Whole-Program Update

Under the *whole-program update* mechanism, an application or operating system is paused, updated in its entirety, and resumed.

This mechanism can provide high updateability because updates can be considered for validity from all old states. The update mechanism does not limit the number of old states from which an update can be safely effected. This wide coverage of old states is also the main challenge of the update mechanism. Safely mapping from the old state to the new for such a large number of old states with little user input needs semantic safety analysis.

This update mechanism also provides two important capabilities that were not provided so far by existing update mechanisms. First, it can update functions that are active on the stack. This is not possible without service interruption using the interrupt-update-restart mechanism, and it is not always possible using binary instrumentation. Second, it can update global variables and local variables active on the stack with no data-access indirection. This is an important benefit because data-access indirection leads to considerable overhead in data-intensive applications.

This update mechanism has been applied by Shanhbhag [62] for portable cross-version checkpointing and recovery for DSU, but some state mappings were not supported. An early form of *stack reconstruction* [63] based on continuation-style programming was also applied to implement whole-program updating for large-scale process-oriented parallel simulations. But it uses data-access indirection and continuously saves execution continuations during program execution.

2.7 Conclusion

The update safety problem is undecidable, which means it is not possible in general to automatically determine if a given old state s is valid for update or to determine a state mapping function $S(s)$ to effect the update and produce s_{new}. In general, user assistance is required both to identify valid states and guide the state mapping. An effective DSU system needs to minimize user involvement. This can be achieved by providing a default state mapping and providing some restricted safety guarantees that are satisfied by an updated program. Some useful safety guarantees are type-safety, transaction-safety, actual and logical representation consistency, and thread-safety.

The *updateability* provided by a DSU system depends on its ability to update from as many old states as possible (coverage), the extend to which it can update program code and state (complexity), its ability to preserve the service provided by an application to other applications (no service interruption), the safety guarantees it can provide, and the input required by the user.

Some update mechanisms are unable to update from many old states but provide useful safety guarantees with little user help. For example, the interrupt-update-restart mechanism cannot preserve volatile state and depending on the application it may not be able to initiate an update immediately. However, it can apply complex updates with the guarantee that updates will be semantically safe with no input from the user if the persistent state is backward compatible. Combined, the mechanisms of indirection and extraction are also unable to update from many old states, but they can apply updates for many single-threaded applications with some user help in identifying quiescent points and long-lived loops. In multi-threaded applications, the capability to update immediately is necessary. Binary instrumentation is a powerful update mechanism but updates applied with it in practice have been limited to mostly small extensions. The whole-program update mechanism has not been substantially explored before this work. Although it provides a wide coverage of old states from which it can update, the main challenge of this mechanism is to minimize the input required from the user in verifying updates are semantically safe.

Chapter 3
RELATED WORK

This chapter surveys various approaches that have been used to apply dynamic software updates. For each approach it indicates which update mechanism, or combination of mechanisms, the approach uses. The approaches studied are extensible design, binary instrumentation, dynamic updating, replication, virtualization, and checkpointing. Continuation-style programming is also studied, which shows promise for applying DSU.

3.1 Extensible Design

A variety of operating systems have been designed with extensibility in mind to facilitate software updates of their core. All of these operating systems essentially use the logical stage extraction and function pointer indirection update mechanisms to apply their new components.

The Synthesis [64] kernel dynamically specializes operating system services to improve performance. This specialization is limited to producing new versions of the services that are backward compatible with the old versions.

Similarly, the Synthetix [65] kernel applies optimistic and incremental specialization to tune system services and applications. It supports concurrent execution and replacement of functions using concurrent dynamic linking (function-pointer indirection) and allows adaptively reconfiguring kernel services at fine granularity. To ensure safety of updates, Synthetix blocks entrance to functions that are executing on the stack. It also assumes only one kernel thread runs per process, which implies there is no possibility of multiple kernel calls concurrently accessing process level data structures. Complex updates of complete subsystems and datatypes are not addressed.

VINO [66, 67] is an operating system specially crafted to be adaptive and extensible. It collects performance data and applies heuristics to adapt the system according to its workload. Safety of user extensions to VINO [60] is provided through transactions, resource accounting, and static checking.

The Exokernel [68] is an operating system that can be extended by untrusted applications using a library, but extensions are not dynamically applied. SPIN [69] is another extensible operating system but it does not allow dynamically updating kernel services either.

K42 [70, 35] is an object-oriented operating system that supports DSU. Objects are accessed indirectly through a common object translation table (function-pointer indirection). Safe update in K42 is supported using a thread-generation counter that detects quiescence of objects before they are replaced. The system is guaranteed to reach a quiescent state because the operating system is specially designed so that all kernel threads are short-lived and non-blocking.

3.2 Binary Instrumentation

The binary instrumentation mechanism has been used for a variety of applications including profiling, debugging, optimization, as well as dynamic updates. Only a few of the systems that use binary instrumentation are focused on dynamic updates, and these systems combine binary instrumentation with data-access indirection. This section describes work that uses binary instrumentation including work that does not address the dynamic software update problem. This allows getting a better understanding of the limitations of this mechanism in general and its potential to be used in future DSU systems.

Static binary rewritters like Mtool [71], ATOM [72], EEL [73] and QPT [74] manipulate the executable instructions of functions to add performance monitoring capabilities. Since static rewriting is applied before an application is started it cannot be used for dynamic software updates.

Dynamic binary instrumentation systems extend this capability to unobtrusively modify applications during runtime. To safely do so, they need to guarantee that the active state is not corrupted. This state is more complex than the state considered by static binary instrumentation systems (only program code) because it includes the values of processor registers, global variables and the stack state of multiple threads. For example, DynInst [51] adds performance profiling and debugging capabilities in user-level and parallel applications. KernInst [52] applies dynamic instrumentation, and control-flow and live register analysis, in a live operating system kernel on the fixed instruction-length SPARC architecture. GILK [53] applies similar instrumentation and analysis on the variable instruction-length i386 architecture. DTrace [75] and the Dynamic Kernel Modifier [76] leverage a kernel that has been recompiled to contain trace points to simplify instrumentation. These systems focused on unobtrusively interposing performance monitoring and debugging code, and the updated code generally preserves application semantics. Dynamic binary instrumentation approaches have also been used extensively for program optimization by Dynamo [77], Diota [54] and Pin [78]. The instrumentation has been limited to either interposing code at the basic block level or producing optimized code that is backward compatible with the old version of the code.

In general, dynamic binary instrumentation have not addressed safe updates of active functions and data structures or of complete subsystems. Supporting such updates is possible but complex. It requires consulting the compiler ABI, such as the calling convention for function calls and the order of variable alignment on the stack, to be able to update local variables and formal parameters. It also requires modifying the code of active functions to use the appropriate offsets when accessing updated local variables. If the datatypes of variables or the return value are modified, the stack may need to be enlarged or reduced. Additionally, updating just one function on the stack may require updating the code of callees of that function to adjust its offsets, for example if variables are passed to the callees by reference. This instrumentation can be inconclusive if it is applied on self-modifying code, functions that contain code-in-data or data-in-code, or if functions have been compiled without frame pointers. Finally, mapping optimized binary code to source code to reason about execution continuation can be challenging.

DynAMOS [33] applies safe multi-threaded DSU in commodity operating system kernels through a binary instrumentation approach of *adaptive function cloning*. It does not update active functions but it allows multiple versions of functions to run simultaneously and invokes user-supplied adaptation handlers to control the updates. However, DynAMOS may need to wait indefinitely for a safe update point where a newer version of a function can be activated. It supports updates of data structures active on the stack through *shadow data structures* but this approach requires data access indirection and cannot be used when updates do not preserve the semantics of the data.

KSplice [37] examines kernel image files at the binary level to automatically generate DSU patches (mainly security fixes) and it applies the patches dynamically. It uses shadow data structures to update data, does not update active functions, and does not apply safe multi-threaded updates.

3.3 Dynamic Update

Quite a few DSU systems have been designed to update applications. They rely primarily on function-pointer indirection to update code, and require the application to reach a quiescent point before they can safely update active code or data. The DSU systems that have been more successful also use data-access indirection to apply updates.

Gupta [23] formalized a software updating model and proved that determining if a change is valid is generally undecidable. This model requires that functions are not present on the stack when they are updated. To ensure valid updates, he proposes two approaches: (a) restricting the types of changes permitted, such as only to functional enhancements (which limits the complexity of updates that can be

applied), and (b) reaching the expected updated state some time after the update (which can disrupt the service provided by an application either indefinitely or until that state is reached). Most DSU systems follow these two approaches and share the limitations that accompany these solutions. A practical implementation of Guptas' system [79] followed the interrupt-update-restart mechanism. It applied software updates by creating a new process and transferring the state of the old program to the new. This approach interrupts the service provided by the running application. For example, open network connections are lost.

DYMOS [24] is the first DSU system ever created. It can add, remove, and update interfaces of modules at runtime. It groups variables in program modules and updates entire modules when the modules are quiescent by modifying the process symbol table (which uses function-pointer indirection). It offers the programmer the capability to describe the valid states in which an update should be applied. For example, module P should be updated only when Q and R are idle. However, in the presence of multiple threads or blocking code updates may need to wait indefinitely before they can be applied.

Online Patches and Updates for Security (OPUS [26]) is a DSU system targeting security patches. It modifies the gcc compiler to produce updates of C programs at a function level and injects the updates (using the ptrace API) after ensuring none of the functions are active on any thread's stack. The update mechanism used is function-pointer indirection (activated with binary instrumentation). Static analysis is used to detect possibly unsafe changes and alert the programmer, but the analysis is highly conservative. It does not permit updates to long-running and top-level functions, function signatures and inline functions. It also does not allow updates to global data structures, updates to another function's stack frame, files or sockets, and does not allow updates to alter the outcome of a function's return value. As a result the updates applied are of low complexity.

The Procedure Oriented Dynamic Update System (PODUS [27]) applies updates to programs by dynamically modifying function call-sites (using binary instrumentation) to redirect execution to newer versions and can only update single-threaded programs. Detours [57] and Vulcan [58] also apply indirection for function-level updates. The shortcoming of all these systems is that they rely on quiescence to safely effect updates and do not support updates of active data.

The Powerful Live Updating System (POLUS [28]) applies DSU using function indirection (activated with binary instrumentation). It allows both old and newer versions of code and data to coexist. During an update, data accesses to either version of global data are trapped using the single-step debug flag of the processor and synchronized for data consistency using a state mapping function automatically provided by a patch generator. This is accomplished by write-protecting both the old

and the new versions of data and associating a signal handler to catch write attempts to either version. POLUS cannot update active function code and stack-resident data and does not account for safety in the presence of multi-threading.

DLpop [59, 80] is a DSU system that permits multiple versions of a datatype to coexist in a program. To update a datatype it creates a copy of the data of the old datatype, opening the possibility of old code operating on stale data. It does not automatically update function pointers, active data or long-running loops. The update mechanism used is dynamic-linking (essentially function-pointer indirection).

Ginseng [32] has been successful in applying DSU with a combination of function-pointer indirection, data-access indirection, and logical-stage extraction. It also automatically produces state mappings with a patch generator. Although it does not support updates of functions active on the stack, it offers update support for infinite or long-lived loops (similar to ERLANG [29]). Users need to anticipate such loops (and the loop post-ample) and manually annotate them for *loop extraction* of their body into a separate function that can be updated before the next loop iteration begins. To safely update multi-threaded applications [34, 36] users may also need to manually break some loops in multiple logical stages (like DynAMOS [33]). To support data updates Ginseng uses padding and data-access indirection and ensures their safety through static analysis that computes universally type-safe [38] update points. It also uses static analysis to offer transaction-safety [39]. Compared to other DSU systems [24, 23, 59, 26, 27, 28, 33, 37] these analyses improve safety and updateability. However, they are conservative approximations that still limit coverage and cannot guarantee immediate continuation.

Linux hot-swapping [81] uses function-pointer indirection to safely swap kernel modules when the modules are quiescent. But it is not capable of dynamically updating core kernel services, like the scheduler and memory manager, or datatypes.

3.4 Replication

Replication of hardware and software resources is the main approach for dealing with hardware failures. It is also used to support runtime updates. To that end, replication uses the interrupt-update-restart mechanism to gradually move the service provided by an application to the replicas providing the new version of the application. As discussed in Chapter 2.3, the ability to update immediately all execution threads is necessary to avoid service interruption. Replication can manage to avoid service interruption because it is often implemented only for program code when the persistent state of the application does not change.

Replication is usually implemented in one of two ways: one confines replication to program code; the other allows the replication of both the program code and program state. These two approaches are discussed next.

When only the program code is replicated, all replicas access a shared state that is persistent on disk (e.g. a file server, or a single database instance). This functions on the assumption that the application does not maintain in-memory state information (e.g. the NFS protocol, which is state-less) or the in-memory state can be discarded with no effects on application consistency (e.g. shutdown a web-server that communicates with a database). Replication is implemented by redirecting new service requests to the new replicas using a layer of indirection when requests are received or by changing a DNS entry to point to a different IP address. This indirection essentially ensures that the update is applied at a quiescent point. The old replica continues to service existing requests, shares the persistent state, and may need to wait indefinitely before the replica can be deactivated. If the update changes the semantics of the persistent state (e.g. the persistent state expected by the new replicas is not backward compatible) then replication cannot help because it violates logical representation consistency.

When both the program code and program state are replicated, the replicas need to communicate to maintain consistency of the program state (e.g. a distributed database system). New replicas are enabled by starting from a clean persistent state and requesting from existing replicas a copy of their persistent state. If the semantics of the application have not changed, the new replica builds its own copy of the persistent state as communicated from the existing replicas and then participates in providing service. However, if the semantics of the state are updated (e.g. a database schema evolves and is not backward compatible), the state communicated to the new replica is no longer understood by the replica and the update violates logical representation consistency.

Postgres-R [82] implements eager multi-master replication on top of the PostgreSQL database management system. It also supports crash recovery and partial replication by extracting a database schema and its data, installing it to a new replica, and executing pending consistency messages before new clients are allowed to connect to the new replica. However this replication model only addresses recovery. It cannot apply updates that change the semantics of the database schema.

UpStart [48, 83] is a DSU system for distributed systems modelled as objects communicating with Remote Procedure Calls (RPC). Updates are applied only at the granularity of top-level objects (only entire nodes are updated). Older and newer versions of objects are allowed to coexist and run in mixed-mode using a thin layer of simulation objects. Simulation objects invoke the appropriate version of an object to match the version of an incoming RPC call if cross-version interoperation is possible. If updates are incompatible they cannot be applied immediately; they need appropriate update scheduling from the user. Additionally, updates under this model do not preserve volatile state.

3.5 Virtualization

Virtualization separates an operating system or application from the underlying system resources. DSU has been implemented using virtualization following both the interrupt-update-restart mechanism, and a combination of the function-pointer indirection and data-access indirection mechanisms.

The Microvisor [49] virtual machine supports online updates of operating systems. It runs applications on one virtual machine while it starts and reconfigures a second virtual machine with an upgraded OS and applications. However, updates do not preserve volatile state and interrupt the provided service (open network connections are lost).

LUCOS [61] uses virtualization to update live operating systems without requiring quiescence following an approach similar to POLUS [28]. It uses two features of virtualization to apply updates. First, it relies on the ability of the Virtual Machine (VM) to write protect the memory areas that hold old and new global variables. Access to these variables raises an exception that returns control to the VM. Second, the VM offers the ability to gain control of execution after a write operation to the data finishes. This is implemented by having the VM unprotect access to the global variables and immediately enabling the single-step debug flag of the processor. When the write operation completes a debug exception is raised that returns control back to the VM. Using these virtualization features LUCOS allows coexistence of both the old and new versions of variables and consistency between them is synchronized using state transfer functions. It cannot update active functions and stack-resident data and does not support updates to multi-threaded kernel subsystems.

JVOLVE [84] extends the Jikes RVM [85] with DSU capabilities. It uses three features of virtualization to apply updates. First, it uses classloading to add, delete, or change existing classes, including adding, removing, or replacing fields and methods. This is similar to binary instrumentation. Second, it uses a subset of VM safe points as DSU update points, because the VM guarantees that at these points it is safe to perform garbage collection and apply updates on the heap. Third, it uses on-stack replacement [86] to recompile updated methods that refer to classes that are updated. However, these approaches have limitations. Classes that need to be updated (instead of classes that refer to classes that need to be updated) cannot be updated if they are active on the stack. There is also no guarantee that JVOLVE will reach a DSU safe update point. Other JVM-based DSU systems are also unable to update active code [87, 88, 89] or require data-access indirection.

3.6 Checkpointing

Checkpointing saves the current state of an application to disk and allows resuming an application from that state at a later time. It has been implemented in single-threaded applications [90], distributed systems [91], high performance computing clusters [92, 93], and for heterogeneous systems [94, 95, 96]. It supports saving the state of pointers [97] and forces all threads to stop in order to safely checkpoint the state of a multi-threaded application [96]. The major limitation of checkpointing is that it interrupts the service provided by the application, but some of the existing working on checkpointing has interesting aspects that could be extended to support DSU. In particular, some of the techniques used in this dissertation originated in work on checkpointing.

Fünfrocken [98] applies source-to-source transformations to instrument Java applications for checkpointing and migration. Source-to-source transformation is chosen, instead of modifying the Java virtual machine, because the Java virtual machine forbids stack manipulation by bytecode for security reasons. The transformation inserts multiple if-statements that allow bypassing code when state is rebuilt. Function signatures are extended to supply as a parameter to every function call the state that will be restored. Checkpoints are manually inserted (based on a programmer's estimate) only in methods that might encounter these points when the application runs. A checkpoint is acquired when all threads stop at the checkpoints, but the possibility remains open that one of the threads will block or never reach a checkpoint, delaying the acquisition of a checkpoint and initiation of migration indefinitely. This approach leads to modest overhead 4%-19% in synthetic applications but it has not been applied in real-world applications. It also enlarges the final bytecode by 350% due to the implementation of state saving.

c2ftc [94] applies portable checkpointing for heterogeneous architectures. During normal execution mode, the instrumentation continuously saves a continuation point (discussed in more detail in Chapter 3.7) that can be used to rebuild the stack when restoring the checkpoint. When a checkpoint is requested the local stack state is saved using a macro. The stack is recursively unrolled by such macros (one per function) which save each stack frame. When restoring, a top-level switch-statement restores each stack frame and the continuations. Since this approach uses macros for stack saving, and multiple checkpoints could be placed in a function, the state-saving instrumentation through macros can enlarge the application size considerably (like [98]).

Karablieh and Bazzi [97] propose an approach similar to c2ftc [94] that is truly heterogeneous and extends checkpointing to multi-threaded applications. It modifies function signatures to pass the checkpointing mode variable as an additional parame-

ter but this introduces overhead and is problematic with library functions (like [98]). Both this work and c2ftc have not been applied in real-world applications.

Shanhbhag [62] checkpoints an application (based on [97]) and transforms the checkpoint file to be readable by the new version of the application. The new version is restarted using the transformed checkpoint file. An equivalence algorithm detects and maps persistent program states, but the algorithm is not control-flow sensitive and makes conservative assumptions that limit the complexity of updates that can be applied. For example updated functions are only allowed to access global variables but not modify them (similar to OPUS [26]) and datatype updates are not supported.

