# Portable Software Update: A Framework for Dynamic Software Updates in C-Language Programs

by

Marcus Karpoff

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

# Abstract

A Dynamic Software Update (DSU) system enables an operator to modify a program without interrupting the execution of the program. Programs written in certain programming language, in particular programs written in C, are difficult to modify while they are executing. This thesis presents Portable Software Update (PSU), a new framework for the creation of C-language DSU programs. PSU offers a simple program interface to build DSU versions of existing programs. Once a program is built using PSU, updates can be applied using background threads that do have negligible impact on the execution of the program. PSU supports multi-threaded and recursive programs without the use of *safe-points* or thread blocking.

PSU uses function indirection to redirect DSU functions calls to the newest version of the function code. DSU functions in a PSU program execute to completion with new calls to DSU functions always executing the newest version. This simple mechanism allows PSU to load updates rapidly. PSU borrows automatic memory management techniques from garbage collection systems to unload obsolete version of DSU functions after they are no longer in use. PSU is the first DSU system for C-language programs which is able to unload older versions of code. This use of resources enables many patches to be applied to a long-running application.

The overhead of the DSU-enabling features is evaluated using a series of custom synthetic micro benchmarks and of a DSU-enabled version of the MySQL database storage engine. The MySQL storage engine maintains 95%

of the performance of the non-DSU version while allowing the entire storage engine to be updated while the database continues executing. The process of modifying the storage engine to enable DSU is simple and straightforward.

# Preface

The design of the Portable Software Update framework has been filed as a collaborative work by Kia Ting Wang, Brice Dobry, Marcus Karpoff, and José Nelson Amaral for patent with the United States Patent and Trademark Office with application number 15861132 and patent ID 85465103. The application's The current listed status of the application *to be filed* with the Patent Cooperation Treaty, with patent ID 85465112.

*To Jordan*

*For supporting me from the very beginning and every day after.*

*Always and Always*

*Naught Without Labour*

— Greek Proverb

# Acknowledgements

First I would like to thank my supervisor, Dr. José Nelson Amaral, for his constant guidance. If not for his inspiration and encouragement, I likely would have not even started this journey let alone finished it. I've learned more from you than the rest of my education combined.

Thank you to Jordan for constantly supporting me and helping me to be the best I can be. I've weathered many tough challenges, that would have been impossible alone, because of you.

Thank you to my parents for raising me with the mentality that education is a critical part of my life. There was never a doubt that I would go to university, just debate on what I would study.

Thank you to my fellow lab members who were always willing to help with any challenges I was stuck on. Thank you whether you were a rubber duck or a full blown debugger.

Thank you to (Amy) Kai Ting Wang, Brice Dobry, Rayson Ho, and Peng Wu for the feedback and advice over the course of this project. Your experience helped me get farther into this project than would have been possible if I had restrict my studies to the academia.

Thank you to the Huawei Compiler Research Team for the resources you provided me. Tens of thousands of hours were spent running benchmarks on Huawei servers.

Thank you to Mitacs Accelerate Program for funding, in part, this work.

# Contents

# List of Tables

# List of Figures

# Listings

# Glossary

**Dynamically Linked Library (DLL)** Dynamically Linked Libraries are libraries whose contents are loaded at program startup. Also known as Shared Libaries. 11–13, 15, 61

**Dynamic Software Update (DSU)** The process of modifying a program at runtime. ii, iii, xi–xiv, 1–5, 7–20, 22, 24–66

**Hardware Transactional Memory (HTM)** A memory concurrency technique which implements transactional memory using specialized hardware. 23, 24

**Procedure Linkage Table (PLT)** A table of function pointers for dynamically linked functions that are initialized at runtime. 11–14

**Software Transactional Memory (STM)** A memory concurrency technique which implements transactional memory using software structures. 23, 24

**Thread-Local Storage (TLS)** "Thread-Local Storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread" [6]. 17, 24–26, 33, 39, 46, 64

**Transactions Per Minute (TPM)** The number of transactions a database can perform in a minute. 52

**Micro Service** A subsystem of a larger system responsible for a specific task in the larger systems overarching goal. 3

**Micro-Service Architecture** "The micro-service architecture is an approach to developing an application as a set of small independent services." [14]. 3

**Monolithic Architecture** A design principle that relies on highly coupled components in a single large application. 3

**MySQL** A SQL database with an open source codebase. ii, xi, 2, 39, 51–54, 66

**Safe-Point** A location in a DSU application where a thread may be executing code while an update may occure without causing errors. ii, 5, 7–9, 55–58

**TPC-C** A benchmark used to evaluate the performance of a databases. 52

# PSU Specific Glossary

**Dynamic Function Versioning Table (DFVT)** A Table in the *update supporting library* containing all versions of PSU functions. 16–19, 31

**Portable Software Update (PSU)** PSU Framework for building DSU applications in C. ii, xi, xiv, 2–6, 14–18, 21, 25–27, 31–39, 41, 46, 47, 49–51, 53–66

**Application Program** An application that a user wants to make dynamically updatable. xiv, 15, 17–19, 21, 27, 31, 33–36, 39, 48, 50, 51, 56, 59

**Memory Reclamation Thread** A thread in the update supporting library responsible for unloading obsolete DSU code without interfering with the *application program*. 16, 17, 19, 26

**Patch Loader Thread** A thread in the update supporting library responsible for loading new DSU code. 16, 17, 27, 28, 30

**PSU Function Counter** A custom counter for reference counting DSU function in PSU. 16–19, 21, 25–27, 30, 32, 33, 35, 38, 39, 46, 57, 64

**PSU Function List** A List of functions that the update supporting library should be able to update at runtime. 31, 32, 53

**PSU Logging System** A logging system in the update supporting library used to provide information about the state of a PSU application. 17, 48, 50

**PSU Patch Id** A Patch Identifier used by the update supporting library to report which patches have been loaded and unloaded. 18, 32, 33, 53

**PSU Thread Intercept** A redefinition of the thread creation function that allows PSU to control how threads are launched. 25

**Update Supporting Library** A dynamically linked library that contains the runtime components of the PSU framework. xiv, 15–18, 20, 26, 31, 32, 34–36, 39, 41, 48–50, 56, 64

**Updating Weaver** An automated code transformation tool built to make PSU applications easier to write. 34–36, 53, 64, 66

# Chapter 1

# Introduction

Software systems are continuously evolving. Often new features are added and bugs are fixed in programs after they are deployed for use. This fluidity is even more prevalent in recent years as software has rapidly transitioned from local applications to online services [2]. Individual applications running on a personal device may be stopped and relaunched regularly causing impact to a single user. However, when an online service is stopped and relaunched all users connected to that service are affected. Downtime of an online service has significant impact for users of the service. Moreover, in several industries that rely on information technology some software applications are mission critical. Updating such applications requires a very careful orchestration between an existing running version and a new version, which must be warmed up and prepped to run immediately upon switch over. Such switch-overs can be both complex and risky, leading operators to be very conservative about installing upgrades. In other situations online services are down for a period of time on a regular basis to allow regularly scheduled upgrades. Dynamic Software Update (DSU) systems provide a way to update programs while they are executing. DSU allows operators to load fixes to bugs in the software, apply security patches, and add new features without stopping the execution of the applications.

Cérin, Coti, Delort, *et al.* show that the downtime of online services, even for short periods of time, can cost significant amounts of money to the service providers[3]. Between 2007 and 2014 these downtimes are estimated to have

cost almost 670 million USD for just 38 services. Unplanned downtime due to failures, bugs, or security violations cannot be predicted. However, the data from Cérin, Coti, Delort, *et al.* shows a need for a way to also mitigate planned downtime to online services. DSU systems allow these services to avoid planned downtime by updating the service's software without requiring software restarts. Moreover, with the ability to upgrade applications while they are running, DSU enables operators to fix issues immediately and, thus, potentially avoid unplanned downtime occurrences.

Another issue with service downtime is the loss of program state from prior to the program's shutdown. Many programs rely on caching commonly used variables in memory to improve performance. An example of this is MySQL's query caching system. MySQL databases employ a method of saving commonly executed queries in a cache memory. This system significantly improves the performance of the database by avoiding the need to access disk memory. To initialize these caches MySQL employs a *warm-up* period where the database loads the common queries into the cache. Park, Do, Teletia, *et al.* show that until a database's cache has been properly warmed up, the database may only perform at a fraction of it's peak performance [19]. Park, Do, Teletia, *et al.* also discuss the use of query caches in other databases. Enabling DSU in a database application allows these databases to maintain their application state, such as the query cache, during an update. The alternative to DSU is the current state of the art where the updating of a mission-critical database application requires a complex warming up of the new application followed by a switch over.

This thesis presents Portable Software Update (PSU), a framework for building portable DSU applications in C with minimal impact to memory consumption and performance while adding minimal complexity. Kemerer shows that as the complexity of software increases the cost of maintaining that software also increases [12]. This phenomenon is easily understood as the number of bugs due to programmer error increases with the complexity of the software. A core principle in PSU's design is to minimize the increase in complexity the programmers must manage while providing the functionality

of a DSU system. The design of PSU is explained in detail in Chapter 4.

A recent trend in software systems is to favour *micro-service architecture* over larger *monolithic architecture*. In large monolithic architecture systems, all components of an application are built into a single program. This is the inverse of micro-service architecture systems. In micro-service architectures the application is composed of several sub-components called *micro services*. Each of these micro services are responsible for a singular task in the overarching system. These micro services communicate with each other through strictly defined interfaces. This approach allows the components to have independent task scopes, drastically reducing the complexity of the overall application. This reduced complexity helps reduce the likelihood of software issues due to programmer error.

The use of a micro-service architecture methodology in the core design of PSU is reflected in the encapsulation of all components of DSU inside a separate runtime library that is responsible for the loading and unloading of all updates. By encapsulating this functionality, programmers who use PSU are able to gain the benefits of PSU without having any need to understand how the update happens. Section 4.2 explains the design details of this runtime library.

The micro-service architecture design of PSU requires a communication structure that enables the runtime library to encapsulate all requisite functionality. The specifications of this communication is demonstrated in Section 4.3.

To reduce the complexity of PSU's interface further, PSU has a specialized C-source-to-C-source compiler that is designed to add the necessary library interface calls to a program. This compiler provides programmers the ability simply to specify which portions of the application are DSU without having to consider how to make these portions DSU. The details on this specialized compiler are detailed in Section 4.4.

PSU uses system interfaces that are either operating-system agnostic or have synonymous libraries in most operating systems. Basing the implementation of PSU on common libraries allows programmers to use PSU without concerns for cross system compatibilities. The prototype for PSU is imple-

mented for POSIX systems using the *pthread* and *dlfcn* libraries. Only the functionalities in these libraries, which are also readily available in other systems, are used.

PSU is designed to be a near full application DSU system. A full application DSU system is able to update any component of the application. Full application systems are very versatile because they can be applied to any application. These systems are extremely difficult to build because they require the system to be able to modify components of the application that might be currently executing. This complexity led several previous systems to suffer from race conditions and deadlocks [5], [13], [24].

PSU avoids the complexity of full application DSU systems by only applying updates to components that are specified to be dynamically updatable. Programmers that use PSU specify which functions they desire to be updatable. In PSU, after an update, only future invocations to the updated functions can use the updated code. Once an instance of a function is invoked, it must run to an exit point using the same version of the code that was loaded in the system when it was invoked. Moreover, all callees of a DSU function in PSU also run using the version that was current when the caller was invoked. This constraint allows a single DSU function to update an entire section of the program's call tree. Section 4.3.3 examines this interaction in more detail.

The performance impact of PSU is measured using a series of custom synthetic programs that were created with the specific purpose of exposing each of the aspects of PSU performance. By design, PSU applications must perform more work than a non-DSU system. This extra work is necessary to allow the applications to update during runtime. The synthetic programs measure these overheads. Individual overheads may appear significant when examined in isolation but in a realistic use case the increase in work that a PSU application must perform is insignificant in comparison with a non-PSU application. Chapter 5 explains the design of each benchmark and discusses their results in detail.

# Chapter 2

# Dimensions of Dynamic Software Update

There are multiple approaches to introduce DSU to an application. DSU systems may be characterized as points in a multi-dimensional space where the dimensions are: (1) multi-threading capability, (2) ability to handle recursion, (3) dynamic-update granularity, (4) abstraction layer, (5) blocking requirement, (6) *safe-point* requirement, (7) ability to maintain multiple versions running at the same time, (8) and requirement for regeneration of binaries to enable DSU.

Table 2.1 lists these dimensions and lists several other features of common DSU systems as well as PSU. The following sections will discuss and explain each dimension. The remaining features as well as the different implementations will be discussed in more detail in the Related Work in Chapter 6.

## 2.1 Multi-Threading Capability

DSU updates may be applicable to single-threaded programs or to multi-threaded programs. In multi-threaded programs an important issue is the coordination of the execution of the multiple threads with the updating. This issue is more relevant when multiple threads are executing the same source code that is subject to the updating [17]. The multi-threading capability interacts with several of the other design dimensions, such as the blocking requirement, the multi-version capability, and safe-point requirements.

| | **Ginseng** | **Kitsune** | **Ksplice** | **Upstare** | **POLUS** | **ISLUS** | **PSU** |
|---|---|---|---|---|---|---|---|
| Update Load Method | Dynamic Library | Dynamic Library | Dynamic Library | Dynamic Library | Process Hijacking + Dynamic Library | Process Hijacking | Dynamic Library |
| Update Method | Function Indirection | API calls | Binary Rewriting | Function Indirection | Binary Rewriting + Function Indirection | Binary Rewriting | Function Indirection |
| Safe-Point | Manual | Manual | Automatic | Manual | Automatic | None | None |
| Multi-Threading | Added | Yes | Yes | Potential Deadlock | Yes | Yes | Yes |
| Recursion | Yes | Yes | Cannot update functions on stack | Yes | Yes | Yes | Yes |
| Blocking | Partial | Yes | Stops All Cores | Yes | Partial | No | No |
| Abstraction Layer | Source Code | Source Code | Source Code | Source Code | Process | Process | Source Code |
| Update Granularity | Future Function Calls | Future Function Calls | Future Function Calls | Stack Unwinding | Future Function Calls | Stack Recon-struction | Future Function Calls |
| Multiple Code Versions | Yes | No | No | No | Yes | Rollback System | Yes |
| Memory Growth | Yes | Yes | Yes | Yes | Yes | Yes | No |

Table 2.1: Comparison of prevalent approaches and their features

## 2.2 Ability to Handle Recursion

Cycles in a call graph, often referred as recursive calls, create a problem for DSU because performing an update in the middle of a recursion requires the system to keep track of dynamic function frames in the stack so that the correct version of the code associated with each frame is executed. To enable this matching between stack frames and code, multiple versions of the same function must be maintained within the system. Some systems avoid this complexity by avoiding performing an update while a recursion is under execution [1]. However, such a design choice creates the need to detect and track recursions, which may be not trivial in the presence of indirect calls through function pointers. Other systems simply ignore or wish away the existence of recursion. Systems that rely of safe-points can usually avoid issues with recursion by simply not setting such points inside recursive code. However, in some important practical applications it may be difficult to establish an execution point that is safely outside recursive execution.

