© 2011 Ryan M. Lefever

DIVERSE PARTIAL MEMORY REPLICATION

BY

RYAN M. LEFEVER

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair
Associate Professor Vikram S. Adve
Professor David M. Nicol
Associate Professor Sanjay J. Patel

Abstract

An important approach for software dependability is the use of diversity to detect and/or tolerate errors. We develop and evaluate an approach for automated program diversity called Diverse Partial Memory Replication (DPMR), aimed at detecting memory safety errors. DPMR is an automatic compiler transformation that replicates some subset of an executable's data memory and applies one or more diversity transformations to the replica. DPMR can detect any kind of memory safety error in any part of a program's data memory. Moreover, DPMR is novel because it uses partial replication within a single address space, replicating (and comparing) only a subset of a program's memory. We propose and evaluate two strategies for handling pointers stored in memory, a key challenge to DPMR. We also perform a detailed study of the diversity mechanisms and state comparison policies in DPMR (a first of its kind for such diversity approaches), which is valuable for exploiting the high flexibility of DPMR. Finally, we explore the use of Data Structure Analysis to eliminate nearly all restrictions on input programs that would otherwise be necessary.

To my parents, Michael and Joyce, my brother, Darin, and my sister, Shannan,
for their unwavering love and support

Acknowledgments

Graduate school has been quite a journey for me. It's not a journey that I planned to take, but it's a journey upon which I am glad I embarked. Along the way, many people have supported me, and I would like to thank them. First, I would like to thank my advisers, Professors William H. Sanders and Vikram S. Adve, for their support and guidance. I was fortunate to have two such advisers. It was a great opportunity to learn from two experts with different perspectives and different technical backgrounds. Both have shaped who I am as a researcher and as a person. I will always be grateful for their trust, respect, and encouragement.

I would like to thank Professors David M. Nicol and Sanjay J. Patel for serving on my Ph.D. committee. Their insights and feedback have unquestionably strengthened my research.

I am thankful to Jenny Applequist for her help in editing not only this dissertation but all of my papers while I was in graduate school. She is extremely talented and has greatly improved the quality of my work.

I would like to thank the funding agencies that financially supported my research. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-04-06351, CNS-06-15372, and 0086096. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. The research reported was also sponsored by the Air Force Research Laboratory (AFRL), Motorola, and AT&T.

While in graduate school, I had the good fortune of working in the PERFORM research group. Foremost, I would like to thank its members for their friendship. The PERFORM group is filled with many talented people, and the discussions I had with its members were invaluable. I would like to thank the following members: Kaustubh Joshi, Eric Rozier,

Sankalp Singh, Shravan Gaonkar, Adnan Agbaria, Robin Berthier, Ramesh Chandra, Shuyi Chen, Amy Christensen, Graham Clark, Tod Courtney, Michel Cukier, David Daly, Nathan Dautenhahn, Daniel Deavours, Salem Derisavi, Jay Doyle, Doug Eskins, Michael Ford, Mark Griffith, Vishu Gupta, Michael Ihde, Ken Keefe, Sudha Krishnamurthy, Vinh Lam, Elizabeth LeMay, James Lyons, Jason Martin, Michael McQuinn, Prashant Pandey, Hari Ramasamy, Jennifer Ren, Paul Rubel, Mouna Seri, Fabrice Stevens, Aaron Stillman, Patrick Webster, and Saman Aliari Zonouz.

I would like to thank my parents, Michael and Joyce, and my brother and sister, Darin and Shannan, for their love and support. They have been there for me through all of my ups and downs. I never question their support, which is extremely empowering. I have also been blessed with amazing friends that support me just like my family.

Last but certainly not least, I want to thank God, without whom none of this would be possible.

Table of Contents

List of Tables viii

List of Figures ix

List of Abbreviations xi

List of Symbols xii

Chapter 1 Introduction 1
- 1.1 Proposed Approach 2
- 1.2 Diverse Partial Replication 4
- 1.3 Scope of DPMR 5
- 1.4 Terminology 7
- 1.5 Related Work 8
- 1.6 Contributions 17
- 1.7 Organization 18

Chapter 2 Diverse Partial Memory Replication 20
- 2.1 Partial Replication 21
- 2.2 Shadow Data Structures 23
- 2.3 Augmented Functions 29
- 2.4 Code Transformation 32
- 2.5 Detection Conditions 42
- 2.6 Diversity Transformations 45
- 2.7 State Comparison Policies 47
- 2.8 External Code 49
- 2.9 Limitations 51

Chapter 3 Construction and Evaluation of Diverse Partial Memory Replication . . . 54
- 3.1 Details Pertaining to C 54
- 3.2 Tool Design 62
- 3.3 Experimental Framework 63
- 3.4 Fault-Injection Framework 64
- 3.5 Variant Builds 66
- 3.6 Evaluation Metrics 68

3.7 Diversity Results 71
3.8 State Comparison Policy Results 75

Chapter 4 Mirrored Data Structures 82
4.1 The Mirrored Data Structures Approach 82
4.2 Code Transformations for Mirrored Data Structures 84
4.3 External Code under Mirrored Data Structures 89
4.4 Limitations of Mirrored Data Structures 90
4.5 Mirrored Data Structures Results 91

Chapter 5 Scope Expansion Through Static Analysis 102
5.1 Data Structure Analysis 102
5.2 Behavior Forbidden by Mirrored Data Structures 106
5.3 Eliminating Limitations 108
5.4 Using Data Structure Analysis with External Code Support Libraries 111
5.5 Unknown DSA Behavior 112
5.6 Remaining Issues 113

Chapter 6 Future Possibilities 116

Chapter 7 Conclusion 118

References 120

Author's Biography 126

List of Tables

2.1 Shadow type definition . . . 25
2.2 Examples of shadow types . . . 26
2.3 Augmented type definition . . . 29
2.4 Example of an augmented type . . . 30
2.5 Composed type definition for $(\mathbf{st} \circ \mathbf{at})(t)$ . . . 34
2.6 DPMR transformation – added behavior . . . 35
2.7 DPMR transformation – transformed behavior . . . 36
2.8 Diversity transformations . . . 46
2.9 Temporal load-checking transformation . . . 48

3.1 Testbed specifications . . . 64
3.2 Measurement components . . . 69
3.3 Mean time to detection of diversity transformations . . . 76
3.4 Mean time to detection of state comparison policies . . . 81

4.1 MDS augmented type definition . . . 83
4.2 Example of an MDS augmented type . . . 83
4.3 MDS transformation – added behavior . . . 84
4.4 MDS transformation – transformed behavior . . . 85
4.5 Mean time to detection of diversity transformations under MDS . . . 97
4.6 Mean time to detection of state comparison policies under MDS . . . 100

List of Figures

1.1 Example motivating the use of diversity . . . 3
1.2 DPR applied to a race condition . . . 6

2.1 Implicit diversity . . . 22
2.2 Replication using non-comparable pointers . . . 23
2.3 Replication making pointers in memory comparable . . . 24
2.4 Replication using shadow data structures . . . 24
2.5 Algorithm for getShadowType . . . 28
2.6 Algorithm for getAugType . . . 31
2.7 Algorithm for getAugFunTypeImpl . . . 32
2.8 Algorithm for getShadowAugType . . . 33
2.9 Transformation of `createNode()` . . . 40
2.10 Transformation of `getSum()` . . . 41
2.11 External function wrapper for `strcpy()` . . . 50

3.1 DPMR memory for command-line arguments . . . 55
3.2 Disambiguated call type . . . 57
3.3 Declaration of the `qsort()` external function wrapper . . . 62
3.4 DPMR tool chain . . . 63
3.5 Compilation strategy . . . 67
3.6 Mean heap array resize coverage of diversity transformations . . . 72
3.7 Mean immediate free coverage of diversity transformations . . . 72
3.8 Mean heap array resize conditional coverage of diversity transformations . . . 73
3.9 Mean immediate free conditional coverage of diversity transformations . . . 73
3.10 Overhead of diversity transformations . . . 75
3.11 Mean heap array resize coverage of state comparison policies . . . 77
3.12 Mean immediate free coverage of state comparison policies . . . 77
3.13 Mean heap array resize conditional coverage of state comparison policies . . . 78
3.14 Mean immediate free conditional coverage of state comparison policies . . . 78
3.15 Overhead of state comparison policies . . . 79
3.16 Exploiting periodicity to improve temporal load-checking overhead . . . 80

4.1 MDS transformation of `createNode()` . . . 88
4.2 MDS transformation of `getSum()` . . . 88
4.3 Side-by-side diversity transformation overheads of SDS and MDS . . . 92

4.4 Side-by-side comparison policy overheads of SDS and MDS 93
4.5 MDS overhead of diversity transformations 94
4.6 MDS overhead of state comparison policies 95
4.7 Mean MDS heap array resize coverage of diversity transformations 95
4.8 Mean MDS immediate free coverage of diversity transformations 96
4.9 Mean MDS heap array resize conditional coverage of diversity transformations 96
4.10 Mean MDS immediate free conditional coverage of diversity transformations 97
4.11 Mean MDS heap array resize coverage of state comparison policies 98
4.12 Mean MDS immediate free coverage of state comparison policies 99
4.13 Mean MDS heap array resize conditional coverage of state comparison policies 99
4.14 Mean MDS immediate free conditional coverage of state comparison policies 100

5.1 Examples of pointer-to-int and int-to-pointer behavior 104
5.2 Reachable object 105
5.3 Result of storing a pointer masquerading as an integer 107
5.4 Situation that could lead to an update omission 107
5.5 DS graph for $F1$ 109
5.6 DS graph for $F2$ 110
5.7 Algorithm for markX() 110
5.8 Candidate for inlining 114

List of Abbreviations

DPMR	Diverse Partial Memory Replication
DPR	Diverse Partial Replication
DSA	Data Structure Analysis
MDS	Mirrored Data Structures
NSCP	N self-checking programming
NVP	N-version programming
NVS	N-version software
NSOP	Next shadow object pointer
ROP	Replica object pointer
SDS	Shadow Data Structures

List of Symbols

type() Function that returns the type of a value stored in a virtual register.

sizeof() Function that returns the number of bytes of memory that are reserved when the input type is allocated. The number of reserved bytes includes any alignment padding.

st() Function that maps types to shadow types.

at() Function that maps types to augmented types.

rpt() Function that maps types to replica parameter types.

spt() Function that maps types to shadow parameter types.

$I()$ Mapping from true and false to 1 and 0.

$\phi()$ Function that converts structure indices to shadow structure indices.

$\gamma()$ Mapping of a virtual register to a set of virtual registers for use in function declarations and function calls.

$\pi()$ Mapping of a function F's return value type to parameters for use by the augmented version of F.

$\sigma()$ Function that maps a type t to a structure that is only composed of scalar types and is structurally equivalent to t.

Chapter 1

Introduction

In recent years, the dependability of computer systems has become a major concern among research communities, businesses, and the public at large. There are many reasons for the increased attention. First, computers are becoming ubiquitous in many areas of society. Computers are relied on for control of aerospace systems, communication, storage of business data, business sales, publications, and entertainment, among other things. The increased usage has led not only to an increase in the number of failures but also to prolific consequences. Computer failures are not just an inconvenience; they can result in loss of life, money, time, and irreplaceable information. Second, particularly with the advent of the Internet, dependability has had to face a new source of failures: malicious entities. Statistics from the CERT Coordination Center [1], a center for Internet security expertise, show that the number of reported vulnerabilities topped 5,500 every year from 2005 to 2007,[1] a tremendous increase from 171 in 1995. Although attacks have always been possible, the Internet has brought about an omnipresent exposure to them. A third reason that dependability has gained recent attention is that applications have become increasingly complex as the use of computers expands. The increase in complexity has led to an increase in failures. Finally, we have seen Moore's law hold for decades, resulting in dramatic improvements in hardware performance and capacity. With such improvements, computer systems can now afford to devote resources to goals such as dependability, while still achieving their primary functionality at acceptable performance levels.

Early dependability research focused on hardware faults with a good deal of success, through a myriad of techniques that included formal methods, self-checking components, and redundancy, such as in TMR systems. Software dependability, grown out of hardware

[1] 2007 is the last fully reported year.

dependability, has enjoyed some success with techniques such as checkpoint-rollback, process replication, exception handling, and various forms of reboot. However, there is still much more to be done, as evidenced by notable software failures that have cost individual companies millions, and in some cases billions, of dollars [2]. A 2002 study by the National Institute of Standards and Technology (NIST) estimated that software errors cost the U.S. nearly 60 billion dollars annually [3]. Another study estimates that 40% of unplanned application downtime can be attributed to application failures [4].

Mismanaged memory in unsafe programming languages, such as C and C++, is a notorious source of errors. Mismanaged memory can lead to buffer overflows, dangling pointers, and uninitialized reads. Like other software bugs, those errors can result in crashes and incorrect output. In addition, memory errors are notorious for opening vulnerabilities that attackers can use to compromise a system. Many buffer overflows have made their way into production systems and lend themselves to being exploited in a variety of ways [5]. In more recent times, exploitable dangling pointers have shown up in applications such as the CVS revision control system [6], the MIT Kerberos authentication daemon [7], the Opera web browser [8], and the IIS web server [9].

Software memory errors have been difficult to detect, diagnose, and correct. Comprehensive static analysis is intractable, and practical static analysis and testing have proven to be insufficient for eliminating memory errors. For some systems, type-safe languages, like Java, are appropriate. However, C and C++ are still widely popular and appropriate for certain systems because of their performance, flexibility, and compatibility. In addition, large amounts of legacy code are written in C and C++. Therefore, there remains a need to develop techniques to detect memory errors in systems built with unsafe languages.

1.1 Proposed Approach

In this dissertation we contend that diversity and replicated runtime execution can be utilized in an automated way to detect software memory errors in an effective and tunable manner. In order to evaluate that thesis, we propose and develop an approach called *Diverse Partial Memory Replication (DPMR)* that utilizes diversity and replicated runtime

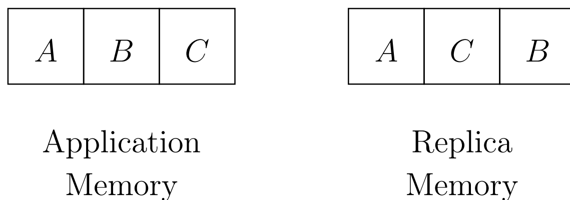

Figure 1.1: Example motivating the use of diversity

execution to detect software memory errors. More specifically, DPMR is an automatic compiler transformation that replicates a subset of an application's data memory and applies a diversity transformation to the replica. The goal of the diversity is to cause memory errors to manifest differently in application memory and replica memory. Errors are detected by comparing application memory and replica memory. In order for such a system to work, the diversity transformation is designed such that the states of the application memory and replica memory do not diverge under error-free execution.

To better understand the premise behind DPMR, consider the memory layouts in Figure 1.1. In the figure, the original memory subsystem contains objects A, B, and C, laid out in that order. The replica memory subsystem, under a diverse environment, has the same objects, but they are laid out in the order of A followed by C followed by B. If the original program contained a buffer overflow out of object A, it would corrupt object B in the original memory subsystem and object C in the replica memory subsystem. If we later compared the value of object B (or object C) in the original memory to that of object B (or object C) in the replica memory, we would detect the occurrence of a memory error.

Unlike traditional replication approaches, DPMR utilizes a *partial replication* approach that limits replication to an application's memory subsystem. By doing so, DPMR eliminates replication of non-memory computations. The results of those computations are stored in both the original memory and the replica. To make the approach efficient and correct, DPMR combines both the original behavior and the replica in a single process. That is a key difference from previous systems, which generally use separate OS processes for the original and the replica behavior.

Another important aspect of DPMR is tunability, and that tunability is one focus of this dissertation. In particular, we analyze how design decisions, diversity transformations, and

state comparison policies affect the performance and dependability attributes of DPMR. The study is important because different applications may perform better under different designs, diversity transformations, and state comparison policies. Furthermore, different deployment environments may have different performance and dependability requirements. For example, it may be desirable to spend more resources on dependability when a new version of an application is deployed than when an older, trusted version is deployed. In addition, different industries have different requirements for their computer systems. A web server handling financial transactions likely has stronger dependability requirements than a web server providing sports news.

1.2 Diverse Partial Replication

DPMR falls under a broader category of strategies that we term *Diverse Partial Replication (DPR)* strategies. As with DPMR, DPR replicates the portion of a system that is relevant to the target fault model and diversifies the replica. The intended purpose of the diversity is to cause errors to manifest differently in the original program behavior and in the replica behavior, while not causing differences between the states of the original program and the partial replica, under error-free execution.

Three important qualities of DPR make it an attractive dependability strategy in general. First, DPR can detect deterministically activated faults that make it into production systems. A *deterministically activated fault* is a fault F in a program P, defined such that if F can manifest as an error when input set I is applied to P, then F will always manifest as an error when I is applied to P. While deterministically activated faults may be easier to fix because they can be predictably repeated in a debugging environment, they have proven to be extremely difficult to detect once they have made it into production systems. Traditional dependability approaches such as process replication, checkpoint-rollback, and reboot cannot handle deterministically activated faults because they rely on multiple executions of the same piece of code in the same environment. Unfortunately, deterministically activated faults manifest the same way in each of those executions, rendering the techniques ineffective.

Second, DPR is tunable, making it adaptable for the different requirements of different deployment scenarios. In Section 1.1, we discussed using interoperable diversity transformations and interoperable state comparison policies as one method of tunability. Another strategy for tunability is to design partial replicas based on component priorities. Components of a system that are affected by the fault model of concern can be prioritized by factors such as susceptibility to the fault model, cost to deploy DPR, and the detection capability afforded by DPR. Then, given finite resources, DPR can be optimized so that it replicates high-priority components before lower-priority components.

The third attractive property of DPR is that it is a strategy that can be applied to *arbitrary* fault models. In this dissertation, we focus on memory errors; however, DPR could be used to detect errors such as race conditions and violations of library assumptions. To illustrate the flexibility of DPR, we will briefly examine a race condition example.

Consider a banking system in which incoming requests, such as a deposit, withdraw, or balance check, are placed in a queue that is serviced by several worker threads. The system specification requires that requests made to the same account be processed in the order in which they arrive. It is the bank's policy to charge overdrawn accounts a $15 penalty. Let's say an account owner has a balance of $100. She makes a request X to deposit a $200 check into her account, followed by a request Y to withdraw $250 in cash from her account. In a faulty implementation, the system may have a race condition that allows requests made to the same account to be processed out of order. Figure 1.2(a) demonstrates how such a fault could manifest as an error. To detect such an error, DPR could be used to replicate the threaded execution as well as the data upon which it operates. Then, a diversified scheduler could be used to produce the replica execution shown in Figure 1.2(b). The error could be detected by comparing the account balances at the ends of the original faulty execution and the diverse replica execution.

1.3 Scope of DPMR

We intend for DPMR to be used for both production systems and debugging environments. Such a scope is viable due to the tunability of DPMR. DPMR's dependability focus is the

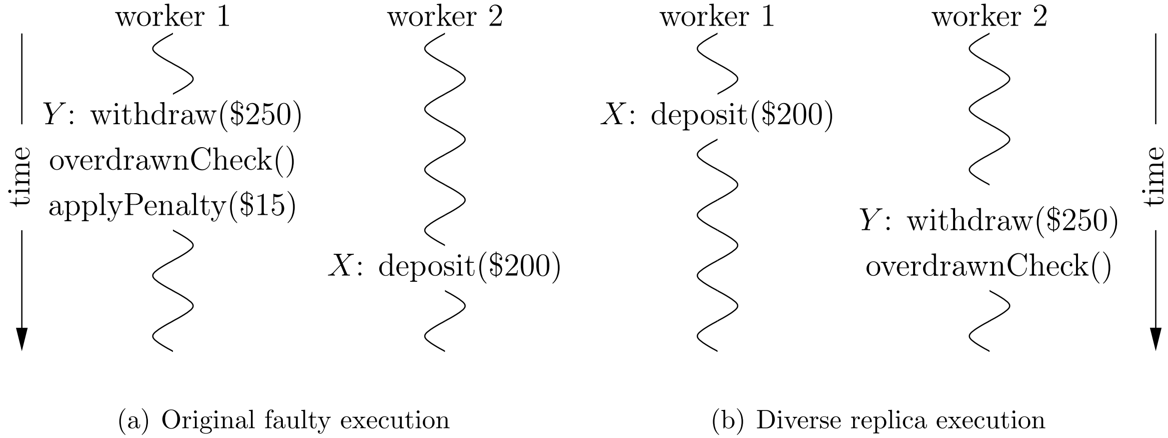

Figure 1.2: DPR applied to a race condition

detection of spatial and temporal software memory errors in all segments of application memory, including the stack, the heap, and global variables. However, it is important to recognize that DPMR also detects hardware memory errors. The ability to handle hardware memory errors is important because it allows us to deploy a single strategy to handle hardware and software memory errors, which can lead to easier-to-manage systems, and cost reductions.

Below we define the classes of software memory errors that DPMR detects.

- *out-of-bounds access*: An out-of-bounds access occurs when a read, write, or free intended for an allocated memory buffer O occurs outside the valid bounds for such an operation. Valid reads and writes to O are bounded by the memory occupied by O. Valid frees are constrained to pointers that point to the start of O. Out-of-bounds writes are typically called *buffer overflows*. Out-of-bounds frees are a subset of what is sometimes referred to in the literature as *invalid frees*.

  The intended target of an out-of-bounds read or write can live in global variables, stack memory, or heap memory, while the target of an out-of-bounds free must reside on the heap. An out-of-bounds write, i.e., a buffer overflow, can be damaging because it can corrupt memory that it is not intended to modify. An out-of-bounds read is dangerous because it can read arbitrary data values. Out-of-bounds frees can result in immediate crashes, memory corruption, and premature deallocation of memory.

- *dangling pointer use*: Dangling pointers are pointers to memory buffers that are no longer valid because they have been deallocated. Dangling pointers are restricted to the stack and the heap, since global variables are never deallocated. Usage of dangling pointers becomes dangerous if the referenced memory has been reallocated to hold other data. Dangling pointer writes, reads, and frees have similar consequences to out-of-bounds writes, reads, and frees. In the literature, a dangling pointer free is sometimes referred to as a *double free*.

- *wild pointer use*: A *wild pointer* is a pointer that is not derived from a valid allocation source. Valid allocation sources include `malloc`, `alloca`, and global variables. Uninitialized pointers are a common source of wild pointers. Wild pointers can point to an arbitrary memory address. The use of a wild pointer can result in the same negative consequences as an out-of-bounds access or a dangling pointer use.

- *uninitialized read*: An uninitialized read occurs when the value of an object is read from memory before that object has ever been given an initial value. Uninitialized reads can occur to global variables, stack memory, and heap memory. Uninitialized reads are dangerous because the read value is arbitrary.

There is one class of software memory errors that is not detectable by DPMR. It consists of memory leaks. Memory leaks occur when memory objects are no longer used by a program but are not deallocated. Memory leaks can degrade performance. However, they do not alter program behavior until the memory address space has been exhausted, at which point the program is either informed or aborted. In that sense, memory leaks are self-detectable.

1.4 Terminology

In this section, we define some general terminology that will be used throughout this dissertation.[2]

[2] The definitions for *failure*, *fault*, and *error* are based on those in [10].

- *failure*: A failure occurs when realized system behavior deviates from correct behavior. Correct behavior is defined on a system-by-system basis. A system that produces incorrect output is considered to have failed. A system that crashes may or may not be considered a failure, depending on the system specification.

- *error*: An error is a deviation in system state that could lead to a failure. For example, the execution of a buffer overflow, i.e., a memory write beyond the bounds of the intended memory object, constitutes an error.

- *fault*: A fault is the potential cause of an error. In a C program, a `malloc` call instruction that does not always allocate a buffer large enough to hold the data it is intended to hold is a fault.

- *process replication*: Process replication is a form of replicated execution. Process replication involves simultaneous execution of two or more copies of a process P, with the same input, whose goal is to produce the results of one non-faulty copy of P. The copies of P execute from copies of code that are identical except in auxiliary functionality required for simultaneous execution. The salient properties of process replication are (1) that replicas produce the same output given the same input, in the absence of environment faults, and (2) that replicas are full copies of P.

- *replica*: In this dissertation, unless qualified more specifically, the term *replica* refers to a duplicate of a software component. That component may comprise an entire system or one piece of a system. We often use the term to refer to a block of memory whose purpose is to hold the same data as an application block of memory.

1.5 Related Work

We now discuss work related to this dissertation. In particular, we consider two areas of research: software memory safety and software diversity.

1.5.1 Software Memory Safety

As discussed at the beginning of this chapter, mismanaged memory is a serious threat to software dependability. Therefore, software memory safety has garnered much attention among researchers. Below, we discuss five categories of software memory safety research. We focus our discussion on techniques that address both spatial and temporal memory errors.

Runtime Checking Most memory safety approaches maintain metadata and use it to detect memory errors by performing checks at runtime. Two of the most well-known of those approaches are Valgrind's Memcheck [11] and Purify [12]. Both tools can detect the same types of memory errors as DPMR, as well as memory leaks. Memcheck uses dynamic binary instrumentation to track which bits in memory have valid data values and which bytes have been allocated. Purify instruments compiled object code to maintain a two-bit state for each byte in memory. The state indicates whether a byte is allocated and, if so, whether the byte is initialized. Both tools check metadata at memory accesses to determine if an access is valid. Memcheck and Purify are known as debugging tools because of the large overheads that they can incur. In addition, neither tool is able to detect dangling pointers to reallocated memory or out-of-bounds pointers to valid memory, both of which are error classes that DPMR can detect. To help minimize the dangling pointer shortcoming, Purify delays the reallocation of freed buffers. Two more advantages that DPMR has over Memcheck and Purify are that DPMR can detect hardware memory errors, and DPMR's detection capabilities are tunable.

Another popular method for performing runtime memory checks uses per-pointer metadata. One of the earliest approaches that uses per-pointer metadata is Safe-C [13]. The metadata for a pointer p include the bounds of the object to which p points, as well as a capability that is tracked in a Global Capability Store (GCS). When p's object is deallocated, p's capability is removed from the GCS, letting us know that p is temporally invalid. Safe-C uses fat pointers to track metadata. Fat pointers replace pointers with a struct that contains the original pointer as well as the metadata. One of the major drawbacks of Safe-C is that fat pointers are not compatible with libraries that

have not been processed by Safe-C. In addition, the per-pointer metadata approach is not capable of detecting uninitialized reads.

Patil and Fischer [14] and Xu et al. [15] improve on Safe-C by eliminating the use of fat pointers. Instead, they maintain metadata separately from pointers. However, neither approach allows arbitrary pointer-to-pointer casts, a restriction that DPMR overcomes with the design described in Chapter 4. Patil and Fischer and Xu et al. also disallow int-to-pointer casts,[3] which DPMR overcomes as described in Chapter 5. Another major difference between DPMR and the approaches that track per-pointer metadata is that DPMR is suited for detecting hardware memory errors, while the per-pointer metadata approaches are not.

The final runtime-checking approach that we examine is BGI [16]. BGI is slightly different from the approaches discussed above and does not detect errors at the same granularity as those approaches. BGI uses access control lists to enforce fault isolation boundaries between the kernel and kernel extensions. BGI is not intended to enforce internal memory properties such as the absence of array bounds violations and dangling pointer errors. In addition, it does not have any facilities for detecting uninitialized reads. However, at very low overheads it may be an appropriate technique to deploy when more comprehensive detection is not required.

Static Analysis and Runtime Checking In addition to the runtime checking methods discussed above, there is a class of methods that combines static analysis with runtime checking. One such tool is CCured [17]. Using static analysis, CCured classifies pointers as safe pointers, sequence pointers, or wild pointers. At runtime, safe pointer accesses are null-checked. Sequence pointer accesses are bounds-checked and null-checked. Wild pointer accesses are bounds-checked, null-checked, and checked for type consistency. In order to handle dangling pointer errors, CCured uses conservative garbage collection. Unfortunately, garbage collection avoids dangling pointer errors instead of detecting them. In addition, garbage collection can impose severe performance penalties and can unnecessarily increase memory usage. Another drawback to

[3] In the term int-to-pointer, int refers to an integer.

CCured is that it deploys fat pointers for sequence pointers and wild pointers. As mentioned earlier, fat pointers are not generally compatible with library code. Finally, CCured does not detect uninitialized reads in general.

SAFECode [18] is a second approach that combines static analysis and runtime checking. The goal of SAFECode is to enforce memory safety at a level that guarantees the following compiler analysis results: a call graph, a points-to graph, and type information, such that the points-to graph and type information result from a flow-insensitive, field-sensitive, and unification-based pointer analysis. However, SAFECode has been extended by other work from the SAFECode authors to detect array out-of-bounds errors [19] and dangling pointer errors [20].