AutoPod [99, 50] facilitates updating operating systems by combining checkpointing with a virtualized environment of isolated process domains [100] implemented with system call interception and chroot(). It suspends, checkpoints, and migrates processes across different versions of the operating system. However it does not preserve the application state maintained by the operating system, such as open network connections, and does not update the applications.

3.7 Continuation-Style Programming

Continuation-style programming [101] is a programming style in which no function is ever allowed to return and functions always accept the current state as an explicit parameter: the continuation. This *execution continuation* is an important part of the state of the application. The ability to update the continuation can be useful in applying DSU because it can force the new version of an application to resume from a different execution point than the execution point from which the old version of the application was paused. Resuming from a different execution point is necessary because it corresponds to updating the Program Counter and the return address ra of a stack frame. This programming style has been used to apply DSU through a general source-to-source transformation. The transformation converts applications that have not been developed in continuation-style to become continuation-style applications.

Execution continuation was offered first in LISP using the call/cc (call-with-current-continuation) control operator [102]. It has been been applied in ML [103] and Mach 3.0 [104], and, to some extent, in Java [98, 105, 86] and K42. It has also been applied for task management [106, 107] and for device drivers [108].

An early design of *stack reconstruction* based on continuation-style programming has been applied for efficient large-scale process-oriented parallel simulations [63]. Under this design, stack state for local variables is allocated on the heap, accessed through data indirection and remains alive until global simulation time. Continuation points are continuously saved during execution. This work could have been extended for DSU by incorporating stack unrolling, preserving local variables on the stack, and addressing the problem of forcing all threads to stop. Still, it improves over

previous simulation work with continuations [109] or a thread-based approach [110] that used the POSIX setjmp()/longjmp() calls for context-switching. Note that while the setjmp()/longjmp() calls can be used to save and restore stack frames they cannot update the functions that are restored or map to a new state.

Sekiguchi et al. [105] proposed an approach for Java and C++ that applies source-to-source transformations to emulate continuations and apply checkpointing. This approach unrolls stack frames using the exception handling feature of the programming language. Thus it cannot be applied generally to other languages that do not offer exception handling, like C. However it could have been extended for DSU had it addressed the need to safely stop multi-threaded applications. The approach creates a method and a class for every instrumented method. The created class represents the possible execution states of the method. Function signatures are extended (similar to Fünfrocken's approach [98]) to supply as a parameter to every function call the state that will be restored. This work identifies that checkpoints should not be captured if library functions are on the stack, which is common in graphical user-interface applications that use callback functions, but does not safeguard from this condition. In comparison to Fünfrocken [98], it employs a top-level switch-statement instead of multiple if-statements when state is restored, which reduces the size of instrumented code. It further reduces the size of instrumented code by restoring local variables just once per method, instead of per continuation.

Légaré [111] also describes a source-to-source transformation of Java programs to use continuations. In comparison to other Java approaches [98, 105], it does not need to modify function signatures or introduce extra classes to restore state. Although it can unroll the logical stack saved for the current stack frame, it is unable to unroll the execution stack of the virtual machine. Hence it is unable to support continuations of functions on all stack frames.

Only recently has continuation-style programming been explored for DSU. Shanhbhag's [62] checkpointing-based updating system (see Chapter 3.6) relies on continuations for restoring application state. It could have applied DSU without service interruption if stack unrolling had been developed. A recent DSU software architecture for Haskell [112] based on continuations proposes separating between code and state and supplying the state as a parameter when updating (similar to [98, 105]). But it is unclear how existing applications could be easily converted to follow this architecture. More recent work also outlines how continuations have the potential to apply DSU [113, 114].

3.8 Conclusion

Existing work has explored DSU through extensible design of operating systems, binary instrumentation, dynamic updating, replication, virtualization, checkpointing,

and continuation-style programming. Most of these approaches have two main limitations. They are either unable to update active functions and data or do not apply updates immediately. Through its application for checkpointing, continuation-style programming has demonstrated that it can be used as the underlying mechanism for applying DSU, although it has not been tested on real-world applications.

A lot of systems update only function code with binary instrumentation [51, 52, 53, 57, 58], but updates are limited, as these systems aim to provide performance profiling and debugging capabilities. Another popular approach is to use table indirection [115, 116]. Most systems use one of these approaches to allow old and new versions of code and data to coexist [117, 116, 80, 32, 28, 33, 37] but may require indirection to access the data [116, 32, 33, 37]. To update data safely, some systems rely on object encapsulation [24, 31, 35]. Other systems may not be able to update both the data and active code without quiescence [24, 23, 80, 35, 61, 28] unless they are supplemented with conservative analyses that provide safety guarantees [26, 32, 38, 39]. Few systems can update long-lived loops [29, 32, 33, 36].

The only DSU system that can apply an update without waiting indefinitely for a safe update point is the K42 operating system [35]. However, K42 is specially crafted to be updateable and its approach cannot be generally applied to existing operating systems and applications without significant re-engineering. Recent work proposes a specially crafted architecture for applications [112].

Continuation-style programming has been adopted for checkpointing and can be used for DSU. Checkpointing systems extend function signatures to pass the continuation state [98, 105, 97, 62] and continuously track the continuation point [94, 62, 97]. Some systems rely on constant data-access indirection [63, 97], on the exception handling mechanism of the programming language [105] to unroll the stack, or add excessive code to save stack state[94, 98].

Chapter 4
DYNAMIC SOFTWARE UPDATE SYSTEM

This dissertation provides a DSU system that implements the *whole-program update* mechanism described in Chapter 2.6.4. Recall that the whole-program update mechanism allows the mapping of the whole state of the old version of the application to a state of the new version of the application.

The DSU system provided, UpStare, incorporates a number of elements to provide immediate dynamic updates for multi-threaded (including multi-process) applications. To *map the state* of one thread, it provides a stack reconstruction mechanism that allows the unrolling of the execution stack of the old version and then reconstituting a stack of the new version. Stack reconstruction applies a default mapping that transforms global variables, stack frames, and execution continuation points of the old version to global variables, stack frames, and execution continuation points of the new version. These default mappings can be overridden by the user through an interface provided by the system. To support the update of multithreaded applications, UpStare forces all executing threads to block and then applies stack reconstruction to every thread. To provide *bounded delay*, UpStare transforms blocking system calls into non-blocking calls. To provide additional *safety guarantees*, UpStare enforces user-defined safety constraints to update safely from a particular application state.

The rest of this chapter describes the architecture of UpStare, explains how stack reconstruction works, how the default state mappings are calculated, how the user can override these default mappings through a well-defined interface, and how updates of multi-threaded applications including those with blocking system calls are supported. Chapter 5 describes how the system can enforce user-defined safety constraints.

4.1 System Architecture

UpStare is an architecture (and operating system) independent tool suite comprising a *compiler* to generate updateable programs, a *runtime environment* for dynamically applying updates, a *patch generator*, and a *dynamic updating tool*. It compiles multi-threaded applications to be updateable, automates generation of dynamic updates with some help from the user, and applies updates safely. UpStare is designed as a DSU solution for the widely used C programming language. C is a relatively low-level

language and it is expected that a solution for a low-level language can be applied in principle to higher-level object-oriented languages like C++.

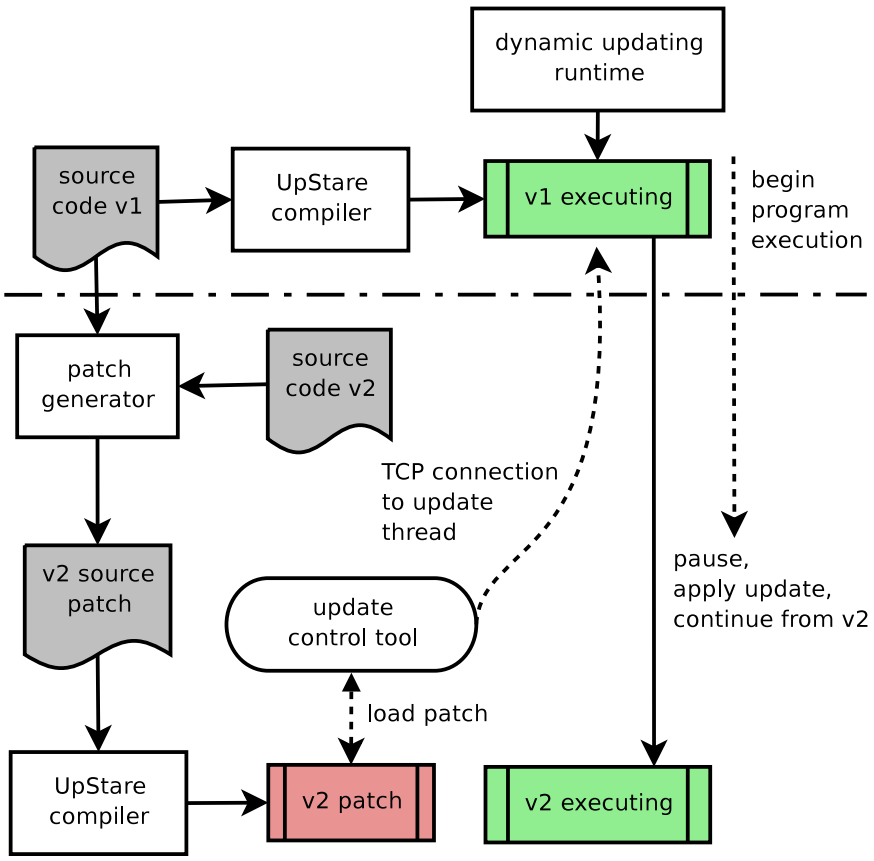

Fig. 1. UpStare System Architecture.

Figure 1 shows the high-level system architecture, which is similar to those of other DSU systems. Users compile the source code of the original version of an application with the UpStare compiler and start the application. When a newer version of the application becomes available, users supply the source code of both the original version and the new version to the patch generator to prepare a dynamic software update patch in source code format. Users compile the source code update patch with the compiler to produce an executable update patch and place the patch locally on the machine executing the application that will be updated. The update is initiated with the dynamic updating control tool which issues update requests through a TCP connection to the executing application. The update requests are serviced by the dynamic updating runtime environment, which is automatically added in the original version of the application by the compiler.

Applying updates involves (1) pausing the application, (2) mapping the application state, and (3) resuming execution. These three actions are coordinated by the runtime environment. First, the runtime environment forces the application to block all threads (and all processes) with the help of *update points* (described next), after it verifies that the update will not violate user-defined safety constraints. Ensuring all threads are blocked is necessary to guarantee that the update will be atomic for the application. Second, the runtime environment puts the application in stack reconstruction mode which unrolls stack frames one by one and saves their contents until the main() function is reached. At this point the entire old program code Π and old program state s are available for modification by calling the state transformer function S. Global variables are mapped to their new version and the stack is then reconstituted by replacing old versions of functions, local variables, and formal parameters with their new versions. Finally, control returns to the runtime environment that puts the application back in normal execution mode coordinates resuming all threads (and all processes).

4.1.1 Compiler

The compiler applies high-level, source-to-source transformations that convert applications written in C to be dynamically updateable. It is written in OCaml using the CIL framework[118] 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 the GNU C Compiler (gcc), with calls to the UpStare compiler (hcucc.pl). No source code modifications by a user are required in existing applications to make them updateable. The compiler transforms applications to implement the whole-program update mechanism using stack reconstruction. It also inserts *update points* in the applications, which are the points where the application can be paused for the update to be applied. The update points are a subset of possible Program Counter locations for the application.

4.1.2 Runtime Environment

The runtime environment is written in C, has a small (64KB) memory footprint, and is statically linked into the updateable application. Requests for an update are transmitted to the runtime on a TCP connection and result in the runtime loading the dynamic update patch using dlopen(). The runtime then uses a collection of routines and data structures to coordinate a safe update by forcing all threads to block and enforcing user constraints. To determine if all threads have been blocked, the runtime tracks the locking activity of threads in its data structures since the application first begun executing.

4.1.3 Patch Generator

It is important to provide a mapping to the new state with minimal user involvement. A DSU system that minimizes user input is more likely to be adopted. UpStare provides a patch generator that partially automates this state mapping. The default mapping (see Chapter 4.3) produced by the patch generator is effective in practice, and can be further fine-tuned or bypassed by the user.

Given the source code of the old and updated programs, the patch generator automatically produces the source code for a dynamic update patch. The patch includes the newer versions of functions, new global variables, and the old and updated datatype definitions of modified variables, either global or declared on the stack. The patch generator also automatically produces data transformers to map global variables and stack transformers to map stack-resident variables. The patch generator does not generate execution continuation transformers, but the runtime environment attempts a simple mapping that preserves continuations.

4.2 Stack Reconstruction

The UpStare compiler automatically instruments programs with code that will be able to unroll and reconstruct the stack. The instrumentation is applied through a source-to-source transformation and requires no user input. This instrumentation produces a program that is semantically equivalent to the uninstrumented version of the program. The instrumented code that adds the ability to unroll and reconstruct the stack is not executed under normal execution mode. This code is guarded by global flags that activate the code only if the program is in stack reconstruction mode. A program is placed in stack reconstruction mode when all threads are blocked, after an update is requested by a user. Stack reconstruction consists of two major steps which are described next:

1. Saving the existing stack state when unrolling;

2. Restoring the mapped state when reconstructing.

Saving and Unrolling. Before each stack frame is unrolled, the stack state is saved automatically by the instrumentation code. For each function, the instrumentation produces one wrapper function that efficiently saves the stack state of the function. The wrapper function is called to save the stack state and then a return instruction is issued to unroll the stack frame to its caller. This continues recursively until the entire stack is saved and unrolled for every thread. At that point the entire state of the application is saved and is available to be mapped. A state transfer function is invoked by the runtime to automatically map the state of the old version

of the application to the state of the new version. This state transfer function is automatically generated by the patch generator but can be fine-tuned or overridden by the user. The state transfer function can be implemented to map the entire state of the application, but UpStare chooses to map only the global variables at this point to simplify the implementation. The default state mapping is described in more detail in Chapter 4.3.

Stack reconstruction needs no input from the user to define how high should stacks be unrolled, or which threads should be reconstructed: by default all threads are unrolled to the top, and they are all updated. However, stack reconstruction is flexible enough to apply updates defined at a fine grain. The user can define which threads should be updated, and how far up should the stack of a thread be unrolled. For example, a user may decide to reconstruct the stack of only one thread to apply a minor security fix to only one function instead of update the whole program.

```
1
2
3
4
5
6
7
8  functionA()
9  {
10     char a;
11     int param;
12     ...
13
14     functionB(param);
15
16
17
18
19
20
21
22 }
```

(a) Non-Instrumented

```
1  typedef struct {
2    char a;
3    int param;
4  } stack_functionA_v1_t;
5
6  (*functionB_ptr) (int) = &functionB_transformed;
7
8  functionA_transformed()
9  {
10   stack_functionA_v1_t locals;
11
12   ...
13   functionB_6_before:
14   functionB_ptr(locals.param);
15   if (may_reconstruct && must_reconstruct()) {
16     if (must_unroll_up(``functionA'')) {
17       save_frame__functionA(&locals, 6);
18       return;
19     }
20     goto functionB_6_before;
21   }
22 }
```

(b) Instrumented

Fig. 2. Transformation of Function Calls for Stack Reconstruction.

Figure 2 shows an example of how functionA() is transformed to check upon returning from the callee functionB() whether the stack should be reconstructed. Note that may_reconstruct (line 15) is a global flag raised only in reconstruction mode to improve performance. If must_reconstruct() is true (line 15; this thread should participate in reconstruction) and execution should be unrolled (line 16; 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 (line 17) and functionA() returns to its caller (line 18). 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 (line 16; must_unroll_up() is false). A goto statement (line 20) resumes execution from functionB_6_before (line 13) and descends in functionB() for reconstruction (line 14).

Mapping and Reconstructing. After the global variables are mapped, the stack of each thread begins being reconstructed. Reconstruction restores the local variables and formal parameters of a function, and also its execution continuation point. Reconstruction calls the new version of the main() function of the application, passing the updated formal parameters, to initialize the function on the stack. As soon as this function is entered, reconstruction automatically restores the stack state of the function by invoking a wrapper function, automatically produced by the instrumentation, that efficiently restores all local variables in the function. This wrapper function invokes a stack transformer (which can be fine-tuned or overridden by the user) that maps the state of the old version of the stack frame to the state of the new version of the stack frame. The execution continuation is also restored using a switch-statement that forces execution to descend (using a goto jump, and then calling the callee) into a callee function as part of the reconstruction process. This is repeated recursively until the entire stack is restored. At this point the update is complete and the program is ready to resume execution.

Figure 3 shows an example of how execution is resumed from functionA(). If on function entry the stack should be reconstructed downwards (line 6), the stack frame is restored (line 7). A switch-statement (line 8) maps the continuation point 6 (line 13) to continuation label functionB_6_before (line 19) using a goto statement (line 14). Execution flow continues by calling functionB() (line 20). When the update is complete (line 21; may_reconstruct is false: execution is no longer in reconstruction mode) and functionB() finishes, execution continues normally (from line 28).

Supporting thread entry-points. Stack unrolling recursively issues a return instruction until all stack frames are unrolled. But issuing a return instruction in the main() function or the start routine passed to a pthread_create() would result in the application (or thread) to terminate permanently. Still, it is desirable to be able to update these functions as well, which means it is necessary to issue a return instruction in these functions.

To allow the update of main() or thread entry-points, calls to such functions are initiated from a wrapper function. Stack unrolling recursively issues a return instruction until the wrapper function is reached but at that point unrolling stops to avoid application (or thread) termination. When reconstruction begins, it first descends into main() or a thread entry-point and allows these functions to be updated. To accurately discover thread entry-points UpStare uses the points-to alias analysis provided by CIL.

```
1  functionA()
2  {
3      char a;
4      int param;
5
6
7
8
9
10
11
12
13
14
15
16
17
18     ...
19
20     functionB(param);
21
22
23
24
25
26
27
28
29 }
```

(a) Non-Instrumented

```
1  functionA_transformed()
2  {
3      stack_functionA_v1_t locals;
4      int continuation;
5
6      if (may_reconstruct && must_reconstruct()) {
7          restore_frame__functionA(&locals,&continuation);
8          switch (continuation) {
9              ...
10             case 3:
11               goto try_to_update_3_after;
12             ...
13             case 6:
14               goto functionB_6_before;
15             ...
16         }
17     }
18     ...
19     functionB_6_before:
20     functionB_ptr(locals.param);
21     if (may_reconstruct && must_reconstruct()) {
22         if (must_unroll_up(``functionA'')) {
23             save_frame__functionA(&locals, 6);
24             return;
25         }
26         goto functionB_6_before;
27     }
28
29 }
```

(b) Instrumented

Fig. 3. Transformation of Function Entrypoints for Stack Reconstruction.

Supporting signal handlers and library functions. Signal handlers and library functions are incompatible with stack reconstruction and need special support. Issuing a return instruction in a signal-handler returns execution control to the operating system and prevents unrolling from completing. Similarly, unrolling does not complete if a return instruction is issued in a function that has been invoked by a library function because it returns execution control to the library.