In DSU systems recursion is a common problem that breaks most approaches. Recursion is ignored because of the complexity it introduces [16]. Several systems [13], [15], [17] have solutions to this problem that are specific to their approach. Our approach uses a simplification of the problem to handle recursion similar to others [15], [17].

## 2.3 Dynamic Update Granularity

Some existing DSU systems are restricted to updating the entire program. Some systems allow for a subset of the program to be dynamically updated. Others systems only allow pre-specified subsets of the program to be updated. Entire program DSU provides maximum flexibility for bug fixes and unplanned changes. However, it may require safe-points or blocking at the DSU point [1], [10], [11], [13], [15], [17]. Requiring the specification of program segments that may be updated provides more control over sensitive portions of an application that an organization may want to prevent from being dynamically updated.

A hybrid approach that allows both whole-program and subset updates would provide the advantages of both systems.

Our approach focuses on function level changes to new calls to a function. Existing functions are allowed to complete their execution while future calls will always running the newest version of code available.

## 2.4 Abstraction Layer

Most existing solutions require that source code be modified to make the target program DSU. The point of modification or transformation differs with the majority being at a source to source level [10], [11], [13], [15], [17]. Some approaches operated at the code object level [1]. Modifications at the code object level allow these approaches to support programming languages that compile to the same code object format but suffer in portability to systems that do not support those formats. Approaches that operate at the source level gain the benefit of expanded portability to varying machines as most are able to be ported to other systems easily but suffer in that they rely on C specific language modification.

## 2.5 Safe-Point Requirement

In many DSU systems *safe-points* must be identified prior to updating. The update can only occur when the execution of the application reach these safe-points [1], [10], [11], [13], [15], [17]. A safe-point may be explicitly declared by a programmer in the source code [10], [11], [13], [15], [17] or it may be automatically discovered by a program analysis [1]. Systems that employ safe-points benefit from being able to perform updates with a guarantee that the system will not result in a program failure. A more versatile DSU systems allows updating at any execution point [5]. Such systems have a great advantage over safe-point-based systems because discovering safe-points is a non trivial task both for a programmer and for an automated analysis. A safe-point based system also must ensure that there is no unbounded amount of execution in between safe-points. There may also be a deadlock on loading an update if a

program has a thread that never reaches a safe-point [5].

## 2.6 Multiple-Version Support

There are several approaches to handle multiple versions of the same software unit in a DSU. One approach is to only allow one version of DSU code to be in execution at any given time. Once an update is completed the previous version can be offloaded from the system memory. An alternative approach is to allow a limited number of versions of the same software unit in the system. For instance, after updating, two versions could be executing at the same time; however, a new updating would not be permitted until the now old version is no longer executing. A more versatile version would allow an unbounded number of versions of the same software unit to be executing simultaneously. In such system, another design decision is the management of memory resources. Some systems are not designed to offload versions that become inactive [15], [17]. A more efficient design will have a mechanism to detect that an old version of the software unit is no longer in execution and will then offload such old versions. Trade-offs, similar to the ones encountered in the automated recovery of memory resources through garbage collection in managed memory systems, are encountered here: how often does the system checks for inactive versions and how much memory resources are free for use. Another important aspect of tracking which versions are active is the linking between executing threads and the software unit versions. An efficient system should minimize synchronization overheads for this tracking.

## 2.7 Binary Regeneration Requirement

Some DSU systems require that the application be written with DSU annotations and, thus, have limited applicability to legacy code [1], [10], [11]. Other systems can be applied to existing code but do require that a one-time recompilation and restart of the application takes place before DSU can occur. Some DSU systems require only recompilation of existing source code to generate DSU-enable, binaries [15], [17]. Yet other systems can perform DSU in

existing pre-compiled binaries [4], [5].

## 2.8 Summary

The seven dimensions of design for DSU presented in this chapter will be revisited in the Related Work discussion where the DSU design presented in this thesis is compared and contrasted with other designs described in the literature. The discussion of these design dimensions in this chapter is also important for a better understanding of both the design and the design decisions made for the DSU system presented here. For instance, the next chapter details why the use of an existing linkage table would be a poor design decision.

# Chapter 3

# Procedure Linkage Table

When contemplating the implementation of a DSU system it is natural to consider leveraging the use of the Procedure Linkage Table (PLT) from a Dynamically Linked Library (DLL) system. The PLT should be considered because such a system already manipulates function pointers and the DSU system could leverage this function-pointer manipulation to implement the replacement of function versions. However, such a DSU implementation has significant disadvantages that make it unsuitable for a practical DSU system. This Chapter describes how a DSU system could be implemented to leverage the PLT and then discusses the disadvantages that make such a solution impractical.

## 3.1 Dynamically Linked Libraries

DLLs use a process where the location of a function can be determined and changed at runtime. This functionality has been around for a long time because of the obvious advantages of only linking code that is actually executed and linking against the most recent version of a library. DLL also enables the use of shared objects that can be used by multiple processes.

A DSU system can leverage this functionality to change the version of an invoked function while a program is running. Thus, it makes sense to use the same data structures as DLL for DSU. In particular, this DSU solution leverages the PLT. The PLT is a table of function pointers that are initialized to point to "dummy code" or stubs. These pointers are changed to point to

the code of the function on the first invocation of each function.

When the executable is loaded, all of the DLLs that the executable depends upon are also loaded in to memory. The dynamic linker will then replace all of the entries in the PLT with pointers to the location of the DLLs. The executable is then able to start execution.

## 3.2 Lazy Linking

The process of replacing all the PLT entries can take a long time; thus, delays the launch of the executable and leading to the concept of *lazy linking*. In the lazy linking model, the actual functions are linked into the PLT only when they are referenced. A call to a function that has not yet been linked triggers the linker. The linker then resolves the symbol and patches the PLT entry such that the next time that the same function is invoked, the PLT entry contains a pointer to the location in the loaded shared library. Thus, after the initial call to a function in the PLT, all future calls can jump directly to the dynamically linked library.

Lazy linking drastically reduces the launch time for executables because linking to PLT entries is delayed, to be performed on demand when each function is invoked. Moreover, if a PLT function is not called during a given execution of the program then the linking overhead for that function is avoided.

## 3.3 PLTHook

A method for building a DSU system is to use the PLT to store entries for each DSU function. Each function has a 0 version, or start function, that is initially linked using the standard dynamic linking process. When a new version of a DSU function becomes available at runtime, its PLT entry is then replaced with a pointer to this new version. This process is made relatively simple because of the availability of an open-source tool called *PLTHook*[22].

PLTHook was designed to search the PLT for an entry and then replace that entry with a specific pointer. PLTHook has a simple and straightforward

API that makes it easy to use and perfectly suited for the task of swapping PLT entries at runtime.

## 3.4 Procedure Linkage Table Unsuitability

Using the PLT to store DSU functions and then using PLTHook to replace them at update time suffers from disadvantages that make the combination of the PLT and PLTHook unsuitable for a DSU system.

The primary issues with using PLT entries for a DSU system is that the code for DSU functions must be removed from the main executable and placed in a DLL. This DLL then needs to be linked to the executable. Placing DSU code in a DLL would require every DSU executable to be composed of two binaries: the executable containing the non-DSU code and the DLL with the initial version of the DSU functions. The path to these binaries must be known at compile time. If both the executable and DLL of DSU function are not available or are not in the right location at runtime, then the dynamic linker is unable to find the DSU code and the executable will not launch.

Likewise, requiring two separate binaries has the disadvantage that users must have two sets of source code: the non-DSU code and the DSU code. The compilation process would require two passes. A first pass to compile the DSU functions into a DLL, and a second pass to compile the non-DSU code while concurrently linking the DLL of DSU functions. This approach is inflexible because of these extra steps and additional source codes.

Tracking multiple versions of DSU functions using the PLT is also a challenge. While the PLT would be suitable for a stack-tracing tracking method, such methods are not feasible for large software systems where the stack can be very large. The alternative tracking method consists of using reference counting. However, using the PLT with this method would require a separate set of tables for reference counters. These reference counters are not functions and, as such, are not suited for placement in the PLT. A solution would be to create a custom table, with similar functionality as the PLT, to store the reference counters. Maintaining two tables leads to unnecessary duplication

of code in the DSU system. Therefore, even though the use of the PLT to store active version of functions in a DSU application is intuitively appealing, the need for multiple binaries and multiple sources, the difficulties in tracking multiple version, and the lack of portability make it infeasible for a large scale DSU system.

## 3.5 Summary

Now that this chapter has established that using the PLT to implement the DSU system would be a poor design decision, even though it is intuitively appealing, the next chapter provides a description of the PSU design, including the mechanisms that were created to keep track of the use of versions of code for each DSU function and how obsolete versions can be reclaimed.

# Chapter 4

# Description of PSU

## 4.1 Overview of PSU

PSU is a framework for running and building DSU programs. The two primary parts in a PSU system are the *update supporting library* and the *application program*. Each of these parts can be broken into several smaller components. Figure 4.1 shows a diagram overview of the PSU.

The application program (8) is the program that a user wants to make DSU. The PSU framework is designed to minimize the performance impact of DSU on the code in the application program and to provide the user a similar development experience. Typically, the vast majority of the application program is composed of non-DSU code (7). The DSU sub-section of the application program contains several pieces of code for each DSU function in the application program.

Each DSU function has its own trampoline function (5). This trampoline is used to enclose all the steps to make a call to a DSU function in a single location and then to make this functionality available to the rest of the application program in a single interface. Each DSU function has global pointers to the active version of the DSU function (2) and the active counter for that function (1). The trampoline uses these global pointers to increment/decrement the counters using PSU functions (13) (14) and to call the DSU function through the global pointer. Patches for a PSU system (20) are created by compiling a DLL containing all the DSU functions that are to be replaced (19). They can be compiled using the same source as the original executable or a separate

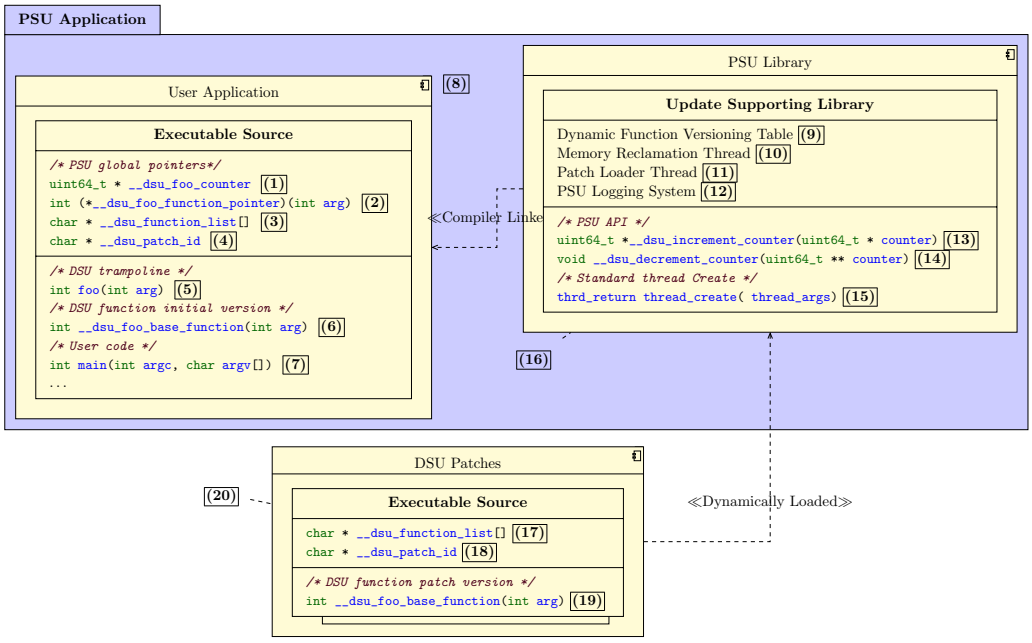



Figure 4.1: Overview of PSU system

source.

The second part of the PSU system is the update supporting library (16) . The update supporting library is responsible for the majority of the runtime work in the PSU framework. The update supporting library watches for new patch files added to the system, maintains references to obsolete versions of DSU functions, loads new DSU functions, and unloads obsolete versions when they are no longer being referenced. In order to maintain a record of all of the loaded versions of DSU functions, the update supporting library maintains a custom table called the Dynamic Function Versioning Table (DFVT) (9), which contains references to every loaded version of a DSU function as well as its associated *PSU function counter*s.

The update supporting library contains two threads. The first thread, called the *patch loader thread* (11), loads in new versions of DSU functions. The second thread, called the *memory reclamation thread* (10), offloads obsolete versions of functions when they are no longer in use.

When a new patch becomes available, the patch loader thread parses through the patch, loads all the DSU code from the patch, and initializes PSU function counters and DFVT entries for each new version of each DSU

function. When the patch loader thread has finished all of these tasks it overwrites the global pointers to the active DSU functions and PSU function counter making them available to the application program.

The memory reclamation thread scans through the DFVT for entries that have obsolete versions of DSU functions with PSU function counters which are set to zero. It will then free up the counters and DFVT entry. When there are no DSU functions that reference a patch, the memory reclamation thread offloads that patch.

When there is no work to be done by the patch loader thread and the memory reclamation thread, they both sleep until a new patch is available. Whenever the patch loader thread finishes loading a new patch, it triggers the memory reclamation thread to start its scans. If there are no obsolete versions of DSU functions in the system, the cleaner goes back to sleep. Otherwise, it repeatedly scans through the DFVT. To reduce the performance impact of these repeated scans, the memory reclamation thread sleeps between each scan. The amount of time that the memory reclamation thread sleeps between scans increases at an exponential rate governed by a threshold.

The update supporting library contains a subsystem, called the *PSU logging system*, that is used to monitor the status of the PSU application [(12)]. The PSU logging system communicates the status of patch loading and unloading in a PSU application, to users via a log file. By avoiding the use of standard input, standard output, or standard error, the PSU logging system can communicate without interrupting the application program.

The update supporting library overrides the thread-create function for the standard threading library [(15)]. When the thread creation function is called from the update supporting library, a Thread-Local Storage (TLS) variable is created to store an index for the PSU function counter. This TLS variable allows each thread in the application program to access and increment the PSU function counter for their thread without a chance of a race condition.

## 4.2 Update Supporting Library

The previous section described how the invocation of a DSU function will always use the most recent version of the function that was loaded into the system. However, for a long-running application, it is also important to release resources that are used by obsolete versions that are no longer executing and that will never be invoked again because they were superseeded by newer versions. This section describes a logging system based on reference counting that accomplishes this resource reclaiming.

### 4.2.1 PSU Logging System

Once invoked, a DSU function must execute to completion using the same version of the code that was active when it was invoked. A DSU patch may be unloaded only after all function invocations that use code from that particular patch have finished executing. Therefore, a patch may stay loaded for an undetermined amount of time depending on the design of the application program. Each PSU patch is provided with an optional *PSU patch id*. If the PSU patch id is provided, then PSU is able to report information about what patches have been loaded and unloaded at runtime. Because PSU operates as a framework for providing DSU functionality to a application program, PSU must be able to provide information without interrupting the I/O of the application program. Therefore, the PSU logs the output for information in a log file using the PSU patch id for reference.