In [19], Dhurjati and Adve improve upon the work of Jones and Kelly [21] and Ruwase and Lam [22] for detecting array out-of-bounds errors. The work by Jones and Kelly uses a splay tree to track object bounds, and runtime checks are used to detect out-of-bounds memory accesses. Ruwase and Lam extend the work of Jones and Kelly to handle a key shortcoming of it, namely, that it does not allow invalid pointers, even if the pointers are not accessed. The major contribution of Dhurjati and Adve is the use of provably distinct memory pools to reduce the size of the Jones and Kelly splay trees. In addition, they eliminate many load and store checks required by Ruwase and Lam by forcing out-of-bounds pointers to point to protected memory. Thus, accesses of the out-of-bounds pointers will result in hardware traps.

In [20], Dhurjati and Adve propose an approach for detecting dangling pointer errors in which each heap object is mapped to a new virtual page. When a heap object is freed, the corresponding virtual page is protected from future reads and writes. In order to make that approach efficient, when a heap object is freed, the underlying physical page is unmapped, even though the underlying virtual page is not. There are two major drawbacks to the approach. First, significant overhead is moved to memory allocation and deallocation points. Therefore, the approach may not be suitable for applications that perform many allocations and deallocations. Second, the approach utilizes a memory pool strategy that may exhaust virtual memory if pools are long-lived. The

authors discuss several techniques to handle long-lived pools; however, such techniques may reduce detection capabilities or require garbage collection or programmer support.

Randomized Memory Layouts A third approach to spatial and temporal memory safety is the use of randomized memory layouts. DieHard [23] is one of the first approaches to utilize a randomized memory layout for memory safety. DieHard tries to emulate the semantics of an infinite heap by over-allocating the heap and randomizing the layout of heap objects. In an infinite heap, buffer overflows are benign, because there is infinite space between heap objects. DieHard simulates that property, with a high probability, by placing empty space between heap objects. Dangling pointer errors are also benign in an infinite heap because heap objects never need to be freed. DieHard emulates that behavior by randomizing the selection of free objects to be allocated, making it unlikely that a recently freed object will be reallocated. DieHard detects uninitialized reads by using process replication and by randomizing the value of initial heap data. In order to make the allocation of heap objects efficient, DieHard partitions the heap into pools of fixed-size objects.

One drawback to DieHard is that it does not protect the stack or global variables. Consequently, it is incapable of handling arbitrary wild pointers. Another major difference between DPMR and DieHard is that DieHard's approach is to avoid memory errors, while DPMR's approach is to detect them. Both avoidance and detection have their advantages and disadvantages, and the deployment scenario will dictate which is more appropriate. Although DieHard and DPMR are different approaches to the same problem, each one can borrow aspects from the other. For example, DPMR could utilize DieHard's heap allocator to achieve diversity. On the other hand, when in replication mode, DieHard could utilize DPMR's partial replication strategy.

Archipelago [24] extends DieHard with the aim of reducing DieHard's physical memory consumption. As described above, DieHard leaves large portions of unallocated virtual memory between allocated objects. Unfortunately, the unallocated virtual memory consumes physical memory. They key to Archipelago is that if unallocated memory between objects occupies an entire page of virtual memory, that virtual page does not

have to be committed to physical memory. Therefore, Archipelago places each heap object on its own virtual memory page and leaves one or more pages of unallocated memory between objects. Those pages of unallocated memory are not committed to physical memory. In addition, Archipelago uses a coloring algorithm that prevents the degradation of cache performance. The coloring algorithm ensures that even though each object occupies an entire page of memory, different objects fall on different cache lines.

Another approach that extends DieHard is DieHarder [25]. DieHarder's goal is to thwart heap-based attacks. One technique DieHarder uses to do so is sparse allocation of objects throughout the heap. The sparse allocation technique leaves guard pages between objects, which when accessed crash the program. Another technique DieHarder deploys is destruction of freed object data.

The last approach that we examine that uses randomized memory layouts is Heap Server [26]. Heap Server was developed concurrently with DieHard and uses similar concepts. It randomizes the layout of the heap by placing random padding between heap objects and by randomizing the selection of the chunks of deallocated memory that will be allocated for new objects. The main difference between Heap Server and DieHard is that Heap Server stores heap metadata in a separate process. Because it does so, the metadata are protected from attacks.

Error Correction A fourth approach to memory safety is correction of memory errors. The most prominent project in that category is Exterminator [27]. Exterminator combines the DieHard allocator with the use of *fence posts* and *canary values*. Fence posts are special data values that are placed between objects to detect buffer overflows, while canaries are special data values that are placed in unallocated memory to detect dangling pointer writes. When Exterminator detects an error, it dumps a heap image. Multiple heap images are statistically analyzed to diagnose the source of a heap error. After the source of an error has been identified, a patch can be automatically generated to address errors. It is possible to patch a buffer overflow by increasing the amount of space that is allocated for the buffer. It is possible to patch dangling pointer errors by

delaying deallocation of the objects pointed to by the dangling pointers. It is important to note that Exterminator is not a panacea. It attempts to correct a symptom; however, the underlying fault may manifest in other ways not handled by a patch. We also note that Exterminator does not address uninitialized reads, global variables, and stack variables.

A second approach to memory error correction is Rx [28]. Rx uses a checkpoint-rollback scheme. However, after detecting an error and rolling a program back to a checkpoint, Rx executes the program in a diverse environment that is designed to avoid the error in the second execution. For example, if a buffer overflow is detected, the overflowed buffer can be padded. In [28], Qin et al. focus on the architecture and do not present new methods for detection of memory safety violations. Therefore, DPMR and Rx could aid each other. DPMR could be used to detect memory errors, and Rx could be used to recover from the detected errors.

Diverse Replication The final category of memory safety approaches is diverse replication. This category includes DPMR and Samurai [29]. In Samurai, a programmer identifies certain memory on the heap as critical. Critical memory is protected through a combination of intra-process replication and the diversity afforded by DieHard. Samurai can detect errors with two replicas and can recover with three replicas.

There are two major differences between Samurai and DPMR. First, the intra-process replication that is used by DPMR is automatically chosen, while the intra-process replication that is used by Samurai is programmer-specified. Identification of the critical memory that is the basis for Samurai's replication is not trivial. Another nontrivial task is the identification of critical stores. Although it is safe to identify a store as critical when it is not, doing so adds unnecessary branches to the code to check if the memory is actually critical memory. The second major difference between Samurai and DPMR is that Samurai does not consider uninitialized reads, double frees, global variables, and stack variables, while DPMR does handle all of those error classes.

1.5.2 Software Diversity

The other category of work that is closely related to DPMR is software diversity. Software diversity is a much-touted mechanism for dealing with deterministically activated faults and malicious faults. Because of growing concerns over security, many papers have been published on software diversity, including several in the last 15 years. The software diversity research most relevant to the work presented in this dissertation falls into two categories: diverse redundant software and diverse software execution.

Diverse Redundant Software Diverse redundant software is a group of techniques for constructing dependable software from diverse redundant components. The most prevalent diverse redundant software approach is *N-version software (NVS)* [30]. NVS is a system for fault tolerance in which multiple diverse replicas are simultaneously executed. The states of the replicas are periodically compared to detect faulty replicas, which can often be corrected upon detection. More recent takes on NVS, such as N-variant systems [31], Orchestra [32], and proactive obfuscation [33], aim to advance the state of the art in NVS to tolerate security attacks. An important problem in NVS is that of determining which replicas are correct and which are not. Many decision algorithms have been developed to address that problem, including majority voting [34,35], consensus voting [35,36], maximum likelihood voting [37], and $t/(n-1)$-variant programming [38]. DieHard's active replication scheme, Exterminator's active replication scheme, Samurai, and DPMR are all examples of NVS systems.

Two other diverse redundant software techniques are *recovery blocks* [39–41] and *N self-checking programming (NSCP)* [42]. While NVS is an actively redundant strategy, recovery blocks are a cold spare strategy, and NSCP is a hot spare strategy. Recovery blocks work as follows. An application takes a checkpoint C. It then executes a variant X_1 of a component X, after which an acceptance test is applied to determine whether X_1 ran correctly. If it did not run correctly, the application rolls back to C and executes a different variant of X. That process is applied recursively until a variant of X passes an acceptance test or all variants have failed. Under NSCP, an application simultaneously executes several variants of a software component. One of

the variants is named as the primary. If the primary fails, then one of the active spares takes over. It is different from NVS, because the different variants of a component are not compared with each other to detect errors.

In order for diverse redundant software systems to accomplish their goals, they require sources of replica diversity. One of the first approaches for generating replica diversity was *N-version programming (NVP)* [30,34]. NVP is an approach in which independent programming teams develop N different versions of a software component. However, its ability to produce useful diversity, which is necessary for diverse redundant software, has been brought into question [43–46]. Another method for achieving component diversity is to use diversity that occurs naturally when multiple vendors develop competing products. That is the approach taken by BASE [47]. The usefulness of such an approach is legitimized by studies of fault diversity in systems such as SQL database servers [48] and operating systems [49]. A third way to obtain replica diversity is through data diversity. That is the approach taken by DieHard's active replication scheme, Exterminator's active replication scheme, Samurai, and DPMR. It is also utilized by the N-variant systems work [50].

Diverse Software Execution Diverse software execution encompasses a group of techniques in which software is executed in diverse execution environments. Here we discuss four categories of diverse software execution.

One category of diverse software execution approaches utilizes diversity for recovery. We already discussed one example of such approaches, Rx. Two more examples are the portable checkpointing approaches Porch [51] and PREACHES [52]. Portable checkpoints allow a process that fails in one environment to recover in a heterogeneous environment. A second category of diverse software execution approaches consists of approaches that utilize randomized memory layouts. In addition to DieHard and Exterminator, there are several approaches that randomize the layout of address spaces [53–57]. The goal of the address space randomization techniques is to foil attacks that require address space knowledge. The third category of diverse software execution approaches is that of randomized instruction sets [58, 59]. Randomized in-

struction sets attempt to prevent code-injection attacks by rendering the injected code useless. The final category of diverse software execution approaches involves operating system-enabled diverse execution. For example, various aspects of system calls can be randomized [53, 54]. As with many of the other diverse software execution techniques, the goal is to thwart attacks.

1.6 Contributions

At the highest level, the contribution of this dissertation is an automated and tunable approach to detecting software memory errors called *Diverse Partial Memory Replication (DPMR)*. More specifically, the contributions of this dissertation are as follows:

- We introduce DPMR, an automated and tunable methodology for detecting software memory errors. It detects both spatial and temporal memory errors in all forms of memory, including global variables, the stack, and the heap. DPMR can also detect memory hardware faults, although that is not the focus of this dissertation.

- We introduce the concept of partial replication, which improves replication overhead by automatically replicating only the software components that are relevant to the targeted fault model, and places a replica in the same process as the corresponding original component.[4] There are many advantages to intra-process replication, as discussed in Section 2.1. Our primary contributions regarding partial replication have been to show how to do it and to evaluate key design choices. Comparison with full replication is left to future work.

- A key challenge when performing intra-process memory replication is handling pointers stored in memory. In particular, if pointers stored to replica memory are identical to those stored in application memory, we lose the ability to traverse replica data structures. If pointers stored to replica memory permit the traversal of replica data structures, we lose the ability to compare pointers stored in memory. We propose

[4] The automatically chosen, intra-process replication presented here is similar in concept to the programmer-specified replication used in Samurai [29].

two strategies for handling pointers stored in memory. One strategy, Shadow Data Structures, permits comparison of pointers stored in memory. The other strategy, Mirrored Data Structures, reduces overhead and memory consumption in general but loses the ability to compare pointers stored in memory.

- We develop the algorithms, compiler transformations, and design details needed to implement DPMR for the C programming language.
- We provide a first-of-its-kind study of diversity mechanisms and state comparison policies for diverse redundant software that addresses memory errors. This study is instrumental in fulfilling DPMR's goal of tunability, as well as fulfilling that goal for other systems that use diverse redundant software for software memory errors.
- We design a framework for reasoning about DPMR in the presence of an incomplete program and unrecognized pointer behavior.
- We develop algorithms to extend the scope of DPMR to most of C. Those algorithms include solutions to the load comparison problem, the int-to-pointer cast problem, and the problem of storing pointers that masquerade as integers.

1.7 Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the design and development of DPMR. In particular, it proposes an approach for managing pointers stored in memory, which is a key challenge for DPMR. It also includes several proposed diversity transformations and state comparison policies that are later evaluated in Chapter 3. While Chapter 2 describes the salient features of DPMR, Chapter 3 addresses details relevant to the construction of a tool based on DPMR. It also presents an experimental evaluation of that tool. Chapter 4 considers an alternative DPMR design whose goal is to reduce the overheads reported in Chapter 3. Chapter 5 examines an approach that eliminates many of the limitations and restrictions imposed by initial DPMR designs. The approach utilizes a

compiler analysis called *Data Structure Analysis*. Chapter 6 discusses future DPMR research that is beyond the scope of this dissertation. Finally, Chapter 7 concludes the dissertation.

Chapter 2

Diverse Partial Memory Replication

In this chapter, we discuss the design of DPMR, first introduced in [60]. We start by defining partial replication in Section 2.1. Section 2.2 introduces and proposes a solution to a key challenge to DPMR, i.e., the handling of pointers stored in memory. Section 2.3 presents a method for mapping application memory to replica memory across function boundaries. Sections 2.4 devises the compiler transformations used to instrument an application with DPMR. Section 2.5 examines the conditions under which DPMR will detect an error. Sections 2.6 and 2.7 describe interoperable comparison transformations and state comparison policies, which are later evaluated in Chapter 3. Section 2.8 discusses a strategy to handle external code. Finally, Section 2.9 identifies the limitations of the approach described in this chapter. Many of those limitations are mitigated with the contributions of Chapters 4 and 5.

Throughout this dissertation, we make several assumptions about input programs in order to describe various data type rules and code transformations. The assumptions described here are typical of the kinds of assumptions that are made by the intermediate representations that are used by compilers. Unless explicitly stated otherwise, e.g., when discussing details pertaining to C, the assumptions described here apply. When defining types, if an element of a derived type is null, then that element drops out of the derived type. If all of the elements of a derived type are null, then the derived type is null.

We assume that the type system of input programs contains primitive integer and floating point types of predefined sizes. The type system also includes a void type and five derived types: pointers, structures, unions, arrays, and functions. All pointer types have the same predefined size. Array types, designated by square brackets, do not imply a pointer as they do in C. Therefore, the type `struct{int32; int32; int32;}` is equivalent to `int32[3]`.

In addition to the assumptions we make on the type system, we assume an architecture in which virtual registers and memory are distinct. Virtual registers can never be accessed through load and store operations, while programs can only interact with memory through loads and stores. We assume that programs can allocate three types of memory: heap memory (via `malloc` calls), stack memory (via `alloca` calls), and global variable memory (via `global` declarations). In doing so, we assume that all global variables are pointers to memory. Our final assumption is that virtual registers can hold only scalars, i.e, they can hold integers, floating point values, and pointers. As a result, each load operation loads one scalar value. Each store operation stores one scalar value. Functions return up to one scalar value, and function parameters are scalars.

2.1 Partial Replication

A key aspect of the DPMR design is partial replication. At its heart, partial replication implies that a portion of a system and/or its components are replicated. Such replication is in contrast to traditional process replication techniques. Three factors motivate a partial replication design. First, if our target fault model affects only a portion of a system, then it is unnecessary to replicate the entire system. Second, for replication techniques that utilize state comparison, if the state of a component cannot be reasoned about, then it is unnecessary to replicate that component. Third, in an effort to support tunability, it may be desirable to disable replication of some components, creating even more of a trade-off between performance and dependability.

The first factor mentioned above is the premise for the partial replicas discussed throughout the majority of this dissertation. In DPMR, only the memory subsystem is replicated, because it is the portion of an application that is directly affected by memory errors. The second factor, i.e., the one pertaining to components whose state cannot be reasoned about, is utilized in Chapter 5 to refine the partial replication used by DPMR. The third motivating factor, i.e., the ability to restrict replication to tune the performance-dependability trade-off, is only explored in this thesis through various state comparison policies that limit state comparisons. However, we feel that further development of such partial replicas could

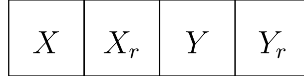

Figure 2.1: Implicit diversity

be a substantial contribution of future research efforts.

In DPMR, partial replication is built on the concept of intra-process replication. Replicated behavior is placed in the same process as original program behavior. The intra-process partial replication scheme used in DPMR has many advantages over a full-process replication scheme. First, intra-process replication eliminates many sources of overhead that would exist in a full replication scheme. It eliminates overhead from the replication of non-memory-related instructions. It eliminates overhead spent on costly state transfers between replicas, which would be necessary to distribute input, the results of unreplicated computation, and data for state comparisons. Finally, it eliminates the time spent synchronizing at application-environment interactions.

The second major benefit of intra-process replication over full-process replication is that intra-process replication eliminates OS and library support that would otherwise be required to replicate system interactions. The third advantage is that intra-process replication leads to implicit diversity. Because original application-allocated memory and replicated memory occupy the same virtual address space, it is unlikely that the memory immediately following (or preceding) an application object X will be occupied by an object paired with the object immediately following (or preceding) X's replica object, unless explicitly manufactured. For example, if an application object X is allocated, followed in sequence by the allocations of X's replica (X_r), another application object Y, and Y's replica (Y_r), those objects will often be laid out as shown in Figure 2.1. In the figure, the object following X is X_r, and the object following X_r is Y. If there is a buffer overflow out of X, once the store operation that creates the overflow has been replicated, two unpaired objects, X_r and Y, will be corrupted.

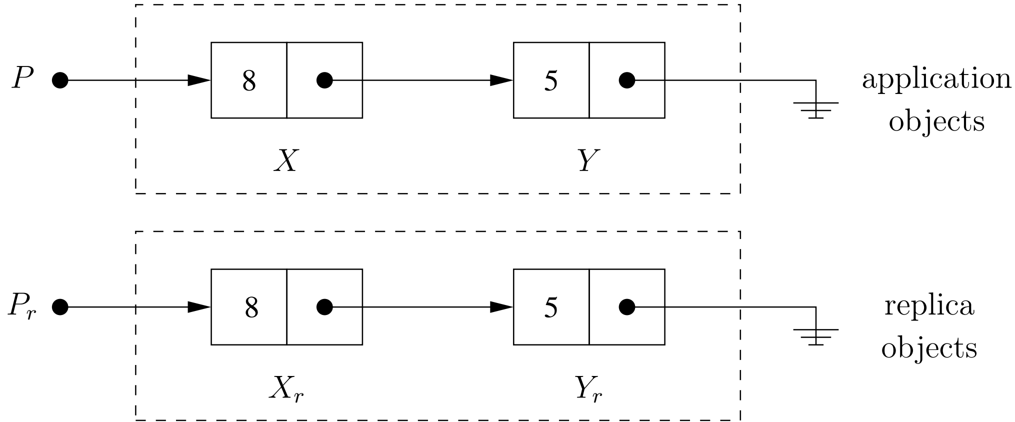

Figure 2.2: Replication using non-comparable pointers

2.2 Shadow Data Structures

The main mechanism to achieve DPMR's intra-process partial replication is a compiler transformation that replicates memory allocation, deallocation, load, and store instructions. Doing so necessitates a runtime method for tracking application object-replica object pairs. In DPMR, such tracking is done by also replicating pointer operations. However, a program may store pointers to memory. Therefore, when a program stores a pointer to memory, we also need to store a replica pointer to memory. One method for doing so is to store the replica pointer to replica memory, yielding memory layouts such as that in Figure 2.2. In the figure, object X contains a pointer to Y; therefore, X_r contains a pointer to Y_r.

Unfortunately, the pointer value stored in X is not comparable to the pointer value stored in X_r. In order to improve detection capabilities, DPMR discards that scheme for a scheme in which pointers are comparable. In other words, the same pointer values that are stored in application objects are also stored in replica objects, as shown in Figure 2.3. However, that leaves us without a mapping from application object pointers to replica object pointers for pointers stored in memory. In Figure 2.3, if the pointer stored in object X, pointing to Y, is traversed, we are left without a method to get a pointer to Y_r.

One solution to that problem, and the solution DPMR utilizes, is to keep a shadow data structure that holds replica pointers. For each application object-replica object pair, a third shadow object is maintained, as illustrated in Figure 2.4. For each pointer that is stored to an application object, we store a replica pointer to that object's shadow object. Consider

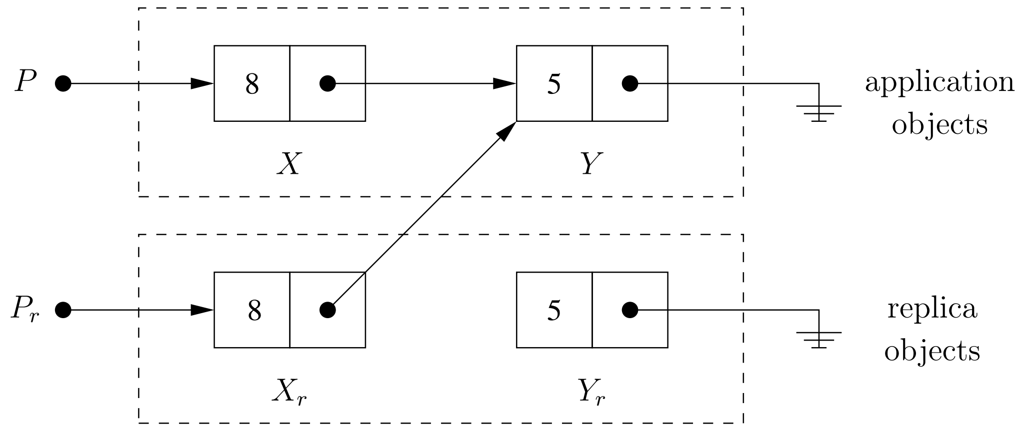

Figure 2.3: Replication making pointers in memory comparable

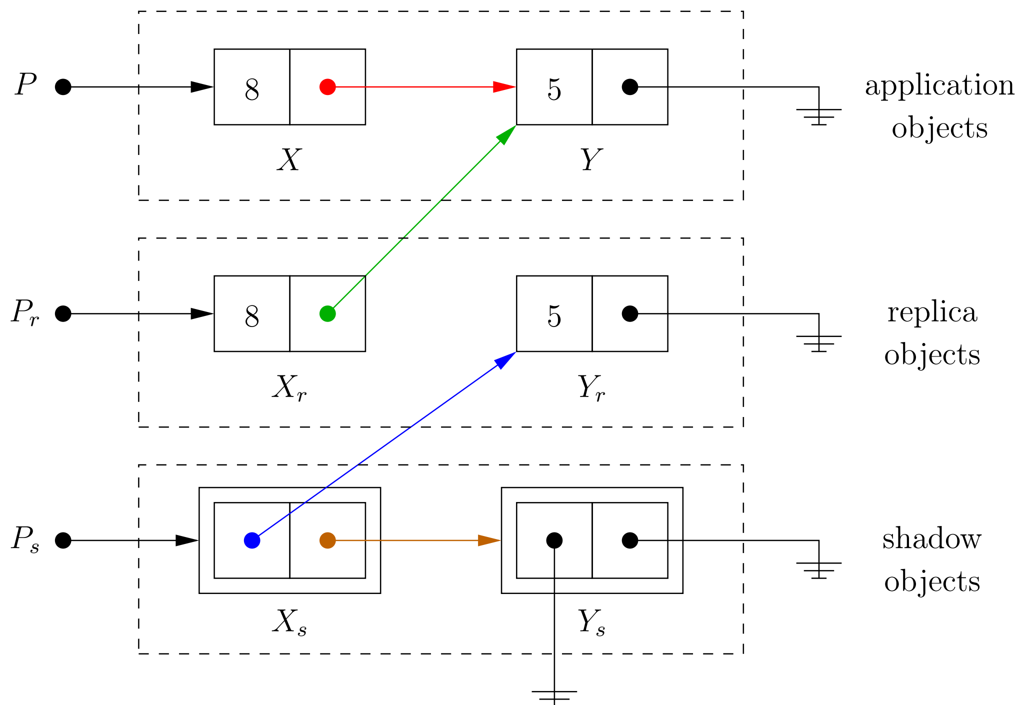

Figure 2.4: Replication using shadow data structures

Table 2.1: Shadow type definition

Description	**Original Type** t	**Shadow Type** $\mathbf{st}(t)$
array	$\tau[]$	$\mathbf{st}(\tau)[]$
struct	struct$\{\tau_0; \ldots; \tau_n;\}$	struct$\{\mathbf{st}(\tau_0); \ldots; \mathbf{st}(\tau_n);\}$
union	union$\{\tau_0; \ldots; \tau_n;\}$	union$\{\mathbf{st}(\tau_0); \ldots; \mathbf{st}(\tau_n);\}$
pointer	$\tau* : \mathbf{st}(\tau) \neq \varnothing$	struct$\{\tau*; \mathbf{st}(\tau)*;\}$
	$\tau* : \mathbf{st}(\tau) = \varnothing$	struct$\{\tau*; \text{void}*\}$
primitive, function, and void types	τ	$\varnothing$

Figure 2.4. When an application stores a pointer to Y in object X (the red pointer in the figure), DPMR performs three additional store operations.

1. A pointer identical to that stored in X is stored to X_r. The store is represented by the green pointer from X_r to Y. The pointer values stored to X and X_r will be identical under error-free execution.

2. A pointer is also stored to X_s that points to Y_r, represented by the blue pointer. That pointer is necessary so that if the pointer to Y stored in X is extracted, we will be able to extract a pointer to Y's replica.

3. The final additional store operation stores a pointer to Y's shadow object in X's shadow object, represented by the brown pointer in the figure. The pointer to Y_s is needed so that if the pointer to Y is extracted from X, we will have a handle to Y's shadow object.

Therefore, for every pointer to object Q stored to an application object O, we store two pointers in O's shadow object: a *replica object pointer (ROP)* pointing to Q's replica and a *next shadow object pointer (NSOP)* pointing to Q's shadow object. We call the replication design using shadow objects *Shadow Data Structures (SDS)*.

SDSs are typed using *shadow types (*$\mathbf{st}()$*)*, as defined in Table 2.1. The definition can be divided into three categories: aggregate and union types, pointer types, and all other types. The shadow type for an aggregate or union type is constructed by applying the aggregate or union type, respectively, to the shadow types of its elements. The shadow type of a pointer

Table 2.2: Examples of shadow types

Original Type	Shadow Type

`int8[]*;`	`struct int8ArrayPtrSdwTy{`
	`  int8[]* rop;`
	`  void* nsop;`
	`};`

`int8[]**;`	`struct int8ArrayPtrPtrSdwTy{`
	`  int8[]** rop;`
	`  struct int8ArrayPtrSdwTy* nsop;`
	`};`

`struct LinkedList{`
`  int32 data;`
`  struct LinkedList* nxt;`
`};`
	`struct LinkedListSdwTy{`
	`  struct{`
	`    struct LinkedList* rop;`
	`    struct LinkedListSdwTy* nsop;`
	`  } nxtSdwObj;`
	`};`

`struct dir;`
`struct file{`
`  int8[]* name;`
`  int32 size;`
`  struct dir* parent;`
`};`
	`struct dirSdwTy;`
	`struct fileSdwTy{`
	`  struct int8ArrayPtrSdwTy nameSdwObj;`
	`  struct{`
	`    struct dir* rop;`
	`    struct dirSdwTy* nsop;`
	`  } parentSdwObj;`
	`};`

p is a structure containing a pointer for p's ROP and a pointer for p's NSOP. The ROP has the same type as p. The NSOP has a pointer type composed of the shadow type of the object that p points to, unless that shadow type is null. In the case for which the shadow type of the object that p points to is null, the NSOP is given a void pointer. The void pointer type is a placeholder in case p is cast to a type $\alpha*$ such that the shadow type of α is not null. Finally, the shadow type for primitive types and function types is null because we do not need to carry any extra metadata for those types.

Table 2.2 contains examples of several shadow types. Consider the type `int8[]*`, a pointer to an array of 8-bit integers. (As stated at the beginning of this chapter, square brackets in a type do not imply a pointer.) Since the shadow type of an integer is null, then the shadow type of an array of integers is null. Therefore, when constructing the shadow type of `int8[]*`, we get a structure composed of a replica pointer of the same type and a `void` pointer, to

replace the pointer to a null shadow type. To compute the shadow type for `int8[]**`, we build on the shadow type for `int8[]*`. Using the rule for pointers, the shadow type for `int8[]**` is a structure composed of a replica pointer and a pointer to the shadow type of `int8[]*`, as shown in Table 2.2.

We now examine the shadow type for a linked list. The linked list is defined using a structure; therefore, its shadow type is a structure composed of the shadow types of its elements. The first element is an integer. The integer's shadow type is null, since there is no shadow data to keep for types that do not contain pointers. The second element of the linked list structure is the recursive pointer `nxt`. Since `nxt` is a pointer, its shadow type is a structure containing an ROP and an NSOP. The ROP has the same type as `nxt`, i.e., a pointer to a linked list structure. The NSOP is a pointer to the shadow type of a linked list, which matches the recursive nature of `nxt`. The last example in Table 2.2 demonstrates a shadow type for a structure with multiple pointers. As shown in the table, each pointer is translated to a structure with two pointers, an ROP and an NSOP, and the integer from the original structure is dropped from the shadow type.