The memory address of signal handlers, defined with sigaction() and signal(), is stored inside the operating system. This information needs to be updated to refer to new signal handlers during an update. To handle this part of the update in an operating system independent way, UpStare avoids resetting the signal handlers and instead initiates calls to the signal handlers using function pointer indirection from a wrapper function. This allows UpStare to update the function pointer in the application when a signal handler needs to be updated. Signal handlers are also 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. 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.

A similar approach is followed for library functions that accept user functions as parameters. Special support is needed for these functions because it is assumed library functions have not been compiled with instrumentation for stack-reconstruction. For example, vsFTPd uses the qsort() library function which accepts a user-supplied sorting function as a parameter. Such user-supplied functions return execution to the library and are incompatible with stack reconstruction. They are handled similar to signal handlers. The difference between library functions and signal handlers is that functions supplied as parameters to library functions do not need to be called from a wrapper function. The wrapper function is not needed because UpStare ensures there is no possibility for the user-supplied function to be updated while a library function is active on the stack.

Redirecting function calls. Unlike other DSU systems, continuously redirecting function calls is not necessary for stack reconstruction. The redirection of function calls is not needed if stack reconstruction executes at a default mode of always unrolling to the thread entry-point wrapper function. It is only the thread entry-point wrapper function that needs this redirection to call the new version of the thread entry-point function. All other function calls can be issued directly. Still, continuous redirection of function calls is implemented to make stack reconstruction a more flexible mechanism that can be used to implement other updating models.

In the current UpStare prototype function-pointer indirection is used to execute function calls. For each function f_v1(), a global pointer variable f_ptr is created that points to &f_v1 and calls to f_v1() are transformed to calls to *f_ptr(). This indirection allows changing the version of the function by modifying directly the value of a pointer in memory. Continuous indirection allows updating a function without forcing all threads to block in a multi-threaded program.

Besides function calls, it is also necessary to wrap invocations of function pointers, because function pointers could be passed as parameters to functions and it is desirable to be able to update to a new version the function they point to. For each function pointer *g_v1(), a wrapper function wrap_g_v1() is created that calls it. This is an effective way of allowing updates to function pointers without points-to alias analysis.

Inserting update points. The runtime environment forces an application to pause with the help of update points that are inserted by the compiler. Update points are automatically inserted at the beginning of each function and each loop. These points are selected as update points because these points are encountered often to allow immediate updates. As will be explained in Chapter 4.5, all locks are also selected as update points to force all threads of an application to pause.

```
1  functionA()
2  {
3      char a;
4      int param;
5
6      while(condition)
7      {
8
9
10
11
12
13
14         ...
15     }
16 }
```

(a) Non-Instrumented

```
1  functionA_transformed()
2  {
3      stack_functionA_v1_t locals;
4
5      ...
6      while(condition)
7      {
8          if (must_update) {
9              coordinate_update_top(&locals, 3);
10             return;
11         try_to_update_3_after:
12             coordinate_update_bottom();
13         }
14         ...
15     }
16 }
```

(b) Instrumented

Fig. 4. Insertion of an Update Point at the Beginning of a Loop.

Figure 4 shows an example update point inserted at the beginning of a loop. When the must_update flag is raised (line 8; an update is requested), the current thread participates in synchronization to block all threads. The current continuation point 3 and the stack frame of functionA() are saved (line 9), and execution returns to the function's caller (line 10). When the stack is reconstructed and functionA() is called again (see lines 10-11 in Figure 3b), execution flow resumes from try_to_update_3_after (line 11).

The current implementation is restricted to a coarse-grain 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. Such a restriction could also be enforced by defining a transaction-safety constraint.

Exporting local variables. The dynamic software update patch is loaded in the application using a programming interface to the dynamic linking loader. The dlopen() library call loads the patch and state transformers included in the patch are accessed using dlsym(). This programming interface is successful 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 and lead to system exceptions when executing state transformers. To address this limitation, the UpStare compiler removes the static keyword from all local variables and exports them to global.

4.3 Default State Mapping

UpStare provides a default state mapping which hopefully matches closely what the user desires, but there are no guarantees for that. The mapping relies on the user for verification of its validity. For the cases that were tested, the mapping has proven to be an effective heuristic that requires minimal user involvement. The mapping is automatically generated and applied at a high-level (source-code format) with the patch generator. Automatically generating the mapping considerably reduces the effort required by the user in producing the mapping and makes it easier to adopt the DSU system. For the cases where automatically producing parts of a mapping is not possible, the patch generator warns the user and the user can supplement the mapping with a custom mapping (as will be described in Chapter 4.4).

The design of stack reconstruction makes it possible to map the program state using a single state transfer function. However, our implementation of mapping the program state uses multiple independent state transfer functions. These are:

1. *Datatype transformers* that map global variables on the heap h,
2. *Stack transformers* that map local variables l and formal parameters p in stack frames T_{sf} (but not the return address ra), and
3. *Execution continuation transformers* that map the Program Counters T_{PC} of each thread and the return address ra of each stack frame.

These state transfer functions are applied together in a single, atomic step. Note that using multiple state transfer functions is not a limitation of the stack reconstruction design. The stack reconstruction design allows applying a single state transfer function after the state of the old version is saved. But the current implementation of multiple state transfer functions is effective in practice and the system allows the user to bypass it if more flexibility is desired.

The remaining of this section describes the default state mapping of UpStare in more detail. It describes the default datatype mapping, default stack mapping, and default execution continuation mapping. Chapter 4.4 describes the interface that is available to the user to override these mappings.

4.3.1 Default Datatype Mapping

For each changed datatype the patch generator produces a datatype transformer. The transformer may be invoked multiple times by other datatype transformers or by stack transformers as needed. The datatype transformer is used to update global variables on the heap h, local variables l, or both, depending on where the datatype is used.

For every global variable whose datatype τ has changed, a new global variable of the new datatype τ_{new} is allocated in h_{new}. If the datatype is a compound datatype (such as a struct or union in C) and it has been extended, the datatype transformer copies the old fields (only new fields must be initialized by the user). If the datatype is reduced, the datatype transformer copies the remaining fields with no user assistance. If the variable is an array, a datatype transformer is applied on all array elements. If the datatype change simply extends an array with more elements (such as the parseconf_uint_array in vsFTPd which is updated to offer more configuration options), a new array with more room is allocated and the values of all old elements are copied.

```
1 void upstare_transformer__vsf_session_v120to121(struct vsf_session_v120 *old,
2                                                 struct vsf_session_v121 *new)
3 {
4   upstare_transformer__mystr(&old->user_str, &new->user_str);
5   upstare_transformer__mystr(&old->anon_pass_str, &new->anon_pass_str);
6   upstare_transformer__mystr(&old->rnfr_filename_str, &new->rnfr_filename_str);
7   ...
8
9   new->p_local_addr = old->p_local_addr;
10  new->p_remote_addr = old->p_remote_addr;
11  new->pasv_listen_fd = old->pasv_listen_fd;
12  new->p_port_sockaddr = old->p_port_sockaddr;
13  ...
14
15  // Must initialize these new fields
16  new->email_passwords_str.PRIVATE_HANDS_OFF_p_buf = 0;
17  new->email_passwords_str.PRIVATE_HANDS_OFF_len = 0;
18  new->email_passwords_str.PRIVATE_HANDS_OFF_alloc_bytes = 0;
19 }
```

Fig. 5. Transformer for Datatype struct vsf_session (vsFTPd v1.2.0 to v1.2.1).

Figure 5 shows an example of a default datatype transformer produced automatically by the patch generator. It transforms datatype vsf_session, which is a large struct, in vsFTPd from v1.2.0 to v1.2.1. This struct contains multiple fields of datatype mystr, thus the vsf_session datatype transformer invokes the transformer for the mystr datatype once for each field of the struct (lines 4-7). For fields of primitive datatypes, the fields are copied directly (lines 9-13). Currently, the datatype transformer does not automatically initialize new fields and leaves that task to the user (lines 16-18). However, this could be supported with more engineering effort in the patch generator since the initialization values are known in the new version of the source code.

In some cases a datatype change suggests that the semantics of the datatype have changed. For example, a datatype may change from a char to a long, or from an int to an int*. In such cases the patch generator produces an empty datatype transformer and warns the user to manually implement the transformer.

After datatypes are updated to their new versions the old versions are never accessed again. Memory allocated for global variables of the old datatypes remains unused. Memory allocated for local variables of the old datatypes is reclaimed when the stack is unrolled and new memory for local variables is allocated when the stack is reconstituted.

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. An approach for accurate tracking of pointer types has been developed in previous work [97]. This approach involves tracking the datatype of pointers when they are cast, continuously tracking all memory pointer assignments, and using indirection through a *memory refractor* to access the pointer values. After an optimization [119] this approach reports overhead of 3.4-5.7% when tested on a program without pointer operations. When tested on the same program (and with the optimization), but with pointer operations, this approach reports overhead of 38-45.5%.

This approach of mapping pointer variables is not yet integrated with UpStare. The performance of tracking pointer types at runtime can be significantly improved if the existing approach can be augmented to track at runtime only the datatype of a pointer. It should not continuously track all memory pointer assignments or use indirection to access the pointer values because that incurs considerable overhead. Instead an alternative approach is proposed. When it's time to map pointer values, two elements can be taken into account for each pointer: the memory address of the pointer, and the size of the tracked datatype. Combined, this information provides a set of ranges of allocated memory areas used for pointers. The offsets of the pointers within these memory areas can be used to determine which datatypes (e.g. fields of structs in C) they were pointing to and how to map them to the new versions.

However this approach may be unable to detect mappings for some pointer variables that were manipulated in unexpected ways by the program, such as dividing a pointer value by two. For such cases, the user can guide the pointer mapping or provide transaction safety constraints that eliminate the occurrence of updates when such pointer values may be alive. It is also possible to identify such cases through static analysis, and automatically produce transaction safety constraints for them.

4.3.2 Default Stack Mapping

The patch generator automatically produces a stack transformer for each function in the application. The same process of generating default datatype transformers is followed for every local variable whose datatype has changed, for every function. By default, each stack transformer preserves all local variables in l_{new} based on their

original values in l. Preserving these values involves invoking the appropriate datatype transformer of a local variable if the datatype is of a compound type (struct or union), or simply copying the variable if it is a primitive type. Only new local variables need to be initialized to a default value by the user. The new formal parameters p_{new} of a stack frame are also automatically preserved from their old value in p. This happens indirectly when stack reconstruction descends into a callee stack frame, through a function call, to reconstitute the callee. If a function signature is changed, such as reduced by one field, the stack transformer preserves the remaining fields with no user assistance. If the function signature is extended by one field, the user needs to initialize only the new field.

Producing stack transformers is a major difference from other DSU systems [32, 28]. Although the patch generator produces a single stack transformer for each function, the transformer may need to map the state differently depending on the execution point at which the function was paused. This complicates the generation of stack transformers because the stack transformer code needs to be able to dynamically map the state differently depending on the execution point. The default stack transformers produced by our patch generator currently do not dynamically account for the execution point.

4.3.3 Default Execution Continuation Mapping

Execution continuation points define a correspondence to return addresses ra and Program Counters T_{PC}. Preserving execution continuation points ensures that after the update execution will resume from where it was paused. There are two challenges in preserving execution continuation. The first is how to select some execution points to be continuation points. The second is how to identify these continuation points in a way that makes it easy to preserve them without, or with very little, user input. Execution continuation point selection and identification are described next.

Selection. UpStare selects as execution continuations points all points prior to function calls and all update points. Update points are inserted in the beginning of a function and in the beginning of loops, because these points are encountered often enough to allow the runtime environment to regain control of the execution and apply an immediate update. Locks are also treated as update points to guarantee immediately regaining control of the execution in a multi-threaded application.

Continuation points are chosen as all points prior to function calls because these points are necessary in reconstructing the stack. During reconstruction, execution control flow needs to be guided to descend to a callee and reconstruct its stack frame, and transferring control flow from the beginning of a function to a callee function call achieves that. Continuation points are also all update points because reaching the update point after the reconstruction finishes (the update completes) is necessary

to resume the program. In general, programs are resumed from the execution point they were paused.

Selecting additional continuations points, such as one in every basic block, is not necessary. That is because any additional code that should be executed after an update, but before the continuation point, can be executed by the stack transformer. Selecting more continuation points is straightforward.

Identification. To identify continuation points, UpStare follows the simple approach of assigning unique numeric ids to continuation points in the order they appear in each function body. By default, UpStare saves and restores the continuation points of each stack frame for all threads. It essentially maps continuation points with the same enumerator in Π and Π_{new}. If the call stack of the application did not change, the loop structure did not change, and no new function calls are issued in the body of functions that are active on the stack then no continuation mappings are needed. This mapping of continuation points is effective in practice for updates.

It is possible to improve the identification (not the selection) of continuation points so that the input required by the user is minimized further. Under the current identification scheme of continuation points, it is expected that continuation points would need mappings mostly when new function calls are issued in the body of functions that are active on the stack, even if the call stack of the application and loop structures did not change. Instead of assigning numeric ids to a continuation point one could instead use a string combining the name of the callee function and the ordinal number of times it is called inside the caller function. This number is necessary because some functions are called multiple times, such as the die() function in vsFTPd or elog_finish() in PostgreSQL, and distinguishing between the continuation points of each one is needed. Under this proposed identification scheme, the addition of new function calls in the body of functions active on the stack would not alter the continuation mapping and would need no input from the user.

4.4 User Interface

UpStare provides an interface to the user for mapping the state from the old version to the new version. The interface allows users to specify datatype transformers, stack transformers, and execution continuation transformers. These transformers expose to the user a copy of the old state of each transformer (e.g. the value of a variable of an old datatype) and allow the user to construct the new state. The transformers developed by the user are supplied to the compiler of updateable programs to produce a dynamic software update patch. Alternatively, if the user wants help in producing the transformers the user can use the patch generator.

The remaining of this section describes how datatype transformers, stack transformers, and execution continuation transformers can be developed by the user.

4.4.1 Datatype Transformers

UpStare provides an interface for implementing a transformer that updates all global variables on the heap h. For each global variable than needs to be updated, it is expected that the user declares a new global variable and implements a mapping that copies the values of the old global variable to the new global variable. Since some global variables may be instances of the same datatype, it is beneficial for a user to implement a general datatype transformer, and then apply this transformer in all variables of that datatype (as already shown in Figure 5). Thus, the transformer for global variables often calls multiple, individual datatype transformers that are used to update global variables, one call per variable instance. These datatype transformers are reusable. They can also be called by stack transformers to update instances of local variables l and formal parameters p on the stack. The global transformer can also initialize global variables.

```
1 extern struct parseconf_str_setting parseconf_str_array[26];
2        struct parseconf_str_setting parseconf_str_array_v200[29];
3        int tunable_no_log_lock;
4        int tunable_ssl_enable;
5        int tunable_allow_anon_ssl;
6        ...
7
8 int upstare_global_transformer_v200()
9 {
10   upstare_transformer__parseconf_str_setting__26to29(&parseconf_str_array,
11                                                      &parseconf_str_array_v200);
12   ...
13
14   // Initialize new global variables
15   tunable_no_log_lock = 0;
16   tunable_ssl_enable = 0;
17   tunable_allow_anon_ssl = 0;
18   ...
19 }
```

Fig. 6. Transformer (Part) for Global Variables (vsFTPd v1.2.2 to v2.0.0).

Figure 6 shows an example of the transformer (part) developed to map global variables in vsFTPd from v1.2.2 to v2.0.0. The user implements an additional datatype transformer (invoked in line 10, but not shown) to map the global variable parseconf_str_array (line 1) from the old version that contains 26 array elements to the new version that contains 29 elements. This new version of the global variable needs to be declared by the user (line 2). The user also initializes (lines 15-17) new global variables (lines 3-5) as needed.

Implementing mappings of global variables is not enough. The user also needs to supply to the compiler of updateable programs the new versions of the program code that use the updated and new global variables. This means that for every updated global variable the user needs to identify program code that uses the variable and

modify that code to use the new version of the variable. This can be time-consuming and error-prone, and it outlines the need for a patch generator that automatically produces such program code.

4.4.2 Stack Transformers

UpStare also provides an interface for modifying the stack of the old version of the application. This is accomplished with a combination of stack transformers and execution continuation transformers. Stack transformers define how an old stack frame is mapped to a new stack frame. Execution continuation transformers (discussed next in Chapter 4.4.3) allow additional operations on stacks, such as inserting and removing stack frames, and changing the Program Counter and return address of stack frames.

The system provides the user with access to the old local variables, and expects the user to construct the state of the new local variables and the new formal parameters. Note that for performance reasons the current UpStare implementation no longer saves and provides the old formal parameters of a stack frame because these values where not used. However, these values should be saved and made available to the user, and reverting the implementation to enable this support is planned.

The old local variables are made available to the user in the stack transformer as a formal parameter. This parameter is a void* pointer to a struct datatype that groups all the local variables used by the old version of a function. A void* pointer is used to simplify the implementation of the runtime system. Similarly, the new local variables are also grouped in a struct pointer formal parameter and it is expected that the user will save in this parameter the values of the new local variables. This means the user needs to properly define the struct datatype of the new local variables. Additionally, the new formal parameters are grouped in a void* pointer to a struct and made available to the user. The user does not need to explicitly preserve the formal parameters, because the values of the formal parameters are set during the function call in the parent stack frames. However the user has the option of modifying the formal parameters if necessary.

Stack transformers are also able to modify values held temporarily in computation registers (e.g. the return value of a function call, which will be passed as a parameter to a different function). Temporary values in registers are available at a high-level as temporary local variables produced by CIL.

Figure 7 shows an example of a stack transformer that maps do_file_send_ascii() in vsFTPd from v1.1.3 to v1.2.0. The update adds the new local variable chunk_size in v1.2.0 (line 9). The user is required to define the struct datatype of the local variables in the old (lines 1-5) and new (lines 7-12) versions. Given these datatype definitions, the user implements the state transfer that preserves existing stack variables (lines 27-30) and initializes the new variable (line 33).

```
1 struct local_do_file_send_ascii_v113_s {
2    struct vsf_transfer_ret ret_struct ;
3    unsigned int num_to_write;
4    int retval;
5 };
6
7 struct local_do_file_send_ascii_v120_s {
8    struct vsf_transfer_ret ret_struct;
9    unsigned int chunk_size; // This is a new local variable
10   unsigned int num_to_write;
11   int retval;
12 };
13
14 void upstare_transformer__do_file_send_ascii(void *transform_stack_to,
15                                              void *transform_stack_from,
16                                              void *transform_params_to)
17 {
18   struct local_do_file_send_ascii_v120_s *stack_to;
19   struct local_do_file_send_ascii_v113_s *stack_from;
20   struct formal_do_file_send_ascii_v120_s *params_to;
21
22   stack_to = (struct local_do_file_send_ascii_v120_s *) transform_stack_to;
23   stack_from = (struct local_do_file_send_ascii_v113_s *) transform_stack_from;
24   params_to = (struct formal_do_file_send_ascii_v120_s *) transform_params_to;
25
26   // Preserve existing stack variables
27   upstare_transformer__vsf_transfer_ret(&stack_from->ret_struct,
28                                         &stack_to->ret_struct);
29   stack_to->num_to_write = stack_from->num_to_write;
30   stack_to->retval = stack_from->retval;
31
32   // Initialize new stack variable
33   stack_to->chunk_size = 65536;
34 }
```

Fig. 7. Stack Transformer for do_file_send_ascii() (vsFTPd v1.2.2 to v2.0.0).