### 4.2.2 Dynamic Function Versioning Table (DFVT)

PSU implements a reference counting approach to track when obsolete versions of DSU functions can be retired. For each DSU function, PSU must store a pointer to the current DSU function version and a PSU function counter that serves as a version reference counter. The DFVT is a custom table of pointers to DSU functions and PSU function counters. The DFVT scales to grow as more versions of DSU functions are loaded and to shrink as these obsolete versions are retired. An update supporting library accesses the DFVT to keep

track of where in the application program the active DSU function pointer and active PSU function counters are stored. The DFVT also stores a list of DSU function pointers and PSU function counters pairs that are no longer the newest version but still have active references.

The outermost layer of the DFVT is a simple hash map. It maps a DSU function's name as a string to a table entry. Each table entry contains a pointer to the location in the application program's memory where the DSU function pointer and the PSU function counter are stored. Each entry also contains a queue of obsolete versions of DSU functions. PSU function counters for obsolete versions of DSU functions cannot ever be increased after that version has been updated. When the last element in the queue hits zero in the PSU function counter, that function can be reclaimed because there are no active stack segments using that DSU function. This scheduling is explained in Section 4.2.6. During an update when a new DSU function is loaded, a new PSU function counter is allocated and the pair are pushed to the front of the queue. The memory reclamation thread periodically checks if there are entries in the queue of obsolete functions and removes them if the PSU function counter for that entry is zero.

### 4.2.3 DSU Function Counter

Obsolete DSU functions must be reclaimed as new DSU functions are loaded. Each DSU function requires memory to store the newly loaded code. If a program containing DSU functions that are updated multiple times runs for an extended period of time, the memory used by obsolete code grows. Thus, the space occupied by obsolete code must be reclaimed to prevent, potentially catastrophic, wasted memory.

Automated memory management is well understood in the context of object-oriented programming languages where a separate system, called the garbage collector, is responsible for reclaiming memory which is occupied by objects that can no longer be accessed by the program. This reclaiming may be performed by sweeping the memory from a known collection of root objects or by using reference counting. This session reviews several approaches for

memory reclamation that could be used in a DSU system.

The first approach is simply to not reclaim memory at all. This approach has a considerable advantage in performance and implementation simplicity as it has no requisite code added other than the base DSU function indirection. Such a system would be ideal for systems that can be restarted semi-regularly. An example of such systems are rapid prototyping systems. DSU without memory reclamation would enable the rapid evolution of such system without requiring a restart for each code change. Unfortunately reclaiming memory is an necessary for systems that run for extended periods of time or that have many updates. This is because the whole system may become slow due to increased page swapping caused by the excessive memory usage.

The second approach is a reference-counting system. In such a system each target location in memory is assigned a counter. When a reference to the location is created, the reference counter is incremented. When that reference is no longer used, the counter is decremented. In a DSU system the number of active invocations associated with a given version of a function are counted instead of counting references to a memory location. This approach is straightforward to implement and has a minimal memory impact. However, in a multi-threaded system all the reference counter updates must be synchronized. A reference counter must be updated in each invocation and each termination of a DSU function.

A third approach is to sweep through memory looking for references to the targeted location in memory. When an active reference is found, it is marked and the system assumes that the location in memory is in active use. These systems have the advantage that they provide the minimal performance impact on the programs they run. The execution pattern of the user threads in this approach is identical to a system where memory is not reclaimed at all. In such a system, the update supporting library would scan through the call stack of the targeted threads looking for the call returns to any DSU functions. When one is found, the update supporting library would then know it could not reclaim that DSU function. The primary issue with scanning the call stack for DSU function calls is that scanning an active call stack is difficult to imple-

ment. Scanning may require the system to pause the execution of the targeted thread during the duration of the scan. While solutions created for concurrent garbage collection in dynamic memory management could be adopted, they are complex to implement and difficult to correctly implement [21]. Another issue with such an approach is that in large systems these call stacks can be quite large, especially in a recursive system.

### PSU Function Counter

Of the approaches listed, PSU implements the reference-counting system. In its core concept, its a relatively simple system to implement. PSU implements a modified version of reference counters called PSU function counters because the PSU is designed to work with multithreaded application programs. Multiple threads trying to concurrently access a reference counter without synchronization constitute a race condition. There are three approaches to deal with this race condition, each with benefits and costs.

```
1  uint64_t * __dsu_foo_counter;
2  int (* __dsu_foo_pointer)(int arg);
3  mutex * __dsu_foo_mutex;
4
5
6
7  int __dsu_foo_function ( int arg) {
8    /* foo body */
9  }
10
11 int foo(int arg) {
12   mutex_lock(__dsu_foo_mutex);
13   uint64_t * cache_counter = __dsu_foo_counter;
14   (*cache_pointer)++;
15   uint64_t * cache_pointer = __dsu_foo_pointer;
16   mutex_unlock(__dsu_foo_mutex);
17
18   int reval = cache_pointer();
19
20   mutex_lock(__dsu_foo_mutex);
21   (*cache_counter)--;
22   mutex_unlock(__dsu_foo_mutex);
23 }
24
25 int main(int argc, char* argv[]) {
26   /* Main body */
27   var_foo = foo(var_bar);
28   /* Main body finish */
29 }
```

Listing 4.1: A mutex based reference counting system for DSU function

The first approach is to place mutex locks before and after calls to DSU functions. This is optimally done through a trampoline function. An example of this approach can be seen in Listing 4.1. Inside the trampoline function, the calling thread will acquire the lock, increment the counter, cache the function pointer, release the lock, and then call the DSU function through the cached pointer.

When exiting a DSU function inside the trampoline function, the calling thread must first cache a copy of the return value from the DSU function, acquire the lock, decrement the counter, release the lock, and then return the value from the DSU function.

A mutex-based approach benefits from having a very strict guarantee that every DSU function is tied to their counter with no race conditions. This leads to a simple and intuitive proof of correct behaviour. This approach also ben-

efits from hardware and software support in existing systems. A limitation of this approach is that the performance impact can be significant. Synchronization via mutex can have substantial overhead. Contention in locks may lead to additional traffic in the system buses, synchronizing instructions such as atomic *test and set* are more expensive to execute. Advanced locking systems such as ticket locks can be used to ameliorate the contention issues. However, even an efficient locking system results in additional memory locations, dedicated exclusively for the locking operations, being accessed and generating additional memory traffic.

The second approach is to use atomic variables as reference counters. Similar to a mutex-based approach, atomic-reference counters would allow multiple threads to increment counters without incurring a race condition. This approach is easier to implement because it simply requires that the type of the counters be changed to an atomic type and the increment function be changed to an atomic increment rather than a conventional increment. However, this approach has a looser pairing between the function version and the counter increment. It also has similar performance overhead issues to the mutex based approach due to the expensive atomic increment and decrement instructions. The atomic-reference-counter approach benefits over the mutex approach in that as contention increases performance does not degrade due to the absence of the spin lock when another thread is incrementing or decrement the counter and making the function call or return.

A third approach is to use transactional memory. Transactional memory is a technique that allows multiple memory loads and stores to be completed atomically. Transactional memory assumes that there will be no concurrent accesses and, thus, perform the operations optimistically. Then, in the cases where a conflict does occur, a transaction rolls back and the sequence of memory operations is attempted again. When an excessive number of rollbacks occur, the system can revert to a lock-based synchronization. There are two forms of transactional memory: Software Transactional Memory (STM) and Hardware Transactional Memory (HTM). STM is implemented using atomic variables and only has benefits when a large number of memory locations are

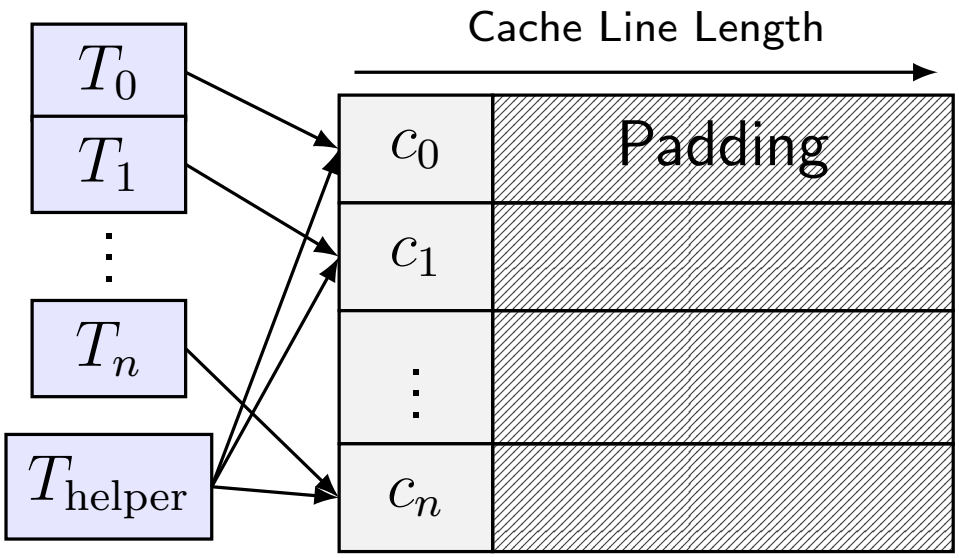



Figure 4.2: Function counter with thread access pattern

modified within a transaction. For the reference counters in a DSU system, there would be no benefit on using STM over atomic reference counters. HTM can have a significant performance benefit over atomic variables and mutex locks when there is low contention to the access of counters, which would be expected in a DSU system. Unfortunately, HTM is only available in a very limited number of machines.

HTM could potentially be the best solution for dealing with the function counter race condition but it is not yet practical.

The final approach is to drop the need for synchronization between threads entirely. The need for synchronization arises due to multiple threads accessing the same reference counter for multiple invocations of a single DSU function. The solution is to split this single counter into a set of counters where each thread maintains its own counter. Therefore, without multiple accesses to the same counter, there is no race condition. Each thread is assigned their own index, which is stored in a TLS variable that is assigned at thread creation. False sharing, where multiple threads could be accessing the same cache line even though they are updating different counter, could result in performance degradation. To prevent false sharing, padding is added around each counter to guarantee that they are located on different cache lines. Figure 4.2 shows

the memory layout of such a system. This approach has the advantage of eliminating a race condition for access to the counters by multiple threads without relying on expensive atomic variables or mutex locks at the cost using more memory. However, there is still a race condition between each individual thread and the thread that reclaims memory used by obsolete functions. The solution to these races is explained in Section 4.2.6.

### 4.2.4 Intercepting Thread Calls

A unique index is associated with each thread and stored in its TLS so that the thread can use this index to access PSU function counters. However, a TLS location can only be accessed by the thread that has access to the TLS. Therefore, this variable can only be assigned after the thread has been created. There are two approaches to solve this problem.

The first solution is to add code that reads the thread index to every function call that may start the execution of a thread. This approach requires a program to systematically go through the source code, discover all calls to thread creation, and add the needed code. This is not a very robust system because it would not be possible to discover function calls that create threads inside libraries. This solution also makes the system much more difficult and error prone to users as it requires a significant amount of user implementation.

The second solution is to intercept calls to thread creation and use a trampoline layer called the *PSU thread intercept*. The PSU thread intercept creates a closure containing a pointer to the function that starts a thread and the values of its arguments. It then calls the original thread creation function with the starting function being a second trampoline function called the *PSU thread initializer*. The closure is passed to PSU thread initializer. The PSU thread initializer then acquires a thread index and assigns it to TLS before finally calling the original target function with its arguments. The acquisition of the thread index must be performed synchronously with every other thread creation call. This adds an increased overhead to thread creation but allows for calls to DSU functions to be performed more efficiently.

The prototype of PSU employs the trampoline approach because it is more

general and more robust than the code-insertion approach. The trampoline approach is also easier to implement in large systems. The PSU prototype is written for a POSIX system and uses the pthread threading library. Thus, in this prototype, calls to `pthread_create` are intercepted to invoke the trampoline. To monitor the use of thread indices, the update supporting library maintains a boolean vector. A PSU thread initializer first acquires a global thread-create lock and then searches for an unset value indicating that the corresponding index is free. Then, it sets a TLS variable to the value of this index. It then releases the thread-create lock and the newly created thread calls the original target function. PSU thread initializer also registers a thread cleanup function called *PSU thread cleanup* that will be called when the thread exists. This function unsets the boolean value at the index's location allowing it to be set again by a new thread.

While the prototype is implemented for POSIX systems, similar functionality exists in other threading libraries. Therefore the techniques described can be used in various multithreading systems.

### 4.2.5 DSU Function Trampoline

PSU implements a reference-counting approach to monitor when DSU function versions become obsolete. PSU also uses function pointers to swap out versions of DSU functions that have been updated. In this approach the PSU function counters must be incremented prior to calls to the DSU function pointer. The PSU function counter must also be decremented after the DSU function returns. This must be done so that the memory reclamation thread knows when a DSU function can be reclaimed. The increment and decrement must happen at every DSU call site. To simplify this, PSU uses a trampoline function call. Inside this trampoline function, a pointer to the PSU function counter is cached and incremented. The DSU function pointer is then dereferenced and called. When the DSU function returns, the cached pointer to the PSU function counter is used to decremented the counter. The return value from the DSU function call is then return. This approach guarantees that the same PSU function counter is incremented when the DSU function call is

made and decremented when the call returns.

An alternative to using a function trampoline would be for PSU users to add code to increment the PSU function counters prior to directly calling the DSU function through the function pointer and to decrement it upon function return. An advantage of this alternative would be a reduced number of function calls required to call a DSU function. However it significantly increases the amount of work required from a DSU user increases the likelihood of user error. Moreover, there is a high probability that the trampoline function will be inlined by a compiler and, thus, the additional function call overhead would be eliminated.

### 4.2.6 Lock-Free Reference Counting for Reclamation

PSU employs a lock-free reference-counting approach and thus there are race conditions between worker threads invoking a DSU function and the updating of the version of the same function by the patch loader thread. This subsection explains how correct operation is guaranteed in the presence of such race conditions. To change the version of a function, the patch loader thread must update both the global pointer for PSU function counter and the DSU function pointer. Logically these two memory locations should be updated atomically to prevent inconsistency because the application program reads both pointers when invoking a DSU function.