Shadow types can be computed with the algorithm shown in Figure 2.5. Much of the algorithm is a straightforward interpretation of the shadow type definition; however, three aspects deserve explanation. First, a dynamic programming approach is utilized for temporal efficiency. Once a shadow type is computed for a type t, it is stored in the global map ST. Second, if a derived type is not composed of pointers outside of function types, then the algorithm can short-circuit and return null. That behavior is performed by the check in line 17. Third, the algorithm must be able to process recursive types. Type recursion can only occur through pointers. The majority of the work of processing a pointer type $\tau*$ consists of computing its NSOP via a recursive call to getShadowTypeImpl(). Prior to making that recursive call, we create a placeholder ph to be used in place of $nsop$, if $nsop$ is needed during the recursive computation of itself. Upon completion of that computation, we resolve instances of ph to the computed value of $nsop$ (line 10). Placeholder resolution can be implemented by assigning a unique type name to the type stored in $nsop$ and replacing instances of ph with that name.

Input t: A type
Output Return the shadow type of t.

```
getShadowType(t)
1:  map P ← ∅
2:  return getShadowTypeImpl(t, P)
```

Input t: A type
$PhMap$: Mapping of types to placeholders
Globals ST: Mapping of a type to its shadow type
Output Return the shadow type for t. The returned type may be partially computed.

```
getShadowTypeImpl(t, ref map PhMap)
1:  if (t ∈ ST) then return ST[t]
2:  if (t = τ∗) then
3:      if (t ∈ PhMap) then
4:          return struct{t; PhMap[t]}
5:      ph ← getPlaceHolder()
6:      PhMap[t] ← ph
7:      st ← getShadowTypeImpl(τ, PhMap)
8:      if (st = ∅) then nsop ← void∗
9:      else nsop ← st∗
10:     resolvePlaceHolderTo(ph, nsop)
11:     rv ← struct{t; nsop; }
12:     if (!containsPlaceHolder(t) ∧ !containsPlaceHolder(rv))
13:         PhMap ← PhMap − {t}
14:         ST[t] ← rv
15:     return rv
16: else
17:     if (!containsPointerOutsideFunType(t)) then rv ← ∅
18:     if (t = array{τ}) then
19:         st ← getShadowTypeImpl(τ, PhMap)
20:         rv ← array{st}
21:     if (t = struct{τ_0; . . . ; τ_n; }) then
22:         for i = 0 to n
23:             st_i ← getShadowTypeImpl(τ_i, PhMap)
24:         rv ← struct{st_0; . . . ; st_n; }
25:     if (t = union{τ_0; . . . ; τ_n; }) then
26:         for i = 0 to n
27:             st_i ← getShadowTypeImpl(τ_i, PhMap)
28:         rv ← union{st_0; . . . ; st_n; }
29:     ST[t] ← rv
30:     return rv
```

Figure 2.5: Algorithm for getShadowType

Table 2.3: Augmented type definition

Description	Original Type t	Replica Parameter Type $\mathbf{rpt}(t)$
pointer	$\tau*$	$\mathbf{at}(\tau)*$
other	τ	$\varnothing$
Description	**Original Type t**	**Shadow Parameter Type $\mathbf{spt}(t)$**
pointer	$\tau* : \mathbf{st}(\tau) \neq \varnothing$	$\mathbf{st}(\mathbf{at}(\tau))*$
	$\tau* : \mathbf{st}(\tau) = \varnothing$	void$*$
other	τ	$\varnothing$
Description	**Original Type t**	**Augmented Type $\mathbf{at}(t)$**
array	$\tau[]$	$\mathbf{at}(\tau)[]$
struct	struct$\{\tau_0; \ldots; \tau_n;\}$	struct$\{\mathbf{at}(\tau_0); \ldots; \mathbf{at}(\tau_n);\}$
union	union$\{\tau_0; \ldots; \tau_n;\}$	union$\{\mathbf{at}(\tau_0); \ldots; \mathbf{at}(\tau_n);\}$
pointer	$\tau*$	$\mathbf{at}(\tau)*$
function types	$r(\tau_0, \ldots, \tau_n)$	$\mathbf{at}(r)(\mathbf{st}(\mathbf{at}(r))*,$ $\mathbf{at}(\tau_0), \mathbf{rpt}(\tau_0), \mathbf{spt}(\tau_0), \ldots,$ $\mathbf{at}(\tau_n), \mathbf{rpt}(\tau_n), \mathbf{spt}(\tau_n))$
primitive and void types	τ	τ

2.3 Augmented Functions

With SDS we have a method for mapping application pointers stored in memory to replica pointers; however, we also need a way to map application pointers that are passed and returned across function calls to replica and shadow pointers. That can be achieved by modifying function types and function calls to allow those replica and shadow pointers to be passed and returned when application pointers are passed and returned, respectively. DPMR transforms all types used by a program into *augmented types (*$\mathbf{at}()$*)*. Under such a transformation, most program types remain the same, with only those that contain function types actually changing.

Augmented types are defined in Table 2.3. Following from our assumptions about program input (see the beginning of this chapter), the only function argument and return value types that can contain pointers are ones that are themselves pointers. Therefore, when transforming function types, we need identify only pointer parameters and pointer return values. For all pointer parameters, we add two additional parameters to the function type, one for an ROP and one for an NSOP. For functions that return a pointer, a shadow object

Table 2.4: Example of an augmented type

Original Type	Augmented Type
`int8[]* (int8[]* s1, int8[]* s2);`	`int8[]* (struct int8ArrayPtrSdwTy* rvSop, int8[]* s1, int8[]* s1Rop, void* s1Nsop, int8[]* s2, int8[]* s2Rop, void* s2Nsop);`

pointer parameter is added.[1] That shadow object pointer is used to point to memory where called functions can store an ROP and NSOP pertaining to their return value. The ROP and NSOP can then be loaded by the caller. Table 2.4 contains an example of the transformation of a function type to an augmented function type. The function type in the example takes two pointers to arrays of 8-bit integers and returns a pointer to an array of 8-bit integers. The augmented function type has a parameter `rvSop` that is a pointer to the shadow type of the return value, which was previously defined in Table 2.2. It also has ROP and NSOP parameters corresponding to the original parameters `s1` and `s2`.

The algorithms for computing augmented types are much more complicated than those for computing shadow types. The algorithms for computing augmented function types are contained in Figures 2.6, 2.7, and 2.8. The getAugTypeImpl() algorithm in Figure 2.6 contains the main loop for computing augmented types. It recursively calls itself and hands the processing of function types to getAugFunTypeImpl(), described in Figure 2.7. At several points during the calculation of an augmented function type, we require the shadow type of the augmented type of an input type. That computation is handled by getShadowAugType(), in Figure 2.8.

Several features of those algorithms deserve discussion. First, as we did when computing shadow types, we use a dynamic programming approach. Completely resolved augmented types are stored to and retrieved from the global map AT. Second, recursive data types are handled using placeholders and placeholder resolution. Given a pointer $\tau*$ for which we want to compute $\mathbf{at}(\tau*)$, we use placeholders to represent $\mathbf{at}(\tau*)$ for recursive references that are required during the computation of $\mathbf{at}(\tau*)$. Lines 5 and 9 from getAugTypeImpl(), in Figure 2.6, and the placeholder map $P2$ comprise the placeholder elements that aid in

[1] In the augmented type definition, an $\mathbf{st}(\mathbf{at}(r))*$ type is added to all function types; however, that type will only be non-null if the function's return value is a pointer.

Input t: A type
Output Returns the augmented type of t.
getAugType(t)
1: map $P1 \leftarrow \varnothing$
2: map $P2 \leftarrow \varnothing$
3: **return** getAugTypeImpl(t, $P1$, $P2$)

Input t: A type
$P1$: Mapping of types to placeholders
$P2$: Mapping of types to placeholders
Globals AT: Mapping of a type to its augmented type
Output Returns the augmented type of t. The return value may be partially computed.
getAugTypeImpl(t, **ref** map $P1$, **ref** map $P2$)
1: **if** ($t \in AT$) **return** $AT[t]$
2: **if** ($t = rt(a_0, \ldots, a_n)$) **then return** getAugFunTypeImpl(t, $P1$, $P2$)
3: **if** ($t = \tau*$) **then**
4:     **if** $t \in P2$ **then return** $P2[t]$
5:     $ph \leftarrow$ getPlaceHolder()
6:     $P2[t] \leftarrow ph$
7:     $at \leftarrow$ getAugTypeImpl(τ, $P1$, $P2$)
8:     $rv \leftarrow at*$
9:     resolvePlaceHolderTo(ph, rv)
10:     **if** (!containsPlaceHolder(t) $\wedge$!containsPlaceHolder(rv))
11:         $P2 \leftarrow P2 - \{t\}$
12:         $AT[t] \leftarrow rv$
13:     **return** rv
14: **else**
15:     **if** (isIntegerType(t) $\vee$ isFloatingPointType(t) $\vee$ $t =$ `void`) **then** $rv \leftarrow t$
16:     **if** ($t = \text{array}\{\tau\}$) **then**
17:         $at \leftarrow$ getAugTypeImpl(τ, $P1$, $P2$)
18:         $rv \leftarrow \text{array}\{at\}$
19:     **if** ($t = \text{struct}\{\tau_0; \ldots; \tau_n;\}$) **then**
20:         **for** $i = 0$ **to** n
21:             $at_i \leftarrow$ getAugTypeImpl(τ_i, $P1$, $P2$)
22:         $rv \leftarrow \text{struct}\{at_0; \ldots; at_n;\}$
23:     **if** ($t = \text{union}\{\tau_0; \ldots; \tau_n;\}$) **then**
24:         **for** $i = 0$ **to** n
25:             $at_i \leftarrow$ getAugTypeImpl(τ_i, $P1$, $P2$)
26:         $rv \leftarrow \text{union}\{at_0; \ldots; at_n;\}$
27:     $AT[t] \leftarrow rv$
28:     **return** rv

Figure 2.6: Algorithm for getAugType

Input t: A function type that equals $rt(a_0, \ldots, a_n)$
$P1$: Mapping of types to placeholders
$P2$: Mapping of types to placeholders
Globals AT: Mapping of a type to its augmented type
Output Returns the augmented type of t. The return value may be partially computed.

getAugFunTypeImpl($t = rt(a_0, \ldots, a_n)$, **ref** map $P1$, **ref** map $P2$)
1: list $ArgList \leftarrow \varnothing$
2: $at \leftarrow$ getAugTypeImpl(rt, $P1$, $P2$)
3: **if** ($rt = \tau*$) **then**
4: $st \leftarrow$ getShadowAugType(rt, $P1$, $P2$)
5: $ArgList = ArgList \cup \{st*\}$
6: **for** $i = 0$ **to** n
7: $at \leftarrow$ getAugTypeImpl(a_i, $P1$, $P2$)
8: $ArgList = ArgList \cup \{at\}$
9: **if** ($a_i = \tau*$) **then**
10: $st \leftarrow$ getShadowAugType(a_i, $P1$, $P2$)
11: $ArgList = ArgList \cup$ getElement(st, 0) $\cup$ getElement(st, 1)
12: $AT[t] \leftarrow at(ArgList)$
13: **return** $at(ArgList)$

Figure 2.7: Algorithm for getAugFunTypeImpl

the computation of $\mathbf{at}(\tau*)$. Placeholders can be implemented as discussed at the end of Section 2.2.

The third interesting feature of the algorithms for computing augmented types is that they do not call either getShadowType() or getShadowTypeImpl(). Instead, they call getShadowAugType(), which is very similar to getShadowTypeImpl() but with a few subtle changes that allow it to compute $\mathbf{st}(\mathbf{at}(t))$ without first computing $\mathbf{at}(t)$. The reason we do not want to compute $\mathbf{at}(t)$ first is that at the point in the algorithm where we would compute $\mathbf{st}(\mathbf{at}(t))$, $\mathbf{at}(t)$ may contain unresolved placeholders. To avoid complications that would arise from processing a type containing unresolved placeholders, we compute $\mathbf{st}(\mathbf{at}(t))$ in one calculation. We can compute $\mathbf{st}(\mathbf{at}(t))$ in one calculation by implementing the composite function $(\mathbf{st} \circ \mathbf{at})(t)$ given in Table 2.5.

2.4 Code Transformation

In this section, we define the code transformations that transform an input program for SDS-based DPMR. The transformations convert a program to use augmented types and shadow

Input	t:	A type
	$P1$:	Mapping of types to placeholders
	$P2$:	Mapping of types to placeholders
Globals	SAT:	Mapping of type x to the shadow type of the augmented type of x
Output	Return the shadow type of the augmented type of t. The return value may be partially computed.	

getShadowAugType(t, **ref** map $P1$, map **ref** $P2$)
1: **if** ($t \in SAT$) **then return** $SAT[t]$
2: **if** ($t = \tau*$) **then**
3:     **if** ($t \in P1$) **then**
4:         $at \leftarrow$ getAugType(t, $P1$, $P2$)
5:         $SAT[t] \leftarrow \text{struct}\{at; P1[t]\}$
6:         **return** $\text{struct}\{at; P1[t]\}$
7:     $ph \leftarrow$ getPlaceHolder()
8:     $P1[t] \leftarrow ph$
9:     $sat \leftarrow$ getShadowAugType(τ, $P1$, $P2$)
10:     **if** ($sat = \varnothing$) **then** $nsop \leftarrow void*$
11:     **else** $nsop \leftarrow sat*$
12:     resolvePlaceHolderTo(ph, $nsop$)
13:     $at \leftarrow$ getAugType(t, $P1$, $P2$)
14:     $rv \leftarrow \text{struct}\{at; nsop; \}$
15:     **if** (!containsPlaceHolder(t) $\wedge$!containsPlaceHolder(rv))
16:         $P1 \leftarrow P1 - \{t\}$
17:         $SAT[t] \leftarrow rv$
18:     **return** rv
19: **else**
20:     **if** (!containsPointerOutsideFunType(t)) **then** $rv \leftarrow \varnothing$
21:     **if** ($t = \text{array}\{\tau\}$) **then**
22:         $sat \leftarrow$ getShadowAugType(τ, $P1$, $P2$)
23:         $rv \leftarrow \text{array}\{sat\}$
24:     **if** ($t = \text{struct}\{\tau_0; \ldots; \tau_n; \}$) **then**
25:         **for** $i = 0$ **to** n
26:             $sat_i \leftarrow$ getShadowAugType(τ_i, $P1$, $P2$)
27:         $rv \leftarrow \text{struct}\{sat_0; \ldots; sat_n; \}$
28:     **if** ($t = \text{union}\{\tau_0; \ldots; \tau_n; \}$) **then**
29:         **for** $i = 0$ **to** n
30:             $sat_i \leftarrow$ getShadowAugType(τ_i, $P1$, $P2$)
31:         $rv \leftarrow \text{union}\{sat_0; \ldots; sat_n; \}$
32:     $SAT[t] \leftarrow rv$
33:     **return** rv

Figure 2.8: Algorithm for getShadowAugType

Table 2.5: Composed type definition for $(\mathbf{st} \circ \mathbf{at})(t)$

Description	**Original Type** t	**Composed Type** $(\mathbf{st} \circ \mathbf{at})(t)$
array	$\tau[]$	$(\mathbf{st} \circ \mathbf{at})(\tau)[]$
struct	$\text{struct}\{\tau_0; \ldots; \tau_n;\}$	$\text{struct}\{(\mathbf{st} \circ \mathbf{at})(\tau_0); \ldots; (\mathbf{st} \circ \mathbf{at})(\tau_n);\}$
union	$\text{union}\{\tau_0; \ldots; \tau_n;\}$	$\text{union}\{(\mathbf{st} \circ \mathbf{at})(\tau_0); \ldots; (\mathbf{st} \circ \mathbf{at})(\tau_n);\}$
pointer	$\tau* : (\mathbf{st} \circ \mathbf{at})(\tau) \neq \varnothing$	$\text{struct}\{\mathbf{at}(\tau)*; (\mathbf{st} \circ \mathbf{at})(\tau)*;\}$
	$\tau* : (\mathbf{st} \circ \mathbf{at})(\tau) = \varnothing$	$\text{struct}\{\mathbf{at}(\tau)*; \text{void}*\}$
primitive, function, and void types	τ	$\varnothing$

data structures. Before presenting the transformations, we introduce some helpful functions in Equations 2.1 to 2.4. $I()$ takes a Boolean value and converts false to 0 and true to 1. $\phi()$ takes a structure type and a pointer field and returns the index of the corresponding field in t's shadow type. $\gamma()$ takes a virtual register and converts it to a set of virtual registers that are used in function calls and declarations under DPMR. $\pi()$ takes the type of a function's return value and converts it to a set of arguments that are added to function calls with that return type.

$$I : \{false, true\} \mapsto \{0, 1\} \tag{2.1}$$

$$\phi(t, f_i) \equiv \sum_{j=0}^{i-1} I(\mathbf{st}(\mathbf{at}(\tau_j)) \neq \varnothing) : t = \text{struct}\{\tau_0\ f_0; \ldots; \tau_n\ f_n;\}, \mathbf{type}(f_i) = \beta* \tag{2.2}$$

$$\gamma(r) \equiv \begin{cases} \{r, r_r, r_s\} & \text{if } \mathbf{type}(r) = \tau* \\ \{r\} & \text{else} \end{cases} \tag{2.3}$$

$$\pi(t) \equiv \begin{cases} \{rvSop\} & \text{if } t = \tau* \\ \varnothing & \text{else} \end{cases} \tag{2.4}$$

The DPMR code transformation is found in Tables 2.6 and 2.7. The cases in Table 2.6 take original program behavior and add to it, while the cases in Table 2.7 take original program behavior and replace it. For a virtual register p in the original program holding a pointer, the transforms create virtual registers p_r and p_s that hold the corresponding ROP and NSOP, respectively. The transform will assign values to p_r and p_s whenever p is assigned

Table 2.6: DPMR transformation – added behavior

Description	Original Behavior		Added Behavior
heap deallocation	`free`(p)		`free`(p_r) `if`$(p_s$ `!= null){ free`(p_s) `}`
store	$*p \leftarrow x$	$\mathbf{type}(x) \neq \tau*$	$*p_r \leftarrow x$
		$\mathbf{type}(x) = \tau*$	$*p_r \leftarrow x$ $(p_s$`->`$f_0) \leftarrow x_r$ $(p_s$`->`$f_1) \leftarrow x_s$
load	$x \leftarrow *p$	$\mathbf{type}(x) \neq \tau*$	`assert`$(x$ `==` $*p_r)$
		$\mathbf{type}(x) = \tau*$	`assert`$(x$ `==` $*p_r)$ $x_r \leftarrow (p_s$`->`$f_0)$ $x_s \leftarrow (p_s$`->`$f_1)$
address of a struct field	$x \leftarrow$ `&`$(p$`->`$f_i)$: $\mathbf{type}(p) = \text{struct}\{$ $\tau_0\ f_0; \ldots; \tau_n\ f_n;\}*$	$\mathbf{st}(\mathbf{at}(\tau_i)) \neq \varnothing$	$x_r \leftarrow$ `&`$(p_r$`->`$f_i)$ $x_s \leftarrow$ `&`$(p_s$`->`$f_{\phi(\mathbf{type}(*p),f_i)})$
		$\mathbf{st}(\mathbf{at}(\tau_i)) = \varnothing$	$x_r \leftarrow$ `&`$(p_r$`->`$f_i)$ $x_s \leftarrow$ `null`
address of an array element	$x \leftarrow$ `&p[`i`]` : $\mathbf{type}(p) = \tau[]*$	$\mathbf{st}(\mathbf{at}(\tau)) \neq \varnothing$	$x_r \leftarrow$ `&`p_r`[`i`]` $x_s \leftarrow$ `&`p_s`[`i`]`
		$\mathbf{st}(\mathbf{at}(\tau)) = \varnothing$	$x_r \leftarrow$ `&`p_r`[`i`]` $x_s \leftarrow$ `null`
address of a function	$p \leftarrow$ `&`fun		$p_r \leftarrow$ `&`fun $p_s \leftarrow$ `null`

Table 2.7: DPMR transformation – transformed behavior

Description	Original Behavior		Transformed Behavior
allocation	$p \leftarrow allocate(\tau)$: $allocate = \{\texttt{malloc}\|$ $\texttt{alloca}\|\texttt{global}\}$	$\mathbf{st}(\mathbf{at}(\tau)) \neq \varnothing$	$p \leftarrow allocate(\mathbf{at}(\tau))$ $p_r \leftarrow allocate(\mathbf{at}(\tau))$ $p_s \leftarrow allocate(\mathbf{st}(\mathbf{at}(\tau)))$
		$\mathbf{st}(\mathbf{at}(\tau)) = \varnothing$	$p \leftarrow allocate(\mathbf{at}(\tau))$ $p_r \leftarrow allocate(\mathbf{at}(\tau))$ $p_s \leftarrow \texttt{null}$
pointer-to-pointer cast	$q \leftarrow (\tau *)p$	$\mathbf{st}(\mathbf{at}(\tau)) \neq \varnothing$	$q \leftarrow (\mathbf{at}(\tau)*)p$ $q_r \leftarrow (\mathbf{at}(\tau)*)p_r$ $q_s \leftarrow (\mathbf{st}(\mathbf{at}(\tau))*)p_s$
		$\mathbf{st}(\mathbf{at}(\tau)) = \varnothing$	$q \leftarrow (\mathbf{at}(\tau)*)p$ $q_r \leftarrow (\mathbf{at}(\tau)*)p_r$ $q_s \leftarrow (void*)p_s$
function declaration	$fun(a_0, \ldots, a_n)$: $\mathbf{type}(fun) = r(\tau_0, \ldots, \tau_n)$		$fun(\pi(r) \cup (\bigcup_{i=0}^{n} \gamma(a_i)))$: $\mathbf{type}(fun) = \mathbf{at}(r(\tau_0, \ldots, \tau_n))$
function call	$x \leftarrow fun(p_0, \ldots, p_n)$: $\mathbf{type}(fun) = r(\ldots)$	$r = \tau *$	$rvSop \leftarrow \texttt{alloca}(\mathbf{st}(\mathbf{at}(r)))$ $x \leftarrow fun(\{rvSop\} \cup (\bigcup_{i=0}^{n} \gamma(p_i)))$ $x_r \leftarrow (rvSop\texttt{->}f_0)$ $x_s \leftarrow (rvSop\texttt{->}f_1)$
		$r \neq \tau *$	$x \leftarrow fun(\bigcup_{i=0}^{n} \gamma(p_i))$
function return	`return` x	$\mathbf{type}(x) = \tau *$	$(rvSop\texttt{->}f_0) \leftarrow x_r$ $(rvSop\texttt{->}f_1) \leftarrow x_s$ `return` x
		$\mathbf{type}(x) \neq \tau *$	`return` x

a value. Below, we describe each case in the transformation.

- *allocation*: When a program allocates memory, whether in the heap, stack, or global variables, then the allocation type must be changed to an augmented type. In addition, DPMR allocates a replica block of memory in the same memory segment, also using the augmented allocation type. If the allocated type contains a pointer (outside of function types), then a shadow object is also allocated. If the shadow type of the augmented allocation type is null, then the corresponding NSOP is set to null.

- *heap deallocation*: When memory is deallocated from the heap, we deallocate the corresponding replica object from the heap. DPMR also inserts a check to see if the NSOP is non-null; if it is, the shadow object corresponding to the application object is deallocated. We perform the null-check at runtime, in case the static (compile-time) type is not precise enough to determine if a shadow object should be deallocated.

- *store*: Whenever the original program stores a value to memory, DPMR stores an identical value to replica memory. If the value that is stored to memory is a pointer, then we store the corresponding ROP and NSOP to shadow memory.

- *load*: When the application loads a value from memory, we load the corresponding value from replica memory and compare the two. If the two values differ, then DPMR has detected a memory error. If the value that is loaded from memory is a pointer, then DPMR loads a corresponding ROP and NSOP from shadow memory.

- *address of a structure field*: If a program computes the address of a structure field for an application object, DPMR computes the address of the same structure field for the corresponding replica object. If the application object contains pointers (outside of function types), then we must also compute the address of the appropriate field in the corresponding shadow object. The $\phi()$ function converts field indices of application structures to field indices of shadow structures.

- *address of an array element*: If the address is computed for an array element stored in memory, then we compute the address of the corresponding array element in replica

memory. If the array contains pointers (outside of function types), then we will also compute the address of a corresponding array element in shadow memory.

- *pointer-to-pointer cast*: When the program under transformation performs a pointer-to-pointer cast of type $\tau*$ on a pointer p, that cast is transformed into an $\mathbf{at}(\tau)*$ cast. In addition, p's ROP is cast to that same augmented pointer type. If the shadow type of the new pointed-to type is non-null, then an $\mathbf{st}(\mathbf{at}(\tau))*$ cast is applied to p's NSOP.

- *address of a function*: DPMR processes operations that take the address of a function a bit differently from other pointer operations. There are no replica functions from which to compute replica pointers. Furthermore, function pointers are never directly dereferenced, i.e., they are only dereferenced by function calls. Therefore, we use the same pointer value for a function pointer and its ROP. In addition, there is no need for the NSOP to reference a shadow object, because it will never be dereferenced. Therefore, it can simply be set to null.

- *function declaration*: DPMR transforms original function declarations to use augmented function types. The transformation adds ROPs and NSOPs corresponding to pointer arguments. In addition, if the return value of the function is a pointer, then a pointer argument is added, so the function can store to it an ROP and NSOP corresponding to the return value.

- *function call*: Function calls are transformed so that ROPs and NSOPs corresponding to input program pointer parameters can be passed to the called function. If the called function returns a pointer, it will need to return a corresponding ROP and NSOP. That is done through the *rvSop* parameter. The calling function allocates memory on the stack to which the callee stores the return pointer's ROP and NSOP. After the function call, the caller loads the ROP and NSOP.

- *function return*: If function return instructions return a pointer, then DPMR stores the pointer's ROP and NSOP in the memory pointed to by the function's *rvSop* argument.

- *global variable initialization*: Global variable initialization is not explicitly specified in

Tables 2.6 and 2.7. Global variable initialization can be thought of as a series of non-pointer operations, pointer arithmetic, and store operations that the compiler performs at compile time. When thought of in that way, global variable initialization can be transformed according to the rules in Tables 2.6 and 2.7.

Figures 2.9 and 2.10 show examples of how DPMR transforms code. Both use the types defined in Table 2.2. We first examine the example in Figure 2.9. The code on the left side is from an input program that creates a linked list node and potentially attaches it to an existing linked list. The code on the right is code transformed by DPMR. The `typedef` declarations at the top of both pieces of code are there for convenience. Below, we discuss how several parts of the original code are transformed.

1. The declaration of `createNode()` is transformed to an augmented type. The transformation includes addition of the `rvSop` argument that points to memory to hold the return value's ROP and NSOP, as well as addition of an ROP parameter and NSOP parameter that corresponds to the argument `last`.

2. The heap allocation in the input program is transformed into three heap allocations: one for an application object, one for a replica object, and one for a shadow object.

3. There are several examples of pointer arithmetic in the original program. We will discuss the one at line 19. In that line, the original program computes the address of the `nxt` field in an `LL` structure. DPMR adds an instruction to compute the address of the `nxt` field in replica memory. It also adds an instruction to compute the address of the corresponding field in the `LLSdwTy` structure.