UpStare also exposes an interface to the user for accessing the old state of other stack frames. Consider an application that contains functions f() and g() on the stack and an update that needs to merge these two functions into one function called h(). The user needs to implement a stack transformer for h(), but the function signature of stack transformers provides access only to the state of f(). In this case the user can use an API provided by the UpStare system that allows access to the state of all saved stack frames, including both f() and g().

4.4.3 Execution Continuation Transformers

UpStare provides the user with an interface that allows overriding the saved continuation points and mapping them to different points. This allows the user to alter the runtime call stack of the program and have it resume with a different call stack after the update. For example, UpStare allows the user to add new stack frames, remove stack frames, or replace a stack frame with the frame of a different function. It also

allows the user to select a different continuation point from which every stack frame should resume.

The capability to modify execution continuation points is necessary in cases where an update requires a thread to "escape" from execution of a long-lived loop or function. If an update completely removes a feature provided by a program, and a thread was paused executing that feature, resuming the update requires the thread to break out of the corresponding code providing that feature and continue from a different execution point that is valid for the update. For example, an update from vsFTPd v1.1.2 to v1.1.3 modifies parts of the function do_sendfile() to execute 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. Modifying execution continuation points is also necessary when functions and loops are merged or partitioned.

```
1 struct vsf_transfer_ret vsf_ftpdataio_transfer_file(
2   struct vsf_session* p_sess, int remote_fd,
3   int file_fd, int is_recv, int is_ascii)
4 {
5   // Continuation point 1
6   if (!is_recv) {
7     if (is_ascii) {
8       // Continuation point 2
9       return do_file_send_ascii(p_sess, remote_fd, file_fd);
10    } else {
11      // Continuation point 3
12      return do_file_send_binary(p_sess, remote_fd, file_fd);
13    }
14  } else {
15    // Continuation point 4
16    return do_file_recv(p_sess, remote_fd, file_fd, is_ascii);
17  }
18 }
```

Fig. 8. Continuation Points in vsFTPd v1.2.2

Figures 8 and 9 show an example of how the user can map the execution continuation when the function do_file_send_binary() is removed from the stack and is replaced with function do_file_send_sendfile() in an update of vsFTPd from v1.2.2 to v2.0.0. This is an update where the function is being renamed, although the new version executes the same task and removes some source code compared to the old version. Updating this function requires mapping the ra to its parent stack frame vsf_ftpdataio_transfer_file(). It requires mapping continuation point 3 from v1.2.2 (Figure 8, line 11) to continuation point 5 in v2.0.0 (Figure 9, line 19), including supplying the new parameters curr_offset() and num_send() (Figure 9, lines 5-6; initialized in the corresponding stack transformer as lines 16 and 18) to the new version do_file_send_sendfile(). Without this mapping an update would incorrectly resume from the saved return address ra=3 in v2.0.0 (Figure 9, line 16), which would load

```
1 struct vsf_transfer_ret vsf_ftpdataio_transfer_file(
2   struct vsf_session* p_sess, int remote_fd,
3   int file_fd, int is_recv, int is_ascii)
4 {
5   filesize_t curr_offset;
6   filesize_t num_send;
7
8   // Continuation point 1
9   if (!is_recv) {
10    if (is_ascii || p_sess->data_use_ssl) {
11      // Continuation point 2
12      return do_file_send_rwloop(p_sess, file_fd,
13                                 is_ascii);
14    } else {
15      // Continuation point 3
16      curr_offset = vsf_sysutil_get_file_offset(file_fd);
17      // Continuation point 4
18      num_send = calc_num_send(file_fd, curr_offset);
19      // Continuation point 5
20      return do_file_send_sendfile(p_sess, remote_fd,
21                                   file_fd, curr_offset, num_send);
22    }
23  } else {
24    // Continuation point 6
25    return do_file_recv(p_sess, file_fd, is_ascii);
26  }
27 }
```

Fig. 9. Continuation Points in vsFTPd v2.0.0.

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 10 shows how the user declares the variable (source code in C) that expresses the continuation mapping to update to vsFTPd v2.0.0. There are two mapping points for vsf_ftpdataio_transfer_file(): 3 maps to 5 (line 5), and 4 maps to 6 (line 6). Continuation points 1 and 2 use a default mapping: they map to their old values of 1 and 2, hence they do not need to be overridden by the user. Also the stack frame of function do_file_send_binary() is replaced with a stack frame for do_file_send_sendfile() (lines 9-10) and execution continues from the replaced function at an offset continuation of -4 (lines 12-16), which indicates that, compared to do_file_send_binary(), some code from the beginning of do_file_send_sendfile() was removed.

4.5 Multi-Threaded Updates

Updating a multi-threaded or multi-process application requires all threads to be blocked. If some threads are executing while reconstruction is underway and global variables are mapped, the possibility remains open for an executing thread to corrupt global data, for example due to a type-safety violation.

UpStare adapts an algorithm that blocks all threads in heterogeneous checkpointing for multi-threaded applications[96] to dynamic updates. The idea is to force all but

```
1 upstare_mapping_t mappings_v200[] = {
2   { "vsf_ftpdataio_transfer_file",
3     "vsf_ftpdataio_transfer_file",
4     2, // 2 continuation points are mapped
5     { { 3, 5 }, // continuation point 3 maps to 5
6       { 4, 6 }  // continuation point 4 maps to 6
7     }
8   },
9   { "do_file_send_binary",
10    "do_file_send_sendfile",
11    5,
12    { { 6, 2 },
13      { 7, 3 },
14      { 8, 4 },
15      { 9, 5 },
16      { 10, 6 }
17    }
18  }
19 };
```

Fig. 10. Continuation Mapping (Part) to Update vsFTPd v1.2.2 to v2.0.0.

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.

Since some threads may block waiting on a lock owned by another thread that is blocked, these threads cannot reconstruct their stack because they have not encountered an update point. Extending all locks to be update points too allows updating such threads.

The algorithm outlined above has been extended to support blocking threads that use counting semaphores[96], but the current implementation does not yet integrate that capability with the dynamic update system.

Multi-process updates. Support for multi-threaded updates is extended in multi-process applications. UpStare keeps track of all processes created by an application and coordinates an atomic update among all the threads of all processes involved in the application.

To keep track of all processes in an application, fork() is replaced with a wrapper call that maintains 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.

4.6 Blocking System Calls

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

Calls to sendfile(), which is used in vsFTPd for file transfer, are segmented into 256KB chunks. Segmentation for for send() is not yet implemented 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: UpStare issues 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 execution should unblock, UpStare writes 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 has not been implemented yet, is to transform programs to 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. The 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().

Another approach to handling blocking system calls is to use worker threads. It is possible to start a few threads that would be available (sleeping in an infinite loop) when a blocking I/O call must be issued and have these threads issue the call. The threads would retrieve through pointer indirection the function that would issue the call. This approach would be a good option when instrumenting long-lived applications that do not frequently fork() additional processes. However, it would not be a good option for TCP servers that frequently fork() short-lived connection handlers because it does not alleviate the cost of pthread_create() to start the worker threads.

Finally, one more approach is to use the Asynchronous I/O (AIO) operations defined by the POSIX.1b standard. A practical limitation encountered implementing this approach is that AIO support was not provided for the accept() call in the Linux environment (Debian 4.0) used to develop UpStare. This support is provided in other operating systems, such as zOS.

4.7 Conclusion

This dissertation provides a DSU system, called UpStare, that implements the whole-program update mechanism. This mechanism is suitable for applying immediate updates. UpStare uses stack reconstruction to map the state of threads and to ensure updates are atomic. It also forces all executing threads to block and transforms blocking system calls to non-blocking to provide bounded delay. The user can provide a general mapping function to map the old application state to any desired state of the new version of the application. This is not practical, so UpStare provides default state mappings by automatically generating datatype and stack transformers and by preserving execution continuation points. The ability to map execution continuation points is unique in UpStare and it is necessary in updates that require a thread to escape a long-lived loop or function. It is also necessary in updates that merge or partition functions and loops. The default state mappings are effective in practice and could be improved to further minimize user input.

UpStare does not yet integrate support for multi-threaded applications that use counting semaphores, or support for mapping pointers, which were developed in previous work.

Chapter 5
RUNTIME SAFETY CHECKING

Chapter 4.2 described the implementation of stack reconstruction which guarantees atomic updates under the whole-program update model. This chapter presents an approach for providing runtime safety checks that can be useful to all updating models. Note that this approach does not rely on stack reconstruction and is architecture (and operating system) independent. It could be incorporated in other DSU systems. This approach is used to enforce transaction-safety in UpStare.

Runtime safety checks are enforced by consulting information about the application call sequences (one per thread) and the call site for every call in these sequences. This information is called the *context-sensitive call stack* information. This information is available at any point during the execution and is maintained using a *dynamic stack tracing* mechanism. This chapter describes how dynamic stack tracing can be used to provide more accurate transaction safety and type-safety. Dynamic stack tracing is used to provide more accurate transaction-safety in UpStare, but it is not used to provide type-safety because UpStare already guarantees type safety under the whole program update model.

It is important to note that unlike other approaches dynamic stack tracing can be used to enforce type-safety and transaction safety *more accurately*. In fact, the context-sensitive call stack has information about the origin of the function calls (it contains the return address of call-sites) and this information is available only during runtime. Existing work *conservatively* enforces type-safety and transaction-safety by computing universally type-safe [38] and transactionally-safe [39] update points. This computation needs to be conservative because it is carried out at compile-time and must account for the multitude of contexts (call-sites) from which functions can be called in an application.

5.1 Dynamic Stack Tracing

Dynamic stack tracing is an approach for capturing the state of the stack of a running program to facilitate runtime safety checks. Programs are instrumented to efficiently and dynamically maintain their stack state at a high-level (source-code) and offer this information to the dynamic software updating runtime environment to enforce safety

checks before an update is applied. The captured state is architecture (and operating system) independent.

The stack trace dynamically captures the names of functions that are active on the stack. For each function that is active on the stack, the instrumentation also saves the execution point from which the next stack frame was created when the callee function was called. The combination of function names and their execution points provides an accurate context-sensitive call stack trace. The execution points captured are equivalent to the continuation points described in Chapter 4.4.3. Using this call stack trace, safety checks such as type-safety and transaction-safety can be enforced more accurately. Type-safety can be enforced if type information is precomputed (statically) for every continuation point. Transaction-safety can be enforced if a user forbids updates from being applied inside specific regions of code which are active on the stack.

An alternative approach would be to extract a stack trace only when it is time to enforce safety checks instead of continuously maintaining it at runtime. This is possible at the lower-level (binary-code) by examining the processor stack pointer (ESP) and identifying the frame pointers of each function [26, 33]. But this approach is more difficult. It depends on the compiler ABI, the system architecture, and requires developing a mapping of binary code to source code. This approach also fails if a program is compiled without frame pointers or if it calls library functions that have been compiled without frame pointers (e.g. several functions in GNU libc). For these reasons UpStare chooses the approach of dynamically tracing the stack at the source-code level.

Design Considerations. Since the runtime safety checks need to consider the safety of multiple threads in an application, the dynamic stack tracing instrumentation needs to be able to capture the stack state of multiple threads. This can be challenging depending on the multi-threading support provided by the operating system. There are two possibilities: (a) saving the state in a global variable shared by all threads, and (b) saving the state locally in each thread. Under both possibilities, the thread state should be saved efficiently. This means a stack tracing implementation should not introduce locking among the threads when maintaining the trace.

Additionally, supporting stack tracing for multi-process applications needs to be considered. The POSIX standard specifies that (a) when a process issues fork() the spawned child process continues using the same memory of the parent process, and (b) the child continues execution of only the thread that issued the fork(). These two specifications imply that if the parent process has multiple threads executing, thus multiple stack traces maintained, the child process needs to re-initalize (free from memory or clean for future use) these traces (of no longer executing threads).

A dynamic stack tracing implementation should efficiently re-initialize stack traces when children processes are spawned.

In conclusion, an ideal dynamic stack tracing implementation should not introduce locking among the threads when maintaining the trace and it should efficiently re-initialize stack traces when children processes are spawned. With these design considerations in mind, the remainder of this section describes the possible options for a dynamic stack tracing implementation.

5.1.1 Stack Tracing in a Global Variable

This section discusses a design that uses a global variable to save the stack traces of all threads. This design is not a good option because using a global variable may require locking to maintain the trace of multiple threads, depending on the threading implementation of an operating system. UpStare does not implement this design.

Saving a thread state in a global variable needs an array of saving per-thread state and relies on the thread id for indexing in such an array. But the numeric range of the thread id reported by an operating system may vary depending on the threading implementation of the system. For example thread ids may be too large (e.g. long values) for indexing in an array. This would make it impossible to allocate enough memory for maintaining such an array.

Consider a threading implementation of the POSIX Threads API, which is the standard for portable operating systems. The maximum number of threads that can be started by a process (PTHREAD_THREADS_MAX) is often set to 16384. This means it should be possible to allocate a small array to maintain such a number of threads. However the ids assigned to each thread by the POSIX Threads implementation in an operating system may not lie in the range of 1-16384. For example, Linux reports thread ids that are unsigned long values (e.g. 3085178560). These values cannot be used to index in an array that keeps per-thread information. Solaris reports thread ids that begin from 1 but the thread ids are not recycled to start from 1 again when PTHREAD_THREADS_MAX is reached, hence these values cannot be used to index in an array that keeps per-thread information either.

It is possible to implement a hash-table for multi-level indexing, but this requires locking for accessing the hash table, which would fail to scale when the number of threads increases. Another possible solution is to change the threading implementation to recycle thread ids, similar to recycling process ids in operating system kernels. But this would make the stack tracing approach dependent on a threading implementation that is capable of thread id recycling, which may not be available in all operating systems.

The POSIX Threads API addresses this problem generally by allowing definitions of thread-local data, as explained in the next section.

5.1.2 Stack Tracing Using the POSIX Threads API

An alternative design relies on the POSIX Threads API to implement stack tracing. This design meets the ideal goal of not introducing locking among the threads when maintaining the trace. However, re-initializing stack traces when children processes are spawned depends on the number of threads that were executing on the parent process. UpStare implements this design and evaluates its performance.

As shown in the previous section, one cannot assume that the implementation of the POSIX Threads API will report thread ids that fall in a specified range and that are recycled. The POSIX Threads API solution to this problem is to allow defining thread-local data using keys. Keys are common to all threads, but the value associated with a key is different between threads. To support stack tracing using keys, first a key is created with pthread_key_create(). For each thread created, a stack trace is allocated on the heap (with malloc()) and it is associated to the thread with pthread_setspecific(). During runtime, the stack trace can be retrieved using pthread_getspecific().

The stack trace is maintained as a fixed-size array of 256 elements and this is enough room to manage stack traces of large applications. For example, the maximum stack depth of PostgreSQL v7.4.16 is only 35 frames. This array is treated as a stack. When a new function is entered an entry is pushed in this stack. Before a function returns the last trace entry is popped from the stack (by decrementing a stack counter). The stack trace saves for each function that is active on the stack the execution point that was visited in the function. This captures accurately the calling context for all active stack frames and the execution point of the stack frame in which the Program Counter currently executes.

Figure 11 shows how programs are transformed for dynamic stack tracing using the POSIX Threads API. In the main entry-point function a stack trace variable pointer *trace is instantiated (line 15). Stack tracing for the main thread is initialized (line 17) by creating a key, allocating the trace on the heap, and associating it with the key. The stack trace pointer is retrieved (line 18) and used to maintain the trace. A trace record is pushed on the stack (line 19) and records that execution is currently in main() (line 20). The trace saves the execution point in main() (line 21) before functionA() is called (line 22). When functionA() is entered, a stack trace pointer is retrieved again (line 6). A new trace record is pushed on the stack (line 7) which records the program is currently executing in functionA() (line 8). Before this function returns to main(), its trace record is popped from the stack trace (line 10). The trace record of main() is also popped from the stack trace (line 23) before the stack trace is freed from memory (line 24) and main() exits.

This approach does not require locking at runtime to maintain the stack trace. A limitation of this approach is that if a program issues a fork() call, the child needs to

```
1  void functionA(void)
2  {
3    int a;
4
5
6
7
8
9    a = 5;
10
11 }
12
13 void main(void)
14 {
15
16
17
18
19
20
21
22   functionA();
23
24
25 }
```

(a) Non-Instrumented

```
1  void functionA()
2  {
3    int a;
4    trace_t *trace;
5
6    trace = pthread_getspecific(trace_key);
7    trace->counter++;
8    trace->function_names[trace->counter] = "functionA";
9    a = 5;
10   trace->counter--;
11 }
12
13 void main(void)
14 {
15   trace_t *trace;
16
17   trace_init("main");
18   trace = pthread_getspecific(trace_key);
19   trace->counter++;
20   trace->function_names[trace->counter] = "main";
21   trace->execution_points[trace->counter] = 1;
22   functionA();
23   trace->counter--;
24   trace_exit();
25 }
```

(b) Instrumented

Fig. 11. Dynamic Stack Tracing Using the POSIX Threads API.

free from the heap any stack traces that had been maintained by the parent. Failure to free this memory could result in the child process, and any sub-children it may spawn, to run out of memory. Freeing stack traces increases the latency of spawning children processes and depends on the number of threads that were executing in the parent. To free the memory of stack traces, fork() is replaced with a wrapper call that cleans up this memory.

5.1.3 Stack Tracing Through Parameter Passing

An alternative design is to pass the whole stack trace as a parameter to every function when it is called. This design meets both goals of an ideal dynamic stack tracking implementation: it *does not introduce locking* among the threads when maintaining the trace and it *efficiently re-initializes stack traces of child processes*. UpStare implements this design and compares its performance to the performance of a stack tracing implementation using the POSIX Threads API. The details of the design follow.

This design saves the dynamic stack trace of each thread locally but without relying on the POSIX Threads API. To maintain the stack trace locally per thread, all thread entry-point functions (like the main() function) are automatically identified and for each one a local variable is declared that is used for maintaining the trace. The variable is passed as a parameter to all callees of the thread entry-point function.

```
1  void functionA(void)
2  {
3    int a;
4
5
6
7    a = 5;
8
9  }
10
11 void main(void)
12 {
13
14
15
16
17
18
19   functionA();
20
21 }
```

(a) Non-Instrumented

```
1  void functionA(trace_t *trace)
2  {
3    int a;
4
5    trace->counter++;
6    trace->function_names[trace->counter] = "functionA";
7    a = 5;
8    trace->counter--;
9  }
10
11 void main(void)
12 {
13   trace_t trace;
14
15   trace_init(&trace_var, "main");
16   trace.counter++;
17   trace.function_names[trace.counter] = "main";
18   trace.execution_points[trace.counter] = 1;
19   functionA(&trace);
20   trace.counter--;
21 }
```

(b) Instrumented

Fig. 12. Dynamic Stack Tracing Through Parameter Passing.

This allows the callees to maintain the stack trace in this variable when they are on the stack. It eliminates the need to call pthread_getspecific() in every function to retrieve a pointer to the stack trace.