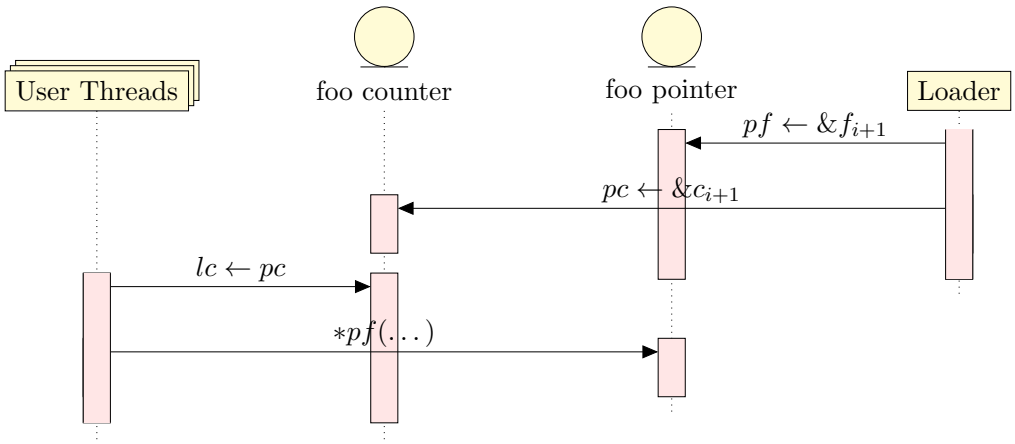
Let $f_i$ and $f_{i+1}$ be two consecutive versions of a DSU function $f$. Let $c_i$ and $c_{i+1}$ be two consecutive PSU function counters for $f$ associated with versions $f_i$ and $f_{i+1}$. Let $pf$ be the global function pointer used by the application program to call $f$. Let $pc$ be the global PSU function counter pointer used by the application program during a call to $f$.

When updating $f$ from $f_i$ to $f_{i+1}$ the patch loader thread must perform the following sequence of events in this order:

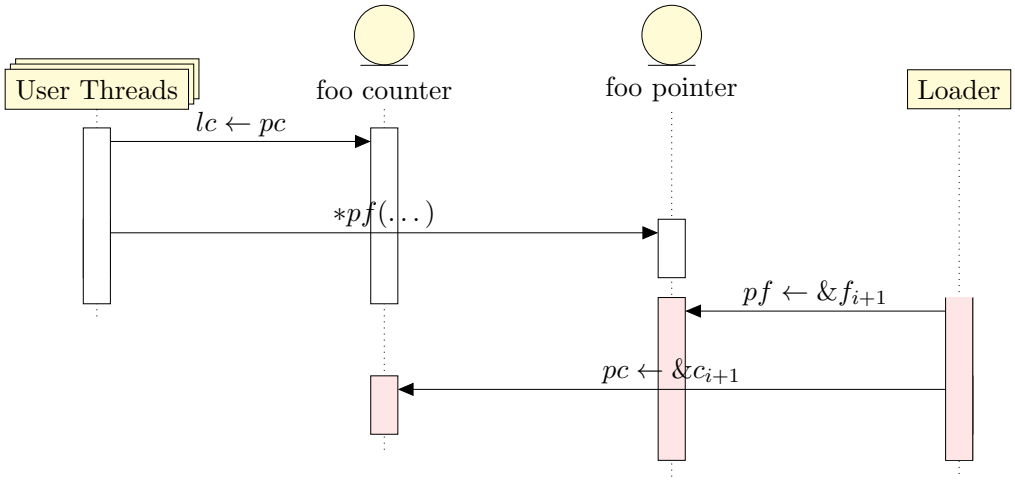
(1) $pf \leftarrow \&f_{i+1}$

(2) $pc \leftarrow \&c_{i+1}$

Where & stands for the "address of" operator. Step (1) sets the global




(a) Patch Loader Thread starts and finishes update prior to DSU function invocation




(b) Application Program threads finish DSU function invocation prior to patch loader thread update

Figure 4.3: Application Program threads and Patch Loader Thread accessing shared PSU Function Counter and DSU function pointer without data race

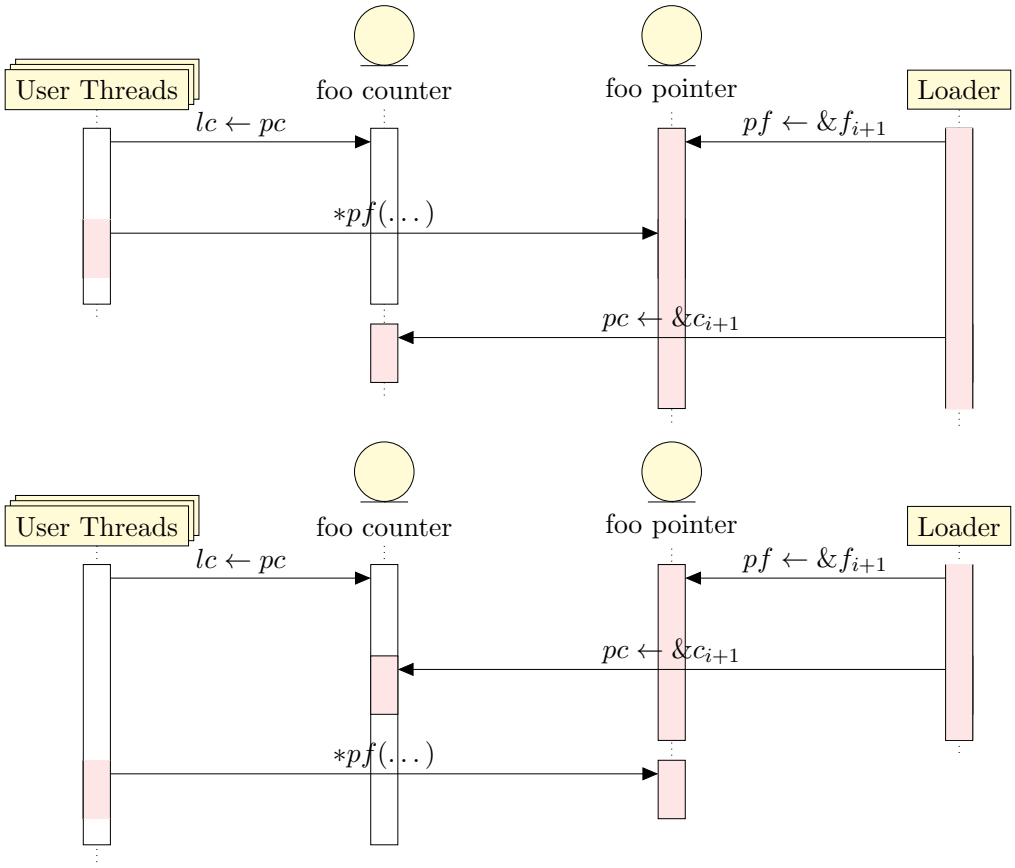



Figure 4.4: Application Program threads and Patch Loader Thread accessing shared PSU Function Counter and DSU function pointer with data race

DSU function pointer $pf$ to point to the new DSU function version $f_{i+1}$ for function $f$. Step (2) sets the global PSU function counter pointer $pc$ to pointer to the new PSU function counter version $c_{i+1}$ for the function $f$.

When invoking a DSU function $f$ a worker thread must execute the following sequence of events in this order

(1) $lc \leftarrow pc$

(2) $(*pf)(\ldots)$

Step (1) creates a local copy $lc$ of global pointer to the counter $pc$. Step (2) dereferences the global function pointer $pf$ to invoke the correct target DSU function.

To address the issue of potential races between the patch loader thread and worker threads' function invocation, it is necessary to consider all interleavings of the steps above. There are two cases to consider as shown in Figures 4.3 and 4.4. Case 1 is when there are no inconsistency because one thread performs both steps before the other thread performs its steps. This occurs with the two interleavings shown on Figure4.3. Case 2 consists of four interleavings, shown in Figure 4.4, that can lead to inconsistent state.

The inconsistency happens when the patch loader thread has updated the function pointer to activate a newer version of the function, but has not yet updated the counter pointer when a worker thread performs its steps to invoke a DSU function $f$. The consequence is that the worker thread increments the counter $c_i$ for a version $f_i$ while actually invoking a version $f_{i+1}$ for the function $f$.

Figure 4.5 summarizes the possible associations between reference counters and function pointers that can be created because of the interleavings shown in Figure 4.4. Given that the steps in both the patch loader thread and the worker threads are strictly ordered, it is impossible for a counter $c_{i+1}$ to be incremented for the invocation of a version $f_i$ of a function $f$. However, a counter $c_i$ may be incremented when a version $f_j$, $j > i$ is invoked. This means that the reference counter can only overcount the number of invocations of a version $f_i$ and undercount the number of invocations of a version $f_j$. Both

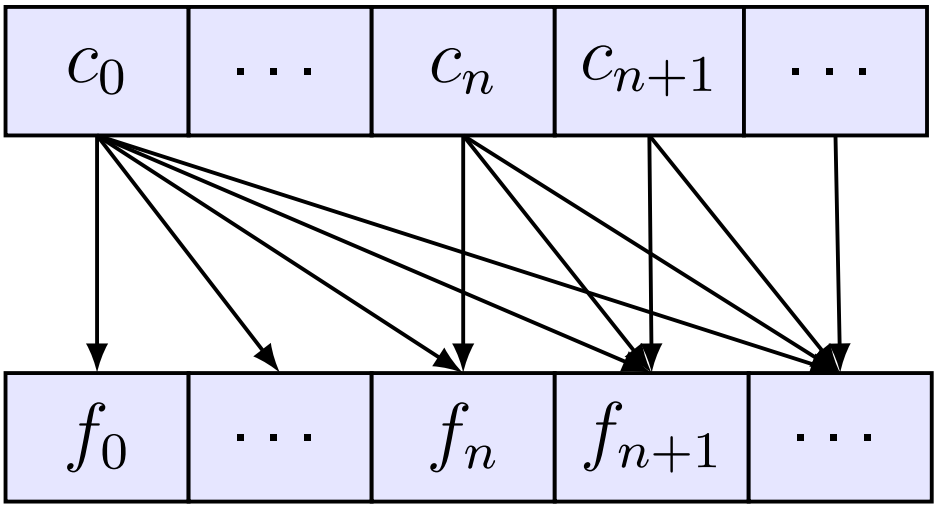



Figure 4.5: Association between function pointers and counters due to the potential data race

overcounting and undercounting could lead to issues when reclaiming versions $f_i$ and $f_j$. However, the overcounting only leads to a temporary delay in the reclaiming of $f_i$ because, eventually, when the invocation $f_j$ terminates the counter $c_i$ will be decremented. To prevent reclaiming $f_j$ too soon because of the undercounting, a version $f_j$ can only be safely reclaimed when the counter $c_j$ is equal to zero and all counters $c_k$, $k < j$ are also zero.

## 4.3 Application Program Interface

The PSU system is a framework for building DSU applications. PSU interfaces with application programs using a set of data structures and functions as well as a linked library. These interfaces must be declared following a specific standard. This rigid standard makes these functions easily detectable by the update supporting library at run time.

### 4.3.1 DSU Function List

Every DSU function in the application program must have its name listed in a globally declared list of functions called the *PSU function list*. When a PSU program is launched, the update supporting library scans the PSU function list and creates entries in the DFVT for each DSU function. The DSU function

name listed in the PSU function list is also used by the update supporting library to locate the information associated with that DSU function such as the global DSU function pointer and the global PSU function counter pointer.

### 4.3.2 DSU Function Declaration

```
1 char * __dsu_function_list[] = [..., "foo", ..., NULL];
2 char * __dsu_patch_id = "...";
```

Listing 4.2: Example of the requisite data structures and frameworks for a DSU patch

```
1  uint64_t * __dsu_foo_counter;
2  int (*__dsu_foo_function_pointer)(int arg);
3
4  /** DSU_FUNC */
5  int foo(int arg) {
6    uint64_t * counter
7      __attribute__((cleanup(__dsu_decrement_counter))) =
8      __dsu_increment_counter(
9          &__dsu_foo_counter[THREAD_ID * CACHE_LINE_SIZE]
10       );
11   return __dsu_foo_function_pointer(arg);
12 }
13
14 int __dsu_foo_base_function(int arg) {
15   /* foo function body */
16 }
```

Listing 4.3: Example of the requisite data structures and frameworks for a DSU function `foo`

```
1 /** DSU_FUNC */
2 int foo(int arg) {
3   /* foo function body */
4 }
```

Listing 4.4: Example of the a DSU function before being transformed by the Updating Weaver

Listings 4.2 and 4.3 depicts a DSU function, named `foo`, and all requisite data structures for the PSU framework. Listing 4.2 Line 1 shows the PSU function list with the DSU function name. The PSU function list is defined as a NULL terminated list of character arrays. Listing 4.2 Line 2 shows a PSU patch id. The PSU patch id is used to provide information about the state of the

PSU program through its execution. The PSU patch id can be any string of characters.

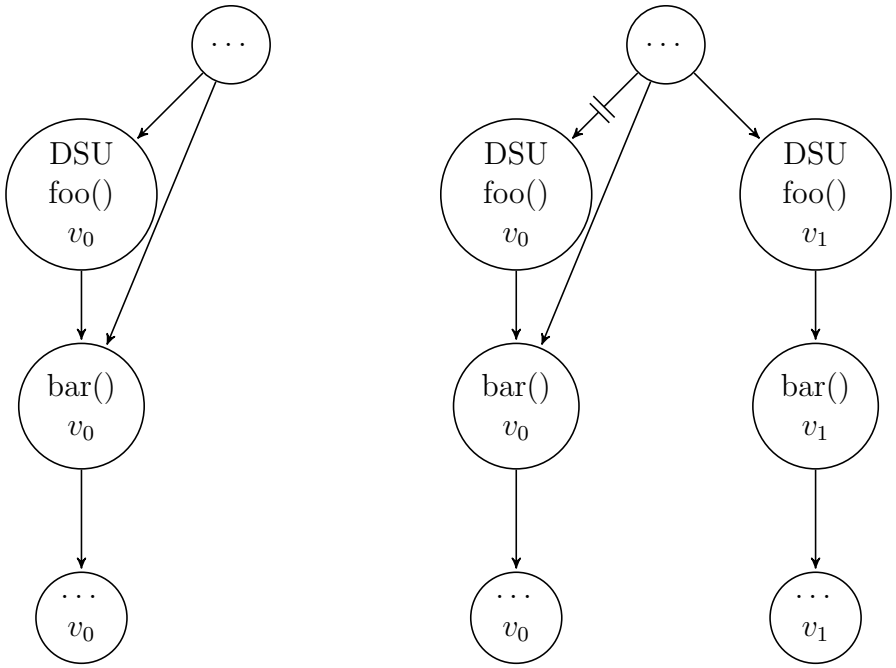
Listing 4.3 lines 1 and 2 show a global DSU function pointer and a global PSU function counter pointer. Both the global DSU function pointer and global PSU function counter pointer are named following these steps: (1) prepend `__dsu_` to the function name `foo`, (2) append `_f_pointer` for the function pointer, (3) and append `_f_counter` for the PSU function counter. The `THREAD_ID` in Line 9 is a thread specific global variable stored in TLS. By storing the variable in TLS, the interface for the trampoline is identical for each thread but the indexed value is unique to that thread. During the creation of a thread in the application program the `THREAD_ID` is set to unique value. The `CACHE_LINE_SIZE` in Line 9 is a global constant representing the size of a cache line for the machines specific hardware.

The DSU function trampoline is named using the original DSU target name `foo`. It's content must follow the definition Listing 4.3 lines 5-12 exactly with the exception of the trampoline name on Line 5. The original function is renamed by prepending `__dsu_` and appending `_function` to the function name. Listing 4.3 Line 14 shows the base version for the DSU function `foo`.