4. The original code contains both a non-pointer store and a pointer store. The data stored to application memory in line 17 are also stored to replica memory in line 18. In line 33, the original program stores a pointer to memory. DPMR adds three additional stores. It stores the same pointer to replica memory in line 34. It then stores a corresponding ROP and NSOP to shadow memory, in lines 35 and 36.

```
1 typedef struct LinkedList LL;
2
3
4
5
6
7
8 LL* createNode(int32 data, LL* last){
9
10
11   LL* n = malloc(LL);
12
13
14   int32* dataPtr = &(n->data);
15
16
17   *dataPtr = data;
18
19   LL* nxtPtr = &(n->nxt);
20
21
22
23   *nxtPtr = NULL;
24
25
26
27   if(last){
28     LL* lastNxtPtr = &(last->nxt);
29
30
31
32
33     *lastNxtPtr = n;
34
35
36
37   }
38   return n;
39
40
41 }
```

(a) Original code

```
1 typedef struct LinkedList LL;
2 typedef struct LinkedListSdwTy
3   LLSdwTy;
4 typedef struct{
5   LL* rop;
6   LLSdwTy* nsop;
7 } LLPtrSdwTy;
8 LL* createNode(LLPtrSdwTy* rvSop,
9 int32 data, LL* last, LL* last_r,
10 LLSdwTy* last_s){
11   LL* n = malloc(LL);
12   LL* n_r = malloc(LL);
13   LLSdwTy* n_s = malloc(LLSdwTy);
14   int32* dataPtr = &(n->data);
15   int32* dataPtr_r = &(n_r->data);
16   void* dataPtr_s = NULL;
17   *dataPtr = data;
18   *dataPtr_r = data;
19   LL* nxtPtr = &(n->nxt);
20   LL* nxtPtr_r = &(n_r->nxt);
21   LLPtrSdwTy* nxtPtr_s =
22     &(n_s->nxtSdwObj);
23   *nxtPtr = NULL;
24   *nxtPtr_r = NULL;
25   nxtPtr_s->rop = NULL;
26   nxtPtr_s->nsop = NULL;
27   if(last){
28     LL* lastNxtPtr = &(last->nxt);
29     LL* lastNxtPtr_r =
30       &(last_r->nxt);
31     LLPtrSdwTy* lastNxtPtr_s =
32       &(last_s->nxtSdwObj);
33     *lastNxtPtr = n;
34     *lastNxtPtr_r = n;
35     lastNxtPtr_s->rop = n_r;
36     lastNxtPtr_s->nsop = n_s;
37   }
38   rvSop->rop = n_r;
39   rvSop->nsop = n_s;
40   return n;
41 }
```

(b) Transformed code

Figure 2.9: Transformation of `createNode()`

```
1 int32 getSum(LL* n){
2
3   int32 sum = 0;
4   while(n){
5     int32* dataPtr = &(n->data);
6
7
8     int32 v = *dataPtr;
9
10    sum += v;
11    LL* nxtPtr = &(n->nxt);
12
13
14
15    n = *nxtPtr;
16
17
18
19  }
20  return sum;
21 }
```

(a) Original code

```
1 int32 getSum(LL* n, LL* n_r,
2 LLSdwTy* n_s){
3   int32 sum = 0;
4   while(n){
5     int32* dataPtr = &(n->data);
6     int32* dataPtr_r = &(n_r->data);
7     void* dataPtr_s = NULL;
8     int32 v = *dataPtr;
9     assert(v == *dataPtr_r);
10    sum += v;
11    LL* nxtPtr = &(n->nxt);
12    LL* nxtPtr_r = &(n_r->nxt);
13    LLPtrSdwTy* nxtPtr_s =
14      &(n_s->nxtSdwObj);
15    n = *nxtPtr;
16    assert(n == *nxtPtr_r);
17    n_r = nxtPtr_s->rop;
18    n_s = nxtPtr_s->nsop;
19  }
20  return sum;
21 }
```

(b) Transformed code

Figure 2.10: Transformation of `getSum()`

5. The final transformation we examine is the transformation of `createNode()`'s return instruction. Since the original program returns the pointer `n`, DPMR adds two instructions to store `n`'s ROP and NSOP to the memory pointed to by `rvSop`. A caller of `createNode()` can load the ROP and NSOP from memory after calling `createNode()`.

We now examine the code transformations that take place in Figure 2.10. Several of the transformations that are made to `getSum()` have already been illustrated with the transformation of `createNode()`, so we will focus on the loads, which were not previously illustrated. In line 8 of the original `getSum()` code, an integer is loaded from memory. The DPMR transformation adds a replica load in line 9, and the replicated load value is compared with the original program load value. If the two values are not equal, a memory error has been detected, and DPMR aborts the program. Line 15 from the original input program contains a load whose loaded value is a pointer. DPMR adds three instructions as a result. In line 16, DPMR replicates the load from replica memory and compares the extracted pointer with the pointer loaded from application memory. If the two pointers are not equal, then DPMR aborts the program. In addition, DPMR loads a corresponding ROP and NSOP from shadow memory.

2.5 Detection Conditions

Now that the transformations for DPMR have been defined, we examine how software memory errors behave in the presence of DPMR. During that examination, we assume that an application's operating environment is fault-free. Only the application can be faulty.

As defined in Section 1.3, our error model consists of out-of-bounds accesses, dangling pointer uses, wild pointer uses, and uninitialized reads. DPMR will detect one of those errors only if the error manifests such that an application load and the replicated load added by DPMR produce different values. Below, we will examine conditions under which the various errors will lead to DPMR detection.

2.5.1 Write Errors

Our error model includes three types of write errors, i.e., buffer overflows, writes after free, and wild pointer writes. In this section, we examine the ways that those write errors can manifest and which of those ways DPMR will detect.

- *unpaired corruption of replicated memory*: Let A and B be two bytes that are paired via DPMR replication. Assume that a write error corrupts memory such that A and B have different values, producing an unpaired corruption of replicated memory. If one-half of a replicated load pair reads A at load offset i, and the other half of the replicated load pair reads B at load offset i, then DPMR will detect the error. Note that if a replicated load reads A at offset i and B at an offset other than i, then that is a read error, which we discuss below.

- *paired corruption*: If a write error corrupts two bytes in memory that are paired via DPMR replication, and the same data value is written to both bytes, DPMR will not be able to detect the corruption of those bytes. Since the corruption is not immediately detectable by DPMR, it can lead to arbitrary behavior.

- *shadow object corruption*: If an ROP is corrupted, it becomes a wild pointer. If it is loaded, it can lead to additional memory errors. If an NSOP is corrupted, it can lead to loading of wild ROPs and wild NSOPs. Additional memory errors will occur if those wild ROPs and NSOPs are used to access memory.

- *other corruptions*: Memory that is not replicated and does not hold a shadow object can also be corrupted. Such memory includes heap metadata, stack metadata, and variables that are pushed to memory after the DPMR transformation, due to register pressure. If a write error corrupts such memory, then arbitrary behavior can occur.

2.5.2 Read Errors

We now consider read errors. Our target error model includes four read errors: out-of-bounds reads, reads after free, wild pointer reads, and uninitialized reads. Three situations

can result from read errors.

- *different values*: A read error can cause a replicated load to read two different values from memory. If that happens, DPMR will detect the error.

- *same correct value*: A read error can cause a replicated load to read two identical values from memory that match the value that should be loaded from memory, even though the replicated load targets two incorrect memory locations. In that case, the error will not lead to a failure and will not be detected.

- *same incorrect value*: The last way that a read error can manifest is by causing a replicated load to read identical values from memory that are not correct. If such a case arises, DPMR detection will not detect the error at the time of the replicated load, resulting in arbitrary behavior. That could happen, for example, if a read error causes a replicated load to read from two locations in memory that are paired via DPMR replication. However, one goal of the diversity that DPMR employs is to reduce the likelihood of such situations.

2.5.3 Free Errors

Finally, we examine free errors, which include out-of-bounds frees, double frees, and wild pointer frees. Free errors can produce the following behaviors.

- *heap buffer free*: If at least one heap buffer is freed by a free error, then the error can be detected if the buffer is reallocated. Let A be a heap buffer that is erroneously freed by a free error. Let B be the buffer that is paired with A at the time of the erroneous free. If A is reallocated and paired with a buffer other than B, then the use of replicated pointers that point to A and B, that would be valid if the free error had not taken place, will cause additional write, read, and free errors that can be detected by DPMR.

- *protected memory pointer free*: If a pointer points to protected memory and is freed, then the result will be a crash.

- *free of other pointers*: If a freed pointer does not point either to the start of a heap buffer or to protected memory, then it could lead to a crash or memory corruption. A crash would occur if error checking in the heap allocator detects that the free is invalid. If a crash does not occur, memory corruption could occur. Many heap allocators store heap metadata in freed buffers. Therefore, if p is erroneously freed, heap metadata may be written into the buffer to which p points, causing a corruption. Such corruption could be detected by DPMR much as other write errors are.

2.6 Diversity Transformations

In this section, we describe several diversity transformations that are evaluated with DPMR in subsequent chapters. The transformations explicitly introduce diversity into replica behavior, beyond the implicit diversity afforded by intra-process replication, as discussed in Section 2.1. Although the diversity transformations discussed here affect the allocation and deallocation of heap objects, similar techniques could easily be applied to stack memory and global variable memory in a production version of DPMR. The diversity mechanisms presented here were strategically chosen, as discussed below; however, there are certainly other reasonable transformations that could be used in the future.

The diversity transformations evaluated under DPMR are defined in Table 2.8. The second column of the table contains original program behavior before DPMR has been applied. The third column contains replica behavior as defined by the standard DPMR transformations described in Section 2.4. Each diversity technique transforms the replica behavior as defined in the fourth column. Any other behavior defined in the standard DPMR transformations still applies.

The first diversity transformation that DPMR uses is *pad-malloc*. Pad-malloc increases the request size for replica heap allocation requests by a static amount. In the experimental studies presented in this dissertation, increases of 8, 32, 256, and 1024 bytes are evaluated. Pad-malloc was chosen as a mechanism explicitly targeting buffer overflows. Buffer overflows out of application objects can immediately corrupt other objects, while the initial portion of an overflow out of a replica object will write to unused padding. Even if the overflow runs

Table 2.8: Diversity transformations

Description	**Original Behavior**	**Replica Behavior**	**Diverse Replica Behavior**
pad-malloc-y	$x \leftarrow$ malloc(τ)	$x_r \leftarrow$ malloc(**at**(τ))	$x_r \leftarrow$ (**at**(τ)*)malloc(int8[**sizeof**(**at**(τ)) + y])

zero-before-free	free(p)	free(p_r)	tmp1 $\leftarrow$ heapBufSize(p_r)
			tmp2 $\leftarrow$ 0
			while(tmp2 < tmp1){
			  ((int8[]*)p_r)[tmp2] $\leftarrow$ 0
			  tmp2 $\leftarrow$ tmp2+1
			}
			free(p_r)

rearrange-heap	$x \leftarrow$ malloc(τ)	$x_r \leftarrow$ malloc(**at**(τ))	B $\leftarrow$ global(void*[20])
			tmp1 $\leftarrow$ randInt(1,20)
			tmp2 $\leftarrow$ 0
			while(tmp2 < tmp1){
			  B[tmp2] $\leftarrow$ malloc(**at**(τ))
			  tmp2 $\leftarrow$ tmp2+1
			}
			$x_r \leftarrow$ malloc(**at**(τ))
			tmp2 $\leftarrow$ 0
			while(tmp2 < tmp1){
			  free(B[tmp2])
			  tmp2 $\leftarrow$ tmp2+1
			}

beyond the padding, the remaining portion will potentially corrupt fewer bytes of actual data than the overflow from application memory will, causing the two overflows to manifest differently.

The second diversity transformation that DPMR utilizes is *zero-before-free*. Zero-before-free writes zeros to the bytes of a replica buffer prior to deallocation. Zero-before-free was chosen because of its potential to cause dangling pointer errors to manifest differently. In particular, if a read after free occurs to an application object X, before the memory that X and its replica X_r occupied is reallocated, the read from X will read data, while the read from X_r will read a zero value. Assuming the values stored in X are not zero, the two reads will differ, signaling an error.

The final diversity mechanism is *rearrange-heap*. When a replica heap object is allocated, rearrange-heap gives it a random location as defined in Table 2.8, rather than the standard location the heap allocator would normally give it. In the definition, the buffer `B` is a global variable that is allocated for use by rearrange-heap and that holds up to 20 pointers. Rearrange-heap is designed to help detect dangling pointer errors. By randomizing the location of replica objects, it decreases the likelihood that an application object X and its replica X_r will occupy the same pair of memory locations as a previously allocated (and freed) application object Y and its replica Y_r. If X is given the memory that Y previously occupied and a dangling pointer to Y and Y_r exists, when that dangling pointer pair is used, the result will be a function of X but will most likely not be a function of X_r.

2.7 State Comparison Policies

In this section, we examine the state comparison policies that DPMR deploys. With respect to the policies, a *load check* involves performing a replica load and comparing the result to an application load. It should be noted that under SDS there is no point in performing a replica load unless the result is compared to an application load. Therefore, either both the replica load and subsequent comparison occur, or neither occurs.

The default state comparison policy used by the standard DPMR transformations is the *all loads* policy, in which all of an application's loads are replicated and compared. In

Table 2.9: Temporal load-checking transformation

maskCounter ← global(int32) initGlobal(maskCounter, 0)

if(($mask$ << (64 - *maskCounter - 1)) >> (64 - 1)){
  assert(x == *p_r)
}
*maskCounter ← (*maskCounter + 1) % 64

Table 2.6, the comparison policy is executed with the assertion statements added by the load transformation. Since state comparison policies are one of the primary tunable components of DPMR, the goal of other state comparison policies is to create a performance versus dependability trade-off among all comparison policies. That can be done by designing comparison policies that limit the frequency of load checks.

One policy that limits that frequency is the *temporal load-checking* policy. Under the temporal load-checking policy, a temporal fraction of all application loads, during runtime execution, are replicated and compared. Temporal load-checking is accomplished by transforming the assertion statements from Table 2.6, into the code in the bottom half of Table 2.9. The code in the top half of Table 2.9 defines a global variable that is used by the code in the bottom half. The temporal load-checking code simply keeps a counter that runs through the bits in a mask. If the current bit is one, then the current load is replicated and compared; otherwise, the load check is skipped. DPMR evaluates temporal load-checking with 64-bit masks of `0x80808080` (checking 1/8 of the time), `0xAAAAAAAA` (checking 1/2 of the time), and `0xFEFEFEFE` (checking 7/8 of the time).

The final state comparison policy that DPMR evaluates is *static load-checking*. Under static load-checking, DPMR adds replication and comparison code for a load at compile time, with a given probability. For example, if load checks are performed 10% of the time, then for each load in the original program, DPMR generates a random floating pointer number r in $[0, 100)$ and inserts the check if $r \geq 90$.

2.8 External Code

DPMR is an interprocedural program transformation, i.e., it operates on a whole program. A problem with many interprocedural program transformations is that an entire program may not be available for transformation. For example, the source code may not be available for libraries installed on a system, or it may be undesirable to transform those libraries.

External code, to which DPMR has not been applied, can be broken down into two components: external functions and external global variables. During our treatment of those components, we often refer to *application-visible memory*, i.e., memory to which an application may receive a handle. It is differentiated from memory that an external function uses internally but from which the external function never allows a handle to escape to the calling application. For correctness, when accommodating external code, DPMR need concern itself with only application-visible memory. In addition, the discussion below does not pertain to the heap allocation and deallocation functions `malloc` and `free`, as they are specific cases in the DPMR transformation.

We now consider external functions. Non-transformed external functions pose three primary problems for applications that are transformed with DPMR: (1) If an external function allocates application-visible memory, corresponding replica and shadow memory will not be allocated. (2) Stores to application-visible memory performed by external functions will not be mimicked in an appropriate way to replica and shadow memory. (3) If an external function returns a pointer, a corresponding ROP and NSOP will not be computed and returned to the calling function. There are also two secondary problems: (1) If application-visible memory is deallocated by an external function, the corresponding replica and shadow memory will not be deallocated. That, of course, is a memory leak. (2) When an external function performs memory loads, DPMR will miss out on opportunities to perform load checks. While those missed load checks do not impact correctness, they may impact error coverage.

DPMR handles external functions by using *external function wrappers* when transforming a program. An external function wrapper W for an external function E is typed using the augmented type of E's function type: $\mathbf{at}(\mathbf{type}(E))$. When an application is being transformed, any references to E are replaced with references to W. An external function

```
1 int8[]* strcpy(int8[]* dest, int8[]* src);
2 int8[]* strcpy_efw(int8ArrayPtrSdwTy* rvSop,
3                    int8[]* dest, int8[]* dest_r, void* dest_s,
4                    int8[]* src, int8[]* src_r, void* src_s){
5   assert(strcmp(src, src_r) == 0); // src is read
6   int8[]* rv = strcpy(dest, src);  // perform original function
7   strcpy(dest_r, dest);            // dest is written
8   rvSop->rop = dest_r;             // set return value rop
9   rvSop->nsop = dest_s;            // set return value nsop
10  return rv;
11 }
```

Figure 2.11: External function wrapper for `strcpy()`

wrapper has two high-level responsibilities. First, it performs the behavior of its corresponding external function, most likely by calling that external function. Second, it performs application-visible DPMR behavior that would occur if the corresponding external function were transformed by DPMR. That application-visible DPMR behavior includes the following:

1. An external function wrapper is required to allocate replica and shadow memory corresponding to application-visible heap memory allocated by the external function.

2. Any store by the external function to application-visible memory should be mimicked in corresponding replica and shadow memory.

3. If the external function returns a pointer, a corresponding ROP and NSOP should be stored to the memory pointed to by the wrapper's *rvSop* argument.

4. Replica and shadow memory on the heap should be deallocated for any application-visible heap memory that is deallocated by the external function.

5. If application-visible memory is loaded by the external function, then the wrapper should load and compare corresponding replica memory.

An example of a wrapper is shown in Figure 2.11. The `strcpy()` function takes a pointer to an array of bytes, `src`, and copies the bytes to the memory pointed to by `dest`. The bytes pointed to by `src` are assumed to be terminated by a byte with a `'\0'` value. `strcpy()`

copies all bytes from `src` up to and including the first `'\0'` value. `strcpy()` completes by returning `dest`. `strcpy()`'s external function wrapper performs four tasks. Since `strcpy()` reads the bytes pointed to by `src`, `strcpy_efw()` verifies that `src_r` points to bytes with the same values. Second, the wrapper calls the original external function. Third, the wrapper writes to `dest_r` data that are equivalent to the data written to `dest` by `strcpy()`. Finally, the wrapper stores the ROP and NSOP corresponding to `strcpy()`'s return value to the memory pointed to by `rvSop`.

Since DPMR does not analyze the external functions upon which external function wrappers are based, it cannot auto-generate the wrappers. Therefore, wrappers must be distributed with DPMR or written by DPMR users. Either way, wrappers for an external library *lib* can be compiled into an *external code support library*, *libDpmrSupport*. *libDpmrSupport* can be linked into a program that links with *lib* and is transformed by DPMR. Such a support library has been constructed with wrappers for most functions in the C `libc` library.

In addition to external functions, DPMR must also handle external global variables. The problem that DPMR encounters with external global variables (that it does not process) is that corresponding replica memory and shadow memory are not allocated and initialized. That problem can be solved by declaring and statically initializing global variable memory for that purpose in DPMR external code support libraries. If the initial values of replica and shadow memory, corresponding to external global variables, are not known a priori, runtime initialization functions can be provided by DPMR external code support libraries to be called upon program start. Note that if external functions read and write external global variables, then the corresponding wrappers should read and write to the appropriate replica and shadow memory.

2.9 Limitations

Unfortunately, the SDS-based DPMR approach is not without its limitations. Those limitations are substantially relaxed with alternative designs discussed in Chapters 4 and 5. The SDS-based approach is evaluated first because, when applicable, it may be the preferred approach since it allows loads that return pointers to be compared. Below, we discuss input

program restrictions that are sufficient to allow application of SDS.

- *memory allocation*: At a memory allocation site A, the allocated type t must satisfy the equation below, such that T is the set of all types that the result of A could hold during execution.

$$\mathbf{sizeof}(\mathbf{st}(\mathbf{at}(t))) \geq \max_{\tau \in T}(\mathbf{sizeof}(\mathbf{st}(\mathbf{at}(\tau))))$$

  The above requirement ensures that shadow memory associated with A will be large enough. It should be noted that another method to ensure that enough shadow memory is allocated is to allocate $2 \times \mathbf{sizeof}(\mathbf{at}(t))$ bytes of memory for the shadow object. That allocates enough shadow memory to handle the case in which the allocated type contains only pointers. However, such an allocation strategy is quite wasteful.

- *store*: When pointers are stored to memory, they must be typed as pointers.[2] When non-pointer values are stored to memory, they must be typed as non-pointers. If those conditions are met, we can apply the appropriate transformation for stores as shown in Table 2.6.

- *load*: When pointers are loaded from memory, they must be typed as pointers,[3] and when non-pointers are loaded from memory, they must be typed as non-pointers. Much like the requirements for stores, those requirements allow us to choose the appropriate load transformation from Table 2.6.

- *structure and array pointer arithmetic*: For a pointer p to which structure or array pointer arithmetic is applied, p must be typed in a manner that allows us to appropriately address shadow memory. To precisely define that restriction, let $\sigma()$ be a function such that $\sigma(t)$ is a structure that is composed of only scalar types and is structurally equivalent[4] to t. Let $\mathbf{type}(p) = s*$ and $\sigma(\mathbf{at}(s)) = \text{struct}\{t_0\ f_0; \ldots ; t_n\ f_n;\}$. Let i be the index of the field to which the pointer arithmetic advances p. Let D be the set

[2] That does not mean that the pointers need to be precisely or even accurately typed.
[3] Again, the pointers do not need to be precisely or even accurately typed.
[4] In this context, pointers and non-pointers are not structurally equivalent.

of types to which p may point at runtime, in the original program. For each $d \in D$, where $\sigma(\mathbf{at}(d)) = \text{struct}\{\tau_0\ e_0; \ldots; \tau_m\ e_m;\}$, we require that there exist a field e_j such that

$$\mathbf{sizeof}(\text{struct}\{t_0; \ldots; t_{i-1};\}) = \mathbf{sizeof}(\text{struct}\{\tau_0; \ldots; \tau_{j-1};\}).$$

If such a field does exist, then we also require that

$$\sum_{k=0}^{i-1} \text{isPointer}(t_k) = \sum_{l=0}^{j-1} \text{isPointer}(\tau_l).$$

- *pointer-to-pointer casts:* For a pointer-to-pointer cast $p = (\alpha*)q$, we require type restrictions if q originated from pointer arithmetic. If

$$\sum_{k=0}^{n} \text{isPointer}(t_k) = 0 : \mathbf{type}(q) = \beta*, \sigma(\mathbf{at}(\beta)) = \text{struct}\{t_0; \ldots; t_n;\},$$

  then we require that

$$\sum_{l=0}^{m} \text{isPointer}(\tau_k) = 0 : \sigma(\mathbf{at}(\alpha)) = \text{struct}\{\tau_0; \ldots; \tau_m;\}.$$

  That restriction ensures that if the pointer to q's shadow object has a null value, then we will not cast q to a type whose shadow object has a nonzero size.

- *int-to-pointer casts*: Int-to-pointer casts are not allowed, because DPMR would have no way to set corresponding ROPs and NSOPs.

- *external code*: SDS requires external function wrappers for all external functions utilized by an input program. In addition, replica and shadow memory must be allocated and initialized for all external global variables with which a program interacts or the external functions that it calls interact.

Chapter 3

Construction and Evaluation of Diverse Partial Memory Replication

This chapter discusses the construction of a DPMR-based C compiler, followed by an experimental evaluation of DPMR. In particular, Section 3.1 addresses issues that arise when applying DPMR to C programs, and Section 3.2 details the design of a DPMR-based C compiler built on the LLVM [61] compiler framework. The next four sections present the experimental framework, fault-injection framework, build procedures, and metrics that are used when experimentally evaluating DPMR. Sections 3.7 and 3.8 analyze results from that experimental evaluation.

3.1 Details Pertaining to C

In this section, we discuss several details that pertain to the application of DPMR to C programs. Those details include how we handle `main()`, functions with variable-length argument lists, convenient modifications that support use of `libc`, external global variables from `libc`, and interesting external function wrappers.

3.1.1 Handling Main

There are two nonstandard details for which DPMR must account when transforming the `main()` function of a C program. First, the function type of `main()` must not be changed. Second, at the beginning of a program, replica memory and shadow memory will not exist for command-line arguments. To handle the first problem, DPMR renames `main()` to a unique name *mainAug*, prior to transforming a program. Any references to `main()` are replaced with references to *mainAug*. Then, DPMR performs its standard transformations,

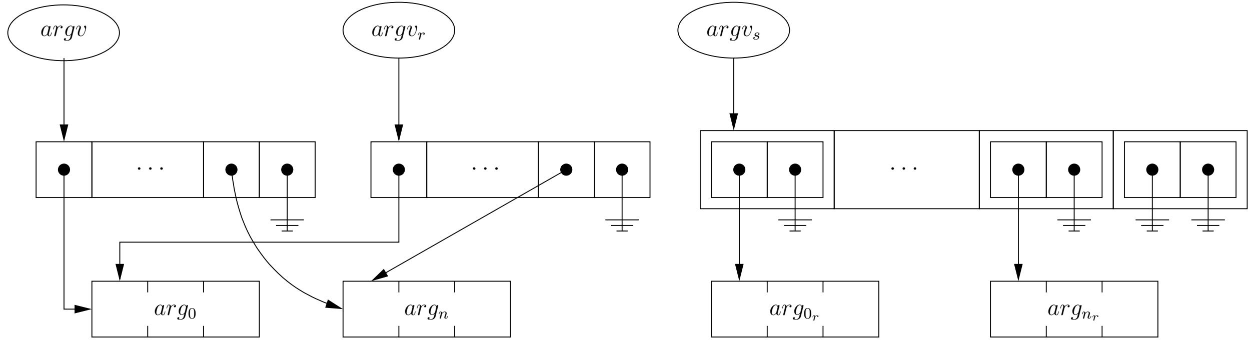

Figure 3.1: DPMR memory for command-line arguments

the result of which contains the C function declaration:

int *mainAug*(int argc, char** argv, char** argv_r, {char*, void*}* argv_s)

Afterwards, DPMR creates a new `main()` that calls *mainAug*. That leads us to the second problem: command-line arguments have not been replicated.

Unfortunately, the compiler cannot replicate command-line arguments at compile time because it does not know their values. Therefore, prior to `main()`'s calling of *mainAug*, DPMR inserts code to create and initialize replica and shadow memory corresponding to the program's command-line arguments. The formatting of that memory is depicted in Figure 3.1. `argv_r` points to an array containing values identical to those in the array to which `argv` points. `argv_s` points to an array that contains two pointers for every pointer in the array pointed to by `argv`. `&(argv_s[i].first)` points to a replica of the ith command-line argument.

3.1.2 Functions with Variable-Length Argument Lists

Another aspect of C that DPMR must plan for is the ability of functions to have variable-length argument lists. Before discussing how DPMR can address variable-length argument lists, we must review the constructs that C has put in place to handle them. A `va_list` is a C data structure that maintains data that are necessary to access a variable-length argument list. `va_start()`, `va_end()`, and `va_copy()` are macros that are used to initialize, destroy, and copy a `va_list`. `va_arg()` is a macro that extracts an argument from a variable-length argument list.

A strategy for handling variable-length argument lists is to have DPMR leave `va_list` objects, `va_start()` calls, `va_end()` calls, and `va_copy()` calls untransformed. `va_arg()` calls that extract non-pointer arguments can also be left untransformed. For a `va_arg()` call that extracts a pointer argument, we transform the call so that an ROP and an NSOP, as well as an application pointer, are extracted from a variable-length argument list. Since DPMR transforms function arguments so that a pointer argument is immediately followed by the ROP and NSOP corresponding to that pointer, we can replace `va_arg()` calls that extract pointer arguments with three `va_arg()` calls: the first will extract an application pointer; the second will extract the corresponding ROP; and the third will extract the corresponding NSOP.

The above strategy works fine for code that is transformed by DPMR; however, a problem presents itself in external function wrappers for functions that accept variable-length argument lists. Consider a function call `printf("%p %d",` p`, 1)`, to which p is a pointer, and `printf()` has the C type `int (const char*, ...)`. Under DPMR, that call will be transformed to `printf_efw("%p %d", "%p %d", NULL,` p`,` p_r`,` p_s`, 1)`, where `printf_efw()` has the C type `int (const char*, const char*, void*, ...)`. Inside `printf_efw()`, we would like to call `vprintf("%p %d",` va`)`,[1] where va is a `va_list` referencing a variable-length argument list containing $\{p, 1\}$; however, we do not have a way to construct va from the variable-length argument list $\{p, p_r, p_s, 1\}$ that is passed to `printf_efw()`. To get around such problems, system-specific macros `va_new()` and `va_append()` could be provided for use in external function wrappers. `va_new()` would return a `va_list` that references an empty variable-length argument list. `va_append()` would take a `va_list` O and a data value D as parameters and would return a `va_list` that references a variable-length argument list containing the contents of the list that O references followed by D. Since the implementation of `va_list` is system-dependent, the implementation of `va_new()` and `va_append()` would also be system-dependent.

An alternative solution is to change the way function calls are transformed, so that variable-length argument lists from the original application stay intact. That is the approach

[1] It is not possible for `printf_efw()` to pass its variable-length argument list directly to a `printf()` call. The prescribed solution to such a situation is for `printf_efw()` to create a `va_list` and pass it to `vprintf()`.

```
1 if(fp == f0 || fp == f1){
2   (*fp)(a, b, c);             // fp type is void(*)(void*, ...)
3 }
4 else if(fp == g0 || fp == g1){
5   (*fp)(a, b, c);             // fp type is void(*)(void*, void*, ...)
6 }
```

Figure 3.2: Disambiguated call type

used for the current implementation of DPMR. Table 2.7 proposed that we transform a call site argument list $\{a_0, \ldots, a_n\}$ to $\{a_0, a_{0_r}, a_{0_s}, \ldots, a_n, a_{n_r}, a_{n_s}\}$. In the new solution, we continue to do that for fixed arguments; however, we transform a variable-length argument list $\{v_0, \ldots, v_m\}$ to $\{v_0, \ldots, v_m, v_{0_r}, v_{0_s}, \ldots, v_{m_r}, v_{m_s}\}$.