Figure 12 shows how programs are transformed for dynamic stack tracing. In the main entry-point function a stack trace variable trace is instantiated (line 13) and initialized (line 15) for the main thread. A trace record is pushed on the stack (line 16) and records that execution is currently in main() (line 17). The trace saves the execution point in main() (line 18) before functionA() is called with the stack trace as a parameter (line 19). When functionA() is entered, a new trace record is pushed on the stack (line 5) which records that the program is currently executing in functionA() (line 6). Before this function returns to main(), its trace record is popped from the stack trace (line 8). The trace record of main() is also popped from the stack trace (line 20) before main() exits.

Similar to the approach of stack tracing using the POSIX Threads API, this approach also does not require locking at runtime to maintain the stack trace. An advantage of this approach is that when children processes are spawned no additional effort is required to free the memory of stack traces. That's because stack traces are instantiated on the stack, instead of the heap. When a child process runs, the stack frames of any threads that were executing in the parent simply remain unused.

5.1.4 Stack Tracing and Stack Reconstruction

Three approaches to implement dynamic stack tracing were outlined. Regardless of the approach followed, stack reconstruction can enable stack tracing only when an update is desired. This means that during normal execution of the application (when an update is not desired) stack tracing can be disabled, and there will be no overhead due to stack tracing during this period. The overhead of stack tracing will be incurred temporarily: from the time an update is requested to the time it is applied. The current UpStare prototype does not yet implement this feature.

When an update is requested, the application can be updated to a version that contains a dynamic stack tracing implementation. Enabling stack tracing will be the only difference in the application in this stage of the update, and the stack traces of all threads can be initialized to correspond to the current state of each thread stack. This is straightforward because stack unrolling saves the continuation points (equivalent to the execution points needed by stack tracing) of each stack frame. Combining the names of functions with the continuation points saved provides the stack trace of each thread.

After the application has been updated to maintain dynamic stack traces, the application resumes execution. At this stage, the runtime safety checking constraints provided by the user (such as transaction-safety constraints) can be enforced. The application pauses each of its threads at update points that meet these safety constraints and then initiates stack reconstruction to update to the new version. The new version no longer contains instrumentation for dynamic stack tracing, and the application resumes execution without the overhead of stack tracing.

5.2 Type-Safety

Dynamic type-safety checking is developed using statically computed type information and the context-sensitive call stack maintained through dynamic stack tracing. This section presents a proof-of-concept implementation that shows type-safety can be enforced dynamically.

The static computation of type information is currently implemented at the function granularity. The implementation of static computation can be extended to the execution point granularity for more accurate type-safety checking, but in general an absolutely accurate safety checking is undecidable.

For each function, the computation identifies the types of all local variables, the types of formal parameters, the type of the return value, the types of all variables set as l-values (these could include global variables) and any types on which these types depend on (e.g. as fields in a struct). The computation produces in a variable a listing of all this type information for each function. This is conservative because

it identifies types that may be used anywhere within the function, even if an update may be applied at an execution point from which some types may no longer be used until the end of the function.

When an update is requested and an update point is encountered the runtime environment checks for type-safety. The runtime environment is given the execution point at which an update point is encountered and the context-sensitive call stack trace. It is also supplied with a list of types, computed by the patch-generator, that need to be updated. If one of the types that will be updated is in the list of types that are used by a function that is already on the stack, the update is forbidden since it would violate type-safety.

5.3 Transaction-Safety

The context-sensitive call stack maintained through dynamic stack tracing is also used to develop dynamic transaction-safety checking. The user supplies transaction-safety constraints to the system and the runtime environment ensures these constraints are satisfied before an update is applied.

The safety constraints express for each function of each thread which regions of code should not be active on the stack when an update is applied. They can also express the opposite: which regions of code should be active on the stack during the update. The regions of code are defined at execution point granularity. All of the constraints (among all threads) must be satisfied before an update is applied.

Note that the transaction-safety constraints are supplied to the updating system dynamically when an update is requested. They do not need to be defined before the application begins execution. If a user realizes that the constraints originally defined are incorrect or too strong, leaving no possibility of applying an update, the constraints can be updated while the application is running. This is not possible using an approach that enforces transaction-safety constraints at compile-time.

Stopping a thread at an execution point that does not violate transaction-safety constraints does not guarantee that other threads will also stop at execution points that do not violate transaction-safety constraints. Addressing this problem is beyond the scope of this dissertation.

5.4 Conclusion

UpStare provides runtime safety checking using a new approach of dynamic stack tracing. This approach does not rely on stack reconstruction and could be incorporated in existing DSU systems. Dynamic stack tracing dynamically maintains a context-sensitive call stack for each thread of the application. This information can be used to more accurately, instead of conservatively, enforce type-safety and transaction-safety.

Another benefit of dynamic stack tracing is that safety constraints can be dynamically modified during application execution. Stack traces combine the names of functions active on the stack with their execution points.

An ideal dynamic stack tracing design should not introduce locking among the threads when maintaining the stack trace and it should efficiently re-initialize traces when children processes are spawned. Two designs are presented that apply source-to-source transformations in applications to dynamically maintain stack traces. The first approach relies on the POSIX Threads API and the second passes a pointer to the stack trace variable as a parameter between function calls. Dynamic stack tracing incurs some overhead and, as will be shown in Chapter 6, relying on the POSIX Threads API performs better than parameter passing in the single-threaded, multi-process applications that are examined. When combined with stack reconstruction, the overhead of stack tracing can be incurred temporarily, from the time an update is requested to the time the update is applied.

Chapter 6
EVALUATION

This chapter describes the evaluation of UpStare on three applications. The data-intensive KissFFT, the vsFTPd server, and the PostgreSQL database management system. It demonstrates that stack reconstruction is necessary to apply atomic updates and that converting blocking calls to non-blocking is necessary to provide bounded delay. A limitation of the evaluation is that it does not study applications that are multi-threaded. However, it demonstrates updates of applications that are multi-process, and support for multi-process updates is an extension of support for multi-threaded updates. It gives a detailed analysis of the sources of overhead of the instrumentation, such as runtime overhead, memory footprint, and network overhead.

6.1 KissFFT

The KissFFT [1] Fast Fourier Transform library is a small (1,936 lines of code) data-intensive application. This application is studied because it performs no disk or network I/O and the aim is to identify in detail the sources of overhead of the instrumentation. The performance of UpStare is also compared with the performance of Ginseng [32] because Ginseng has been successful in applying DSU with a combination of function-pointer indirection, data-access indirection, and logical-stage extraction. KissFFT source code instrumented with Ginseng was made available to us. This application is not updated, but it is compiled to be updateable. The comparison examines the execution time, the memory footprint, and the instrumentation size of an updateable KissFFT instrumented with UpStare against an updateable KissFFT instrumented with Ginseng.

6.1.1 Execution Time

This application was used to get a better understanding of the sources of overhead introduced by the instrumentation. Various various instrumented versions of the application were prepared that selectively and progressively added UpStare instrumentations to isolate the overhead introduced by each stage of the instrumentation. Performance was measured using various versions of compilers on various processors.

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

This application was compiled and ran in the following configurations (the command-line arguments supplied are shown in parenthesis):

1. Using the original compiler.
2. Using CIL.
3. When only wrapper functions to save/restore stack frames are produced (–no-stack-reconstruction –no-function-call-indirection).

   From this point on, additional instrumentation is progressively enabled and the cumulative impact of each instrumentation stage is measured.
4. When functions are called through pointer indirection(–no-stack-reconstruction).
5. When if-statements without a body (the body contains four "no-operation" – nop – assembly instructions) are inserted for (a) update points (Figure 4), (b) the switch-statement prologue (Figure 3b), or (c) upwards stack unrolling (Figure 2b); here the aim is to measure the overhead of branch checks when the must_update and must_reconstruct() flags are not raised (–simulate-stack-reconstruction).
6. After adding the body of these if-statements. This is the full UpStare instrumentation.
7. Using Ginseng to insert only update points (–doupdate –update-points=returnonly).
8. As made available to us, already instrumented with Ginseng. This source code had been manually modified to optimize away some redundant uses of versioned pointer indirection. This optimization could be implemented automatically in Ginseng but the analysis for this optimization had not been implemented yet.

KissFFT v1.2.0 was compiled (at -O3) using float datatypes to be dynamically updateable and was used to perform 100,000 iterations on 20,000 points. Figure 13 shows the impact of the presence of reconstruction-aware code in the program and compares this instrumentation with Ginseng. To compare the results the evaluation identifies 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, we speculate that the overhead of UpStare is rooted at missed optimization opportunities due to the stack-reconstruction instrumentation.

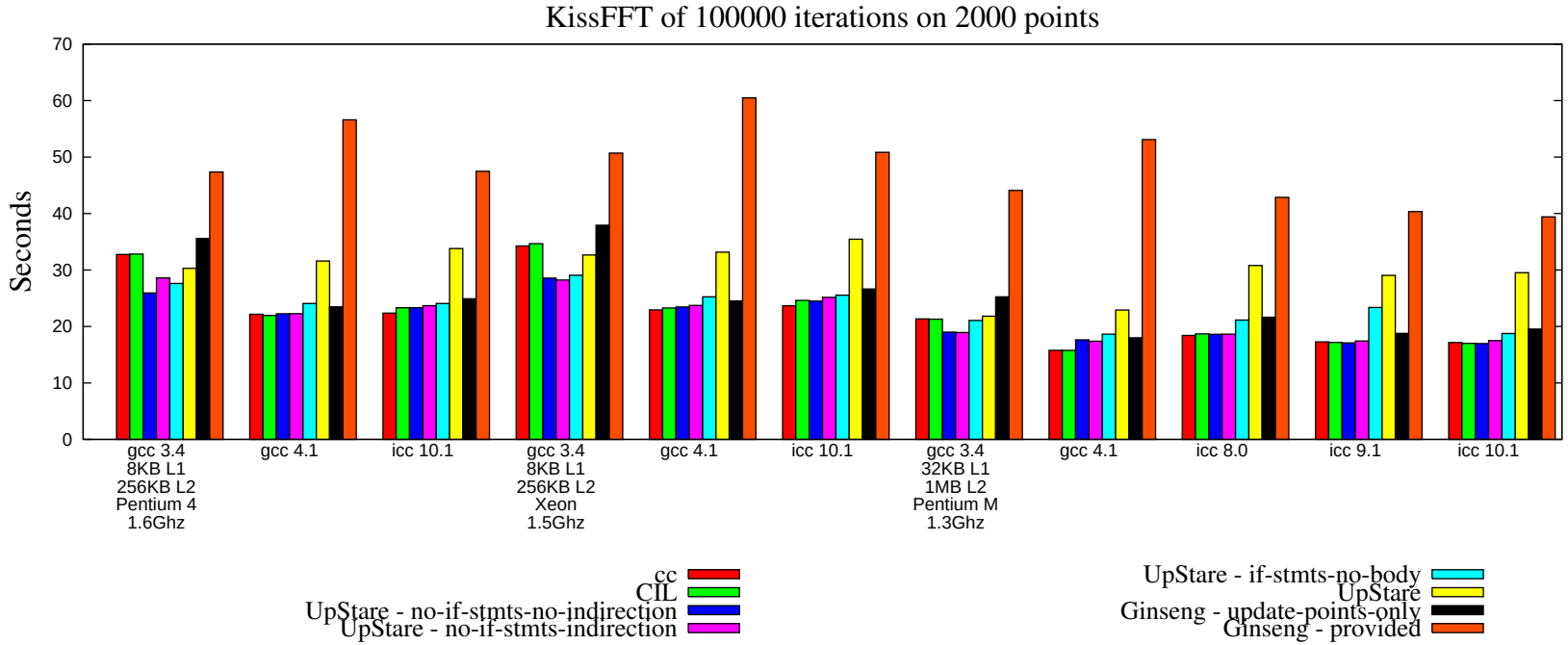

Fig. 13. KissFFT: Impact of Reconstruction Code on Running Time.

CIL. CIL parses a program, builds an internal representation that can be used for analysis, and outputs the program according to this internal representation. This output does not match precisely the original program. For example, temporary variables are introduced as the return value of each function call, and these variables can take space on the stack if they are not optimized away by a compiler. Generally, CIL doesn't alter the performance of the application. 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. Wrapper functions that save and restore stack frames are produced outside the .text segment. Producing these functions in an application should introduce zero overhead because the functions would not pollute the instruction cache. However, compared to the overhead of CIL, producing these functions reported 11.8% overhead with gcc 4.1 and 11.0% improvement with gcc 3.4, on a Pentium M. This high variability in the gcc results suggests a problem with gcc. This is in contrast to the Intel compilers which report no overhead under all processors, as was expected by this type of instrumentation.

Function indirection. Functions called through pointer indirection should incur very little, and 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). Function indirection is also used by Ginseng and other DSU systems and it is expected Ginseng has similar overhead. Also note that, as described in Chapter 4.2, function indirection is not needed by UpStare. Function indirection is a feature the current UpStare prototype provides to allow implementing updating models other than the whole-program update model, hence this overhead could be eliminated entirely.

If-statements without a body. In UpStare, if-statements are added for (a) update points, (b) reconstructing, and (c) unrolling. In comparison, Ginseng instrumentation adds only update points, which are implemented as function calls. Adding if-statements without a body (the body contains four "no-operation" – nop – assembly instructions) should not have significant impact on the function body size or the execution time. 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 if-statements in UpStare incur comparable overhead.

If-statements with a body. This is a version of the application fully instrumented with UpStare, which increases the size of functions. 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.

6.1.2 Sources of Overhead

Three possible sources of overhead are suspected in UpStare. First, the increase in function size due to instrumentation could lead to ITLB misses. Second, adding if-statements for the update points and the big switch-statement could lead to branch mispredictions. Third, the instrumentation for saving and restoring the stack state could lead to missed optimization opportunities, especially in computationally-bound loops. Each of these possible sources of overhead were investigated further to better understand their impact in the final overhead, as described next.

ITLB misses and branch mispredictions. To study the possibility of ITLB misses, OProfile[2] was used to collect processor performance statistics on the Pentium 4 with gcc 4.1 (overhead 31.3%). Performance statistics were compared between inserting if-statements without a body and inserting if-statements with a body (full UpStare instrumentation). Inserting if-statements with a body observes a 15% increase in the number of ITLB translations and an 11% increase in the number of instruction fetch requests from the branch prediction unit. Other events like ITLB misses, retired mispredicted branches and page walks showed no significant deviation. Although higher numbers of ITLB translations and instruction fetch requests are observed, these numbers are not sufficient to justify the 31.3% overhead on this processor, especially since there is no deviation in the number of ITLB misses which are more expensive. This experiment concludes that ITLB misses are not a significant factor in the performance overhead of the instrumentation of UpStare.

Two additional experiments were executed comparing the performance between inserting if-statements without a body and inserting if-statements with a longer body which still does not contain the UpStare instrumentation. If the increase of the function size is a significant factor leading to ITLB misses, then systematically increasing the function size should yield higher overhead. In the first experiment, if-statements were inserted that contained up to 2000 nop instructions, instead of just 4 nop instructions, but no change in the overhead was observed. In the second experiment, if-statements were inserted containing large amounts of code performing floating point arithmetic, but this still had no effect in the overhead. These experiments conclude that an increase in the function size does not affect performance.

Missed optimization opportunities. To study the possibility that the instrumentation forces compiler optimizations to be more conservative, OProfile was used to identify the most frequently used function in KissFFT. This function, kf_bfly4(), executes a computational loop that contains 16 computation statements (add, subtract, multiply) performed against an array. We speculate that since KissFFT performs matrix computations, and has a relatively long loop body, it presents more optimization

[2]http://oprofile.sourceforge.net

opportunities. Binary executables were produced when if-statements without a body were inserted, and using the full UpStare instrumentation. The executables were disassembled, and the assembly code produced for kf_bfly4() was compared between these two versions. The body of the loop was observed to use different assembly instructions between these two versions, which indicates the optimization of the compiler was affected by the full UpStare instrumentation.

Two experiments were executed on a Pentium M with gcc 4.1 investigating the possibility that the instrumentation that saves and restores stack frames could be affecting the compiler optimization. The first experiment executed the full UpStare instrumentation minus the calls to the functions that save stack frames in an update point. Compared to inserting if-statements without a body this instrumentation reports overhead of 13.7%. The second experiment executed the full UpStare instrumentation minus the calls to the functions that restore stack frames in the big switch-statement. Compared to inserting if-statements without a body this instrumentation reports overhead of 21.2%. These two experiments indicate that the majority of the overhead stems from the addition of the code that saves and restores the stack frames. We speculate that instrumentation by UpStare introduces high overhead in applications containing relatively long loop bodies that present more optimization opportunities because the instrumentation changes the control flow so that the compiler can no longer assume it is safe to take advantage of those optimization opportunities:

- When restoring stack frames, the compiler needs to be more conservative because the altered control flow allows for the possibility that data can be written-to externally (when a user requests an update). We believe a compiler could properly handle this case by producing code that checks at runtime this control flow dependency (the may_reconstruct and must_update flags of UpStare).

- Saving stack frames should not conflict with the optimization. However the implementation of saving stack frames was implemented generally to supply to the runtime environment a pointer to the stack frame. We speculate that the compiler needs to be conservative because it cannot guarantee that the pointer contents will not be written-to by the runtime environment which is statically linked as a library. To handle this case, the function calls that save stack frames could be annotated to indicate that the contents of the stack frame pointer are not written-to (e.g. icc supports such annotations; also the flag -fargument-noalias-global).

Another experiment instrumented the Bubblesort application which implements two nested loops in very little code. The computation in the loops accesses float

numbers from an array and swaps two elements (1 computation statement, in a relatively short loop body) and we speculate that there is little optimization opportunity in this application. The overhead of a Bubblesort (compiled at -O3) instrumented with if-statements sorting 100,000 randomly generated numbers was compared to a Bubblesort containing the full UpStare instrumentation. On a 1.6Ghz Pentium with gcc 4.1, the full instrumentation reported overhead as low as 0.9%. This indicates that for an application that has little opportunity for optimization UpStare does not introduce significant overhead.

6.1.3 Memory Footprint

Similar to the evaluation of execution time, the memory footprint of this application was also measured at the various stages of the instrumentation. The unit of measure was the resident set size, which is the non-swapped physical memory the application has used. Studying the memory footprint is different from studying the increase in the size of the application executable file due to instrumentation. This is because not every component of the instrumentation added to the application will be used during normal execution mode of the application. Some instrumentation is used only when the application is updated.

Figure 14 reports the impact of instrumentation on the memory footprint. CIL does not increase the resident set size. 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.

6.1.4 Instrumentation Size

To study the increase of the function size due to instrumentation, the size of an updateable KissFFT executable was measured at the various stages of instrumentation.