### 4.3.3 DSU Call tree

In a PSU application, all functions can be transformed to be DSU except the `main`. This would allow for the future invocation of every function to be updatable at any point. This would incur an unnecessary performance penalty. When a DSU function is compiled into a patch, all requisite sub-functions that will be invoked will be compiled into the patch and invoked by the DSU function. In PSU, a DSU function marks a point in the call tree that should be able to be transferred to a new version.

Figure 4.6 depicts an example of a DSU function in PSU. Function `bar` may be referenced from a non-DSU section of the program or from a DSU function. When a DSU patch is applied to the application, `foo` is updated from $v_0$ to $v_1$. When `foo` makes a call to `bar`, it will call the version of bar associated with it. In Figure 4.6a, `bar` is invoked from both the non-DSU code and by the DSU




(a) Subsection of a call tree with a DSU function

(b) Subsection of a call tree with a DSU function after a patch has been applied

Figure 4.6: Subsection of a call tree of a DSU program

function `foo`. In Figure 4.6b, `bar` exists in two forms. The non-DSU code will invoke the original version of `bar` ($v_0$). The updated DSU function `foo` ($v_1$) will invoke the version of `bar` ($v_1$) that was compiled with it.

An optimal use case for PSU in a program is to map sections of the application programs which are optimally run from invocation to termination in a single DSU version. This subsection of the application program's call tree must not be neither too large, as it would not be updated often enough, nor too small, as it would incur an excessive performance penalty.

## 4.4 Automated Transformation

The rigid nature of the PSU interface is not only important for the runtime interface of the update supporting library, but it also increases the likelihood of a user error. To simplify this process, PSU has a source-to-source transformation tool called the *updating weaver*. The updating weaver is written using the *Clang LibTooling* interface [23]. The Clang LibTooling system is de-

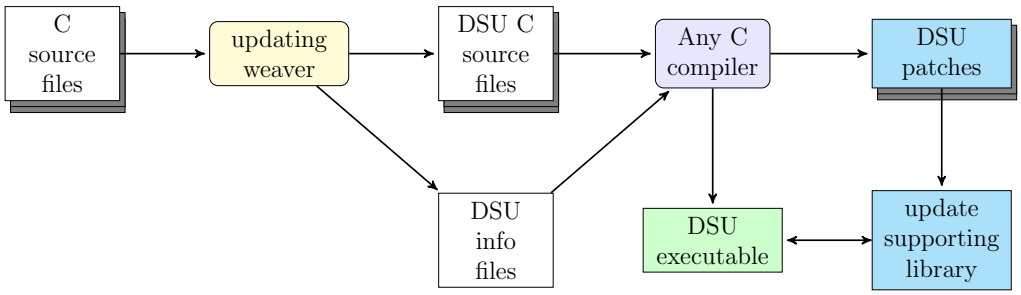



Figure 4.7: The compilation flow for a PSU application with shared source files

signed to allow developers to use the clang parser and write their own backend programs. This is typically used to create linting tools as well as code transformation tools. The updating weaver scans the application program code base and searches for a PSU specific function tag shown in Listing 4.4 Line 1. It then transforms the function into a DSU function through the following steps: (1) rename the function to the DSU base version format, (2) insert the global DSU function pointer, (3) insert the global PSU function counter pointer, (4) insert the DSU function trampoline using the original function name, (5) and insert an `include` statement for the update supporting library into any file that has a DSU function declaration.

The addition of the updating weaver to the PSU framework is a tool of convenience. It allows for applications, not originally written with DSU in mind, to be able to easily be rebuilt with entire subsection of them DSU. It also allows users to build new applications without the extended complexity of the PSU system. The updating weaver also allows users to run static analysis tools on their code bases without the PSU interface interfering with them.

## 4.5 Compilation of PSU application

With the updating weaver, PSU is able to make a non-DSU program easily DSU. The steps involved are displayed in Figure 4.7 and listed as follows: (1) add DSU Function Tag to the application program using `/** DSU_FUNC */` or `//! DSU_FUNC`, (2) run the updating weaver on the application program code base, (3) and compile the generated code using a standard compiler.

The DSU function tags in step (1) only need to be added if a partial DSU application is being generated. The updating weaver can transform every function in the application program, except the main function, into a DSU function. This would allow for a near whole program to be DSU.

As manually annotating every function desired to be DSU may be tedious, PSU allows users to use an alternative way of listing functions to be targeted for transformation. A user simply creates a header file which contains function declarations for each function to be targeted. This header file is then passed to the updating weaver during step (2). This approach allows a large number of functions to be declared in a single place.

The updating weaver can perform an in-place transformation to overwrite the source files. Alternatively the updating weaver is able to generate a DSU copy of the application program source files. This flexibility allows developers to adopt a system that is suitable for their use case.

PSU applications are written in standard C and do not use compiler specific functions. This allows PSU applications to be compiled using any C compiler. The update supporting library requires that DSU functions be compiled with their relocation data still available at runtime. In GCC and Clang this is done by compiling with the flag `-rdynamic`.

An alternative to compiling with relocation data is to register the location of each DSU function component prior to launching the main function. This registering would be done through a call-back system. This call-back system would be a series of function calls from the application program that would have to occur after the update supporting library is initialized but prior to the application program's main function being invoked. This approach would increase the complexity of the PSU system and would reduce the compatibility with other complex systems but would potentially allow for better performance of the PSU application.

The compilation process employed by PSU is simple and easy to use. This allows PSU to add little extra complexity for a developer. PSU manages to add the additional DSU functionality with minimal impact to the compilation process. This is tested in Chapter 5.

## 4.6 Summary

This chapter presented the design of PSU describing how invocations of DSU functions are executed indirectly through function pointers, how a carefully designed low-overhead multi-threaded reference counting mechanism can be used to reclaim resources occupied by obsolete versions of DSU functions, and how a non-DSU function is recompiled with a weaver to make it DSU. All these modifications will impose an overhead in comparison with the original version of the program that could not be updated dynamically. The next chapter presents an experimental evaluation that quantifies these overheads both using synthetic programs designed specifically to measure each type of overhead, and then the overall overhead experienced by a database application.

# Chapter 5

# Experimental Evaluation

The PSU system must perform more work than a normal C program. This extra work encompasses all the extra steps that must be executed in order for the program to update themselves when patches are available. Therefore the experimental evaluation of a DSU system must determine how much is performance is lost in these additional steps.

The evaluation of the DSU overhead requires the study of several components of PSU programs: (1) the transformed DSU function, (2) creation of threads, (3) destruction of threads, (4) program initialization, and (5) patch loading. With each of these components a small amount of performance is lost. This evaluation proposes several questions focused on each of these components as well as the system as a whole.

The First component is the transformed DSU function. Each DSU function is transformed from a direct function call to an indirect function call with PSU function counters accesses. This adds a delay to the target program in order to be able to execute the target function's body. The Function-Call Experiment, in Section 5.2.1 is designed to measure the precise cost in performance introduced by the transformation from a direct function call to a DSU function call.

The PSU function counters are implemented to have no cross thread impact in performance between user threads. The Multi-threaded Function-Call Experiment, in Section 5.2.2, proves that user threads have no cross thread impact. This allows PSU to scale to applications with large numbers of threads.

During thread creation, each thread is assigned a TLS variable. These variables are used for PSU function counter accesses and must be assigned using expensive mutex locks. The Thread-Creation-Destruction Experiment in Section 5.2.3 is designed to measure the precise cost of this sequential startup of threads.

The update supporting library must be initialized prior to the main function of the application program may be invoked, which adds a delay to the launch of the application. The DSU Initialization Experiment, in Section 5.2.4, measures the delay before the user's main application starts running as the target application scales in number of DSU functions.

When a new DSU patch is loaded, new versions of DSU functions are executed, on the first function call, immediately after they have finished loading. The time it takes to load a DSU patch varies based on the number of DSU functions that must be loaded. The DSU Patch-Load Experiment in Section 5.2.5 measures the time it takes for a DSU patch to be loaded as the target application scales in number of DSU functions.

The previous experiments mentioned rely on synthetic micro benchmarks to measure the exact cost of the components. They do not fully reflect the performance of a PSU application. The micro benchmarks show the worst-case performance loss of the PSU system and not the performance relative to a real world application. The MySQL Experiment in Section 5.3 is created in order to measure DSU performance in a real world application. In the MySQL Experiment we created a DSU version of used MySQL [18] and used the HammerDB benchmarking software [9] to measure performance.

## 5.1 Experimental Setup

All experiments were run on a system with two Intel Xeon E5–2687W v4 CPUs with the CPU frequency locked at 2.0 GHz, with all forms of speed step disabled, and with 125GB of memory. The system ran Ubuntu 16.04 with the 4.4.0–138–generic Linux kernel. In all experiments that use the GCC compiler, the GCC–7.3.1 April 24, 2018 release is used. All experiments that use the

Clang Compiler use Clang–6.0.0 March 8, 2018 release. All references to to executing with Perf in the subsequent sections, refer to executing the program in question with Perf version 4.4.155 [20] monitoring cache-references, cache-misses, branches, branch miss-predictions, page-faults, cpu-clock, and task-clock.

All experiments which use synthetic benchmarks are compiled using both Clang and GCC at optimization levels o0, o1, o2, and o3 to create a baseline version of the benchmark program. The DSU library inclusion statements are then added and the program is compiled again to create the experimental version of the benchmark program. All synthetic benchmarks are executed 30 times with the average results reported. In all of the following bar graphs a 95% confidence interval is recorded. In several of the graphs the confidence interval is so narrow that the confidence interval not being visible in the graphs.

## 5.2 Synthetic Performance

### 5.2.1 Function-Call Performance

The invocation of a DSU function must perform more work than the invocation of a non-DSU function. The DSU function invocation requires a function pointer dereference as well as the updating of a reference counter. Thus, the performance implications of a DSU function invocation must be understood.

#### Experimental Design

In order to measure the performance loss caused by DSU function invocation, a very simple synthetic program was created. This program has two core components: a target function that dereferences a pointer to an integer and returns the value of the integer and a loop that invokes this target function many times. The time to execute a single invocation of the target function is minuscule. Therefore the invocation has to be repeated a large number of times in order to accurately measure the invocation cost. In different experimental measurements, the loop invokes the target function a different number of times.

The synthetic benchmark is then run at all optimization levels with Perf.

Varying the number of iterations in the loop containing the function invocation provides insight into whether the programs initialization cost may overshadow the DSU function call cost.

The purpose of this experiment is to measure the cost of the indirect function call and reference-count update in relation to the baseline direct function invocation. Thus, function inlining must be disabled for this experiment to prevent the simple function target from been automatically inlined by the compiler. If function inlining was allowed, it would not be possible to compare the two function call costs and the baseline execution time would be very low because the target function is extremely simple. Disabling function inlining is reasonable for these measurements because functions that are optimal DSU targets cannot be inlined.

Each experimental measurement was repeated thirty times and the averages of these thirty executions of each version of the synthetic program are reported for each optimization level, compiler, and number of loop executions. The performance difference between the baseline and the experimental version of the benchmark show the difference in performance for function calls to DSU functions.

**Results**

As the number of loop iterations increases, the ratio of execution time for the baseline version of the program and the experimental version of the program stabilizes. Figure 5.1 shows the ratio of execution time as the number of iterations increases. As the number of loop iterations increases, the impact of program startup and the initialization of the update supporting library on the overall runtime are minimized. A figure of the experiment with the largest number of loop iterations is included in Figure 5.2 because it has the smallest impact from confounding factors. In this experimental evaluation, a DSU function call using the PSU framework takes 2.97× and 4.24× as long to perform the function call compared to a non-DSU function.

This additional overhead is due to (1) the extra function call to the trampoline layer, (2) the reference counter being incremented and decremented

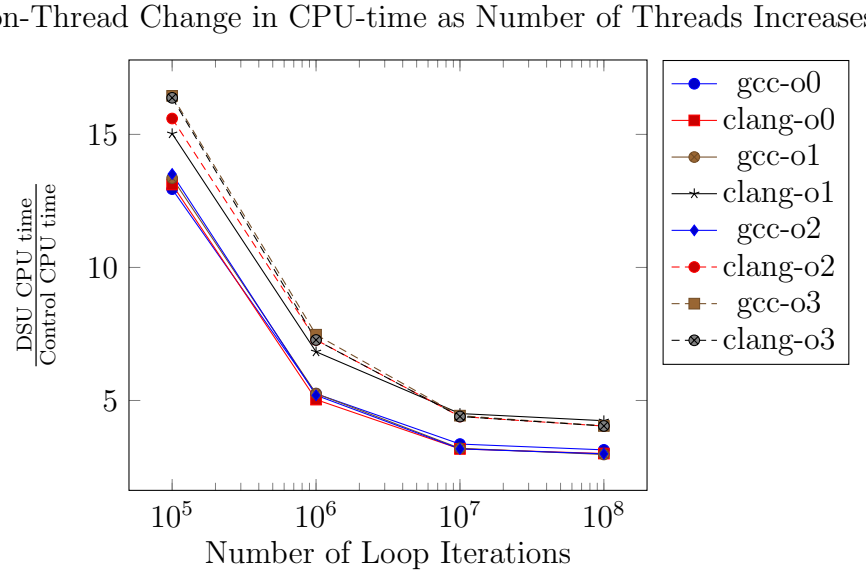



Figure 5.1: Ratio of execution time between DSU function calls vs. non-DSU function calls as the number of loop iterations increases (Log Scale X-axis)

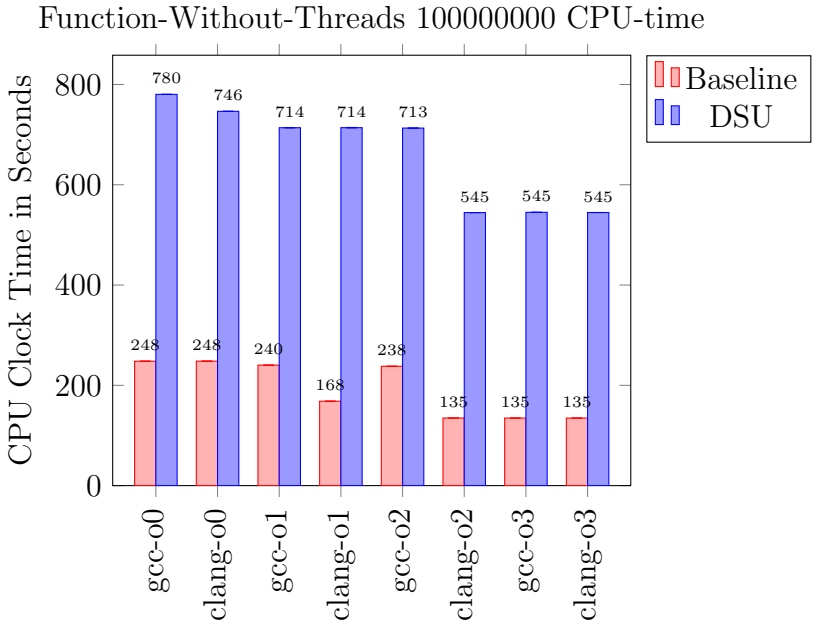



Figure 5.2: Execution time of a DSU function call in a non-DSU system vs. a DSU system.

before and after the call to the DSU function, (3) the call to the DSU function through a function pointer rather than a direct call.