There are two potential problems with the alternative solution. The first potential problem is that it invalidates our strategy for transforming functions that accept variable-length argument lists. However, that was not an issue in the applications that we transformed with DPMR because those applications do not define their own functions that accept variable-length argument lists. The second potential problem arises if a call site calls a function that accepts a variable-length argument list through a function pointer. In that case, if not all of the possible called functions have the same number of fixed pointer arguments, then we will not know which ROPs and NSOPs correspond to variable arguments and should be placed at the end of the call site's argument list. The easiest way to handle that problem is to have the compiler disambiguate function call types. For example, imagine a function call `(*fp)(a, b, c)`, where `fp` could point to `f0`, `f1`, `g0`, or `g1`, and `a`, `b`, and `c` are pointers. Furthermore, the declared type of `f0` and `f1` is the C type `void(*)(void*, ...)`, and the declared type of `g0` and `g1` is the C type `void(*)(void*, void*,...)`. Then, prior to the DPMR transformation, the compiler could transform `(*fp)(a, b, c)` into the code in Figure 3.2. Through that approach, each call site uses one consistent function type.

3.1.3 Convenient Modifications

We now discuss two modifications that are used in the DPMR implementation to simplify its implementation for C input programs. The first modification concerns the C `FILE` type.

In C programs, `FILE` pointers are passed to and/or returned by library functions such as `fopen()`, `fclose()`, and `fprintf()`. However, many input C programs do not dereference `FILE` pointers themselves. If that invariant is upheld, we can treat a `FILE` object as an array of bytes when implementing external function wrappers. Such treatment is convenient because it means that we can set the NSOP associated with a `FILE` pointer (a type that could be system-dependent) to null. Also, when we are writing external function wrappers for external functions that dereference `FILE` pointers, the DPMR behavior corresponding to those dereferences becomes much simpler.

Another modification we make is the use of *external function placeholders*. External function placeholders are wrappers around external code that preserve a functional interface that may be lost during compilation. For example, `libc` has several functions that check a character for certain properties, such as `isblank()` and `isdigit()`. When generating object code for calls to those functions on Linux, `gcc` transforms them into a series of instructions that load memory returned by a call to a function called `__ctype_b_loc()`. It is simpler to create an external function wrapper for `isblank()` than for `__ctype_b_loc()`. Therefore, to preserve the `isblank()` interface, prior to running DPMR, we replace references to `isblank()` in an input program with calls to an external function placeholder, `isblank_efph()`. At the time of that function replacement, we declare `isblank_efph()` but leave it undefined. That way, it appears as external code. Then, when DPMR runs, it can replace calls to `isblank_efph()` with an external function wrapper that calls `isblank()`.

3.1.4 External Global Variables

As mentioned in Section 2.8, replica and shadow memory must be explicitly allocated and initialized for external global variables. Four external global variables from `libc` often show up in applications: `stdout`, `stderr`, `stdin`, and `errno`. DPMR includes functions that initialize the replica and shadow memory associated with those global variables in its `libc` external code support library, rather than including system-dependent static initializations. Thus, after transforming a C program, DPMR inserts calls to those initialization functions at the start of `main()`. DPMR also includes functions in its `libc` external code support

library to update the replica and shadow data associated with `stdout`, `stderr`, `stdin`, and `errno`, for use when writing external function wrappers that write to those global variables. DPMR simplifies the implementation of those initialization and update functions by treating `FILE` objects as arrays of bytes, as suggested in Section 3.1.3.

`errno` is a unique external global variable for two reasons. First, many `libc` functions write to it. Therefore, when writing external function wrappers, it is important to pay attention to the specific conditions under which `errno` may be written. For example, when `strtol()` is called, `errno` may have been set if `strtol()`'s return value equals `LONG_MIN`, `LONG_MAX`, or 0, or if `strtol()`'s base argument is 1, less than 0, or greater than 36. In addition, `malloc` itself may set `errno`. Therefore, DPMR adds checks after every `malloc` call, comparing the result to null. If the result is null, DPMR updates `errno`'s replica.[2] The second unique aspect of `errno` is that if the input program does not read `errno` and does not call the few functions that do read `errno`, such as `perror()`, then as an optimization, we can eliminate any initialization and updates of `errno`'s replica.

3.1.5 Interesting External Function Wrappers

The last aspect of DPMR's application to C programs that warrants discussion is interesting external function wrappers. In Section 2.8, we gave an example of a fairly simple external function wrapper. Most external function wrappers are simple; however, some are not, and we discuss several of those here. As noted in Section 2.8, `malloc` and `free` are not considered during the writing of external function wrappers.

- `printf()` and `scanf()`: The functions in the `printf()` and `scanf()` families require complex external function wrappers because those functions utilize variable-length argument lists that may contain pointers that are dereferenced. When considering the variable-length argument lists passed to those functions, DPMR is concerned only with pointers. In order to find pointers in the variable argument lists and determine if those pointers are used in loads or stores, DPMR must parse the format strings passed to

[2]Since `errno` is an integer stored in memory, there are no shadow data to update. Also, it is not always an error when malloc returns null; however, it does not affect correctness to update `errno`'s replica.

the `printf()` and `scanf()` family functions.

One problem with such an implementation is that format strings may be incorrect. However, if the format string is incorrect, then any erroneous DPMR behavior that results will potentially allow DPMR to detect the improper format string error. Another problem that can occur is that when a wrapper executes a `printf()` or `scanf()` family function, that function may abort due to an environment error, such as an I/O error. The problem is that the wrapper may not know how many of the arguments the `printf()` or `scanf()` family function processed. However, under such circumstances, it is safe for the wrapper to perform load checks and store updates for pointer arguments that would be loaded and stored, respectively, under normal operation.

It is also worth noting that `printf()` and `scanf()` family functions may change `FILE` objects; however, we may not know precisely what portions of those objects are changed by the `printf()` or `scanf()` call, without comparing the `FILE` object's states before and after the call. We can eliminate the comparison by conservatively assuming that the entire `FILE` object is changed by the `printf()` or `scanf()` call. If we also assume that `FILE` objects are byte arrays, as suggested in Section 3.1.3, then we can update a `FILE` object's DPMR data by simply copying the `FILE` object's contents to the replica `FILE` object.

- `strcmp()` and `atof()`: The `strcmp()` and `atof()` wrappers are difficult to write because `strcmp()` and `atof()` do not always indicate how much of their string arguments have been read. For example, when two unequal strings are compared, `strcmp()` will eventually return a nonzero value. However, that return value does not indicate how much of the input strings were read to determine that they were not equal. The DPMR external function wrapper needs to know how much of a string has been read so that it knows how much of that string to compare to its replica. A similar situation occurs with `atof()`. `atof()` converts a string to a floating point value. However, it stops reading an input string when it encounters a character that is not part of that floating point value. A wrapper for `atof()` compares the input string to its replica for as much of the input string as was read. The solution to the problems posed by `strcmp()` and

atof() is to have their external function wrappers emulate their string-parsing behavior. In doing so, the wrappers will be able to determine how much of the underlying string has been read. Note that if we emulate strcmp()'s string parsing behavior to determine how much of its input strings it reads, then we are doing most of the work that strcmp() does. Therefore, we implement strcmp() in its wrapper instead of explicitly calling strcmp() from the wrapper.

There is a common misunderstanding that wrappers can just read input strings until they encounter a '\0'. However, there is no guarantee that input strings will be terminated by a '\0'. In fact, a program may pass nonterminated strings to strcmp() and atof(), which it knows will terminate processing for some other reason.

- fopen() and fclose(): fopen() is unique because it returns a pointer to a FILE object. Therefore, an external function wrapper must allocate corresponding replica memory (and shadow memory if FILE objects are not treated as arrays of bytes). Since we do not know how long the FILE object will live, we allocate its replica on the heap. The question that arises is, when do we deallocate the replica? Since fclose() effectively renders a FILE object useless, we choose to deallocate the replica (and any shadow memory) at fclose(), despite the fact that fclose() does not necessarily deallocate the FILE object passed to it.

- qsort(), memcpy(), and memmove(): qsort(), memcpy(), and memmove() each write to memory in a type-generic manner. The problem with doing so is that it can violate one of the restrictions of DPMR noted in Section 2.9, namely that pointers stored to memory must be stored to memory as pointers. To handle that problem, we include an extra parameter in the wrappers for qsort(), memcpy(), and memmove() that indicates the sizes of shadow objects of parameters of those functions. The extra parameter enables us to appropriately update shadow memory.

  qsort()'s extra parameter indicates the size of the shadow objects that correspond to the elements that qsort() sorts. The modified external function wrapper declaration is shown in Figure 3.3, and the extra parameter is the sdwSize parameter. The extra

```
1 typedef int (*CmpFTy)
2   (void*, void*);
3
4 void qsort(void* base, size_t nmemb,
5   size_t size, CmpFTy cmp);
6
7
8
9
```

(a) Declaration

```
1 typedef int (*CmpAugFTy)
2   (void*, void*, void*,
3    void*, void*, void*);
4 void qsort_efw(size_t sdwSize,
5   void* base, void* base_r,
6   void* base_s,
7   size_t nmemb, size_t size,
8   CmpAugFTy cmp, CmpAugFTy cmp_r,
9   void* cmp_s);
```

(b) External function wrapper declaration

Figure 3.3: Declaration of the `qsort()` external function wrapper

parameter for `memcpy()` and `memmove()`'s external function wrappers indicates the entire size of the shadow memory that corresponds to the memory that is to be copied or moved.

The value of the extra parameter that is passed to the external function wrappers of `qsort()`, `memcpy()`, and `memmove()` is determined by the compiler at each call site based on the real type of the memory passed to each of those functions. In order to determine that type, we must impose additional restrictions similar to the ones presented in Section 2.9.

A function type inconsistency could occur if `qsort()`, `memcpy()`, or `memmove()` were called through a function pointer. That problem never occurred in the applications we transformed. However, if it did, it would be possible to create a per-thread global variable for each of those functions that could store the extra parameter. Then, the extra parameter would not need to be added to the external function wrappers of `qsort()`, `memcpy()`, and `memmove()`.

3.2 Tool Design

DPMR is implemented using the LLVM compiler framework [61]. We make use of three components of LLVM, namely a source code frontend, the `opt` transformation and analysis engine, and a native code backend. The frontend that we utilize is `gcc`-based. It takes C

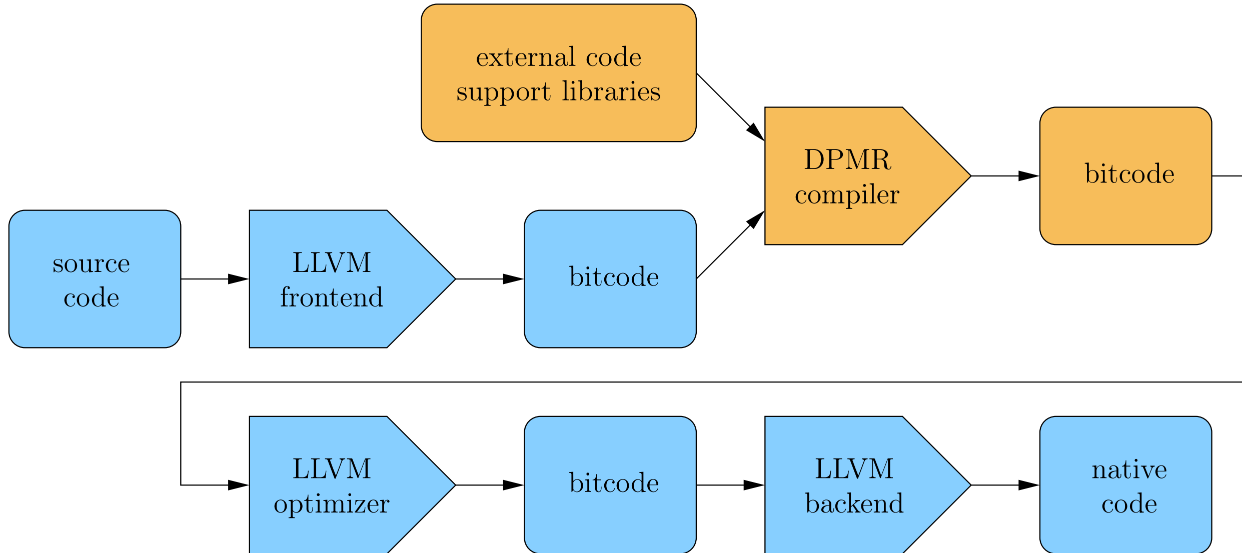

Figure 3.4: DPMR tool chain

code and transforms it to bitcode, LLVM's intermediate representation. The transformation and analysis engine is used to analyze bitcode and/or transform it, while the backend takes bitcode and turns it into a native library or native executable.

DPMR consists of two components. The first is the DPMR compiler, which is built on top of LLVM's transformation and analysis engine. The second is DPMR's collection of external code support libraries. The DPMR tool chain is depicted in Figure 3.4. The source code for an input program is scanned and parsed by the `gcc`-based LLVM frontend, producing bitcode. The bitcode, along with DPMR's external code support libraries, is fed into the DPMR compiler, where DPMR transformations are applied. The output is DPMR-transformed bitcode. That bitcode is passed through LLVM's optimizer (built on `opt`), resulting in optimized bitcode. Finally, that optimized bitcode is passed through a backend to produce a native executable or library.

3.3 Experimental Framework

An important part of this dissertation is the experimental evaluation of DPMR. In particular, we aim to evaluate the performance and dependability characteristics of DPMR under different diversity transformations and state comparison policies. For the experi-

Table 3.1: Testbed specifications

CPU	AMD Athlon XP 2400+
CPU speed	2 GHz
memory size	512 MB
L2 cache size	256 KB
operating system	Ubuntu 8.10 Linux

mental evaluation, we conducted two types of experiments: non-fault-injection experiments and fault-injection experiments. The non-fault-injection experiments were used to evaluate the performance of DPMR, and the fault-injection experiments were used to evaluate the dependability properties of DPMR.

All experiments were conducted on a testbed of identical machines with the setup shown in Table 3.1. The experiments were conducted when the machines were not running any other workloads, other than standard system processes and daemons. Experiments evaluated the following C programs from the SPEC CPU 2000 benchmarks suite [62]. Each application was run under the train workload included with the SPEC benchmarks.

- *art:* Art is a floating-point application that uses a neural network to recognize objects in a thermal image.
- *bzip2:* Bzip2 is the standard compression utility, but it has been modified by SPEC to perform all compression and decompression entirely in memory. It is an integer-based benchmark.
- *equake:* Equake is a floating-point simulation of seismic waves in large basins.
- *mcf:* Mcf is an integer-based benchmark that performs a combinatorial optimization to determine vehicle scheduling for a public transportation company.

3.4 Fault-Injection Framework

As previously mentioned, fault-injection experiments were used to evaluate the dependability properties of DPMR. The fault-injection framework we used is compiler-based and is designed

to simulate software bugs. The idea was to inject into a program faulty code that will be present every time the injected location is executed. That strategy is different from many runtime fault-injection strategies that inject a fault once during an experiment, rather than every time a targeted location in the code executes. Such runtime strategies are insufficient for modeling software memory faults.

All fault injections were inserted into an input program prior to the transformation of that program with DPMR, just as real software bugs would be present prior to transformation. To cut down on the number of program variants needed for the experiments, we compiled a fault-free version of a program and a faulty version that contains faulty and fault-free code at each injection site. Command-line arguments were used to enable faulty code at specified locations. Results in this dissertation were based on the following fault-injection types.

- *heap array resize:* A heap array resize reduces the number of objects requested at a heap array allocation site. The request size can be reduced by a percentage or by a static amount. Heap array resizes lead to out-of-bounds accesses.

- *immediate free:* An immediate free deallocates a heap buffer immediately following its allocation. Immediate frees lead to reads, writes, and frees after free.

Separate experiments were conducted for each combination of fault-injection type and possible fault-injection location. Heap array resizes that reduce heap array requests by 50% were injected at all heap array allocation sites. In addition, immediate frees were injected at all heap allocation sites.

It is important to note that not all enabled injections will manifest as errors. For one thing, the code containing an enabled injection may never execute. It may be in a part of the application that is not exercised by the current workload. It is also possible for a fault to be executed without manifesting as an error. For example, consider a heap array resize in which the initial malloc request is for 24 bytes, and the resize reduces it to a 16-byte request. It is common for malloc implementations to have a minimum buffer allocation size. Let's say that size is 24 bytes. Then, 24 bytes will still be allocated for the reduced 16-byte request. Static analysis is used to filter out parameters for fault injections that will definitely not manifest, but not all request sizes are static.

3.5 Variant Builds

In order to evaluate DPMR, an application was compiled into different variants. Each variant was identified by the configuration under which it was compiled. Configurations varied by whether they were compiled for fault injection and whether they were transformed with DPMR. If a variant was transformed with DPMR, its configuration also included the diversity transformation and state comparison policy applied to it. Variants were broken down into four classes. Figure 3.5 depicts the compilation strategy for each of the four classes. The compilation strategy is presented so that the reader is aware of any transformation to and instrumentation of the original application.

The first class is represented by the rightmost path in the figure. It consists of exactly one variant, which is an unmodified version of the original application compiled under a standard compilation strategy. It is referred to as the *golden* class, or golden application. The second class is the *fi-stdapp* class, represented by the leftmost compilation path. Again, it is composed of one variant. That variant is a version of the golden application that has been transformed for fault injection. In addition, logging and timing support have been added for use with fault injections. The third class is the *nofi-dpmr* class. It represents a group of variants that have not been modified for fault injection but have been transformed for DPMR. The different variants are distinguished by the diversity transformation and state comparison policy that DPMR applies to them. They do not require timing support, since they do not involve fault injections. The third path from the left in the figure illustrates their compilation path. The final class is the *fi-dpmr* class. The second compilation path from the left in Figure 3.5 applies to it. It represents a group of variants that are compiled for fault injection and transformed with DPMR. As with nofi-dpmr, the variants are distinguished by the diversity transformation and state comparison policy that are applied to them. Since fi-dpmr variants are compiled for fault injection, they require initialization for both logging and timing.

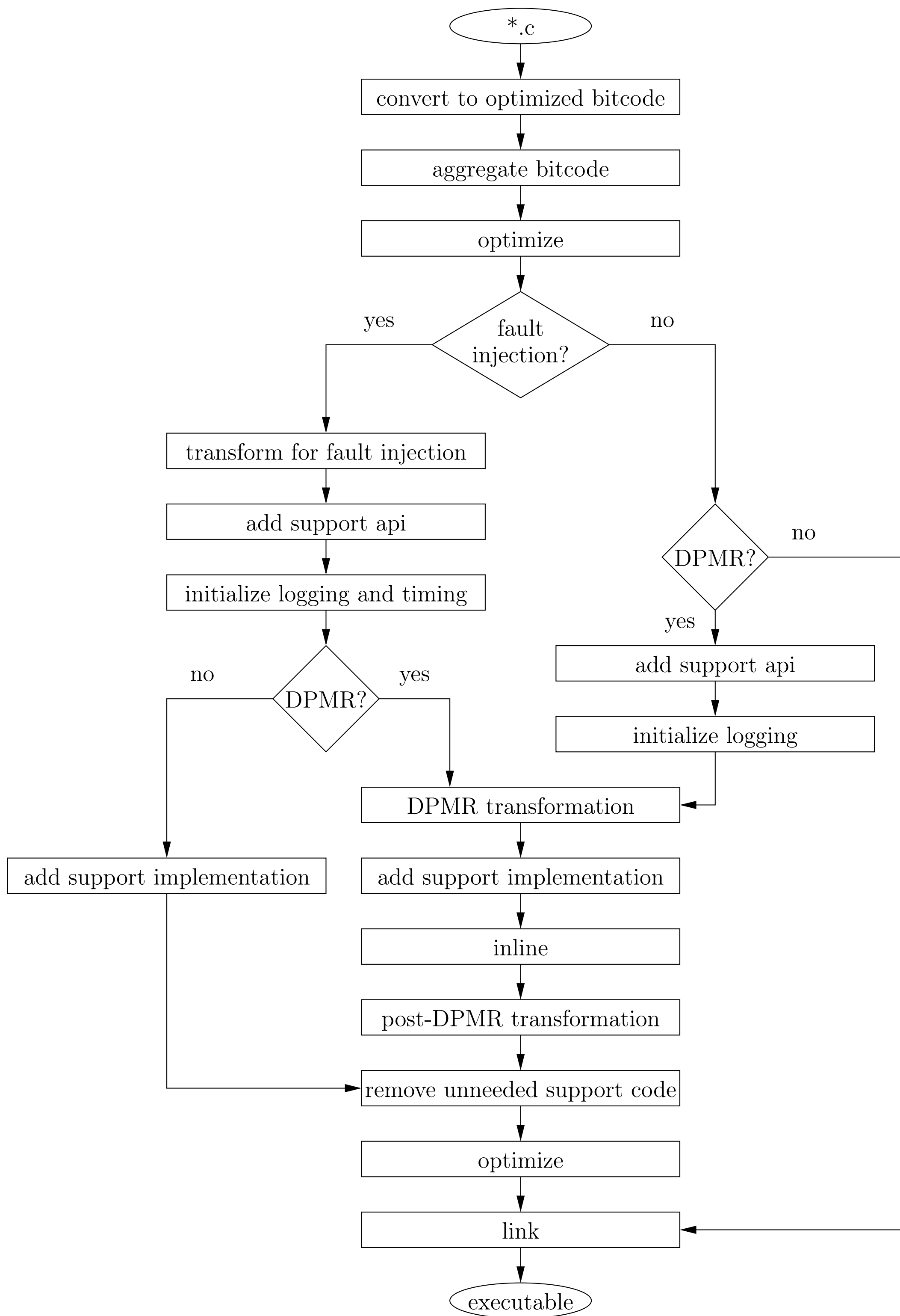

Figure 3.5: Compilation strategy

3.6 Evaluation Metrics

In this section, we discuss the measurements taken in our experiments and how evaluation metrics were derived from those measurements. An experiment is one run of an application variant. Each experiment is identified by the tuple (W, C, D, I, RN). W is the workload. C is the comparison policy applied to the variant if the variant is transformed with DPMR. A comparison policy of $\varnothing$ indicates that DPMR was not used. If a variant is transformed with DPMR, D is the diversity transformation that is applied to it. $\varnothing$ indicates that no explicit diversity is applied. I indicates the type of fault that is injected into the target variant, if any, and RN indicates the run number under the settings (W, C, D, I).

In every experiment, the precise exit status and wall clock running time were recorded. For non-fault-injection experiments, the exit status was always a normal exit; but for fault-injection experiments, it could also have been a signal exit or a timeout. Timeout values were set to approximately 20 times the normal running time of an application under the applied workload. The wall clock time was measured with the Linux `clock_gettime()` system call, using the `CLOCK_REALTIME` parameter. In fault-injection experiments, the time was measured between the start of an application and the first execution of fault-injection code. That measurement is computed using the x86 read time stamp counter instruction (`rdtsc`). Since the `rdtsc` instruction is simply a read of a CPU register, it is preferred over more intrusive system calls. Measurements recorded using `rdtsc` were converted from clock ticks to wall time using the CPU speed, as read from the Linux `/proc/cpuinfo` file.

Table 3.2 describes notation, experiment descriptors, and random variables that are used to define metrics. Most items in the table are self-explanatory, but a few require further explanation.

- *successful fault injection:* A fault is considered successfully injected if the corresponding fault-injection code is executed at least once during the experiment. A successful fault injection does not imply the manifestation of an error. In fact, without intrusive instrumentation, error manifestation cannot be precisely determined. Therefore, error manifestation is not computed.

Table 3.2: Measurement components

Notation	
$P_s[X]$	sample density function for the random variable X
$\mu[X]$	sample mean of X
Experiment Descriptors	
C	current state comparison policy
D	current diversity transformation
F	true for a fault-injection experiment
Random Variables	
T	running time of the current experiment
SF	true for a successful fault injection
CO	true if the experiment produces the correct output
$Ndet$	true if a fault is naturally detected
$Ddet$	true if a fault is detected by DPMR
$T2D$	time to fault detection
$StdNotAllDet$	true if $P_s[(Ndet \mid C = \varnothing, D = \varnothing, F, SF, \neg CO) = 1] < 1$

- *correct output:* Correct output indicates that an experiment produced the output that the golden run would have produced under the same workload. Therefore, incorrect output includes not only “bad” results but also error detection. It is a literal interpretation of the term.

- *natural detection:* Natural error detection is error detection that is implicit to the application under study and can occur in two ways. An error is naturally detected if the application variant in an experiment crashes. An error is also naturally detected if the variant produces application-dependent output indicating that an error was detected. Such output might be an error message written to stderr or an exit with an error-identifying return value.

- *time to fault detection:* Time to fault detection is computed by subtracting the time between an experiment’s start and the first successful fault injection, from the total experiment time. Two points require further clarification. First, the time to error detection is not computed because, as mentioned earlier, there is not a nonintrusive way to determine when a fault has manifested as an error. Second, time to detection cannot be directly computed because, if a variant abnormally exits, there is no way to

record the value of a timer.

- *StdNotAllDet*: *StdNotAllDet* is true for a given fault injection if, under the nofi-stdapp variant, there is at least one run that produces incorrect output but not natural error detection.

Below are the measurements used to evaluate DPMR.

- *overhead:* Overhead is defined in the non-fault-injection case, for a given comparison policy c and diversity transformation d, as the mean execution time divided by the mean execution time of the golden variant. More formally, overhead is given by

$$overhead \equiv \frac{\mu[T|C = c, D = d, \neg F]}{\mu[T|C = \varnothing, D = \varnothing, \neg F]}. \tag{3.1}$$

- *coverage:* A fault is considered covered in successful fault-injection experiments, for a given comparison policy c and diversity mechanism d, if the experiment produces correct output or an error is detected. It is formally defined as

$$coverage \equiv CO \vee Ndet \vee Ddet|C = c, D = d, F, SF. \tag{3.2}$$

  In addition to overall coverage, each of its components is examined, i.e., $[CO|(C = c, D = d, F, SF)]$, $[(Ndet, \neg CO)|(C = c, D = d, F, SF)]$, and $[(Ddet, \neg CO)|\ (C = c, D = d, F, SF)]$.

- *conditional coverage:* Conditional coverage is coverage conditioned on *StdNotAllDet*. It is important because it measures how a variant would perform in cases where fi-stdapp would have produced incorrect output but would not always have detected an error. Conditional coverage is defined as

$$conditional\, coverage \equiv CO \vee Ndet \vee Ddet|C = c, D = d, F, SF, StdNotAllDet. \tag{3.3}$$

- *detection latency:* The final major metric of interest is detection latency. Detection latency is defined in covered experiments that involve some sort of detection. Formally,

detection latency is

$$detection\,latency \equiv T2D | C = c, D = d, F, SF, \neg CO, (Ndet \vee Ddet). \tag{3.4}$$

3.7 Diversity Results

In this section, we present an experimental evaluation of the diversity transformations described in Section 2.6. In particular, we examine eight application variants. One variant is the standard application without DPMR. The second variant has been transformed with DPMR but has had no additional diversity transformations applied to it. It relies on the implicit diversity afforded by DPMR's intra-process replication scheme. Each of the other six variants has been transformed by DPMR and has had one of the following diversity transformations applied to it: zero-before-free, rearrange-heap, pad-malloc 8, pad-malloc 32, pad-malloc 256, or pad-malloc 1024. Each variant that has been transformed with DPMR uses the all loads state comparison policy.

We begin the evaluation of diversity transformations by examining coverage. Mean coverage of heap array resizes is displayed in Figure 3.6, and mean coverage of immediate frees is displayed in Figure 3.7. Coverage results are broken into correct output (blue), natural detection (yellow), and DPMR detection (green). Figures 3.8 and 3.9 offer mean coverage results across all studied applications, conditioned on incorrect output and $StdNotAllDet$.