Figure 15 reports the progressive increase of the .text segment as a result of the instrumented code. As expected CIL generally does not increase the program code size. Wrapper code that saves/restores stack frames also does not affect the program code size because this code is stored outside the .text segment. However, an increase in the code size is observed and it is attributed to the functions provided by the runtime environment. These functions are listed first in the .text segment and do not affect instruction caching, as has been confirmed in Figure 13. This code increase

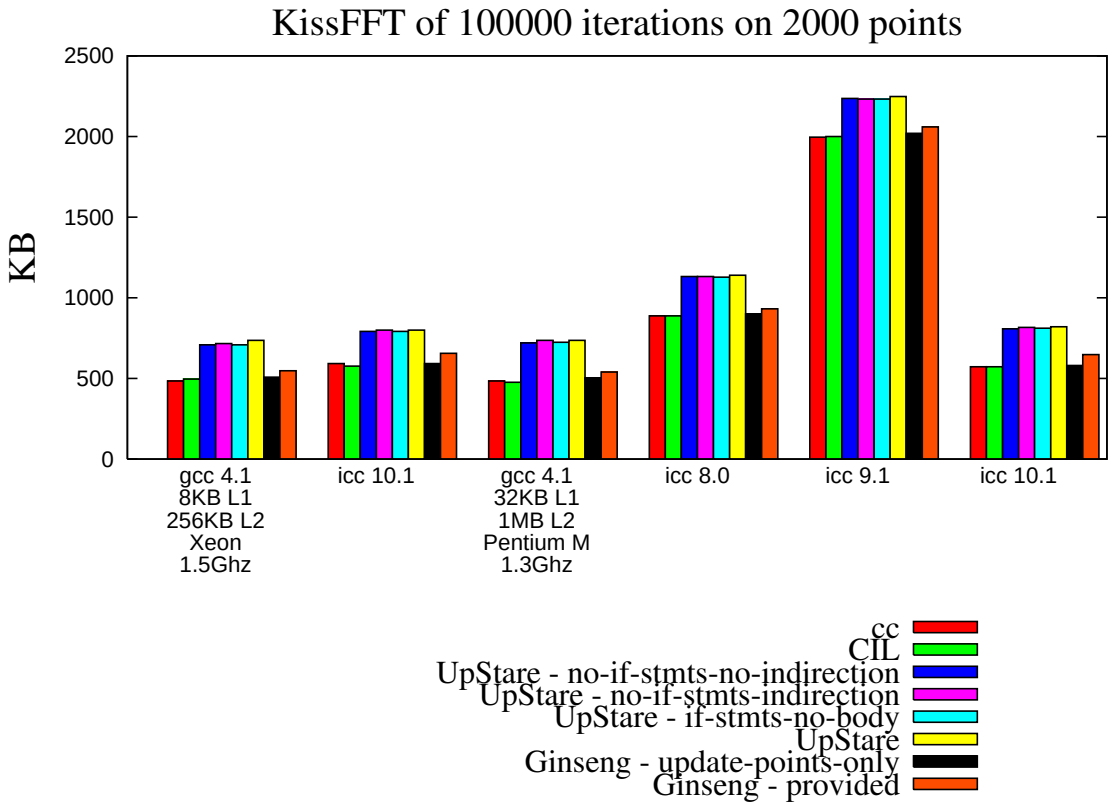

Fig. 14. KissFFT: Impact of Reconstruction Code on Memory Footprint.

is in the range of 52-57KB (depending on the compiler), which is the size of the runtime environment. Since this code does not affect the program execution time, the description of performance that follows provides a percentage of the code size difference compared to the code size produced by CIL.

Calling functions through pointer indirection often decreases the program code (up to -3,296 bytes with gcc 3.4; 36% compared to CIL). Adding if-statements without a body (the body still contains four nop instructions) increases the program code between 1,808 bytes (with icc 10.1; 8.3% compared to CIL) and 4,176 bytes (with gcc 4.1; 60.3% compared to CIL). Finally, adding the body of these if-statements is responsible for most of the code size increase. It increases the code between 8,576 bytes (with icc 9.1; 55.5% compared to CIL) and 20,672 bytes (with gcc 4.1; 225.8% compared to CIL). For the best performing configuration of UpStare with KissFFT (gcc 3.4 on a Pentium M with overhead of 38.2%) the program code increase due to the body of if-statement is 21,280 bytes (307.7% compared to CIL). It is interesting to note that Ginseng also adds considerable instrumentation in .text. For example,

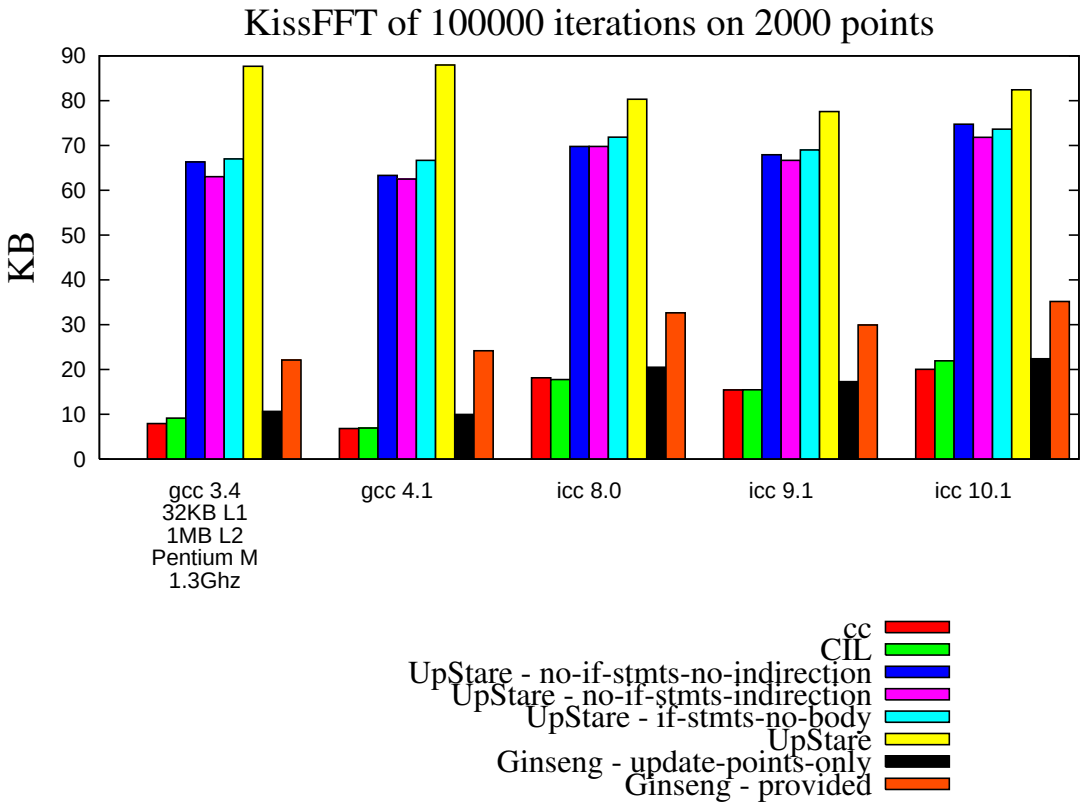

Fig. 15. KissFFT: Impact of Reconstruction Code on Function Size (.text).

the best performing configuration of Ginseng (icc 10.1 on a Pentium M with overhead of 87.1%) increases the program code by 57.1%.

6.2 The Very Secure FTP Daemon

The Very Secure FTP Daemon, vsFTPd[3], is a fast, secure, widely used FTP server. This application is studied because it is a medium-sized (∼12,000 lines of code), real-world, multi-process, server application. The aim is to update all of the versions of this application semi-automatically. As explained in Chapter 2, automatically producing semantically safe updates is a difficult problem, hence some state mappings may need to be produced manually. Given the size of this application it should be able to override some of the automatically-generated state mappings with modest effort. The next sections study the source code evolution of vsFTPd, describe the experience of updating this application, and present performance results.

[3]http://vsftpd.beasts.org

6.2.1 Source Code Evolution

The source code evolution of vsFTPd is studied to gain a better understanding of the nature of the dynamic updates that will be required. vsFTPd evolved over 5.5 years (13 releases) from v1.1.0 (July 2002) to v2.0.6 (February 2008). The study identifies additions, deletions, and updates of all datatypes, variables, and functions. However, the study does not attempt to discover function renamings: functions that are deleted in the new version but added with a different name, same body, and same formal and local parameters. That's because it is desirable to ensure representation consistency: that the application in memory is updated to match the source code of the new version. Hence, UpStare follows a policy of updating functions that are modified and adding new functions, and this subsumes functions which are renamed.

Table III summarizes the results. The results differ from the results already reported in [120]. The difference is that the table shows only datatypes defined by the application (collected without passing the –keepunused argument to CIL), rather than showing unused datatypes imported through system header files. Datatypes are more often added (15 datatypes total were added over all versions) and modified (12 total), rather than deleted (2 total). Variables are frequently added (90 total), sometimes modified (24 total) and rarely deleted (9 total). Functions are updated very often (317 total), new functions are often added (108 total) and they are less likely to be deleted (23 total).

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. For example, 35 variables were added in v1.2.0 and 16 variables were added in v2.0.0. These account for 51 variable additions out of 75 total (56.7%). Similarly, 39 functions were added in v1.2.0 and 35 functions were added in v2.0.0, and they account for 74 function additions out of 108 total (68.5%).

Ver.	Date	LoC[4]	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	105	-	-	-	-	158	-	-	-	-	401	-	-	-	-
1.1.1	2002-10-07	8,468	105	105	0	0	0	161	156	3	0	2	399	383	0	2	16
1.1.2	2002-10-16	8,731	108	103	3	0	2	165	159	4	0	2	410	391	11	0	8
1.1.3	2002-11-09	8,839	108	107	0	0	1	167	164	2	0	1	412	402	2	0	8
1.2.0	2003-05-29	10,011	114	104	8	2	2	201	159	35	1	7	444	341	39	7	64
1.2.1	2003-11-13	10,506	114	112	0	0	2	205	196	7	3	2	449	410	6	1	33
1.2.2	2004-04-26	10,547	114	114	0	0	0	204	202	1	2	1	450	439	1	0	10
2.0.0	2004-07-01	11,527	116	112	2	0	2	218	200	16	2	2	476	384	35	9	57
2.0.1	2004-07-02	11,543	116	116	0	0	0	219	218	1	0	0	476	469	0	0	7
2.0.2	2005-03-03	11,612	117	116	1	0	0	219	219	0	0	0	476	452	1	1	23
2.0.3	2005-03-19	11,743	117	117	0	0	0	226	216	8	1	2	479	444	5	2	30
2.0.4	2006-01-09	11,857	118	117	1	0	0	229	225	3	0	1	482	464	4	1	14
2.0.5	2006-07-03	11,923	118	117	0	0	1	234	228	5	0	1	482	455	0	0	27
2.0.6	2008-02-13	12,202	118	116	0	0	2	239	231	5	0	3	486	462	4	0	20

TABLE III
vsFTPd: Source Code Evolution.

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

6.2.2 Experience

A total of 13 updates to vsFTPd were semi-automatically prepared and applied. All updates were applied without modifying the original source code of vsFTPd. The goal was demonstrating that updates could be applied systematically in this application. Another goal was determining if vsFTPd required updates of functions on the stack. Updates were applied under two use cases which are typical for this type of application. The two use cases examined are:

- **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.

  In this use case the server is blocked indefinitely on a recv() call waiting for client input through the network.

- **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 had not finished transmission. An update was applied.

  In this use case the server is blocked on a sendfile() call. Even though the call offers bounded delay, there is no guarantee as to when the transmission will finish.

Dynamic update patches were prepared automatically using the patch generator and some of them were manually customized. The update patches required a total of 11 user-defined continuation mappings for the two use cases examined. This means that updates were manually verified to be semantically correct from only two update points (there are a total of 613 update points in v2.0.5). Additional mappings and manual customizations will be needed to update from other update points.

The customizations involved some manual initialization of new variables and struct fields in the automatically-generated datatype transformers. This initialization could have been automated with additional engineering effort in the patch generator. For example, added variables that were initialized to a default value could also be automatically initialized by the datatype transformers. The same applies for preserving the temporary variables produced by CIL, which are used to hold the return values of function calls and may differ in their enumeration. For example, tmp___6 in the new version may correspond to tmp___5 in the old version.

The customizations also involved more complicated state transformations executed by the automatically-generated stack transformers. These customizations cannot be automated without semantic analysis, and semantic analysis is not addressed in this dissertation. For example, the update from v1.1.1 to v1.1.2 changes vsf_standalone_main() to allocate a hash table entry for every incoming connection

and maintain data about the connection in callees of this function. The stack transformer of vsf_standalone_main() had to be customized to allocate this entry in order for the new versions of callee functions to work properly. Such manual, complex state transformations had to be defined in a total of 4 updates. From v1.1.1 to v1.1.2 (just described), from v1.1.3 to v1.2.0 (to install additional signal handlers), from v1.2.1 to v1.2.2 (to dynamically allocate memory for storing a client's originating IP address), and from v2.0.3 to v2.0.4 (to DNS resolve the client's IP address and store it in a local variable).

These manual customizations indicate that an immediate update needs to identify code that would have been executed by the new version had the application been started with the new version from the beginning. The state that would have been prepared by such code needs to be properly initialized before the new version continues. This outlines the need for analysis that guarantees semantic safety.

For the two use cases examined, a total of 22 lines of source code were written that required complex transformations that need semantic analysis to automate. 300 lines of source code were written that could have been automatically produced by the patch-generator with additional engineering effort, which included 35 lines of code for preserving temporary variables. All this source code was included in the mapping files that overrode the patch-generator defaults. These mapping files contained additional source code to enable the patch to compile and completely transfer the state (including the remaining portions of automatically-generated stack transformers, datatype-transformers, and datatype definitions), and consumed a total of 3,354 lines of code. After taking these mapping files into consideration, the patch-generator automatically produced a total of 209,845 lines of source code, which included state transformers, the new versions of functions, and old and new datatype definitions.

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, a case was observed where an update applied during a large file transfer possibly needed to escape a loop. During the update from v1.1.2 to v1.1.3 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.

Comparison with Ginseng. While Ginseng can support the update of vsf_session struct, it achieves that with data padding whose limitations have already been discussed. Also note that vsFTPd is an application that forks connection handlers that do not communicate with each other or their parent. Although Ginseng reported

updating vsFTPd, it did not update all processes of the application, or update them immediately.

The updating experience with vsFTPd met the aims set forth in updating this application. All versions of the application were systematically updated with modest effort. However, updating outlined the need for semantic safety analysis to further reduce programmer involvement. Finally, updating identified that applying updates immediately would enable the application to possibly execute new versions of functionality that were pending as forward control flow on the stack.

6.2.3 Performance

Two versions of vsFTPd, v2.0.5 and v2.0.6, were compiled with gcc 4.1 on a 2.4Ghz Xeon and measured the performance of UpStare when enabling various components of its instrumentation in vsFTPd. These measurements don't study in detail the sources of overhead of each stage of stack reconstruction since these were identified in Chapter 6.1.1. Instead the measurements study the instrumentation of stack reconstruction as a whole, the instrumentation for multi-process updates, and the instrumentation for blocking system calls. They also study the instrumentation of dynamic stack tracing using the POSIX Threads API and using parameter passing.

vsFTPd is not compiled with Ginseng to directly compare performance, although its performance is compared with published Ginseng results [32, 34]. Compiling with Ginseng is avoided because Ginseng may be more conservative than necessary in its type-safety checks and could prohibit updates to some datatypes. This would reduce the Ginseng instrumentation involved and would require user intervention to adjust. We did not request vsFTPd source code instrumented with Ginseng as we did for KissFFT. Another limitation of the measurements is that they don't report performance after a streak of applying all updates.

The experiment was setup in a client-server configuration connected with a cross-over cable to eliminate network fluctuations. This setup was necessary to accurately measure performance: in preliminary measurements UpStare reported performance improvement, which was counter-intuitive. vsFTPd was installed 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.

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 [32] but it does not support multi-process updates.

The experiment measured the latency of establishing a connection and retrieving a 32-byte file 1000 times and the throughput of retrieving a 300MB file. Table IV

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 IV
VSFTPD: IMPACT OF INSTRUMENTATION ON LATENCY.

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 ∼0.37-0.50ms (4.9-5.3%), multi-process support by ∼0.65-0.70ms (6.8-7.4%), and support for blocking system calls 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 [32], 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 measurements 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 UpStare reports zero overhead for all instrumentations.

The experiment also measured the latency and throughput of a dynamic stack tracing implementation using the POSIX Threads API and using parameter passing. These measurements were obtained at a later time and we were unable to repeat the results of Table IV (although using the same machine) for a direct comparison with the other instrumentations. The difference is that during this round of measurements the machine reports ∼0.35-0.40ms improvement. To measure the instrumentation using the POSIX Threads API, the instrumentation for multi-process support needs to be enabled to wrap the fork() call for freeing stack traces maintained by the parent.

vsFTPd Configuration	Connection Latency(ms) 32-byte file			
	Hard-disk		Memory	
v2.0.5 - NonInstrumented	9.18		9.09	
v2.0.5 - MultiProcess	9.89	(7.7%)	9.79	(7.7%)
v2.0.5 - StackTracing: PThreads	10.78	(16.2%)	10.56	(16.2%)
v2.0.5 - StackTracing: ParameterPassing	10.18	(10.9%)	10.25	(12.8%)

TABLE V
vsFTPd: Impact of Dynamic Stack Tracing on Latency.

But the multi-process support adds overhead for more than just the support for dynamic stack tracing. Thus the overhead of multi-processing support is measured and subtracted from the overhead of stack tracing using the POSIX Threads API to obtain the overhead incurred only by the stack tracing instrumentation using POSIX Threads.

Table V reports the median of 11 runs of the dynamic stack tracing measurements. Stack tracing using the PThreads API is computed to have overhead of 0.89ms (9.7%) from hard-disk and 0.77ms (12.8%) from memory. Stack tracing using parameter passing has overhead of 1ms (10.9%) from hard disk and 1.16ms (12.8%) from memory. The overhead in throughput is zero.

UpStare has the potential of reporting good performance after a streak of updates. Currently UpStare updates only the functions that have been modified instead of updating all functions of the application. This can have a negative impact on spatial locality of code and data [34] because a combination of old and new versions of active functions are spread between two separate memory regions. The update policy could be extended to update all functions, including all old functions that did not need to be updated. This would reconstitute the entire program (both code and data), and active functions would no longer be spread in separate memory regions.

6.3 PostgreSQL Database Management System

PostgreSQL[5] is the most advanced open-source database management system. This application is studied because it is a large, real-world, multi-process application (∼326K lines of code). The main differences with vsFTPd are that PostgreSQL forks connection handlers that communicate with each other through shared memory and maintains a separate process for checkpointing its write-ahead log. The goal was to update only one version and measure the performance of the instrumentation. Given

[5]http://www.postgresql.org

the size of this application, manually verifying semantic safety for all updates of PostgreSQL, as done for vsFTPd, would have been laborious and error-prone. Studying the evolution of PostgreSQL reveals that automated semantic analysis will be needed to systematically update large applications.

The next sections study the software evolution of PostgreSQL, describe the experience of applying an update for one version of this application, and present performance results.

6.3.1 Source Code Evolution

Even though only one version of PostgreSQL is updated in this experiment, it was beneficial to study the source code evolution of the 7.4.x branch. PostgreSQL is a large application with the postmaster (the database server process) consuming 215K[6]lines of code (source code from src/backend/). Multiple versions of this application are maintained across multiple branches to provide users with new features and defect corrections as soon as possible.