### 5.2.2 Multi-threaded Function-Call Performance

Concurrent calls to mutable memory can have significant performance impact on a multi-threaded system. The DSU approach described in this thesis relies on mutable function pointers being accessible from multiple threads. Thus, an interesting question is "What is the impact of DSU functions in the performance of multi-threaded programs?". A synthetic program is created to to estimate the multi-threaded function invocation overhead by modifying the program used in Section 5.2.1.

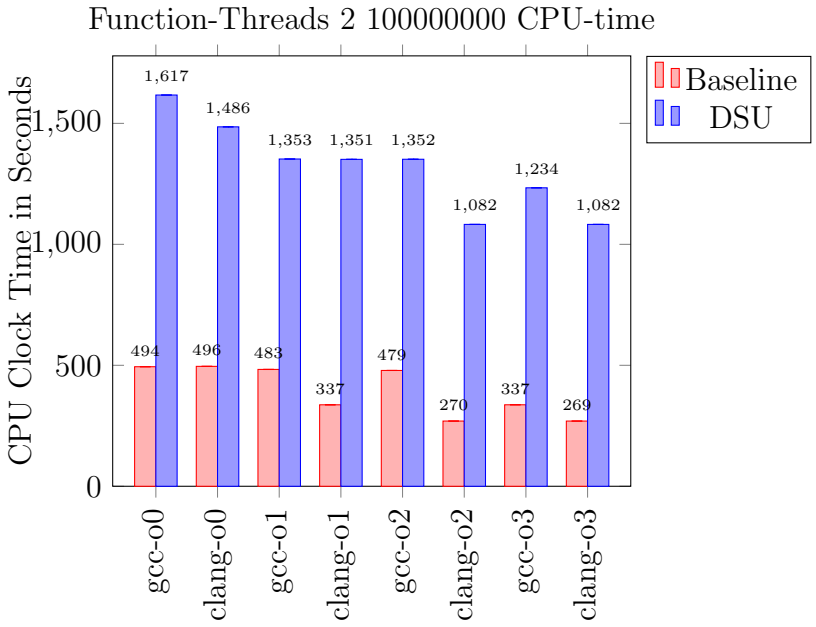
#### Experimental Design

The synthetic program to estimate multi-threaded function invocation overhead spawns multiple threads. Each of these threads run the target loop from the function invocation synthetic program described in Section 5.2.1. The variable dereferenced by the invoked function in each thread is unique to avoid interference from the memory accesses by the other threads. Each loop runs for a set number of iterations. The number of threads spawned can be set for each experiment.

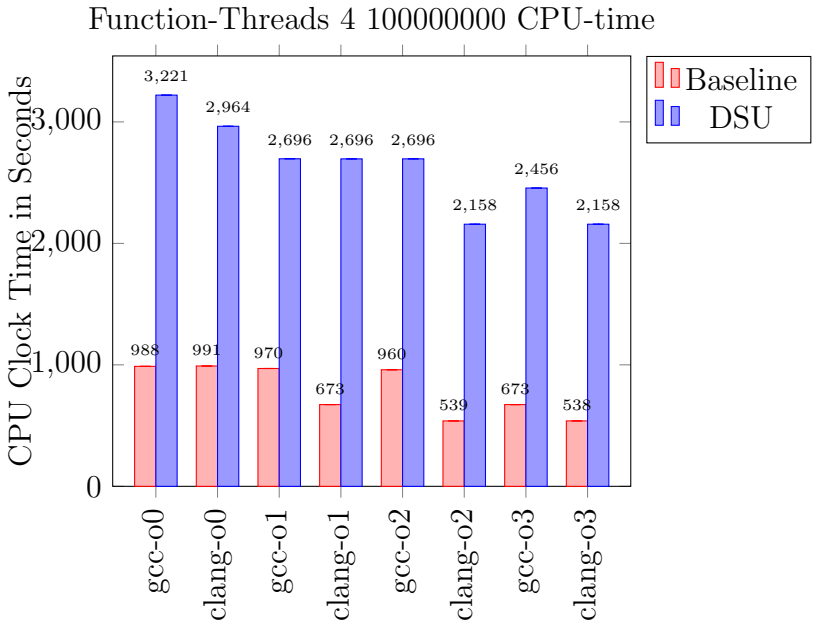
For the same reasons as the Function-Call Benchmark, function inlining is disabled. Then, different numbers of loop executions and threads are executed. By varying both of these we can see the performance impact of making DSU function calls, eliminating the startup cost as a distracting factor, and see the impact of more threads trying to run the same DSU function.

#### Results

The runtime performance of the multi-threaded version of the function-invocation performance study follows the same trend as the single-threaded version: the ratio between the baseline and the DSU versions stabilizes as the number of loop iterations increases because the impact of the startup and shutdown cost is reduced. The results of the highest number of loop iterations

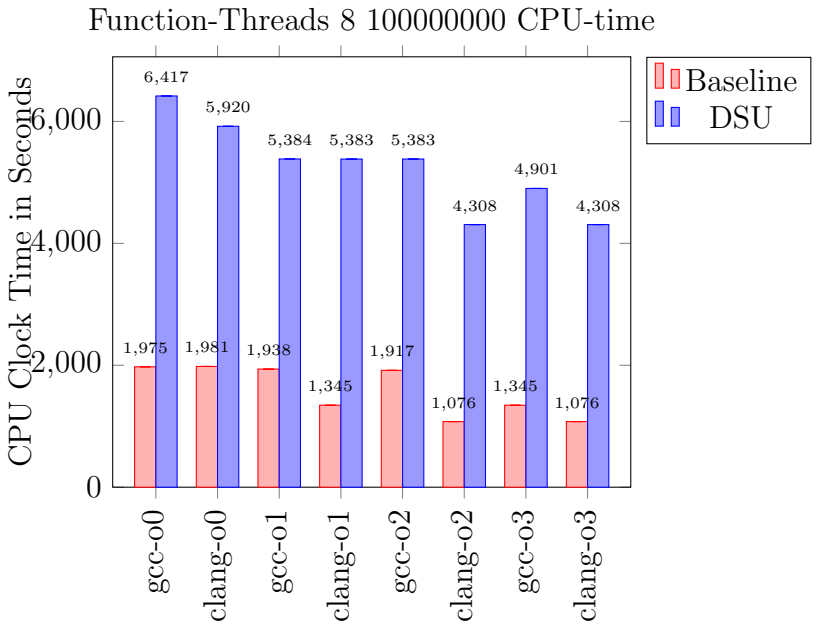



(a) Execution time of a DSU function call in a non-DSU system vs. a DSU system with 2 threads running the target function

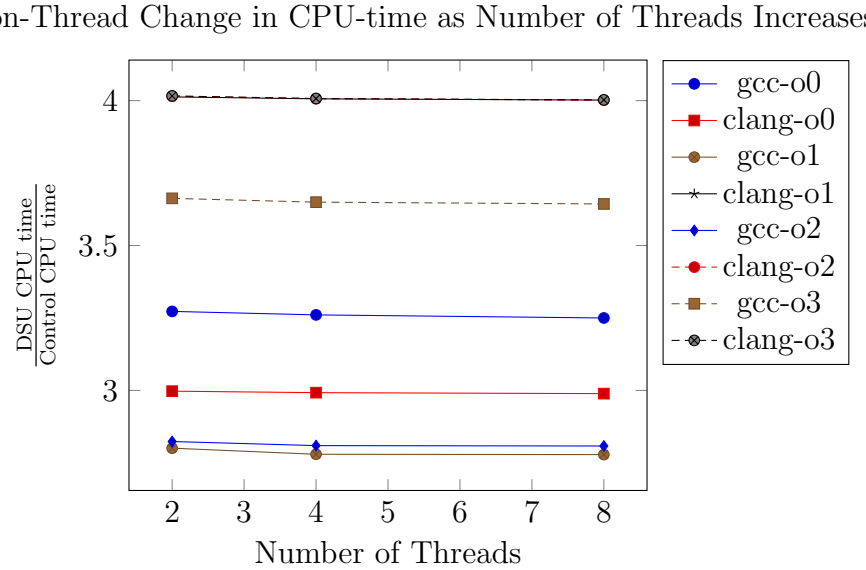



(b) Execution time of a DSU function call in a non-DSU system vs. a DSU system with 4 threads running the target function

Figure 5.3: Experimental results for Multi-Threaded Function-Call micro benchmark




(c) Execution time of a DSU function call in a non-DSU system vs. a DSU system with 8 threads running the target function




(d) Ratio of performance between DSU function calls vs. non-DSU function calls as number of threads increases

Figure 5.3: Experimental results for Multi-Threaded Function-Call micro benchmark

is included in Figures 5.3a, 5.3b, and 5.3c. If the use of multiple threads had an impact on the performance of DSU functions in PSU then the ratio of execution time would increase as the number of threads increased. Figure 5.3d shows that there is no significant change in ratio between the baseline version and the DSU version as more threads are added. Therefore, the addition of DSU functionality in multithreaded applications introduces no performance impact between threads during thread execution of DSU function. This is because in the PSU framework PSU function counters are designed to have no inter-thread interaction with each other.

### 5.2.3 Thread-Creation-Destruction Performance

The creation of a new thread in a DSU program requires the initialization of global DSU function table values and the creation and initialization of TLS variables. This TLS must also be released for reuse by the system when a thread is destroyed. These additional steps are necessary for all threads in a DSU system regardless of the presence of actual DSU functions in the program. Thus, the characterization of a multi-threaded DSU system must determine the additional overhead incurred at thread creation and destruction. A Thread-Create-Destruction synthetic program is used to estimate this overhead in the DSU system described in this thesis.

#### Experimental Design

The synthetic program is composed of a simple thread function that increments a global counter. This function is executed in a thread that is created in the body of a loop. The loop creates the thread and then waits for the thread to join back. A sufficient number of iterations of this loop are executed such that the total execution lasts for a measurable time.

The Thread-Creation-Destruction Benchmark is run at each optimization level and run with perf. The programs were executed with an increasing number of loop body iterations. The variation in loop body execution allows us to isolate the cost of performance in thread creation from the rest of the program.

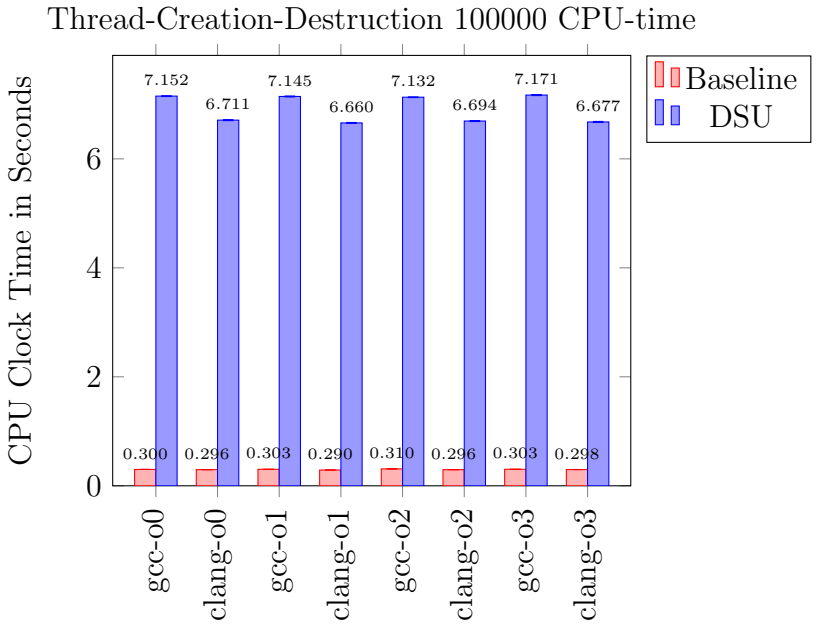



Figure 5.4: Execution time of a DSU function call in a non-DSU system vs. a DSU system.

**Results**

The additional overhead of creating a thread in a PSU application is very significant. Figure 5.4 shows that a thread created in a PSU application takes as much as 24× longer to launch. In a typical application, thread creation is already considered an expensive operation to perform. Therefore, thread creation is typically kept to a minimum and is generally performed at application startup using a technique call a Thread Pool [7]. Thus, in a typical application, the increase in thread creation cost has a minimal impact on the overall performance.

### 5.2.4 DSU Initialization Performance

DSU programs must initialize reference tables with entries for each DSU function during startup. This initialization is done prior to the main method. Each DSU function requires its own entry in the reference table. Therefore, the time it takes to initialize the reference table scales with the number of DSU functions. In order to measure this cost, we created the DSU-Initialization synthetic program.

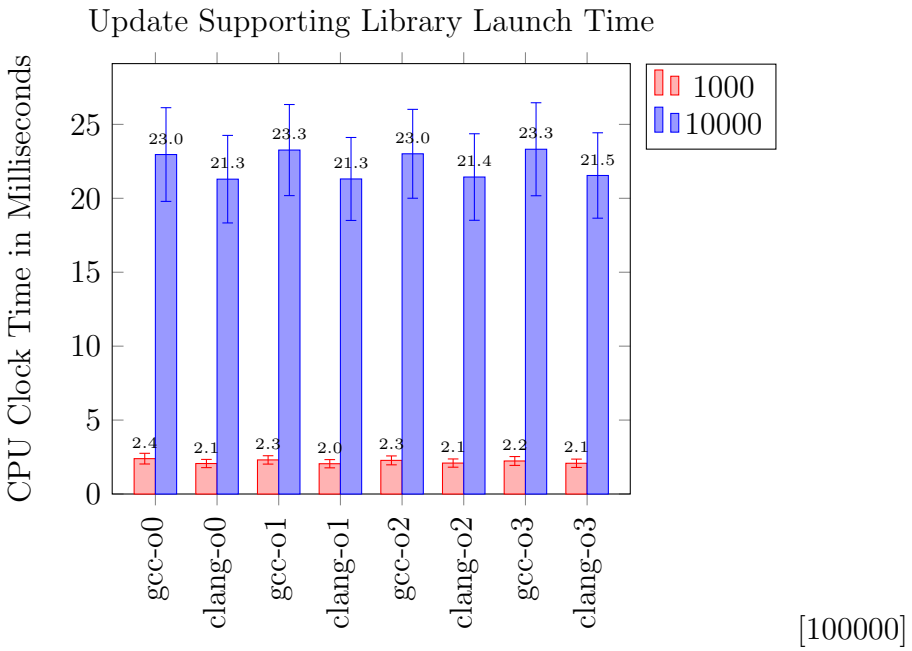



Figure 5.5: Graph of time for Update Supporting Library to initialize as number of DSU functions increases

### Experimental Design

The core of the DSU-Initialization is a simple tool created to generate C functions. These functions do little more than return a random variable. This tool is then used to generate DSU programs that have a variable number of DSU functions. Even though the main function will not actually use the DSU functions, the update supporting library will still load them before the main function executes. Thus, measuring the performance while varying the number of DSU functions that will be loaded provides insight into the load time per DSU function.

The program was compiled with an increasing number of DSU functions. An instrumented version of the update supporting library was used to record the time from the initial launch of the update supporting library to when it relinquished control to the application program. This information was then logged using the built-in PSU logging system in the update supporting library.