We draw several observations and insights from the coverage graphs. First, coverage numbers are high, even in the standard application (without DPMR). Coverage is high in the stdapp variant because many faults lead to natural detection, such as a crash, or to correct output. There are two primary reasons that an experiment might result in correct output despite a successful fault injection. One reason is that memory corruption resulting from a fault injection might not impact application or DPMR data that will be read before the corruption is overwritten. Such behavior affects both heap array resizes and immediate frees. The second reason that a successful fault injection might result in correct output was briefly discussed in Section 3.4. Heap allocators commonly have predetermined fixed

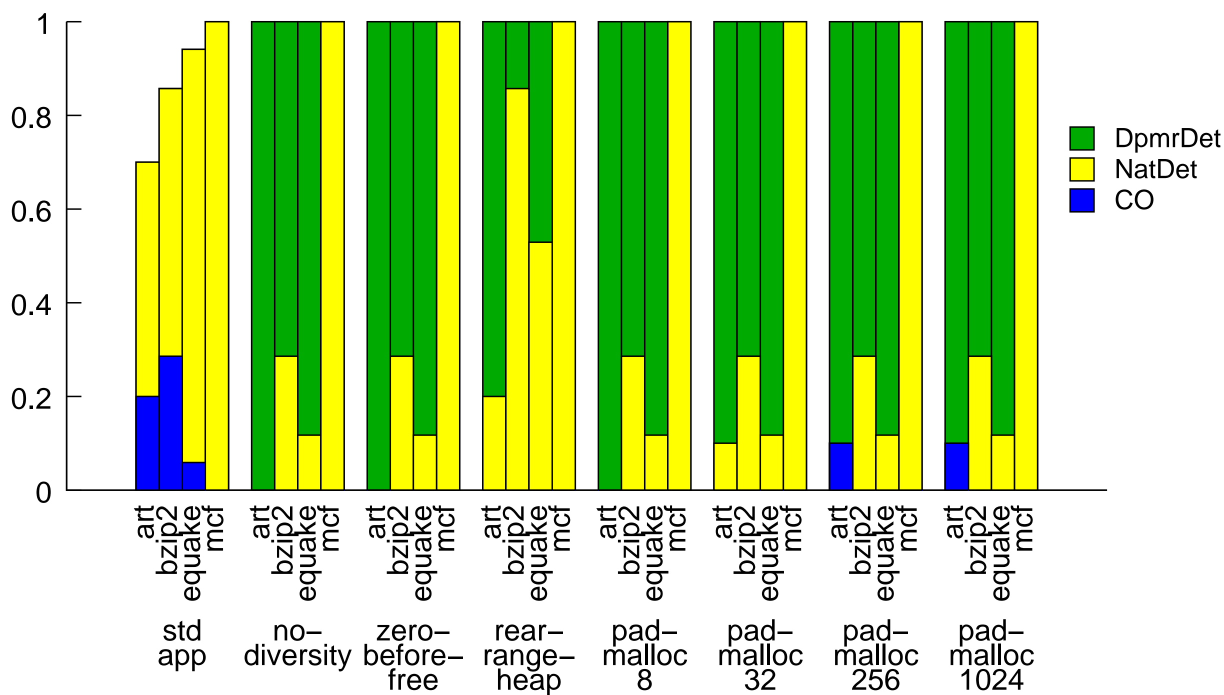

Figure 3.6: Mean heap array resize coverage of diversity transformations

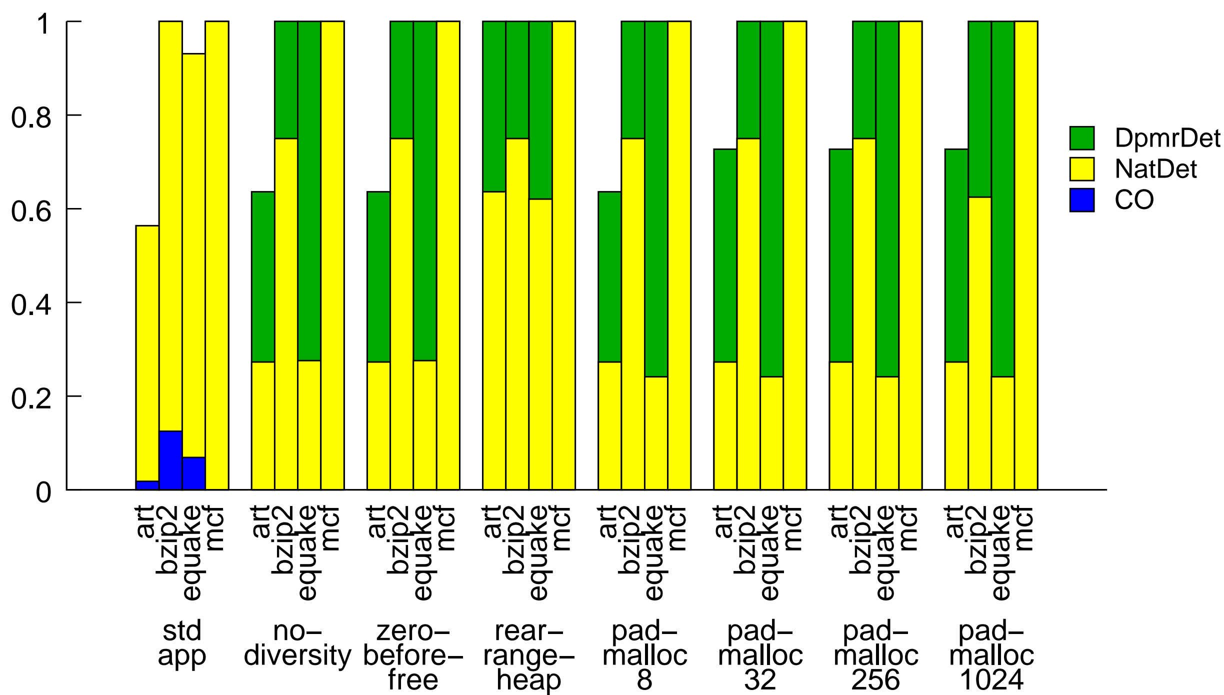

Figure 3.7: Mean immediate free coverage of diversity transformations

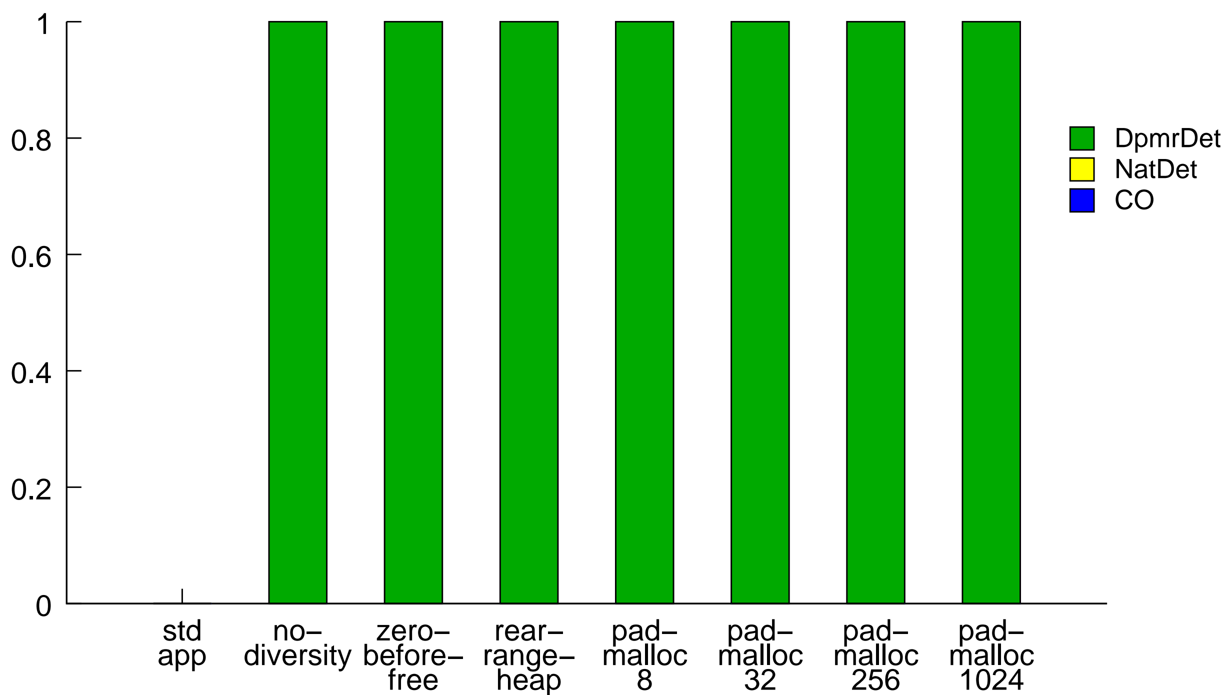

Figure 3.8: Mean heap array resize conditional coverage of diversity transformations

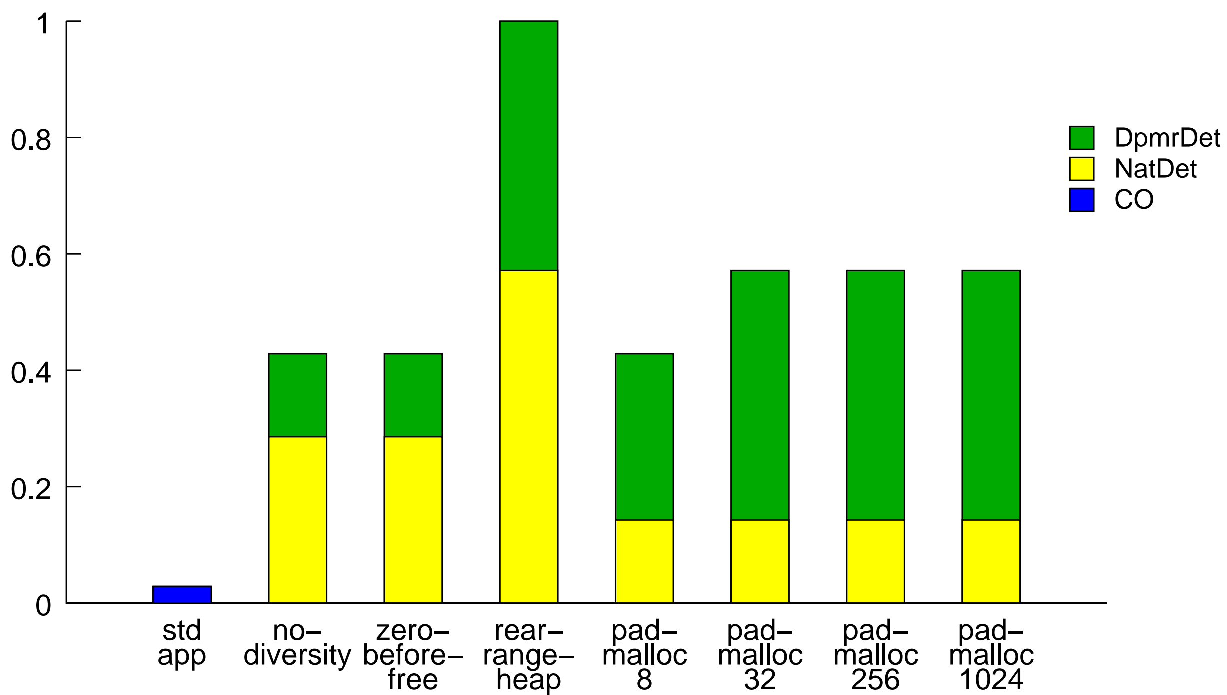

Figure 3.9: Mean immediate free conditional coverage of diversity transformations

sizes of memory that they allocate. Often requests will fall between two of those sizes, and the allocator rounds up. Therefore, even if a `malloc` request does not ask for enough memory for its needs, there is a chance that it will be granted enough memory for its needs. Overallocation primarily affects heap array resizes. That is one reason that correct output occurs more often under heap array resizes than under immediate frees.

The second observation that we make is that buffer overflows resulting from heap array resizes are entirely covered by implicit diversity (the no-diversity variant). That suggests that implicit diversity may be enough to detect buffer overflows. When two objects X and Y are allocated, implicit diversity leads to layouts such as $\{X, X_r, X_s, Y, Y_r, Y_s\}$. That decreases the likelihood that the objects following X and X_r are replicas of each other. Therefore, a buffer overflow out of X (and X_r) is unlikely to corrupt paired objects, and DPMR will be able to detect the corruption.

The third observation we make is that the rearrange-heap mechanism performs the best (in terms of coverage) and is the only diversity mechanism to cover 100% of the injected faults. Rearrange-heap performs well against heap array resizes for the same reasons that all of the diversity mechanisms perform well. It performs well against the dangling pointers created by immediate frees because its randomization decreases the likelihood that an application object-replica pair will occupy the same memory locations as previous application object-replica pairs. Thus, corruptions ensuing from a dangling pointer to one application object-replica pair are not likely to corrupt the memory occupied by another application object-replica pair in an identical way that is not detectable by DPMR.

Our final observation is that rearrange-heap produces less DPMR detection than the other DPMR diversity policies, even though it achieves the best coverage results. Though it has not yet been investigated, it seems reasonable that DPMR detection would tend to result in better fault diagnosis than natural detection would because it offers an opportunity to log process state automatically at the time of detection.

We now turn our attention to overhead. Overhead is displayed in Figure 3.10. We observe that all overheads are between 2x and 5x, with no-diversity and zero-before-free performing the best, and the larger pad-malloc policies performing the worst. We hypothesize that the rearrange-heap and pad-malloc mechanisms often perform worse than the no-diversity and

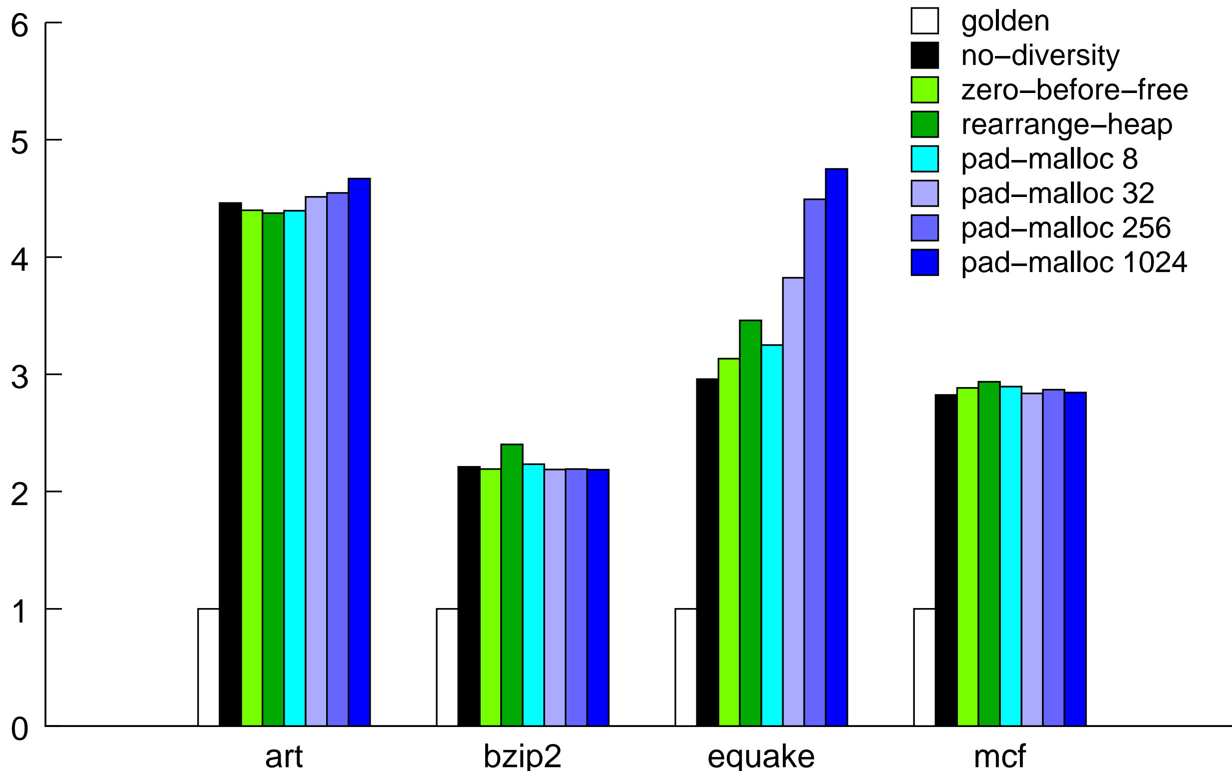

Figure 3.10: Overhead of diversity transformations

zero-before-free mechanisms because they may cause the heap allocator to cross cache page boundaries when allocating and deallocating buffers, with the larger pad-mallocs being the biggest offenders.

The last metric we examine is detection latency, shown in Table 3.3. Detection latency is important because the sooner an executed fault is detected, the sooner recovery can start, and the easier it will be to diagnose the fault. Rearrange-heap drastically outperforms the other policies when examined under art, and it tends to have similar detection latencies to the other policies for the other three applications.

3.8 State Comparison Policy Results

Here, we present an experimental evaluation of the comparison policies discussed in Section 2.7. As with diversity transformations, the experiments presented here involve eight application variants. The first variant is the standard application without DPMR. The

Table 3.3: Mean time to detection of diversity transformations

variant	msecs				
	art	**bzip2**	**equake**	**mcf**	
no-diversity	5773.13	268.30	224.38	6.58	heap array resizes
zero-before-free	5684.66	262.29	220.44	6.80	
rearrange-heap	54.10	128.63	285.07	9.00	
pad-malloc 8	5754.66	261.77	234.74	6.95	
pad-malloc 32	129.78	265.63	257.36	5.68	
pad-malloc 256	154.67	266.65	443.57	8.10	
pad-malloc 1024	195.55	264.34	1043.00	8.12	
no-diversity	8333.10	4354.93	183.76	4.64	immediate frees
zero-before-free	8353.59	4298.64	178.69	90.68	
rearrange-heap	43.31	4888.75	259.01	6.71	
pad-malloc 8	8349.75	4407.54	201.63	7.82	
pad-malloc 32	7475.46	4281.71	225.67	4.79	
pad-malloc 256	7525.73	4348.99	392.98	6.29	
pad-malloc 1024	7749.52	4385.43	902.89	6.09	

stdapp results in this section are identical to those in Section 3.7. They are repeated for convenient comparison with comparison policy results. The remaining seven variants are transformed with DPMR and the rearrange-heap diversity transformation. We use rearrange-heap because it was the best-performing diversity transformation. Each of the seven transformed variants uses one of the following state comparison policies: all loads, temporal load-checking 1/8, temporal load-checking 1/2, temporal load-checking 7/8, static load-checking 10%, static load-checking 50%, and static load-checking 90%.

As with diversity transformations, we start our evaluation of comparison policies by examining coverage. Coverage of heap array resizes is depicted in Figure 3.11, and coverage of immediate frees is shown in Figure 3.12. Coverage combined across all studied applications, conditioned on incorrect output and $StdNotAllDet$, is displayed in Figures 3.13 and 3.14. As discussed during the analysis of diversity transformations, coverage results are high, even for the standard application without DPMR. Two other major results can be gathered from the figures. First, coverage is robust in the face of reduced checking. What that means is that coverage remains high even when the load-checking frequency is reduced. Second, in our experiments, static load-checking is just as viable as temporal load-checking, suggesting that coverage is spatially robust. When we say that coverage is *spatially robust*, we mean

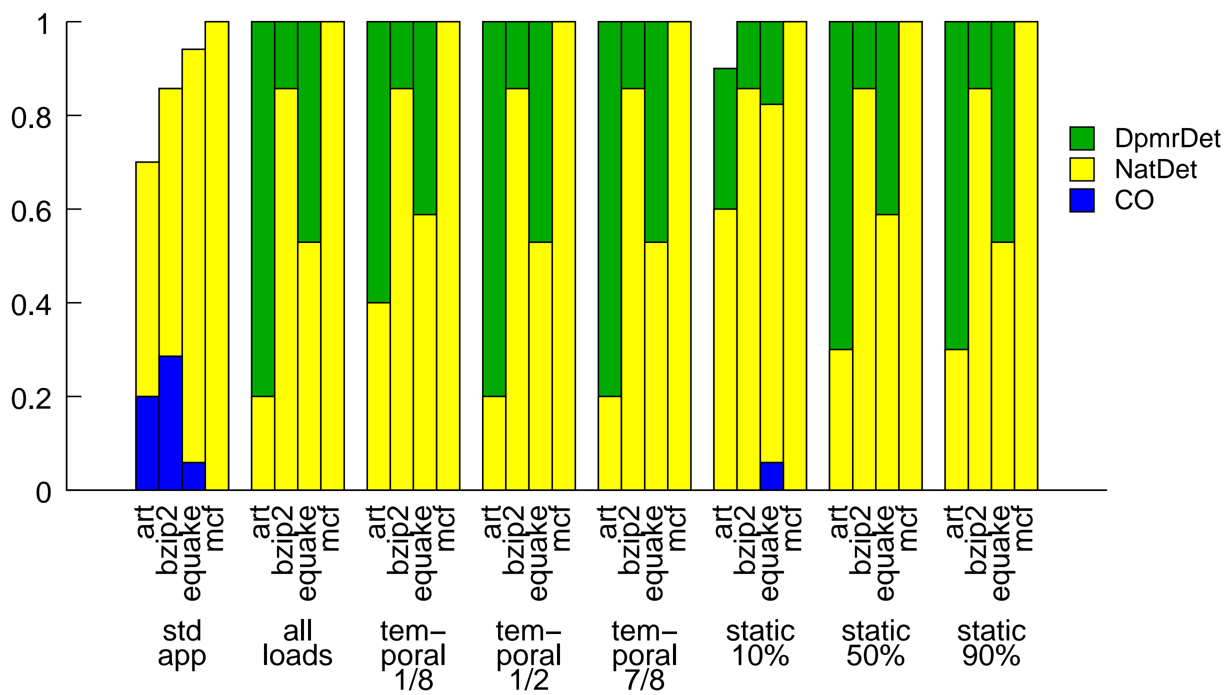

Figure 3.11: Mean heap array resize coverage of state comparison policies

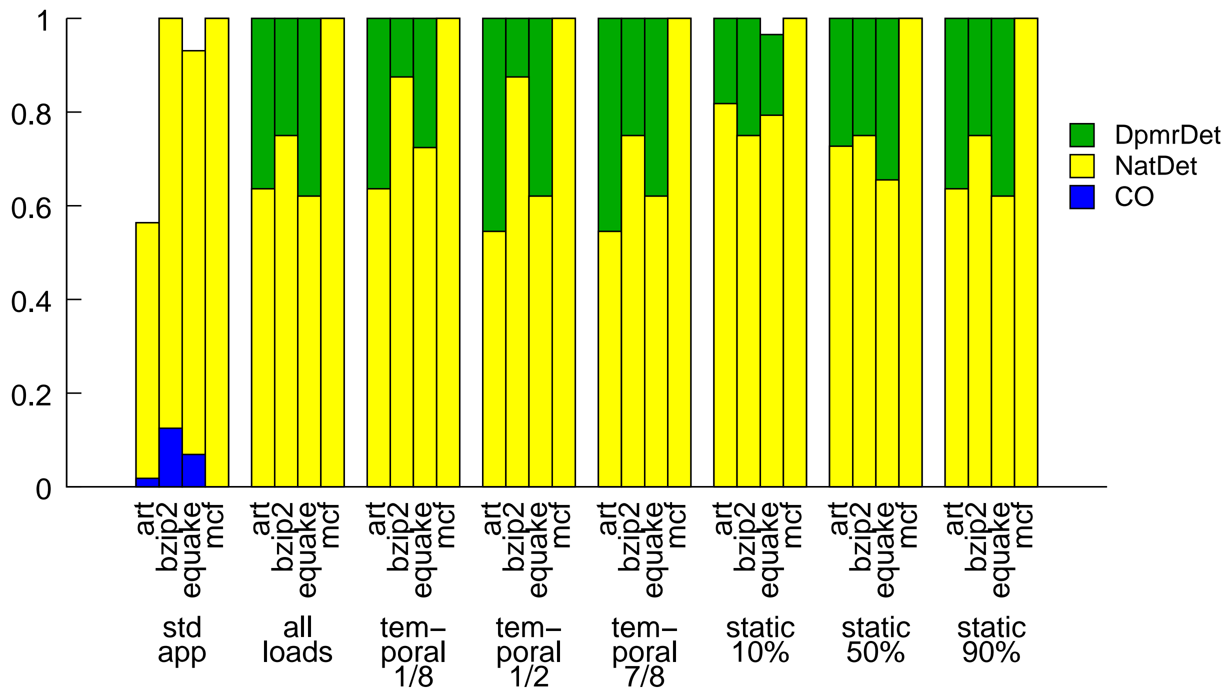

Figure 3.12: Mean immediate free coverage of state comparison policies

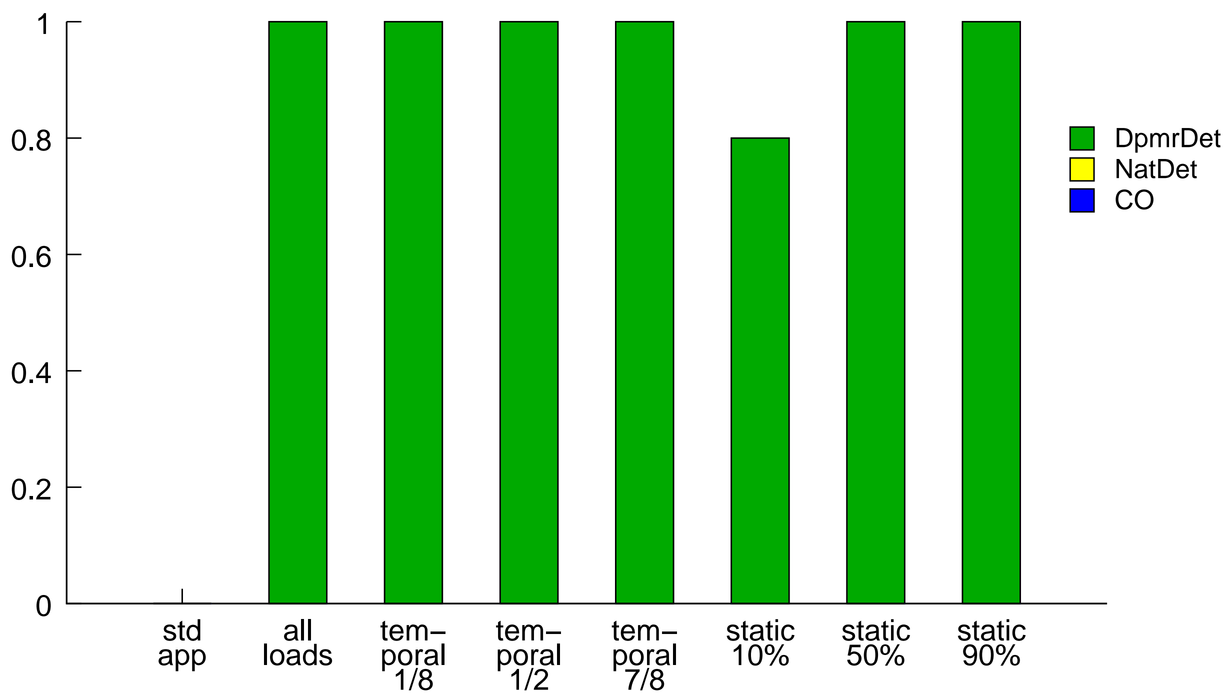

Figure 3.13: Mean heap array resize conditional coverage of state comparison policies

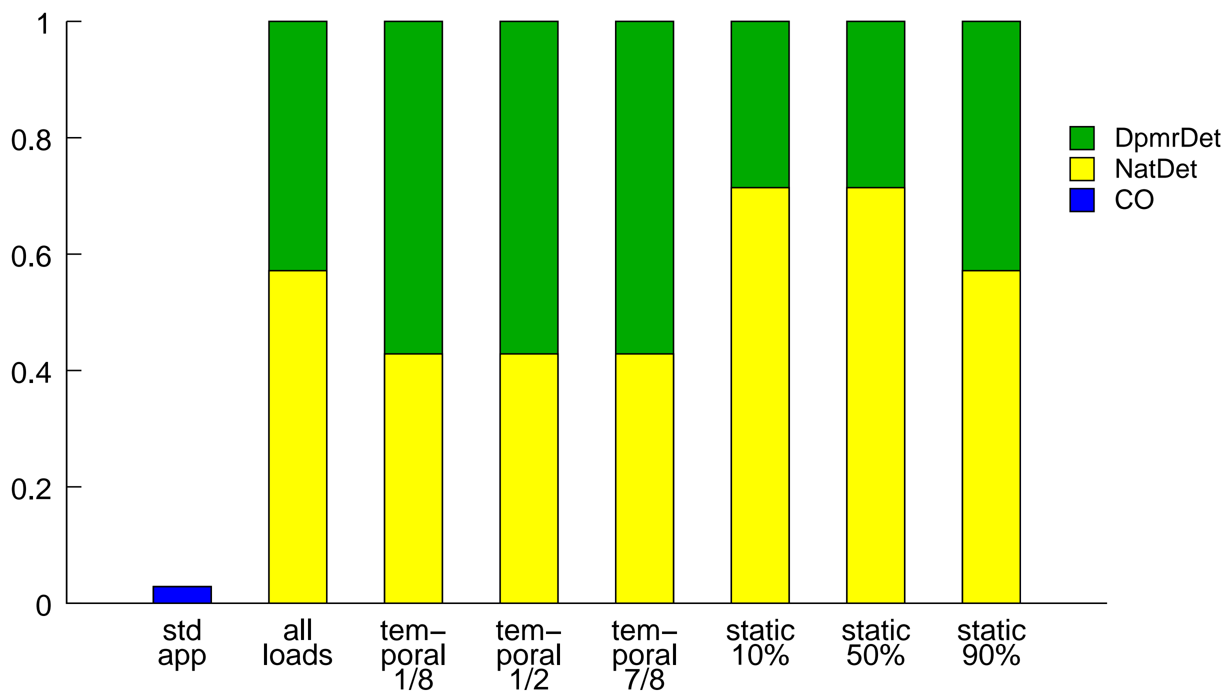

Figure 3.14: Mean immediate free conditional coverage of state comparison policies

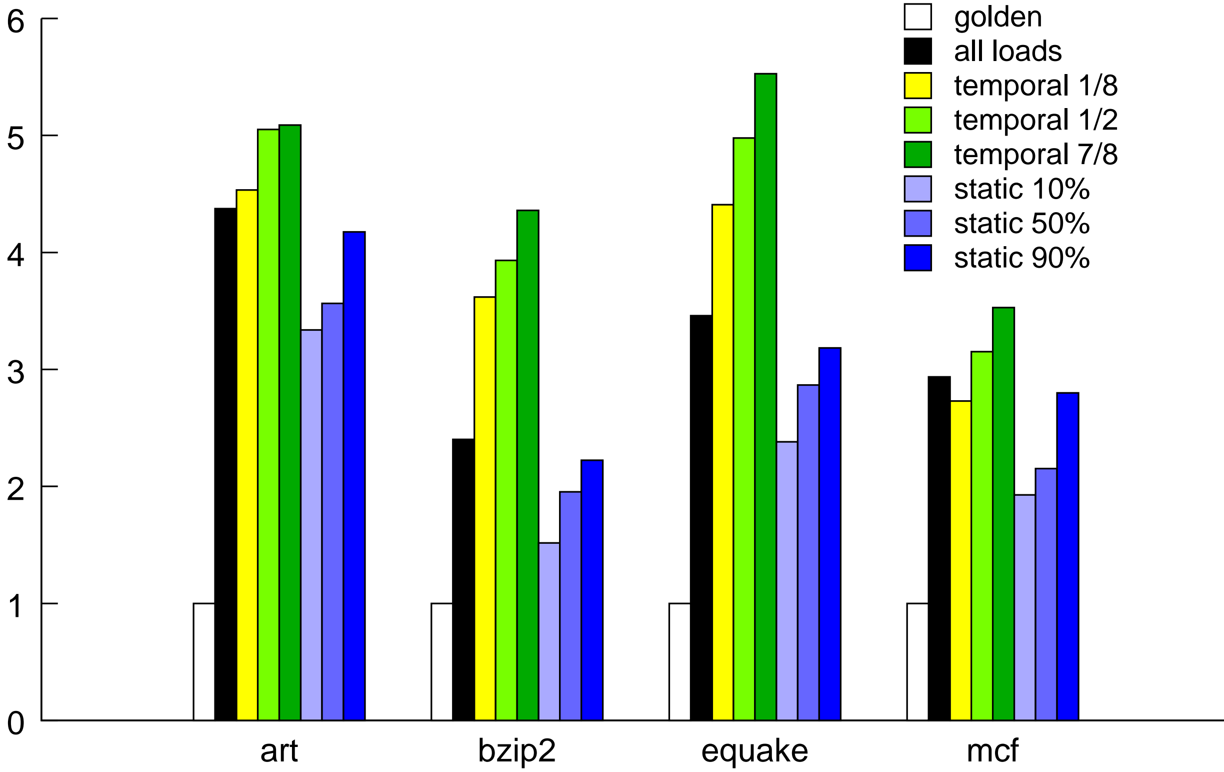

Figure 3.15: Overhead of state comparison policies

that coverage remains high despite a large reduction in the number of load locations that are checked. In our results, coverage was reduced only when we reduced checking to a static 10% of application loads.