For example, branch v7.3.x was being maintained while branches 7.4.x and 8.0.x were being developed. v7.3.21 and v7.4.19 were released the same day (January 2008), while v8.0.0 was released $\sim$2 weeks before v7.4.7 (January 2005). This shows that to meet user needs large applications need to be maintained in multiple branches for a long time until they are stable. For example, the 7.4.x branch was maintained for 4 years. Updating from one development branch to a newer branch is likely to cause instability, because different branches aim to introduce new features that require drastic changes in the structure of the application. It is not uncommon for users to choose to update to a newer version of their originally chosen branch (e.g. from v7.4.6 to v7.4.7) rather than a new branch (e.g. from v7.4.6 to v8.0.0) to minimize regressions due to possible defects of features provided in the new branch.

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

Ver.	Date	LoC	Types					Variables					Functions				
			Tot.	Same	Add.	Del.	Upd.	Tot.	Same	Add.	Del.	Upd.	Tot.	Same	Add.	Del.	Upd.
7.3.21	2008-01-07	189,451	1521	-	-	-	-	789	-	-	-	-	5007	-	-	-	-
7.4.0	2003-11-17	213,869	1721	1029	294	87	398	829	618	124	84	87	5592	1860	940	355	2792
7.4.1	2003-12-22	214,004	1721	1719	0	0	2	827	825	0	2	2	5596	5445	4	0	147
7.4.2	2004-03-08	214,195	1721	1702	1	1	18	829	825	3	1	1	5605	5308	10	1	287
7.4.3	2004-06-14	214,222	1721	1721	0	0	0	829	829	0	0	0	5605	5451	1	1	153
7.4.4	2004-08-16	214,239	1721	1714	0	0	7	829	827	0	0	2	5605	5538	0	0	67
7.4.5	2004-08-18	214,241	1721	1721	0	0	0	829	829	0	0	0	5605	5598	0	0	7
7.4.6	2004-10-22	214,315	1721	1721	0	0	0	830	829	1	0	0	5608	5537	3	0	68
7.4.7	2005-01-31	214,416	1721	1721	0	0	0	830	830	0	0	0	5608	5435	0	0	173
7.4.8	2005-05-09	214,554	1721	1715	3	3	3	830	830	0	0	0	5610	5482	2	0	126
7.4.9	2005-10-04	214,792	1721	1719	0	0	2	831	817	1	0	13	5612	5271	2	0	339
7.4.10	2005-12-12	214,819	1721	1721	0	0	0	830	830	0	1	0	5615	5548	3	0	64
7.4.11	2006-01-09	214,963	1721	1721	0	0	0	836	830	6	0	0	5617	5561	2	0	54
7.4.12	2006-02-14	214,963	1721	1719	0	0	2	836	835	0	0	1	5617	5570	0	0	47
7.4.13	2006-05-23	215,436	1723	1719	2	0	2	839	834	3	0	2	5636	5536	20	1	80
7.4.14	2006-10-16	215,446	1723	1723	0	0	0	839	839	0	0	0	5636	5610	0	0	26
7.4.15	2007-01-08	215,576	1723	1723	0	0	0	839	839	0	0	0	5638	5594	2	0	42
7.4.16	2007-02-05	215,578	1723	1721	0	0	2	839	839	0	0	0	5638	5614	0	0	24
7.4.17	2007-04-23	215,726	1723	1723	0	0	0	840	839	1	0	0	5640	5575	2	0	63
7.4.18	2007-09-17	215,799	1723	1721	0	0	2	842	827	2	0	13	5646	5482	6	0	158
7.4.19	2008-01-07	215,954	1723	1720	0	0	3	844	840	2	0	2	5652	5566	7	1	79
8.0.0	2005-01-19	233,511	1862	1213	234	91	415	905	591	174	113	140	6131	2521	759	280	2851

TABLE VI
PostgreSQL: Source Code Evolution.

Table VI studies the evolution of the PostgreSQL 7.4.x branch (without passing –keepunused to CIL). First note the drastic differences between v7.3.21 and v7.4.0 which confirm that different development branches introduce new features that significantly change the structure of the application. The evolution to v7.4.0 adds 294 new datatypes (out of 300 total; 98.0%) and deletes 87 datatypes (out of 91; 95.6%). It causes the highest number of variable modifications among the entire 7.4.x branch. It results in 124 additions (out of 143; 86.7%), 88 deletions (out of 92; 95.7%), and 87 updates (out of 123; 70.7%). Significant changes are also observed in the evolution of functions among the entire branch with 940 new functions added (out of 1,004; 93.6%), 355 deleted (out of 359; 98.9%), and 2,792 functions updated (out of 5,007 in v7.4.0; 55.8%). Similar differences are also observed between versions v7.4.19 and v8.0.0 which belong to different development branches.

Comparison with vsFTPd. PostgreSQL is a much larger application than vsFTPd and it is more actively developed. PostgreSQL released a new version on average every 2.6 months (19 releases over 50 months) while vsFTPd released a new version on average every 5 months (13 releases over 66 months). If the source code evolution of v7.4.0 is excluded, an average of 105.5 functions are updated by each new version of PostgreSQL while vsFTPd updates on average 8.3 functions per new version. To systematically apply updates in a large application like PostgreSQL semantic analysis will be needed to automatically verify safety.

6.3.2 Experience

Due to the size of PostgreSQL, UpStare was used to apply only one update from v7.4.16 to v7.4.17. The aim was demonstrating that immediate updates (atomic and with bounded delay) could be applied in this application. Another aim was demonstrating that the approach and implementation are robust and can be applied to a large, real-world application.

The update use case considered for this application involved a client connecting to the server, authenticating correctly, waiting idle on the command-line and then applying the update. Under this use case the server has forked a connection handler that is blocked indefinitely on a recv() call waiting for client input through the network. The server is blocked indefinitely on an accept() call waiting for more incoming connections. Also note that the postmaster maintains a separate process for checkpointing its write-ahead log. This process is blocked in a select() call and is configured by default to awake every 5 minutes to produce the checkpoint.

A dynamic update was automatically prepared using the patch generator. v7.4.17 updated 64 functions and added one variable. No user-specified continuation mappings were needed for this update and it was manually verified that for the use case examined the update was semantically safe. User-specified mappings will probably be

needed to update from other update points (9931 update points where automatically inserted in v7.4.16).

UpStare applied an immediate update under this use case. An update request was issued to the parent process, which coordinated atomic stack reconstruction with the forked connection handler and the checkpointing process. The instrumentation forced all these processes to break out of their blocked system calls and apply the update immediately. A major advantage of UpStare compared to other DSU systems is that UpStare can apply this update without having to wait for all of the following three conditions to hold: (a) wait up to 5 minutes for the checkpointing process to wake-up, (b) wait for a new connection to arrive, and (c) wait for the client to issue a query.

Even though the particular update applied did not violate safety among multiple processes, an update to PostgreSQL outlines the need to apply immediate updates. Forked connection handlers in PostgreSQL communicate with each other through shared memory. A datatype update needs to be applied atomically among all forked connection handlers if type-safety is to be guaranteed (as described in Chapter 2.3).

The correctness of the instrumentation was tested using the PostgreSQL testsuite. The instrumented v7.4.16 and the update to v7.4.17 passed 85 (out of 93) tests of the testsuite, both in serial and parallel execution. For the remaining 8 testcases, MPatrol and Valgrind were used to verify 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 a 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 the implementation cannot be guaranteed to work in the presence of such bugs.

6.3.3 Performance

An updateable PostgreSQL v7.4.16 was compiled with gcc 4.1 on a 2.4Ghz Xeon and the performance of UpStare was measured when enabling various components of its instrumentation. The update applied by UpStare 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 experiment 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. This benchmark measured the time to run 100,000 transactions after a ramp-up time of 40,000 transactions. Table VII measures the throughput when the database is loaded both on hard-disk and in memory. Stack reconstruction

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 VII
PostgreSQL: Impact of Instrumentation on Throughput.

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, this experiment also measured the latency for establishing a connection and running only one transaction over the connection. The latency was measured by running a transaction 1000 times (1000 connections were established and torn down). Table VIII 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 again due to missed optimization opportunities by the compiler. Note that the overhead due to reconstruction is comparable to that of KissFFT. We speculate that this is due to the nature of the application (data-intensive). A performance measurement could not be obtained for Ginseng because Ginseng could not compile PostgreSQL but it is

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 VIII
PostgreSQL: Impact of Instrumentation on Latency.

expected that the data accesses through pointer indirection in Ginseng would result in high overhead.

The experiment also measured the latency and throughput when instrumented using dynamic stack tracing. Only the dynamic stack tracing implementation using the POSIX Threads API was measured. The dynamic stack tracing implementation using parameter passing failed to compile PostgreSQL because this implementation is not yet finished. The measurements were obtained at a later time and were unable to repeat the results of Table VIII for a direct comparison with the other instrumentations. The difference is that during this round of measurements the machine reports ∼0.06-0.10ms improvement. Similar to the measurements in vsFTPd, the overhead of multi-processing support is measured and subtracted from the overhead of stack tracing using the POSIX Threads API.

Table IX reports throughput of the dynamic stack tracing measurement. Stack tracing using the PThreads API is computed to have overhead of 12.3% from hard-disk and 7.1% from memory. Table X measures the latency of stack tracing using the PThreads API which is computed to be 119.2ms (779.2%) from hard-disk and 197.4ms (854.9%) from memory.

6.4 Conclusion

UpStare was evaluated on three applications. It was used to study in detail the overhead of stack reconstruction in the data-intensive KissFFT, to systematically apply 13 updates (5.5 years worth of updates) to vsFTPd, the very secure FTP daemon

PostgreSQL Configuration	pgbench throughput (t/s) 100,000 transactions			
	Hard-disk		Memory	
v7.4.16 - NonInstrumented	173.48		323.1	
v7.4.16 - MultiProcess	169.7	(2.2%)	320.4	(8.4%)
v7.4.16 - StackTracing: PThreads	148.39	(14.5%)	273.0	(15.5%)
v7.4.16 - StackTracing: ParameterPassing	-	-	-	-

TABLE IX
PostgreSQL: Impact of Dynamic Stack Tracing on Throughput.

PostgreSQL Configuration	pgbench latency (ms) Average of 1000 transactions			
	Hard-disk		Memory	
v7.4.16 - NonInstrumented	25.56		23.46	
v7.4.16 - MultiProcess	28.60	(11.9%)	26.26	(11.9%)
v7.4.16 - StackTracing: PThreads	227.77	(791.1%)	226.8	(866.8%)
v7.4.16 - StackTracing: ParameterPassing	-	-	-	-

TABLE X
PostgreSQL: Impact of Dynamic Stack Tracing on Latency.

(about 12,000 lines of code), and to apply one update to the PostgreSQL database management system (over 200,000 lines of code). It was also used to study the overhead of the instrumentation in applying multi-process updates, in offering bounded delay in the presence of blocking system calls, and in dynamically maintaining stack traces for dynamic safety checking.

The performance of UpStare was compared to Ginseng [32], which has been successful in applying DSU with a combination of updating mechanisms. For KissFFT, a data-intensive application, UpStare is 60%-140% faster because it eliminates data-access indirection. For vsFTPd v2.0.5, UpStare reports a 15% slower latency than Ginseng, but it reports zero overhead in the throughput of vsFTPd. It also reports 25% overhead in the throughput of PostgreSQL v7.4.16 (35% latency overhead; 89% worst-case), which is a data-intensive application Ginseng fails to compile.

Updates were applied under uses cases that are typical for each application and the updates were immediate. The updates were prepared automatically using the patch-generator, they needed minimal manual adjustments from the user, and they were manually verified for semantic safety for the use cases considered. For the 13 updates applied to vsFTPd, a total of 11 continuation mappings were manually

defined and this number could have been reduced with additional engineering effort. However it was also necessary to define more complex state transformations (22 lines of code total) for 4 updates to prepare the state of the new version as it would have been prepared had the application been started from the new version from the beginning. These more complex state transformations outline the need for automated analysis that verifies semantic safety of updates. Semantic analysis will be necessary to systematically apply updates to large applications, like PostgreSQL.

There are three limitations in the current implementation of UpStare that affect performance. First, the implementation can lead to missed optimization opportunities, and this can lead to high overhead in applications containing relatively long loop bodies that present such optimization opportunities. Second, UpStare offers function-indirection to allow implementing updating models other than the whole-program update mechanism. Function-indirection and the overhead it adds could be eliminated entirely. Third, the support for blocking system calls has not been optimized.

The performance of applying a streak of all updates was not studied. UpStare should be able to report good performance in this scenario because it is capable of reconstituting the entire program (both code and data) so that the application will no longer be spread in separate memory regions.

Two possible implementations of dynamic stack tracing were studied as a mechanism for providing dynamic safety checks. Dynamic stack tracing using the PThreads API outperforms tracing using parameter passing.

Chapter 7
CONCLUSION

The primary contributions of this dissertation are:

- A new dynamic *whole-program update* mechanism that guarantees *representation consistency*: before the update only old code is executed, after the update only new code is executed, and at no time does the application expect different representations of state (such as global variables or stack-frame contents). Also a new DSU mechanism of *stack reconstruction* that implements this update mechanism.
- *Immediate* updates (*atomic* and with *bounded delay*) for multi-threaded and multi-process applications.
- High updateability by allowing complex updates from all application states that are valid for update without interrupting the service provided by the application, given some user input.
- Dynamic software updates with *no data-access indirection*.
- A *dynamic stack tracing* mechanism for building context-sensitive program call graphs, implemented at a high-level (in source-code), to enforce user-supplied safety constraints.
- A practical DSU system implementation for multi-threaded and multi-process C applications. UpStare is the most flexible implementation of a DSU system to date and is as safe as existing DSU systems.

7.1 Future Work

This dissertation identified how the implementation of UpStare can be improved. These improvements are summarized next along with a description of future work that extends the design of UpStare.

The current UpStare implementation can be improved by:

- Transforming applications to allocate I/O data buffers on the heap instead of the stack, in support of blocking system calls. This will eliminate the need to copy data back to the stack when an I/O operation completes.

- Improving the identification (not the selection) of default execution continuation mappings to use strings instead of numeric ids. This will further minimize the input needed by a user in defining continuation mappings.
- Integrating support for automatically mapping pointers, which was developed in previous work, but without continuously tracking memory accesses. Computing through static analysis pointer variables that may be manipulated in unexpected ways and automatically producing transaction-safety constraints for them.
- Integrating support for multi-threaded applications that use counting semaphores, which was developed in previous work.
- Removing function-pointer indirection, since it is not needed for stack reconstruction.
- Reconstituting the entire code of the new version of the applications, instead of continuing to execute old versions of functions that did not need to be updated. This will improve spatial locality of code and data, which should offer good performance after a streak of updates.

Future work includes exploring semantic safety analysis, defining an update description meta-language, applying updates of in-transit data, and applying updates of files. The aim is to reduce the input required by the user in applying DSU and to improve updateability.

Semantic safety analysis will be necessary to systematically apply updates to large applications. It would be beneficial to explore semantic analysis to minimize the amount of manual semantic safety verification that is currently needed by a user, and to minimize the amount of manual state mappings a user needs to provide. One direction is to identify the nature of most updates, classify them, and prove that for these common cases the updates are semantically safe. This would allow a semantic analyzer to automatically verify semantic safety for the majority of updates, thus reducing the verification that the user must manually carry out. Another direction is to leverage program slicing to automatically produce transformations of complex state that cannot be easily automated with a patch-generator. These transformations require preparing program state as it would have been prepared had the application been started from the new version from the beginning. Producing these transformations automatically would reduce the input needed by the user in defining state mappings in large applications.

Semantic analysis can also benefit from user annotations created when application source code is modified. It is possible to define an update description meta-language that can be integrated in the software development process. Through this language a

programmer can provide hints for DSU, such as marking a code change as semantics-preserving, or a field addition as changing the semantics of existing fields.

In-transit data, such as data transmitted through sockets, shared memory or pipes, cannot be updated because the data reside inside the network or the operating system. One way to handle them is to install data transformers at the reception end-points and automatically transform the data to the new representation. Another approach is to consider both the client and the server in the update scope. An update can be applied only after it is detected that no data are still in-transit. This can be implemented by pausing all clients, transmitting a special marker message through each socket, shared memory region, or pipe, and applying the update after all marker messages are received at the other end.

Another open problem is updating files on disk. Applying a data transformation in an atomic step in a large data file (such as the persistent storage of PostgreSQL) would require a long time to complete. Although the update would be applied with bounded delay, a large delay would essentially result in downtime. Since it is desirable for an update to not interrupt the service provided by an application, it is possible to install data transformers (similar to handling in-transit data) at the endpoints where data are accessed (such as read(),write()), apply an update that does not transform data files, and resume application execution. The transformers installed can be temporary. A low-priority datatype transformation thread can be converting the file permanently to use the new data representation. Write requests that include new datatype fields can be preserved in a separate temporary file and consulted by the transformation thread, until they are eventually coalesced into the new representation.

7.2 Program Slicing

Program slicing [121, 122, 123, 124] has the potential of solving the *update safety* problem (see Chapter 2.1). A program slice consists of all statements of a program that might affect the value of a variable x at a point p. Program slicing can be used to identify semantic differences [43, 125], and semantic equivalence [126], between two versions of a program and integrate [127, 128] the two versions together. We expect program slicing will be pivotal in applying semantically safe DSU.

Program slices can be computed statically [129, 130] or dynamically [131, 45, 46]. They are valuable in debugging [132], testing [133, 134], program integration [135], procedure extraction [136], and software engineering and maintenance [137, 44, 46].