#### Results

Figure 5.5 shows the time, measured in milliseconds, that it takes to launch a DSU application using PSU. The number of DSU functions starts at 1000 because with fewer DSU functions the load times were negligible. This experiment shows that PSU has a negligible impact on the start time of an application with a very large number of DSU functions still only taking 0.43 seconds to launch.

### 5.2.5 DSU Patch-Load Performance

A DSU program built using PSU does not need to stop its execution to perform an update. Each function is individually loaded and made available to the executing user code when it is available. An important characteristic of a DSU system is the average time between the installation of a new version of a DSU function and the first time that an invocation of that function uses this new version. This time is called the "patch load time".

#### Experimental Design

A DSU-Patch-Load synthetic program is used to measure the patch load time. The DSU-Patch-Load synthetic program follows the methodology described for the DSU-Initialization synthetic program. It consists of a program with a variable number of DSU functions. However, instead of the empty main function used in the DSU-Initialization synthetic program, the DSU-Patch-Load synthetic program has an infinite loop in the main function. A patch is generated containing a variable number of DSU functions and the load time is measured.

Similar to the DSU-Initialization synthetic program, the DSU-Patch-Load synthetic program was compiled with an increasing number of DSU functions. The DSU library already reports into a log file when it detects a patch file as well as when it has finished loading a patch. Unfortunately, this measurement is not accurate enough for a performance evaluation. To remedy this, we created an instrumented version of the update supporting library that accurately

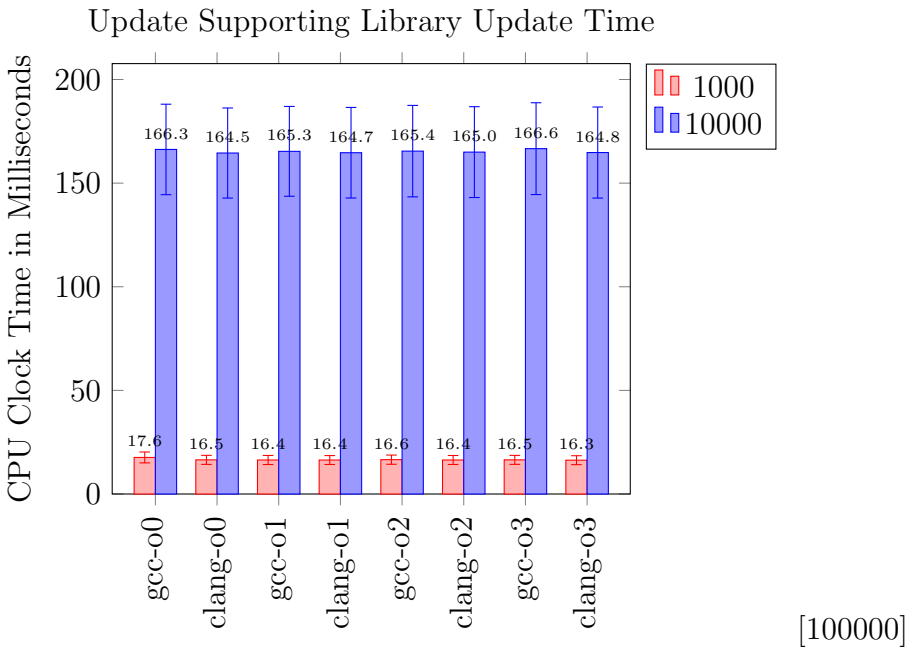



Figure 5.6: Graph of time for Update Supporting Library to load an update as number of DSU functions increases

reports the time from detection of a patch to the completion of the patch loading. The update supporting library recorded the time it took to load a full update. This information was then logged using the built in PSU logging system in the update supporting library. The experiment was run 30 times and the average update time is reported.

**Results**

Figure 5.6 shows the time in milliseconds it takes to update a DSU application using PSU. The number of DSU functions starts at 1000 for the sake of consistency with the DSU-Initialization Benchmark. This experiment shows that updates do not take very long. The extreme examples test taking 2.15 seconds at their peak. These patches take, on average, less than 0.005 milliseconds per DSU function to load. As updates are performed on separate threads concurrently with the application program threads, there is no impact on the performance of the application by performing an update.

## 5.3 MySQL Real World Application Performance

PSU introduces many steps to allow the application program become DSU. The synthetic programs described in the previous sections can be used to characterize the performance effects of these additional steps and are able to show the exact performance loss of the various subcomponent. Examined in isolation, some of these performance effects appear to be significant. However, it is interesting to study their effect on an actual application. This section describes a case study which provides evidence about such effects in a realistic application. This case study consists of transforming the MySQL database into a DSU application.

### 5.3.1 Experiment

#### Experimental Design

To simulate a typical subsystem that might be made fully DSU, the entire storage engine is changed to be DSU. The source code for MySQL was pulled from Github [18]. The program was compiled with GCC using optimization level o2 and the default MySQL build configurations provided with the MySQL sources from Github [18].

The heap storage engine in particular was chosen for this experiment. This choice was made for a few reasons. The first is that the heap storage engine is not dependent on disk access. This allows the experiment to more closely resemble the performance of a database which is already fully warmed up. Park, Do, Teletia, *et al.* show that the performance of a fully warmed up database far exceeds that of a partially warm database [19]. As a database warms up it loads frequently used queries into memory. By using the heap storage engine, which stores data exclusively in memory, the database is able to provide a closer representation to a warm database. The second reason is at the time of this work the MySQL database was transitioning to a C++ implementation. The current implementation of PSU only supports C style functions in C++. At this time the heap storage engine is one of the few

MySQL storage engines which is still implemented in C.

To measure the performance HammerDB [9] was used to run the *TPC-C* benchmark. TPC-C simulates a warehouse with a large amount of database transactions. HammerDB allows for concurrent simulated users to simultaneously interact with the database. HammerDB was configured to run a three-minute ramp-up phase prior to performing the TPC-C benchmark. The ramp-up is a sequence of database queries designed to warm up the database cache. During the ramp-up period, no performance metrics are recorded. HammerDB was scripted to run TPC-C with one, two, and four virtual users for five minutes. A virtual user is a simulated user that will run the benchmark script. This is done to assess the performance of databases with concurrent access.

The TPC-C database configuration was saved from the default values to a database dumb file and was restored prior to each benchmark run. Each configuration was run thirty times. To avoid potential biases due to system variations, a batch containing all the configurations is executed, then the configurations are randomly shuffled before running the next batch.

At the end of every run, the average Transactions Per Minute (TPM) is recorded. The TPM is dependent both on the database's implementation of the TPM counter and on the type of transactions performed. Therefore, in a typical database benchmarking, TPM is generally considered an unreliable measurement for comparison of database performance [8]. However, in this study, TPM is an appropriate metric because the comparison is between different versions of the same database rather than between two different databases. In such case the TPM is measured in the exactly the same way. Moreover, the experiments also use the same benchmark and, thus, the same transactions are performed on the database.

### Results

The results in Figure 5.7 indicate that the DSU modified MySQL database maintains 95.02% of the performance of the standard MySQL database. This is a modest reduction in performance for the functionality that is gained with a DSU system.

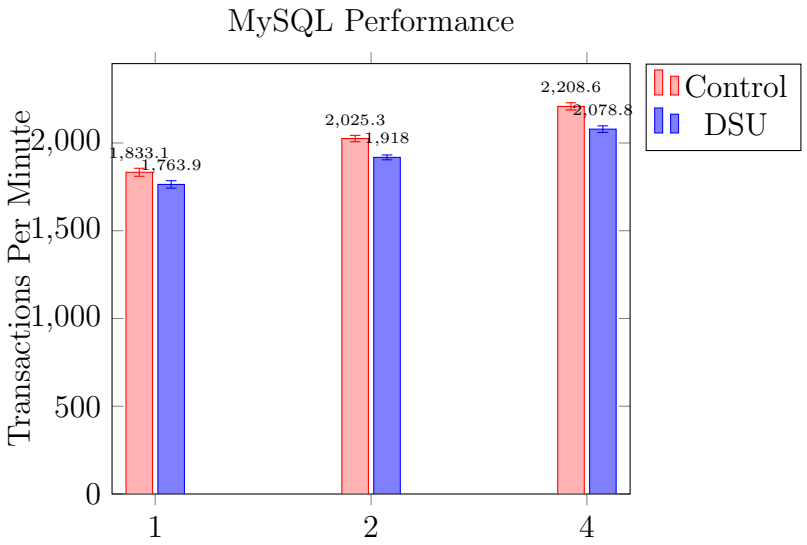



Figure 5.7: The performance of a normal MySQL database and a DSU enabled version of the MySQL database

### 5.3.2 Complexity of Use

As the MySQL experiment is an evaluation of the real world application of PSU, this section focuses on the use complexity of PSU. The necessary steps to transform the baseline version of the MySQL database to a DSU version are listed as follows: (1) a simple header file containing function declarations for each function in the heap storage engine was passed to the updating weaver, (2) the updating weaver then performed an inplace transformation on the codebase, (3) the new C source file containing the PSU patch id and PSU function list was added to the heap storage engine's CMakefile source list, (4) the PSU runtime was added to CMakefile library linkage list, (5) and the `-rdynamic` flag was added to the compiler flags.

As reduced complexity is a core design principle of PSU; each of these steps are quite trivial to perform. This simplicity allows PSU to add DSU functionality to the MySQL database's heap storage engine without overly increasing the compilation complexity. Aside from the modifications listed, MySQL was able to be compiled using its default build process.

### 5.3.3 Summary

The MySQL experiment show that in a real world scenario PSU is able to be used with a moderate 4.98% performance penalty while adding DSU functionality to the whole storage engine. This modification to the storage engine was simple to implement and allowed the majority of the MySQL source code to be left entirely unmodified. This experiment also shows that with proper placement of the DSU functions, the performance can easily be maintained.

# Chapter 6

# Related Work

There are several core design features that distinguish the various approaches to DSU. Chapter 2 outlines many of the common dimensions of DSU systems. Some of these dimensions can be characterized by the system supporting or not supporting a feature: (1) safe-point requirement, (2) multi-threading capability, (3) ability to handle recursion, (4) blocking requirement, (5) and ability to maintain multiple versions running at the same time. Other dimensions are characterized by a system falling in one amongst several possible design points: (6) the abstraction layer and requirement for regeneration of binaries to enable DSU, (7) the dynamic-update granularity, (8) and the memory impact of DSU functionality.

DSU approaches can also be distinguished by the method used to apply the update to a system. The method of updating a program determines other features of the DSU system. Table 2.1 shows an overview of the various distinguishing features of the prevalent DSU systems. The subsequent sections will discuss each in detail.

## 6.1 Update Method

Most approaches have two components to the update method: the method of loading new code and the method of inserting the new code into the running application. Similar to PSU, Ginseng [15], [17], Kitsune [10], [11], Upstare [13], POLUS [4] and Ksplice [1] use dynamic loading to insert new code into the system.

Like PSU, both Ginseng and Upstare rely on function indirection to invoke DSU functions. Therefore, they have the same benefits and hindrances as PSU relating to the impact of function calls on performance.

Kitsune requires programmers to manually insert calls to a Kitsune API, both to invoke and to update DSU functions. Kitsune depends more heavily on manual programmer intervention than PSU, it allows programmers to explicitly control how an update happens.

Ksplice [1], POLUS [4], and ISLUS [5] use binary rewriting to overwrite the code in the target application at run time. Ksplice and ISLUS overwrite a whole target function at runtime. POLUS inserts a jump to the new version of a DSU function into the start of the old version of the DSU function. Therefore, several updates to the same function in POLUS may lead to more significant performance degradation.

## 6.2 Safe-Point

In the vast majority of systems for DSU, an update may only occur when the program execution is at specific points in the code base called a safe-point. If an update is made available while the target program is not in a safe-point the target program will wait for the target program to reach a safe-point.

These safe-points may be inserted by a programmer as in Ginseng [15], [17], Kitsune [10], [11], and Upstare [13] or there can be automatically defined safe-points as in Ksplice [1].

These safe-points are necessary for these approaches to guarantee that a catastrophic failure does not occur when an update happens. A disadvantage of safe-point DSU systems is that an extensive number of safe-points might have to be inserted. Guaranteeing that a thread will not run for a long time without reaching a safe-point is, very difficult. Moreover, these systems require all threads to reach a safe-point prior to performing an update.

PSU does not rely on safe-points. The moment an update is made available, the update supporting library starts updating DSU functions in the application program. This allows PSU to quickly update a program. Existing invocations

of the updated function continue executing the previous version while all new invocations execute the updated code for the function.

## 6.3 Multi-threading

The ability of systems to handle multi-threading can be impacted by the safe-point requirement. For instance, the first implementation of Ginseng [17] was unable to handle multi-threading because it required safe-points. The complexity that was introduced in guaranteeing that all threads are in a safe-point made it difficult to implement DSU in multi-threaded applications. This limitation was not addressed for a couple of years [15]. Other DSU systems that rely on safe-points do support multi-threaded applications; however, they do have limitations.

Chen and Qiang show that Upstare suffer from potential deadlocks in multithreaded applications during the stack reconstruction phase of an application update [5]. This is due to the unique way that Upstare unwinds the stack during an update.

PSU naturally handles multi-threading by simplifying the problem to simple function indirection. The careful engineering of the PSU function counter allows PSU to handle multi-threaded applications without any increased performance overhead in comparison with a single threaded applications during thread runtime. PSU also does not have a risk of deadlocks during updates.

## 6.4 Recursion

Some DSU systems are limited in their ability to handle recursive function calls. The requirement that all threads be executing the same safe-point can lead to difficulties in handling recursion.

In Kitsune [10], [11], safe-points are inserted manually by a programmer. The instructions explicitly prohibit the programmer from placing safe-points in a recursive function. This requires programmers to be very conscientious of where they are inserting the update statements and introduces the opportunity for the insertion of difficult-to-find bugs.

Ksplice [1] is unable to handle updates to DSU functions that have active stack invocations. The consequence is that it cannot dynamically update any recursive function during the execution of the recursion.

PSU handles recursive functions by updating a pointer such that future invocations of DSU functions call the newest version of the DSU function. Existing invocations to DSU functions are allowed to execute to completion as they were invoked. Therefore, in PSU multiple versions of a recursive function may be present in an active recursion stack.

## 6.5 Blocking Requirement

In a simple implementation of a DSU system all threads must typically stop execution to avoid crashes or inconsistency in program state when an update is applied to running program. For instance, if a running thread tries to execute a piece of code while it is being updated, that thread may end up in an inconsistent state and may crash. To avoid such incorrect behaviour, most systems implement some form of thread blocking to prevent the user threads from executing code while the code is being updated. The requirement of thread blocking is strongly correlated with the requirement of safe-points.

Ginseng [15], [17] is almost able to perform updates without interrupting the executing threads by using safe-points. All user threads "check-in" with a runtime library responsible for loading in the update. When a thread reaches the end of the safe-point they must "check-out". An update only occurs when all threads have checked-in. If they reach the end of the safe-point before the update is complete, they then stall until the update completes.