We hypothesize that those results are due to two effects. First, the results may suggest that memory faults logically propagate throughout a program. Even if we do not check the loads that are directly affected by a fault, errors from that fault are likely to affect other loads. That makes sense, since it is likely that the data throughout an application are interdependent. Second, we believe that the results are affected by the periodic nature of software bugs. Even if we do not detect a fault the first time it executes, we can detect future executions, as particular instructions are often executed multiple times during a run of a program.

Now, we focus on overhead. Intuitively, the primary benefit of reduced-checking policies is a reduction in overhead. As seen in Figure 3.15, that was the case with static load-checking, culminating in the achievement of approximately a 1/3 speedup by the static load-checking

```
1 int32[]* a = ...;
2 int32[]* a_r = ...;
3 int32 sum = 0;
4 int32 i = 0;
5 for(i=0; i < 100; ++i){
6   int32 tmp1 = a[i];
7   int8 chkVal = *chkCounter;
8   if(chkVal == 0){
9     assert(tmp1 == a_r[i]);
10  }
11  *chkCounter = (chkVal+1) % 2;
12  sum += tmp1;
13 }
```

(a) Checking code using a counter

```
1 int32[]* a = ...;
2 int32[]* a_r = ...;
3 int32 sum = 0;
4 int32 i = 0;
5 for(i=0; i < 100; ++i){
6   int32 tmp1 = a[i];
7   assert(tmp1 == a_r[i]);
8   sum += tmp1;
9   ++i;
10  tmp1 = a[i];
11  sum += tmp1;
12 }
13
```

(b) Periodic checking code

Figure 3.16: Exploiting periodicity to improve temporal load-checking overhead

10% policy. However, it was not the case when load comparisons were temporally reduced.

To understand why temporal checking increased the overhead, we must understand that it requires an additional check at each load to determine whether the load should be replicated and compared. That leads to three additional sources of overhead. First, it inserts additional computation. Second, some of that additional computation comprises branches that can lead to expensive branch misses. Finally, we conjecture that additional branches may prevent standard compiler optimizations from occurring that would otherwise have taken place.

We do not feel that temporal load-checking is necessarily a lost cause, but it would require aggressive optimizations to make it efficient. In particular, there are some compiler optimizations that could increase its performance if the load-checking were performed periodically. Consider the checking code in Figure 3.16. The code on the left side performs temporal load-checking much as the current implementation does. `chkCounter` is a global variable that determines when a load should be checked. In the figure, every other load is replicated and checked. The code on the right side of the figure exploits the periodic nature of the checking and eliminates the branch decision that determines whether the loaded value `tmp1` should be checked. In addition, the code on the right side eliminates the need to load and store the `chkCounter` global variable.

Finally, we examine detection latency, shown in Table 3.4. The major finding is that static

Table 3.4: Mean time to detection of state comparison policies

variant	msecs				
	art	**bzip2**	**equake**	**mcf**	
all loads	54.10	128.63	285.07	9.00	heap array resizes
temporal 1/8	160.94	266.38	303.00	5.09	
temporal 1/2	178.31	270.12	305.68	5.20	
temporal 7/8	57.63	290.69	311.53	5.09	
static 10%	142.43	115.50	246.63	9.86	
static 50%	160.77	110.87	304.85	5.00	
static 90%	50.04	124.22	285.52	8.61	
all loads	43.31	4888.75	259.01	6.71	immediate frees
temporal 1/8	5190.58	7296.49	267.73	2.91	
temporal 1/2	5711.38	7843.44	275.22	2.91	
temporal 7/8	55.75	8883.58	280.99	4.05	
static 10%	57.08	3155.66	241.88	9.53	
static 50%	68.00	4297.55	253.03	5.43	
static 90%	42.77	4682.83	259.49	6.65	

load-checking detection latency tends to be approximately equal to and in some cases lower than the all loads policy, while it tends to be higher for temporal load-checking than for the all loads policy. The increased detection latency for temporal load-checking makes sense, since a reduction in checking frequency should increase detection latency. On the other hand, the static load-checking result seems anomalous. However, there are two factors that impact detection latency: checking frequency and overhead. Although the reduced checking of static load-checking may increase detection latency, its decreased overhead decreases detection latency. When static load-checking has lower detection latency than all loads, that suggests that the detection latency is dominated more by the reduced overhead than by reduced checking.

Chapter 4

Mirrored Data Structures

In this chapter, we explore an alternative memory layout for DPMR and compare it to the SDS-based approach. Section 4.1 describes the new approach; Section 4.2 examines code transformations under the new approach; Section 4.3 describes changes in the requirements that DPMR imposes on external code support libraries; and Section 4.4 discusses the limitations of the new approach and how they differ from the limitations of SDS. Finally, Section 4.5 looks at the results and places them in the context of SDS results.

4.1 The Mirrored Data Structures Approach

In Section 2.2 we discussed two memory layouts for DPMR and chose the layout based on SDS. Here, we examine the alternative, called *Mirrored Data Structures (MDS)*. It places replica pointers in replica memory, as illustrated in Figure 2.2; replica memory mirrors the layout of application memory. Since MDS does not use shadow memory, the definition of augmented types must change. The new definition for augmented types can be found in Table 4.1. As with SDS, augmented types affect only function types. In particular, augmented types, under MDS, add an ROP after original pointer arguments; and if the return value of a function is a pointer, a pointer argument is added to which the callee can store an ROP corresponding to the pointer return value. An example of an MDS augmented type is displayed in Table 4.2. Because the return type of the function is a pointer, the `rvRopPtr` argument has been added so that an ROP can be returned to calling functions. In addition, ROP arguments are added that correspond to the pointer arguments `s1` and `s2`.

The downside of MDS is that pointers stored in application and replica memory are not comparable, which has the potential to reduce coverage. Indeed, if every value stored to

Table 4.1: MDS augmented type definition

Description	Original Type t	Replica Parameter Type $\mathbf{rpt}(t)$
pointer	$\tau*$	$\mathbf{at}(\tau)*$
other	τ	$\varnothing$

Description	Original Type t	Augmented Type $\mathbf{at}(t)$
array	$\tau[]$	$\mathbf{at}(\tau)[]$
struct	$\text{struct}\{\tau_0; \ldots; \tau_n;\}$	$\text{struct}\{\mathbf{at}(\tau_0); \ldots; \mathbf{at}(\tau_n);\}$
union	$\text{union}\{\tau_0; \ldots; \tau_n;\}$	$\text{union}\{\mathbf{at}(\tau_0); \ldots; \mathbf{at}(\tau_n);\}$
pointer	$\tau*$	$\mathbf{at}(\tau)*$
function types	$r(\tau_0, \ldots, \tau_n)$	$\mathbf{at}(r)(\mathbf{rpt}(r)*, \mathbf{at}(\tau_0), \mathbf{rpt}(\tau_0), \ldots, \mathbf{at}(\tau_n), \mathbf{rpt}(\tau_n))$
primitive and void types	τ	τ

Table 4.2: Example of an MDS augmented type

Original Type	Augmented Type
`int8[]* (int8[]* s1, int8[]* s2);`	`int8[]* (int8[]** rvRopPtr, int8[]* s1, int8[]* s1Rop, int8[]* s2, int8[]* s2Rop);`

memory were a pointer, then no loads would be comparable. However, there are three potential benefits of MDS over SDS. First, the memory footprint for a program under MDS is less than or equal to that of the same program under SDS. They would be equal only if nothing stored to memory is a pointer. In general, the memory overhead for SDS is 2x to 4x, while the overhead for MDS is 2x. Those calculations do not account for the fact that DPMR increases register pressure, which may force more registers to memory. However, the register pressure for SDS is greater than or equal to that of MDS. The second major benefit of MDS over SDS is that MDS imposes fewer restrictions on program input than SDS does, as discussed in Section 4.4. The third advantage of MDS is that it has the potential to induce less overhead than SDS does because of its reduced memory footprint and reduction in the number of additional loads and stores that are required under its transformations, as seen in the following section.

Table 4.3: MDS transformation – added behavior

Description	Original Behavior		Added Behavior
heap deallocation	`free(`p`)`		`free(`p_r`)`
store	`*`$p \leftarrow x$	$\mathbf{type}(x) \neq \tau*$	`*`$p_r \leftarrow x$
		$\mathbf{type}(x) = \tau*$	`*`$p_r \leftarrow x_r$
load	$x \leftarrow$ `*`p	$\mathbf{type}(x) \neq \tau*$	`assert(`x `==` `*`p_r`)`
		$\mathbf{type}(x) = \tau*$	$x_r \leftarrow$ `*`p_r
address of a struct field	$x \leftarrow$ `&(`p`->`f_i`)`		$x_r \leftarrow$ `&(`p_r`->`f_i`)`
address of an array element	$x \leftarrow$ `&`p`[`i`]`		$x_r \leftarrow$ `&`p_r`[`i`]`
address of a function	$p \leftarrow$ `&`fun		$p_r \leftarrow$ `&`fun

4.2 Code Transformations for Mirrored Data Structures

As with SDS, the code transformations for MDS make use of the $\gamma()$ function for transforming function declaration arguments and function parameters. The transformations also make use of the $\pi()$ function for translating a function's return type into an argument set. However, the definitions for $\gamma()$ and $\pi()$ have been slightly modified. The new definitions are found in Equations 4.1 and 4.2. $\gamma()$ no longer refers to a register holding an NSOP, while *rvSop* has been renamed to *rvRopPtr*, in $\pi()$. The change to $\pi()$ is made to reflect the fact that additional augmented function arguments corresponding to pointer return values are pointers to ROPs rather than pointers to shadow objects.

$$\gamma(r) \equiv \begin{cases} \{r, r_r\} & \text{if } \mathbf{type}(r) = \tau* \\ \{r\} & \text{else} \end{cases} \tag{4.1}$$

$$\pi(t) \equiv \begin{cases} \{rvRopPtr\} & \text{if } t = \tau* \\ \varnothing & \text{else} \end{cases} \tag{4.2}$$

The code transformations for MDS are specified in Tables 4.3 and 4.4. The transformations in the first table add to the original program behavior, while the transformations in the second table replace the original program behavior. Explanations of the transformations are

Table 4.4: MDS transformation – transformed behavior

Description	Original Behavior		Transformed Behavior
allocation	$p \leftarrow allocate(\tau) : allocate = \{$ `malloc`|`alloca`|`global`$\}$		$p \leftarrow allocate(\mathbf{at}(\tau))$ $p_r \leftarrow allocate(\mathbf{at}(\tau))$
pointer-to-pointer cast	$q \leftarrow (\tau*)p$		$q \leftarrow (\mathbf{at}(\tau)*)p$ $q_r \leftarrow (\mathbf{at}(\tau)*)p_r$
function declaration	$fun(a_0, \ldots, a_n)$: $\mathbf{type}(fun) = r(\tau_0, \ldots, \tau_n)$		$fun(\pi(r) \cup (\bigcup_{i=0}^{n} \gamma(a_i)))$: $\mathbf{type}(fun) = \mathbf{at}(r(\tau_0, \ldots, \tau_n))$
function call	$x \leftarrow fun(p_0, \ldots, p_n)$: $\mathbf{type}(fun) = r(\ldots)$	$r = \tau*$	$rvRopPtr \leftarrow$ `alloca`$(\mathbf{at}(r))$ $x \leftarrow fun(\{rvRopPtr\} \cup (\bigcup_{i=0}^{n} \gamma(p_i)))$ $x_r \leftarrow *rvRopPtr$
		$r \neq \tau*$	$x \leftarrow fun(\bigcup_{i=0}^{n} \gamma(p_i))$
function return	`return` x	$\mathbf{type}(x) = \tau*$	$*rvRopPtr \leftarrow x_r$ `return` x
		$\mathbf{type}(x) \neq \tau*$	`return` x

as follows:

- *allocation*: When an allocation instruction occurs in the program under transformation, it is transformed to allocate an augmented type. In addition, a replica buffer of memory is allocated.

- *heap deallocation*: When heap memory is deallocated, MDS inserts an instruction to deallocate the corresponding replica memory.

- *store*: If a non-pointer is stored to memory by the original application, then the same non-pointer value is stored to replica memory. On the other hand, if a pointer is stored to application memory, then the corresponding ROP is stored to replica memory. That is a key difference between SDS and MDS.

- *load*: If a non-pointer value is loaded by the original application, then, depending on the state comparison policy, the corresponding value can be loaded from replica memory and compared with the original loaded value. If a pointer is loaded from memory, then the corresponding ROP is loaded from replica memory. In the case of a loaded pointer, a load comparison never occurs, because the pointers are different by definition.

- *address of a structure field*: When a program computes the address of a field in a structure, MDS applies identical addressing arithmetic to the corresponding ROP.

- *address of an array element*: Likewise, if a program addresses an array element, then identical addressing arithmetic is applied to the corresponding ROP.

- *pointer-to-pointer cast*: If a cast instruction from one pointer type to another pointer type occurs, it is transformed into a cast to an augmented pointer type. A similar cast to an augmented pointer type is applied to the corresponding ROP.

- *address of a function*: When a pointer is assigned the address of a function, an ROP is assigned the same address. Since functions are never explicitly dereferenced, i.e., they are only dereferenced via function calls, there are no corresponding replica functions

from which to compute addresses. Thus, we simply use the same address as the application pointer.

- *function declaration*: Function declarations are transformed to use augmented function types, which potentially adds new arguments.
- *function call*: Function calls are transformed so that ROPs are passed to the function for every original pointer parameter. In addition, if the called function returns a pointer, then memory must be allocated to hold an ROP corresponding to the return value. A pointer to that memory is passed along to the called function. When the called function returns, the ROP corresponding to the return value is loaded from memory.
- *function return*: If a function returns a pointer, then a store instruction must be inserted before the function's return instructions. The store writes an ROP to the memory pointed to by the *rvRopPtr* argument.
- *global variable initialization*: As discussed in Section 2.4, global variable initialization can be thought of as a series of non-pointer operations, pointer arithmetic, and store operations that can be executed at compile time. Those operations can be transformed using the rules discussed above.

We now revisit the two code transformation examples presented in Section 2.4, to see how they look under MDS. The first example is shown in Figure 4.1. Line 5 in the transformed code allocates replica memory on the heap that corresponds to the memory allocated by the original program. Line 9 is inserted into the transformed code to store to the replica memory the same non-pointer value that is stored to application memory. In contrast to the store of a non-pointer, line 19 demonstrates that when the original code stores a pointer to memory, MDS does not store the same pointer to replica memory. Instead, it stores an ROP. Finally, since `createNode()` returns the pointer `n`, the corresponding pointer `n_r` is stored to the memory pointed to by `rvRopPtr`.

Figure 4.2 illustrates another code transformation. The transformation of `getSum()` is presented to show load transformations. In particular, line 7 demonstrates that when the

```
1 typedef struct LinkedList LL;
2 LL* createNode(int32 data, LL* last){
3
4   LL* n = malloc(LL);
5
6   int32* dataPtr = &(n->data);
7
8   *dataPtr = data;
9
10  LL* nxtPtr = &(n->nxt);
11
12  *nxtPtr = NULL;
13
14  if(last){
15    LL* lastNxtPtr = &(last->nxt);
16
17
18    *lastNxtPtr = n;
19
20  }
21  return n;
22
23 }
```

(a) Original code

```
1 typedef struct LinkedList LL;
2 LL* createNode(LL** rvRopPtr,
3 int32 data, LL* last, LL* last_r){
4   LL* n = malloc(LL);
5   LL* n_r = malloc(LL);
6   int32* dataPtr = &(n->data);
7   int32* dataPtr_r = &(n_r->data);
8   *dataPtr = data;
9   *dataPtr_r = data;
10  LL* nxtPtr = &(n->nxt);
11  LL* nxtPtr_r = &(n_r->nxt);
12  *nxtPtr = NULL;
13  *nxtPtr_r = NULL;
14  if(last){
15    LL* lastNxtPtr = &(last->nxt);
16    LL* lastNxtPtr_r =
17      &(last_r->nxt);
18    *lastNxtPtr = n;
19    *lastNxtPtr_r = n_r;
20  }
21  *rvRopPtr = n_r;
22  return n;
23 }
```

(b) Transformed code

Figure 4.1: MDS transformation of `createNode()`

```
1 int32 getSum(LL* n){
2   int32 sum = 0;
3   while(n){
4     int32* dataPtr = &(n->data);
5
6     int32 v = *dataPtr;
7
8     sum += v;
9     LL* nxtPtr = &(n->nxt);
10
11    n = *nxtPtr;
12
13  }
14  return sum;
15 }
```

(a) Original code

```
1 int32 getSum(LL* n, LL* n_r){
2   int32 sum = 0;
3   while(n){
4     int32* dataPtr = &(n->data);
5     int32* dataPtr_r = &(n_r->data);
6     int32 v = *dataPtr;
7     assert(v == *dataPtr_r);
8     sum += v;
9     LL* nxtPtr = &(n->nxt);
10    LL* nxtPtr_r = &(n_r->nxt);
11    n = *nxtPtr;
12    n_r = *nxtPtr_r;
13  }
14  return sum;
15 }
```

(b) Transformed code

Figure 4.2: MDS transformation of `getSum()`

original application loads a non-pointer value, DPMR can insert a load check that loads replica memory and compares it to the value loaded from application memory. On the other hand, if a pointer is loaded from memory, no load check is inserted, and replica memory must be loaded, as shown in line 10. The value loaded from replica memory is the ROP for the pointer loaded from application memory.

4.3 External Code under Mirrored Data Structures

Under MDS, we still use external function wrappers and generate and initialize replica memory corresponding to external global variables. However, the responsibilities of external function wrappers are slightly different from their responsibilities under SDS. The modified responsibilities are described below.

1. External function wrappers must perform the functionalities that their corresponding external functions would perform. They typically do so by calling the external functions.

2. External function wrappers are required to allocate replica memory corresponding to application-visible heap memory allocated by the external function.

3. Any non-pointer value stored by the external function to application-visible memory should also be stored to corresponding replica memory. If an external function stores a pointer to application-visible memory, the corresponding ROP should be stored to replica memory.

4. If the external function returns a pointer, a corresponding ROP should be stored to the memory pointed to by the wrapper's *rvRopPtr* argument.

5. For any application-visible heap memory that is deallocated by the external function, corresponding replica heap memory should be deallocated.

6. If application-visible non-pointer memory is loaded by the external function, then the wrapper should load and compare corresponding replica memory.

Not having to deal with shadow memory in external function wrappers helps to simplify the implementations of several wrappers. That is particularly true for the `libc` functions `qsort()`, `memcpy()`, and `memmove()`. In Section 3.1.5 we discussed how those functions require an extra parameter specifying information about the shadow types of the memory they manipulate. The extra parameter was necessary because those functions dealt with memory using generic types. Under MDS, that is no longer a problem. Replica memory has the same layout as application memory within an allocated buffer. Therefore, generic type operations can be applied in the same way to both. However, if one wishes to perform load comparisons in the wrappers for `memcpy()` and `memmove()`, an extra parameter would still be required to indicate what is a pointer and what is not. That need not be the case with `qsort()`, because `qsort()` requires a comparison function that compares the memory to be sorted, and the load comparisons can be left to that comparison function.

4.4 Limitations of Mirrored Data Structures

Below, we consider the restrictions that MDS imposes on an input program.

- *store*: Like SDS, MDS requires that pointers be typed as pointers when they are stored to memory. It does not matter if the pointers are typed accurately or precisely.
- *load*: As with stores, loads of pointers must be typed as pointers. SDS has the same requirement. Again, it does not matter if the pointers are typed accurately or precisely.
- *int-to-pointer casts*: Int-to-pointer casts are not allowed because DPMR would have no way to set corresponding ROPs. SDS also forbids int-to-pointer casts.
- *external code*: External code support libraries must be implemented for all external code with which an input program interacts. The same restriction applies to SDS.

The above list is much shorter than the list of limitations for SDS because MDS eliminates the need for several of the restrictions that SDS required. The following is a discussion of the limitations that were eliminated.

- *memory allocation*: Unlike SDS, MDS does not impose any restrictions on memory allocations. SDS had restrictions on the compile time allocation type so that enough shadow memory would be allocated, which is no longer an issue.

- *store*: It is no longer necessary to type non-pointers as non-pointers at store sites.

- *load*: It is no longer necessary to type non-pointers as non-pointers at load sites.

- *structure and array pointer arithmetic*: There are no type restrictions for pointer arithmetic. That is one of the biggest advantages of MDS over SDS. Restrictions on pointer arithmetic were required under SDS so that we would be able to address shadow memory appropriately. Without shadow memory, restrictions on pointer arithmetic are no longer required, because any type-generic pointer arithmetic that is applied to application memory can be applied to replica memory. That is the case because the layouts of data in application and replica objects are structurally identical.

- *pointer-to-pointer casts*: There are no restrictions on pointer-to-pointer casts. SDS had restrictions on pointer-to-pointer casts so that we could handle NSOPs correctly.

It is also worth noting that the limitations of MDS can be reasoned about using a pointer analysis, while it is necessary to have type information as well in order to reason about the limitations of SDS.

4.5 Mirrored Data Structures Results

We turn our attention to experimental results. The MDS experiments evaluated the same diversity transformations and state comparison polices that we previously evaluated under SDS. When examining state comparison policies, the following must be kept in mind. Consider static load-checking. Let P be the number of loads of pointers in a program, and let N be the number of non-pointer loads. If we only include β of the load checks, then under SDS we will have removed $(1-\beta)(P+N)$ load checks. However, under MDS we will have removed only $(1-\beta)N$ load checks. In terms of reduction of overhead, we therefore expect

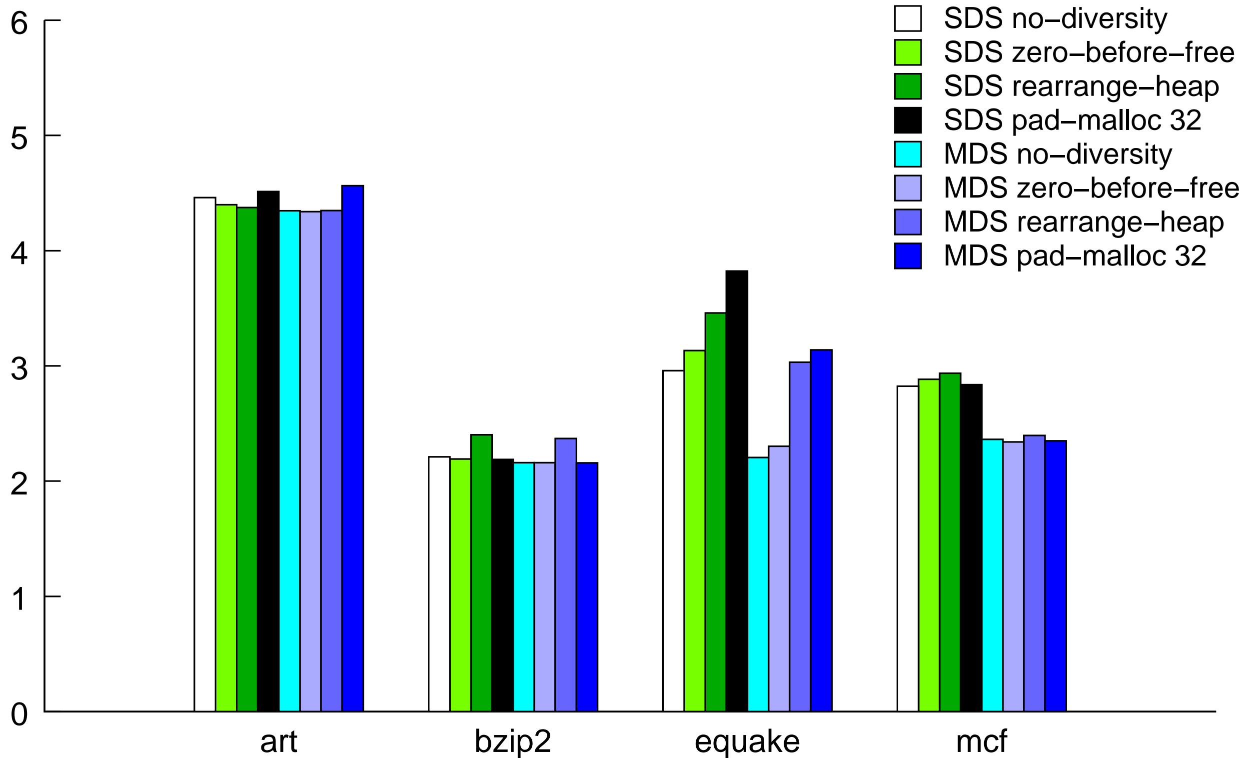

Figure 4.3: Side-by-side diversity transformation overheads of SDS and MDS

that a switch from the SDS all loads policy to the SDS static load-checking β comparison policy will result in greater improvement than could be obtained by switching from the MDS all loads policy to the MDS static load-checking β policy. A similar argument holds for temporal load-checking.