REFERENCES

[1] David Oppenheimer, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A. Patterson, and Kathy Yelick, "Roc-1: Hardware support for recovery-oriented computing," *IEEE Transactions on Computers*, vol. 51, pp. 2002, 2002. 1

[2] Steve Parker, "A simple equation: IT on = Business on," *The IT Journal. Hewlett Packard*, 2001. 1

[3] Meta Group, "IT Performance Engineering and Measurement Strategies: Quantifying Performance and Loss," October 2000. 1

[4] Cisco Systems, "Enterprise Continuance Strategies for the Cisco ONS 15500 Series: Multiservice DWDM Aggregation and Transport Platforms," 2008. 1

[5] Contigency Planning & Management magazine and KPMG, "Business Continuity Study: A review of the factors influencing Business Continuity in the next millenium," Tech. Rep., March 2002. 1

[6] Computer Weekly, "Downtime hits ambulance service," August 2006. 1

[7] Nancy C. Nelson, "Downtime procedures for a clinical information system: a critical issue," *Journal of Critical Care*, vol. 22, no. 1, March 2007. 1

[8] BBC News, "Massive air disruption across UK," June 2004. 1

[9] Woody Baird, "An air traffic control failure is examined," *USA Today*, October 2007. 1

[10] Rachel Stevenson, "Flight delays continue after air traffic control failure," *The Guardian*, September 2008. 1

[11] Mikael Ronstrom, "Database requirement analysis for a third generation mobile telecom system," in *Databases in Telecommunications*, 1999, pp. 90–105. 1

[12] Andy Cress, "Linux Clustering Software for Telecom," *The Telecom System View*, vol. 1.5, no. 14, January 2004. 1

[13] Jim Byrnes and Ruthlyn Newell, "AT&T announces cause of frame relay network outage," *Risk Digest*, vol. 19, no. 72, May 1998. 1

[14] Alex Berenson, "Software failure halts Big Board trading for over an hour," *New York Times*, March 2001. 1

[15] Leo King, "London Stock Exchange confirms network software caused costly outage," *Computerworld UK*, September 2008. 1

[16] Brian Krebbs, "Cyber Incident Blamed for Nuclear Power Plant Shutdown," *Washington Post*, June 5 2005. 1

[17] Jack J. Woehr, "Really remote debugging for real-time systems: A Conversation with Glenn Reeves," *Dr. Dobb's Journal*, November 1999. 1

[18] Alexander C. Calder, "Flight Software Design and On-Orbit Maintenance," Tech. Rep., Computer Sciences Corporation, November 2007. 1

[19] StackSafe, "IT Ops Research Report: Downtime and Other Top Concerns," July 2007. 1

[20] StackSafe, "IT Ops Research Report: Six Out of 10 Organizations Cite Application Changes as Leading Cause of Downtime," July 2007. 1

[21] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Tech. Rep. Planning Report 02-03, May 2002. 1

[22] David E. Lowell, Yasushi Saito, and Eileen J. Samberg, "Devirtualizable virtual machines: Enabling general, single-node, online maintenance," in *ASPLOS*, 2004. 1

[23] Deepak Gupta, Pankaj Jalote, and Gautam Barua, "A formal framework for online software version change," *Software Engineering*, vol. 22, no. 2, pp. 120–131, 1996. 2, 6, 15, 25, 27, 33

[24] I. Lee., *DYMOS: A Dynamic Modification System*, Ph.D. thesis, University of Wisconsin, Department of Computer Science, Madison, April 1983. 2, 19, 26, 27, 33

[25] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles, "Dynamic software updating," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 13–23. 2

[26] Gautam Altekar and Ilya Bagrak and Paul Burstein and Andrew Schultz, "OPUS: Online Patches and Updates for Security," in *14th USENIX Security Symposium*, July 2005, pp. 287–302. 2, 26, 27, 31, 33, 58

[27] Janghoon Lyu, Youngjin Kim, Yongsub Kim, and Inhwan Lee, "A procedure-based dynamic software update," in *DSN*, 2001, pp. 271–284. 2, 19, 26, 27

[28] 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*, Washington, DC, USA, 2007, pp. 271–281, IEEE Computer Society. 2, 19, 20, 26, 27, 29, 33, 46

[29] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams, *Concurrent Programming in ERLANG (Second Edition)*, Prentice Hall, 1996. 2, 19, 27, 33

[30] Dominic Duggan, "Type-based hot swapping of running modules," in *International Conference on Functional Programming*, 2001, pp. 62–73. 2, 8, 20

[31] Chandrasekhar Boyapati and Rasekhar Boyapati and Barbara Liskov and Liuba Shrira and Chuang-hue Moh and Steven Richman, "Lazy Modular Upgrades in Persistent Object Stores," in *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, 2003, pp. 403–417. 2, 33

[32] 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. 2, 3, 19, 20, 27, 33, 46, 66, 80, 81, 89

[33] Kristis Makris and Kyung Dong Ryu, "Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels," in *EuroSys 2007*, March 2007. 2, 19, 20, 25, 27, 33, 58

[34] Iulian Neamtiu, *Practical Dynamic Software Updating*, Ph.D. thesis, University of Maryland, August 2008. 2, 8, 27, 80, 82

[35] 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, USENIX Association. 2, 19, 24, 33

[36] Iulian Neamtiu and Michael Hicks, "Safe and timely dynamic updates for multi-threaded programs," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2009. 2, 19, 20, 27, 33

[37] Jeff Arnold and M. Frans Kaashoek, "KSplice: Automatic Rebootless Kernel Updates," in *EuroSys 2009*, April 2009. 2, 19, 20, 25, 27, 33

[38] 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. 3, 8, 20, 27, 33, 57

[39] 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)*, Jan. 2008, pp. 37–50. 3, 8, 20, 27, 33, 57

[40] Robert Love, "Lowering latency in linux: introducing a preemptible kernel," *Linux Journal*, vol. 2002, no. 97, pp. 1, 2002. 4

[41] Dipankar Sarma and Paul E. McKenney, "Making RCU safe for deep sub-millisecond response realtime applications," in *In Proceedings of the 2004 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004, pp. 32–32, USENIX Association. 4

[42] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni, "Read-copy update," in *Ottawa Linux Symposium*, 2001, pp. 338–367. 4

[43] Susan Horwitz, "Identifying the semantic and textual differences between two versions of a program," *SIGPLAN Notices*, vol. 25, no. 6, pp. 234–245, June 1990. 7, 93

[44] Susan Horwitz and Thomas W. Reps, "The use of program dependence graphs in software engineering," in *International Conference on Software Engineering*, 1992, pp. 392–411. 7, 93

[45] Hiralal Agrawal and Joseph R. Horgan, "Dynamic program slicing," in *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, 1990. 7, 93

[46] Árpád Beszédes, Tamas Gergely, Zsolt Mihaly Szabo, Janos Csirik, and Tibor Gyimothy, "Dynamic Slicing Method for Maintenance of Large C Programs," in *5th European Conference on Software Maintenance and Reengineering (CSMR)*, 2001, pp. 105–113. 7, 93

[47] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou, "Efficient online validation with delta execution," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2009, pp. 193–204, ACM. 7

[48] Sameer Ajmani, *Automatic Software Upgrades for Distributed Systems*, Ph.D. thesis, MIT, September 2004. 18, 28

[49] David E. Lowell, Yasushi Saito, and Eileen J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," in *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, New York, NY, USA, 2004, pp. 211–223, ACM. 18, 29

[50] Shaya Potter and Jason Nieh, "Reducing downtime due to system maintenance and upgrades," in *LISA '05: Proceedings of the 19th conference on Large Installation System Administration Conference*, Berkeley, CA, USA, 2005, pp. 6–6, USENIX Association. 18, 31

[51] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille, "Dynamic program instrumentation for scalable performance tools," *1994 Scalable High Performance Computing*, May 1994. 18, 24, 33

[52] 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. 18, 19, 24, 33

[53] David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder, "GILK: A Dynamic Instrumentation Tool for the Linux Kernel," in *Computer Performance Evaluation / TOOLS*, 2002, pp. 220–226. 18, 19, 24, 33

[54] J. Maebe, M. Ronsse, and K. De Bosschere, "Diota: Dynamic instrumentation, optimization and transformation of applications," in *Compendium of Workshops and Tutorials held in conjuction with PACT '02*, 2002. 19, 24

[55] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere, "Instrumenting JVMs at the machine code level," in *3rd PA3CT-symposium*, September 2003. 19

[56] Jonas Maebe and Koen De Bosschere, "Instrumenting self-modifying code," in *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, September 2003. 19

[57] Galen Hunt and Doug Brubacher, "Detours: Binary Interception of Win32 Functions," in *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999, pp. 135–143. 19, 26, 33

[58] Amitabh Srivastava, Andrew Edwards, and Hoi Vo, "Vulcan: Binary transformation in a distributed environment," Tech. Rep. MSR-TR-2001-50, Microsoft Research, April 2001. 19, 26, 33

[59] Michael Hicks, *Dynamic Software Updating*, Ph.D. thesis, University of Pennsylvania, August 2001. 19, 20, 27

[60] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, "Dealing with disaster: Surviving misbehaved kernel extensions," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, Washington, 1996, pp. 213–227. 19, 23

[61] H.Chen, R. Chen, F.Zhang, B.Zhang, and P-C. Yew, "Live Updating Operating Systems Using Virtualization," in *VEE*, 2006. 19, 29, 33

[62] Amit Shanhbhag, "Portable cross-version checkpointing and recovery for dynamic software updates," M.S. thesis, Arizona State University, August 2003. 21, 31, 32, 33

[63] Kalyan S. Perumalla and Richard M. Fujimoto, "Efficient large-scale process-oriented parallel simulations," in *WSC '98: Proceedings of the 30th conference on Winter simulation*, Los Alamitos, CA, USA, 1998, pp. 459–466, IEEE Computer Society Press. 21, 31, 33

[64] H. Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services*, Ph.D. thesis, Columbia University, 1992. 23

[65] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole, "Fast concurrent dynamic linking for an adaptive operating system," 1996. 23

[66] Seltzer M., Endo Y., Small C., and Smith K., "An Introduction to the Architecture of the VINO Kernel," Tech. Rep. 34-94, Harvard University, Computer Science, 1994. 23

[67] M. I. Seltzer and C. Small, "Self-monitoring and self-adapting operating systems," *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997. 23

[68] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Symposium on Operating Systems Principles*, 1995, pp. 251–266. 23

[69] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, "Extensibility, safety and performance in the SPIN operating system," in *15th Symposium on Operating Systems Principles*, Copper Mountain, Colorado, 1995, pp. 267–284. 23

[70] Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, and Orran Krieger, "System support for online reconfiguration.," in *Proceedings of the 2003 USENIX Technical Conference*. 2003, pp. 141–154, USENIX Association. 24

[71] Aaron J. Goldberg and John L. Hennessy, "Performance debugging shared memory multiprocessor programs with mtool," in *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1991, pp. 481–490, ACM. 24

[72] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," in *ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*. June 1994, ACM SIGPLAN. 24

[73] J.R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," in *ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*. June 1995, ACM SIGPLAN. 24

[74] Thomas Ball and James R. Larus, "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 4, pp. 1319–1360, July 1994. 24

[75] B.M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004. 24

[76] Ronald G. Minnich, "A dynamic kernel modifier for Linux," in *Proceedings of the LACSI Symposium*, September 2002. 24

[77] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, "Dynamo: a transparent dynamic optimization system," in *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, New York, NY, USA, 2000, pp. 1–12, ACM. 24

[78] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI 2005*, June 2005. 24

[79] Deepak Gupta and Pankaj Jalote, "On-line software version change using state transfer between processes," *Software - Practice and Experience*, vol. 23, no. 9, pp. 949–964, 1993. 26

[80] Michael Hicks, Jonathan T. Moore, and Scott Nettles, "Dynamic software updating," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. June 2001, pp. 13–23, ACM. 27, 33

[81] Yueh-Feng Lee and Ruei-Chuan Chang, "Hotswapping linux kernel modules," *Journal of Systems and Software*, vol. 79, no. 2, pp. 163–175, February 2006. 27

[82] Bettina Kemme and Gustavo Alonso, "Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 2000, pp. 134–143, Morgan Kaufmann Publishers Inc. 28

[83] Sameer Ajmani and Barbara Liskov and Liuba Shrira, "Modular Software Upgrades for Distributed Systems," in *European Conference on Object-Oriented Programming (ECOOP)*, July 2006. 28

[84] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley, "Dynamic Software Updates: A VM-centric Approach," June 2009. 29

[85] Bowen Alpern and C. Richard Attanasio and John J. Barton and Anthony Cocchi and Susan Flynn Hummel and Derek Lieber and Ton Ngo and Mark F. Mergen and Janice C. Shepherd and Stephen E. Smith, "Implementing Jalapeño in Java," in *OOPSLA*, 1999, pp. 314–324. 29

[86] Stephen J. Fink and Feng Qian, "Design, implementation and evaluation of adaptive recompilation with on-stack replacement," in *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, 2003, pp. 241–252, IEEE Computer Society. 29, 31

[87] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes, "Runtime support for type-safe dynamic java classes," in *In Proceedings of the Fourteenth European Conference on Object-Oriented Programming*. 2000, pp. 337–361, Springer-Verlag. 29

[88] Tobias Ritzau Linkping and Tobias Ritzau, "Dynamic deployment of java applications," in *In Java for Embedded Systems Workshop*, 2000. 29

[89] Alessandro Orso, Anup Rao, and Mary Jean Harrold, "A Technique for Dynamic Updating of Java Software," in *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002. 29

[90] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, "Libckpt: Transparent Checkpointing under Unix," in *Proceedings of USENIX Winter1995 Technical Conference*, New Orleans, Louisiana/U.S.A., Jan. 1995, pp. 213–224. 30

[91] Richard Koo and Sam Toueg, "Checkpointing and rollback-recovery for distributed systems," in *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, Los Alamitos, CA, USA, 1986, pp. 1150–1158, IEEE Computer Society Press. 30

[92] Georg Stellner, "CoCheck: Checkpointing and Process Migration for MPI," in *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996. 30

[93] Barak Amnon and La'adan Oren, "The MOSIX Multicomputer Operating System for High Performance Cluster Computing.," *Journal of Future Generation Computer Systems*, vol. 13, no. 4-5, pp. 361–372, March 1998. 30

[94] Balkrishna Ramkumar and Volker Strumpen, "Portable checkpointing for heterogeneous architectures," in *In Symposium on Fault-Tolerant Computing*. 1997, pp. 58–67, Kluwer Academic Press. 30, 33

[95] Kuo-Feng Ssu and W. Kent Fuchs, "PREACHES - portable recovery and checkpointing in heterogeneous systems," in *Symposium on Fault-Tolerant Computing*, 1998, pp. 38–47. 30

[96] Feras Karablieh and Rida Bazzi, "Heterogeneous Checkpointing for Multithreaded Applications," in *21st Symposium on Reliable Distributed Systems (SRDS)*, October 2002. 30, 52, 54

[97] Feras Karablieh, Rida Bazzi, and Margaret Hicks, "Compiler-Assisted Heterogeneous Checkpointing," in *20th Symposium on Reliable Distributed Systems (SRDS)*, October 2001. 30, 31, 33, 45

[98] Stefan Fünfrocken, "Transparent migration of java-based mobile agents - capturing and reestablishing the state of java programs," in *In Mobile Agents*. 1998, pp. 26–37, Springer-Verlag. 30, 31, 32, 33

[99] Jason Nieh, "Autopod: Unscheduled system updates with zero data loss," in *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, Washington, DC, USA, 2005, pp. 367–368, IEEE Computer Society. 31

[100] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh, "The design and implementation of zap: a system for migrating computing environments," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, New York, NY, USA, 2002, pp. 361–376, ACM. 31

[101] Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992. 31

[102] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand, "Continuations and coroutines," in *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, New York, NY, USA, 1984, pp. 293–298, ACM. 31

[103] Bruce F. Duba, Robert Harper, and David Macqueen, "Typing first-class continuations in ml," in *Journal of Functional Programming*, 1991, pp. 163–173. 31

[104] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," in *In Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991, pp. 122–136. 31

[105] Tatsurou Sekiguchi, Takahiro Sakamoto, and Akinori Yonezawa, "Portable Implementation of Continuation Operators in Imperative Languages by Exception Handling," in *Lecture Notes in Computer Science*. 2001, pp. 217–233, Springer-Verlag. 31, 32, 33

[106] Atul Adya and Jon Howell and Marvin Theimer and William J. Bolosky and John R. Douceur, "Cooperative task management without manual stack management or, event-driven programming is not the opposite of threaded programming," in *In Proceedings of the 2002 Usenix ATC*, 2002. 31

[107] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek, "Events can make sense," in *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007. 31

[108] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser, "Dingo: Taming device drivers," in *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, Apr 2009. 31

[109] C. J. M. Booth and D. I. Bruce, "Stack-free process-oriented simulation," in *PADS '97: Proceedings of the eleventh workshop on Parallel and distributed simulation*, Washington, DC, USA, 1997, pp. 182–185, IEEE Computer Society. 32

[110] Edward Mascarenhas and Vernon Rego, "Ariadne: Architecture of a portable threads system supporting mobile processes," *Software-Practice and Experience*, vol. 26, 1996. 32

[111] Jean-Sébastien Légaré, "Providing Continuations in Java via Source Code Transformations," December 2006, Department of Computer Science, University of British Columbia. 32

[112] Don Stewart and Manuel M. T. Chakravarty, "Dynamic applications from the ground up," in *Proceedings of the ACM SIGPLAN Workshop on Haskell*. September 2005, ACM Press. 32, 33

[113] Pierre Duquesne and Ciarán Bryce, "A language model for dynamic code updating," in *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008, pp. 1–5, ACM. 32

[114] Jérémy Buisson and Fabien Dagnat, "Introspecting continuations in order to update active code," in *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, New York, NY, USA, 2008, pp. 1–5, ACM. 32

[115] Ophir Frieder and Mark E. Segal, "On dynamically updating a computer program: from concept to prototype," *Journal of Systems Software*, vol. 14, no. 2, pp. 111–128, 1991. 33

[116] Gísli Hjálmtýsson and Robert Gray, "Dynamic C++ classes: a lightweight mechanism to update code in a running program," in *Proceedings of the 1998 USENIX Technical Conference*, Berkeley, CA, USA, 1998, pp. 6–6, USENIX Association. 33

[117] Theodore C. Goldstein and Alan D. Sloane, "The Object Binary Interface–C++ Objects for Evolvable Shared Class Libraries," Tech. Rep., 1994. 33

[118] 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. 36

[119] Feras Karablieh, *Compiler Assisted Application-Level Fault Tolerance in Distributed Systems*, Ph.D. thesis, Arizona State University, May 2005. 45

[120] Kristis Makris and Rida Bazzi, "Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction," in *Proceedings of the USENIX '09 Annual Technical Conference*, June 2009. 76

[121] Mark Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, March 1981, pp. 439–449. 93

[122] Frank Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995. 93

[123] Mariam Kamkar, "An overview and comparative classification of program slicing techniques," *Journal of Systems and Software*, vol. 31, no. 3, pp. 197–214, 1995. 93

[124] David Binkley and Keith Brian Gallagher, "Program slicing," in *Advances in Computers*, 1996, vol. 43, pp. 1–50. 93

[125] Wuu Yang, Susan Horwitz, and Thomas Reps, "A program integration algorithm that accommodates semantics-preserving transformations," *SIGSOFT Softw. Eng. Notes*, vol. 15, no. 6, pp. 133–143, 1990. 93

[126] Wuu Yang, Susan Horwitz, and Thomas Reps, "Detecting program components with equivalent behaviors," Tech. Rep. CS-TR-1989-840, 1989. 93

[127] Wuu Yang, *A new algorithm for semantics-based program integration*, Ph.D. thesis, University of Wisconsin, 1990, Supervisor-Thomas Reps and Supervisor-Susan Horwitz. 93

[128] David Wendell Binkley, *Multi-procedure program integration*, Ph.D. thesis, University of Wisconsin, Madison, WI, USA, 1991. 93

[129] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987. 93

[130] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, New York, NY, USA, 1988, pp. 35–46, ACM. 93

[131] Bogdan Korel and Janusz W. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, 1988. 93

[132] Bogdan Korel and Jurgen Rilling, "Application of dynamic slicing in program debugging," in *In Mariam Kamkar, editor, Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG*, 1997, pp. 43–57. 93

[133] Mark Harman and Sebastian Danicic, "Using Program Slicing to Simplify Testing," *Journal of Software Testing, Verification and Reliability*, vol. 5, no. 3, pp. 143–162, 1995. 93

[134] David Binkley, "The Application of Program Slicing to Regression Testing," in *Information and Software Technology Special Issue on Program Slicing*. 1999, pp. 583–594, Elsevier. 93

[135] David Binkley, *Multi-procedure program integration.*, Ph.D. thesis, Computer Sciences Department, University of Wisconsin, Madison, August 1991. 93

[136] Raghavan Varadhan Komondoor, *Automated duplicated code detection and procedure extraction*, Ph.D. thesis, University of Wisconsin, 2003. 93

[137] K. B. Gallagher and J. R. Lyle, "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pp. 751–761, 1991. 93