Kitsune [10], [11] and Upstare [13] rely both in the use of safe-points and on thread-blocking. When an update is available, all threads will stop execution upon reaching their next safe-point. The update will only commence when all the threads have paused their execution. A flag is used to indicate that an update is available. Whenever a thread executes a safe-point, it checks the flag to determine if it should pause execution or if it should continue executing until the next safe-point.

Ksplice [1] is designed explicitly to operate within the Linux kernel. Thus it uses kernel directives such as `stop_machine` when an update needs to be performed. `stop_machine` stops all cores on the machine while it runs a specified function. In this case the specified function is the Ksplice update directive. `stop_machine` freezes not only the target application but all other applications on the machine. Thus is optimal for updating the OS Kernel as an update to the kernel typically impacts all applications in the OS. However, using `stop_machine` in a user application is not desirable because it causes unrelated applications to suspend their execution.

ISLUS [5] redirects threads into intermediary functions and blocks them while transforming the call stack of the program. ISLUS only redirects functions that will access the function being updated during the update. This allows most threads to avoid blocking.

One of PSU's most distinguishing features is that it does not require the application program to pause the execution of threads while an update is in progress. In contrast, the majority of the competing approaches require the target application to suspend execution. Therefore, in PSU, updates have less impact on the executing program.

## 6.6 Abstraction Layer and Binary Regeneration

The abstraction layer specifies how a DSU system interfaces with the target program. Most solutions rely on transforming the source code to perform DSU. Thus, they cannot modify existing binaries and must regenerate the executables. When transforming a non-DSU application into a DSU application, Ginseng [15], [17], Kitsune [10], [11], Upstare [13], Ksplice [1], and PSU must re-compile a modified application source code into a DSU binary. This requirement makes them unsuitable for applications that have already been deployed.

POLUS [4] hijacks a running application using memory-mapping tools to insert code that dynamically loads the POLUS runtime library into the run-

ning application. POLUS then uses thread traces to redirect threads into the POLUS runtime library, which is responsible for coordinating the rewriting of the binary.

ISLUS [5] uses a similar technique to POLUS to hijack the running application. Rather than inserting a runtime library as POLUS does, ISLUS directly loads new DSU code. ISLUS then inserts jump instructions into the code that is being updated but is executing in a thread. It then rewrites the binary using a semantic mapping.

The approaches used by POLUS and ISLUS has a significant advantage over PSU because those system can apply dynamic updates to existing binary files.

## 6.7 Dynamic-Update Granularity

The dynamic-update granularity is a measurement of the portion of code of an executing program that can be dynamically updated using a DSU system. An optimal system would be able to apply DSU to the whole application; thus, be able to update the largest amount of code in an executable. For a number of reasons, most existing approaches avoid whole program DSU updates.

Ginseng [15], [17], POLUS [4], Kitsune [10], [11], and PSU only allow updates to future invocations of functions. Existing invocations of functions must complete their execution with the same version of the code with which they were invoked. PSU differs from Ginseng and POLUS in that PSU is able to optionally allow some or all of the functions to be DSU. Because these approaches rely on updating only future invocations, the main function is not able to become DSU.

Ksplice [1] only allows updates of functions that do not have calls on the call stack. This prevents it from performing an update to existing invocations of DSU functions. This also prevents it from updating future invocation of DSU functions that might have a permanent function on the call stack.

Upstare [13] unwinds the call stack and reassembles it using special transformation functions. Thus, it is able to update existing calls to DSU functions

on the stack.

ISLUS's [5] use of binary rewriting and state transformation allows it to update the whole program including function calls already on the stack.

## 6.8 Multiple Versions

DSU systems can also be characterized by their ability to enable multiple versions of DSU code to exist concurrently in the system. Several approaches like Ginseng [15], [17], POLUS [4], and PSU allow multiple versions of the same function. Thus, they present a much more versatile DSU model to programmers.

Approaches such as Upstare [13], Kitsune [10], [11], and Ksplice [1] explicitly avoid having multiple versions of functions executing concurrently. This constraint allows them to maintain a consistent state but requires complex state transformers for stack variables.

ISLUS [5] runs multiple versions as part of a rollback mechanism. It combines the use of state transfers and multiple versions. Maintaining multiple versions allows it to recover in the event of a critical failure on a new version of DSU code by falling back to a previous version.

## 6.9 Memory Impact

DSU systems load new code and functionality into programs. This new code must exist in the program's memory and, unless the space used by obsolete code is reclaimed, obsolete versions will consume resources. This may result in performance degradation. Ginseng [15], [17], Kitsune [10], [11], Upstare [13], Ksplice [1], and PSU all load code through DLLs. ISLUS directly loads new code into memory using memory mapping tools. PSU is the only approach that is able to unload versions of DSU functions that are no longer necessary. This provides more efficient management of resources and allows PSU to remain memory efficient indefinitely provided that invocations to earlier versions of DSU functions eventually return.

# Chapter 7

# Caveats

There are several limitations to the types of changes to a program that could be included in an update in PSU. This section discusses some of these limitations and also provides insights into the difficulties of creating a system like PSU for object-oriented languages such as C++.

## 7.1 Updates to Function Signatures

Most changes to a function in a DSU system occur in the code within the body of the function. However, in some cases a programmer may which to change the signature of the function, which is formed by the list of arguments passed to the function and the return value. The signature includes the number of arguments, the types of the arguments and the type of the return value. In PSU programs there are three scenarios to consider for signature changes.

First, a change to the signature of a function that is invoked only in points of the call graph that are dominated by a DSU function $f$. The code invoking these functions will change to match the expected signature changes and will be compiled into the same namespace as $f$. These functions are always invoked within the same version of $f$ and therefore their signature can be changed when the version of $f$ changes.

Second, a signature change to a function that is invoked only in a point of the call graph that is not dominated by DSU functions. These functions will never be updated as there is no code from a DSU function that invokes them. This means that even if their signature is changed the code that invokes them

will never reach the new versions of the functions.

Third, a signature change to a function that is invoked in a point of the call graph that can be reached from a DSU function and in at a point in the call graph that is reached without going through a DSU function. The updated version of the function will only be used by invocations in the sections of code that followed an invocation path through a has a DSU function. Invocations through paths that do not pass through a DSU functions will always invoke the baseline version of the function, thus signature changes have no effect on them.

## 7.2 Global Constant Variables

Constant variables in PSU can only be changed if all references to the variable are in the code of functions that are invoked within a region of the call graph that is dominated by a single DSU function $f$. The new version of the constant variable is created during the compilation of the patch file for $f$. The global constant variables are encapsulated in the name space of the dynamic shared library system. In most scenarios these global constant variables are replaced with constants rather that references to global variables by the compiler.

## 7.3 Global Variables

The management of the name space that the majority of PSU relies on makes it challenging to handle global variables. Global variables may only change if they are exclusively accessed from within the DSU code and their state values can be transferred from non-DSU code to the DSU code. Developers must be cognizant of this constraint when designing their application.

A solution is to create a DSU function interface to all accesses to global variables that will be changed by updates. This DSU function interface would be responsible for converting the variable's data layout to the format that is expected by the invoking function. This conversion includes transforming structures when their definition changes. These interface functions must be DSU functions because they need to be able to handle new transformations

as the application evolves. This mechanism to handle global variables was not explored in the prototype because it may have a non-trivial performance degradation for programs that rely heavily on global variables.

## 7.4 Process Forking

The prototype for PSU does not implement the functionality to support process forking. Support for forking can be added using the same principles used to enable threading support described in Section 4.2.4. To add this support, the original fork function must be overwritten so that calls to the fork function are intercepted in a similar way to thread intercept. The *fork intercept* needs to first call the original fork function, splitting the process. Upon returning from the original fork, the fork intercept would check the process identifier to determine if the process is the original process or the forked child process. In the case of the original process, the fork intercept returns and the process continues it's execution. In the case of the child process, the fork intercept first makes a call to a update supporting library. This update supporting library resets all PSU function counters and reinitializes all TLS thread indexes. Upon completion of this initialization the thread intercept returns and the child process resumes execution.

## 7.5 Compatibility with C++

The design of PSU relies on the accessibility to pointers. While many languages have this support, this section will focus on C++ and the challenges that need to be overcome to support DSU in C++.

First, the C++ function name mangling must be either disabled or made accessible by the DSU system. The update supporting library relies on the names of functions matching supplied names. The updating weaver used to make functions DSU in PSU would need to be able to track the supplied names to the mangled names.

Second, class member functions would have to be transformed with the same trampoline layer that PSU uses in C to allow PSU to function in a

majority of cases. However, this trampoline-layer-based solution could potentially fail in the presence of class inheritance. For class inheritance to be implemented, PSU would need to be modified so that the behaviour of class inheritance of DSU class member functions are specified. The current design of PSU does not contemplate class inheritance. Solutions to incorporate class inheritance could require all classes to have their own version of a DSU member function even in inherited classes or could require all the inherited DSU functions to use the same version. Both of these solutions have potential benefits to be explored. The first provides a fine grained update mechanic, while the second is simpler to reason about.

Third, changes to class definitions in C++ in PSU present a difficult challenge. When a class signature is changed, it must be interpreted throughout the call tree. If the data layout of a class's member variables are changed then the class cannot be correctly interpreted by code from different patch versions.

The PSU prototype can handle C-style functions in C++. While this support is a promising first step, there are many other challenges besides those listed in this section. Thus, creating a system like PSU for languages like C++ will require substantial additional research and development.

# Chapter 8

# Conclusion

This thesis presents Portable Software Update a framework for building DSU applications in C. While PSU is not the first framework for building DSU applications, it presents several key new advantages over previous works.

PSU performs DSU updates to applications through a simple function indirection framework. New versions of DSU functions are loading into a running application using Dynamically Linked Libraries. Obsolete versions of DSU functions in a PSU application are then able to execute to completion and are able to be unloaded from memory once they are no longer executing. Combined with the updating weaver, PSU drastically simplifies the process of building DSU applications; thus, reducing the likelihood of a programmer error.

The design of PSU allows for it to be easily ported to a wide range of systems. This portability comes from PSU's exclusive use of standard functions as well as library functionality that is readily available in most operating systems.

PSU is evaluated using several custom synthetic benchmark programs as well as the MySQL database. The findings from these experiments shows that the PSU framework is able to enable DSU functionality in real world applications with acceptable performance levels.

# Bibliography


[1] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates", in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09, Nuremberg, Germany: ACM, 2009, pp. 187–198, ISBN: 978-1-60558-482-9. DOI: 10.1145/1519065.1519085. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519085. 7–9, 55, 56, 58–61

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility", *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009, ISSN: 0167-739X. DOI: 10.1016/j.future.2008.12.001. [Online]. Available: http://dx.doi.org/10.1016/j.future.2008.12.001. 1

[3] C. Cérin, C. Coti, P. Delort, F. Diaz, M. Gagnaire, Q. Gaumer, N. Guillaume, J. Lous, S. Lubiarz, J. Raffaelli, *et al.*, "Downtime statistics of current cloud solutions", *International Working Group on Cloud Computing Resiliency, Tech. Rep*, 2013. 1, 2

[4] H. Chen, J. Yu, C. Hang, B. Zang, and P.-C. Yew, "Dynamic software updating using a relaxed consistency model", *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 679–694, Sep. 2011, ISSN: 0098-5589. DOI: 10.1109/TSE.2010.79. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.79. 10, 55, 56, 59–61

[5] Z. Chen and W. Qiang, "Islus: An immediate and safe live update system for c program", in *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, Jun. 2017, pp. 267–274. DOI: 10.1109/DSC.2017.50. 4, 8–10, 56, 57, 59–61

[6] (2019). GCC GNU online docs: 6.63 thread-local storage, [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Thread-Local.html (visited on 02/13/2019). xiii

[7] B. Goetz. (2002). Java theory and practice: Thread pools and work queues, [Online]. Available: https://www.ibm.com/developerworks/java/library/j-jtp0730/index.html (visited on 03/05/2019). 47

[8] HAMMERDB. (2018). Comparing HammerDB results, [Online]. Available: https://www.hammerdb.com/docs/ch03s04.html (visited on 10/20/2018). 52

[9] ——, (2018). *HammerDB*. version 3.1, [Online]. Available: https://www.hammerdb.com/index.html (visited on 10/20/2018). 39, 52

[10] C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C", *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, 13:1–13:38, Oct. 2014, ISSN: 0164-0925. DOI: 10.1145/2629460. [Online]. Available: http://doi.acm.org/10.1145/2629460. 7–9, 55–61

[11] C. Hayden, E. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for C", vol. 47, Oct. 2012, pp. 249–264. DOI: 10.1145/2384616.2384635. 7–9, 55–61

[12] C. F. Kemerer, "Software complexity and software maintenance: A survey of empirical research", *Annals of Software Engineering*, vol. 1, no. 1, pp. 1–22, Dec. 1995, ISSN: 1573-7489. DOI: 10.1007/BF02249043. [Online]. Available: https://doi.org/10.1007/BF02249043. 2

[13] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction", in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX'09, San Diego, California: USENIX Association, 2009, pp. 31–31. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855807.1855838. 4, 7, 8, 55, 56, 58–61

[14] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture", *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014. xiii

[15] I. Neamtiu and M. Hicks, "Safe and timely updates to multi-threaded programs", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 13–24, ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542479. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542479. 7–9, 55–61

[16] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming", in *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '08, San Francisco, California, USA: ACM, 2008, pp. 37–49, ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328447. [Online]. Available: http://doi.acm.org/10.1145/1328438.1328447. 7

[17] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C", in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, Ottawa, Ontario, Canada: ACM, 2006, pp. 72–83, ISBN: 1-59593-320-4. DOI: 10.1145/1133981.1133991. [Online]. Available: http://doi.acm.org/10.1145/1133981.1133991. 5, 7–9, 55–61

[18] M. D. team Oracle. (2017). MySQL github repository. version 5.7 commit 8d6b0863, [Online]. Available: https://github.com/mysql/mysql-server. 39, 51

[19] K. Park, J. Do, N. Teletia, and J. M. Patel, "Aggressive buffer pool warm-up after restart in sql server", in *2016 IEEE 32nd International Conference on Data Engineering Workshops (ICDEW)*, May 2016, pp. 31–38. DOI: 10.1109/ICDEW.2016.7495612. 2, 51

[20] (2018). *Perf performance monitor*. version 4.4.155, [Online]. Available: https://pkgs.org/download/perf (visited on 03/05/2019). 40

[21] F. Pizlo, E. Petrank, and B. Steensgaard, "A study of concurrent real-time garbage collectors", in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 33–44, ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375587. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375587. 21

[22] K. Takehiro. (2017). PLTHook github repository, [Online]. Available: https://github.com/kubo/plthook (visited on 08/24/2017). 12

[23] T. C. Team. (2017). Clang 8 documentation LibTooling, [Online]. Available: https://clang.llvm.org/docs/LibTooling.html (visited on 08/24/2017). 34

[24] D. Zou, H. Wang, and H. Jin, "StrongUpdate: An immediate dynamic software update system for multi-threaded applications", in *Human Centered Computing*, Q. Zu, B. Hu, N. Gu, and S. Seng, Eds., Cham: Springer International Publishing, 2015, pp. 365–379, ISBN: 978-3-319-15554-8. 4