Since reduced overhead is one of the potential benefits of MDS, we examine it next. Figure 4.3 shows a side-by-side comparison of diversity transformations under SDS and MDS. As in other evaluations of diversity transformations, results are evaluated with the all loads state comparison policy. In all cases except one (art pad-malloc 32), MDS outperforms its SDS counterpart, albeit marginally so for art and bzip2. Examination of the code for the four benchmarks indicates that the fractions of the allocations that are for memory to hold pointers are larger for equake and mcf than for art and bzip2. That observation fits the overhead results. If very few pointers are stored to memory, then the executions of SDS and MDS are very similar. Without pointers in memory, SDS does not need to allocate shadow memory, and everything in memory will be comparable for MDS. Therefore, the memory

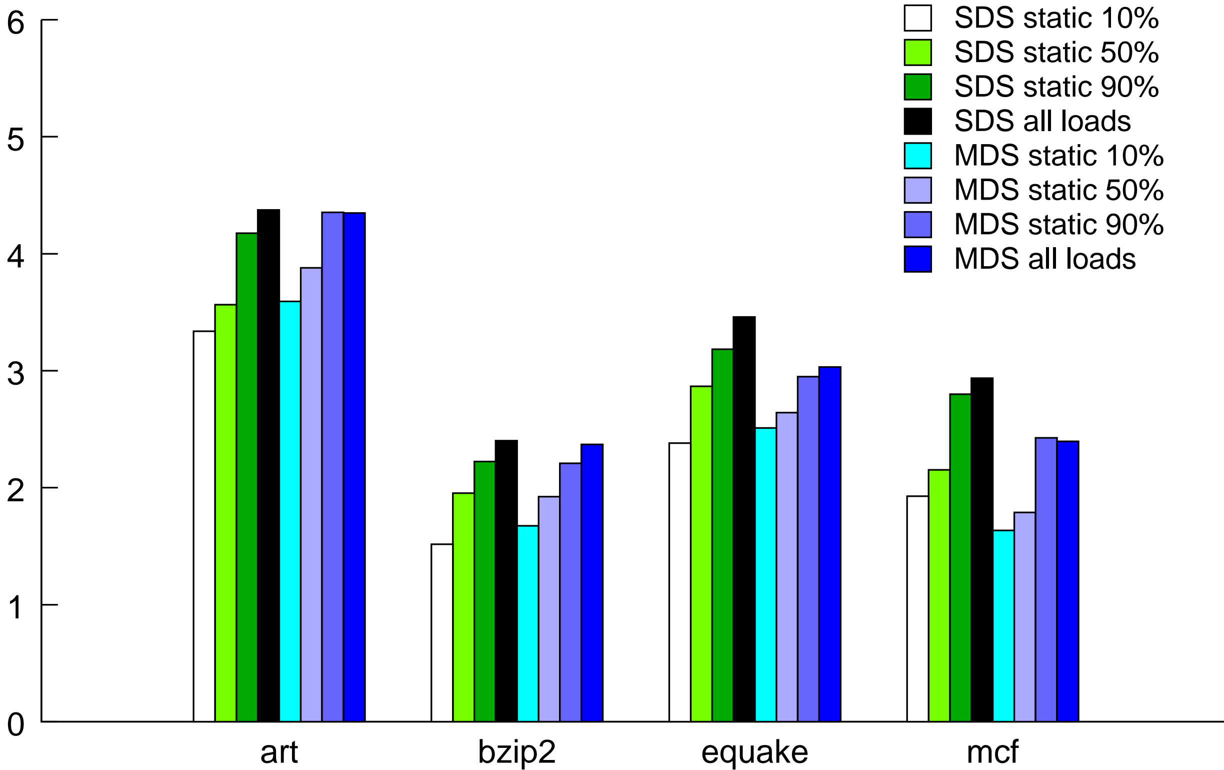

Figure 4.4: Side-by-side comparison policy overheads of SDS and MDS

overhead of SDS and MDS will be the same, and they both will perform comparisons at every load. For the more pointer-heavy benchmarks, equake and mcf, MDS is able to achieve stronger overhead gains. The MDS diversity transformation overheads for equake are 74% to 89% of their SDS counterparts, and the MDS mcf overheads are 81% to 84% of the SDS overheads.

We now compare overhead results for state comparison policies, using the rearrange-heap diversity transformation. Those results can be seen in Figure 4.4. In the side-by-side comparison, we do not include temporal load-checking results, since they are generally worse than the all loads results. Full overhead results for MDS can be seen in Figures 4.5 and 4.6. In Figure 4.4, the pointer-heavy benchmarks, equake and mcf, tend to see performance gains from MDS. For bzip2, the overheads under MDS tend to be about the same as those under SDS. For art, SDS tends to outperform MDS; however, we attribute that to caching effects and the random nature of the load-checking selection process under static load-checking.

In order not to marginalize its benefits, MDS must be capable of achieving reasonable

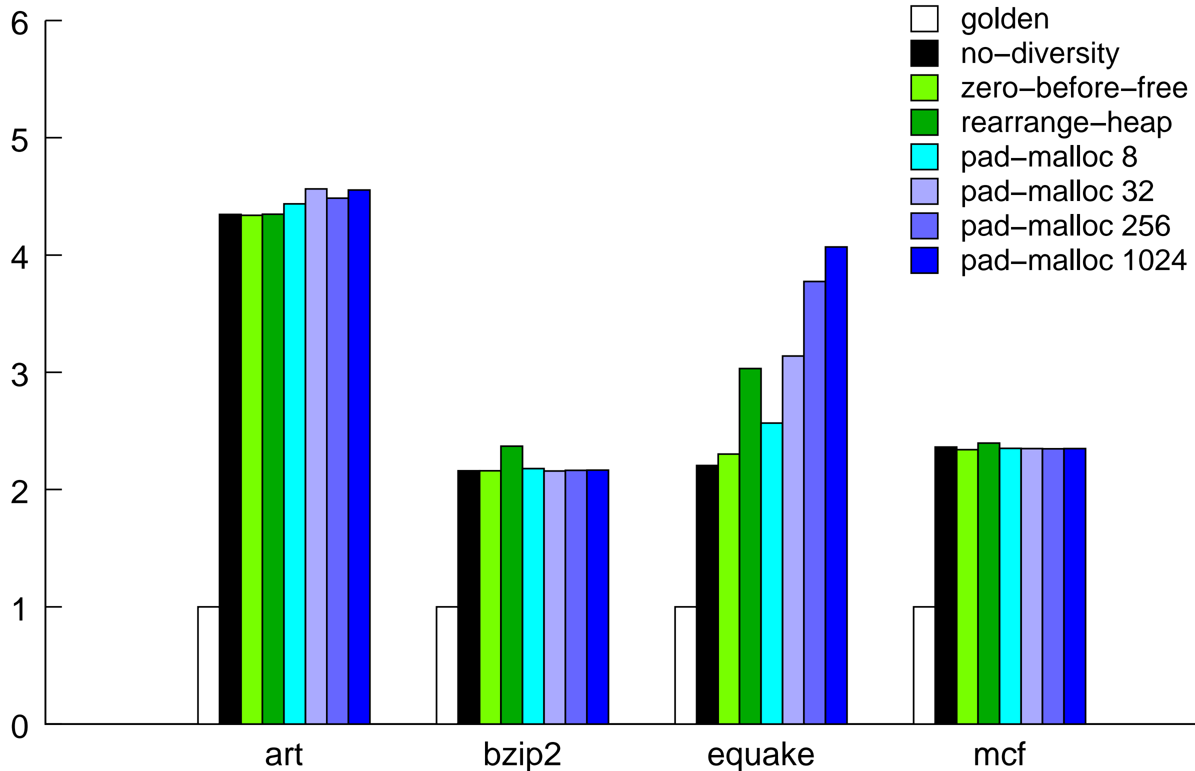

Figure 4.5: MDS overhead of diversity transformations

dependability. Figures 4.7 and 4.8 show mean coverage results of diversity transformations for heap array resizes and immediate frees, respectively, while Figures 4.9 and 4.10 show mean conditional coverage. The coverage results are shown on a per-application basis, and the conditional coverage results are combined across all applications. As with SDS, all heap array resizes are covered with implicit diversity, i.e., the no-diversity policy. On the other hand, the rearrange-heap policy is the only policy to detect all immediate frees. We conjectured in Section 3.7 that the randomness of the rearrange-heap transformation improves dangling pointer detection by decreasing the likelihood that an application object-replica pair occupies the same memory as a previous application object-replica pair.

The last metric we examine is detection latency. Results for diversity transformations are displayed in Table 4.5. When compared to the SDS detection latencies from Table 3.3, the results are very similar. Rearrange-heap tends to have similar detection latencies to other policies for bzip2, equake, and mcf; however, rearrange-heap has much lower detection latencies for art.

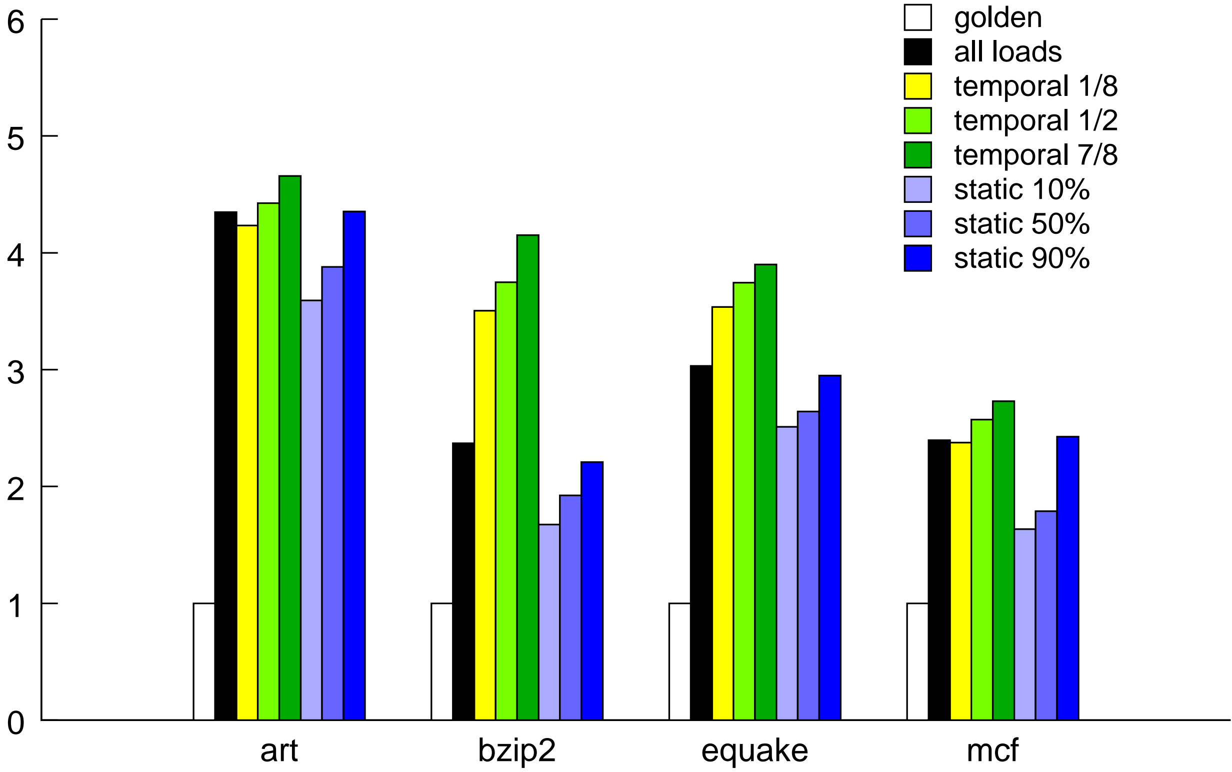

Figure 4.6: MDS overhead of state comparison policies

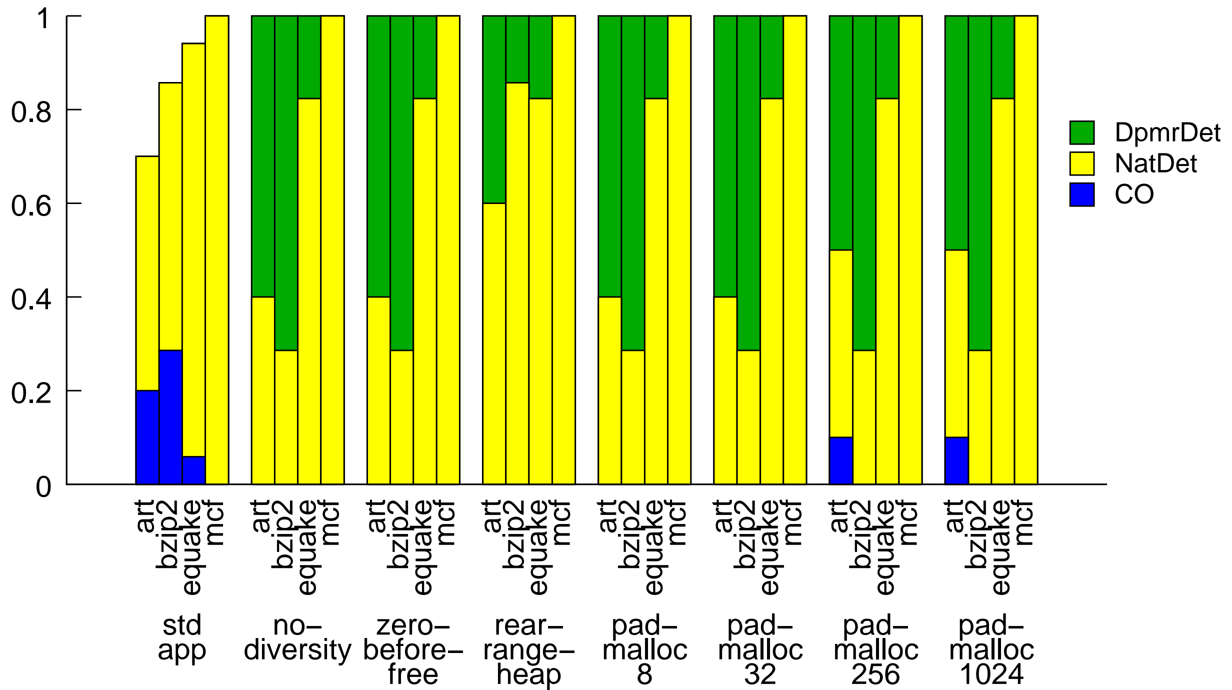

Figure 4.7: Mean MDS heap array resize coverage of diversity transformations

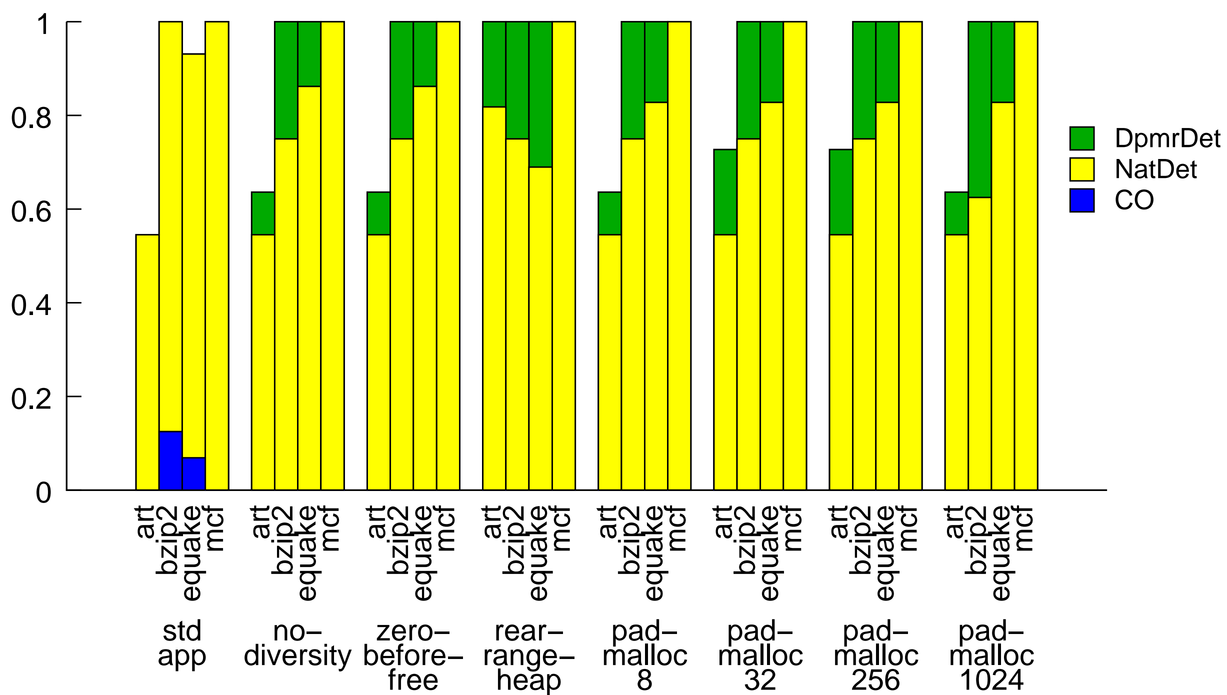

Figure 4.8: Mean MDS immediate free coverage of diversity transformations

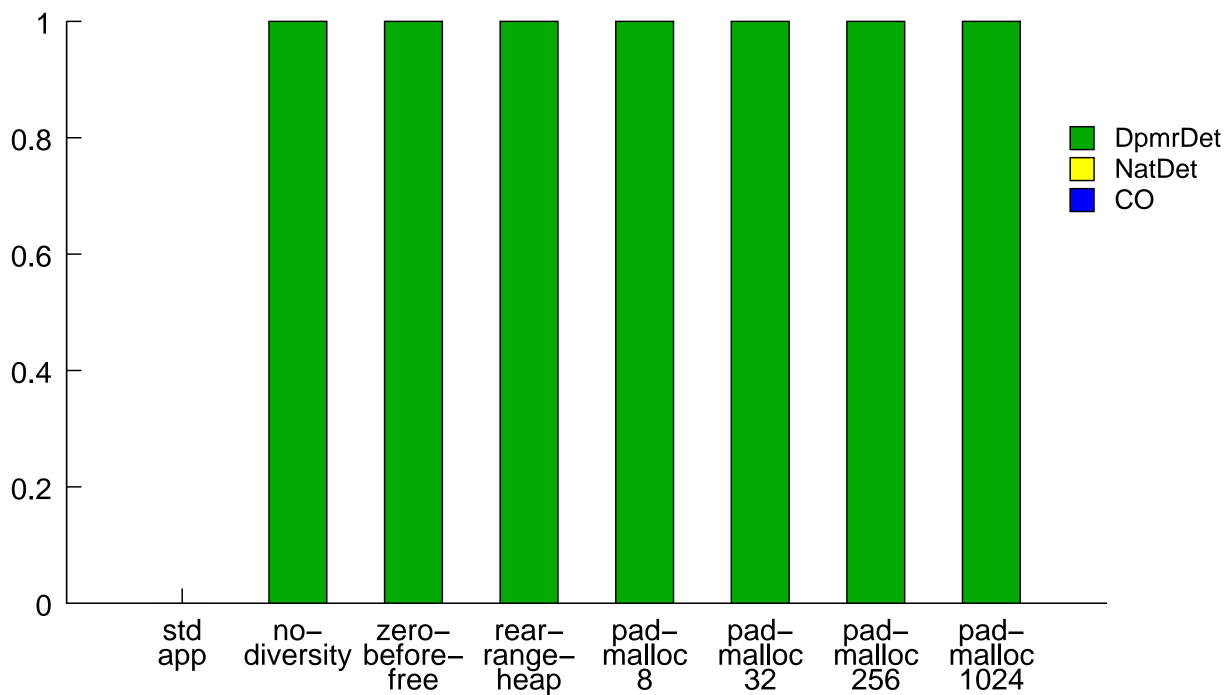

Figure 4.9: Mean MDS heap array resize conditional coverage of diversity transformations

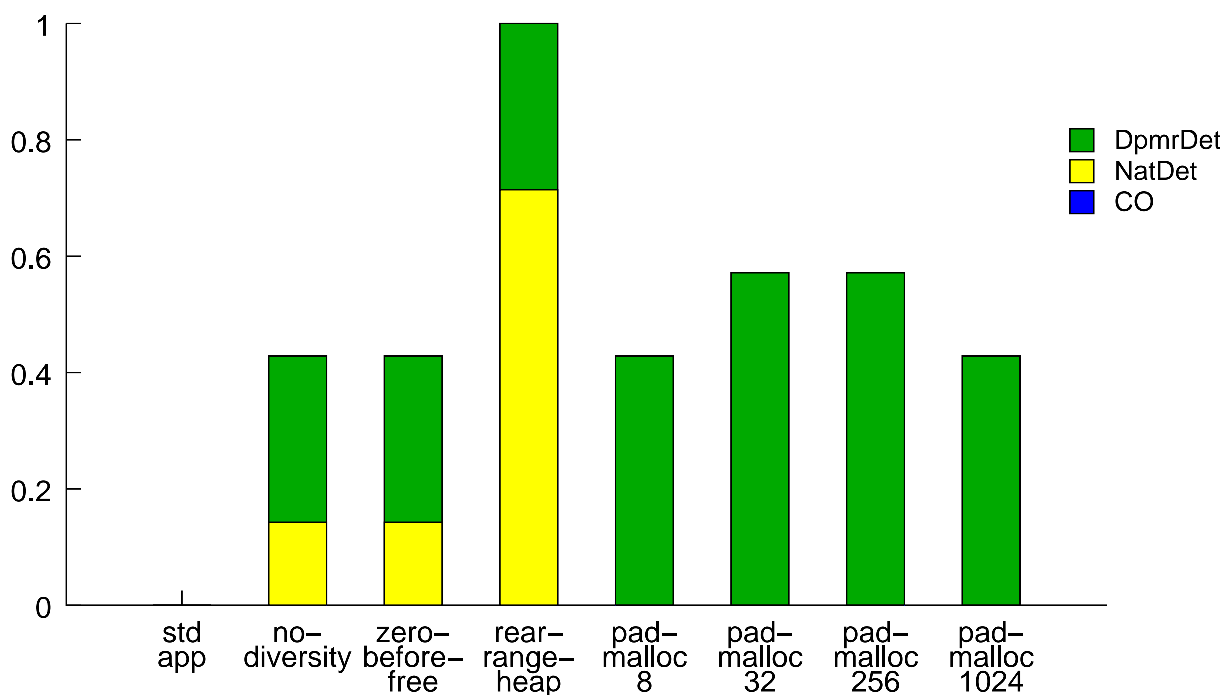

Figure 4.10: Mean MDS immediate free conditional coverage of diversity transformations

Table 4.5: Mean time to detection of diversity transformations under MDS

variant	msecs				
	art	**bzip2**	**equake**	**mcf**	
no-diversity	5380.62	262.42	175.27	4.13	heap array resizes
zero-before-free	5501.68	254.51	168.18	4.10	
rearrange-heap	53.48	123.81	175.28	4.16	
pad-malloc 8	5662.55	259.95	176.06	4.11	
pad-malloc 32	5795.81	255.93	199.22	4.16	
pad-malloc 256	138.27	256.14	359.25	4.22	
pad-malloc 1024	178.92	254.13	827.85	4.10	
no-diversity	8019.51	4346.01	140.84	6.37	immediate frees
zero-before-free	7891.38	4273.82	138.36	92.24	
rearrange-heap	21.79	4722.31	273.40	4.25	
pad-malloc 8	8083.59	4318.01	163.51	5.70	
pad-malloc 32	7369.63	4322.33	183.66	6.66	
pad-malloc 256	7764.75	4290.65	333.04	5.66	
pad-malloc 1024	50.62	4346.71	804.23	7.57	

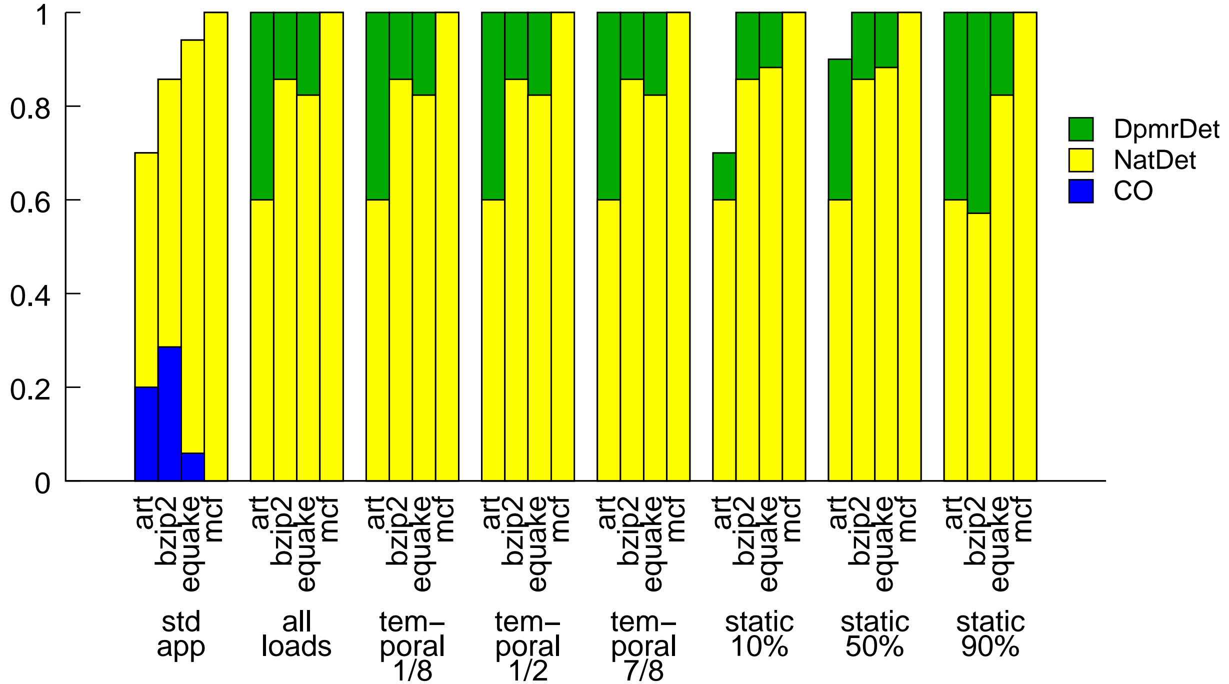

Figure 4.11: Mean MDS heap array resize coverage of state comparison policies

Figures 4.11 and 4.12 plot the mean coverage of state comparison policies under the rearrange-heap diversity mechanism, and Figures 4.13 and 4.14 plot mean conditional coverage. As was the case earlier, coverage results are shown on a per-application basis, and conditional coverage results are combined across all applications. The results are similar to those for SDS. Coverage remains robust in the face of reduced checking. In the MDS results, temporal load-checking looks to be more robust than static load-checking. That is most likely due to the nature of static load-checking. A reduction of 50% of the load checks at compile time could end up removing nearly all or almost none of the load checks during runtime. The drop in coverage of the 10% and 50% static load-checking policies could also be an indication that certain faults end up impacting only certain load sites. Under temporal load-checking, each load will most likely get checked at some point, while under static load-checking, if a load check is removed, then that load site will never be checked.

MDS detection latency is captured in Table 4.6, for state comparison policies. MDS detection latency exhibits similar behavior to SDS detection latency. Static load-checking tends to have similar or lower detection latencies than the all loads policy does, and temporal

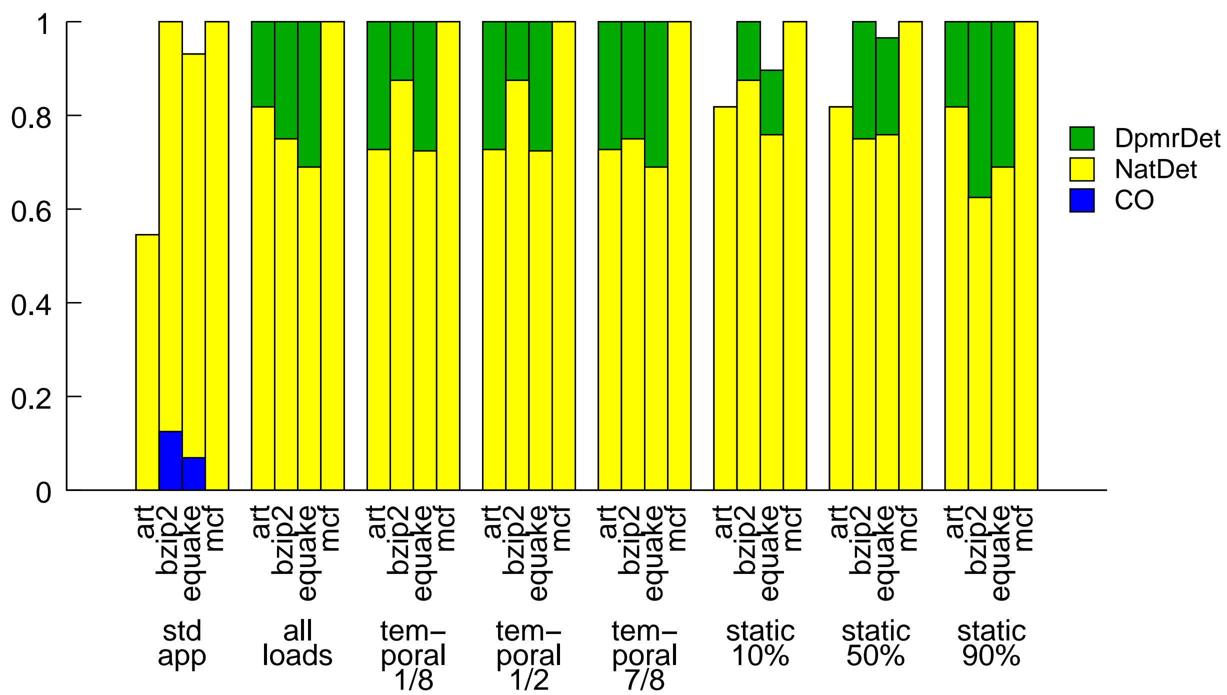

Figure 4.12: Mean MDS immediate free coverage of state comparison policies

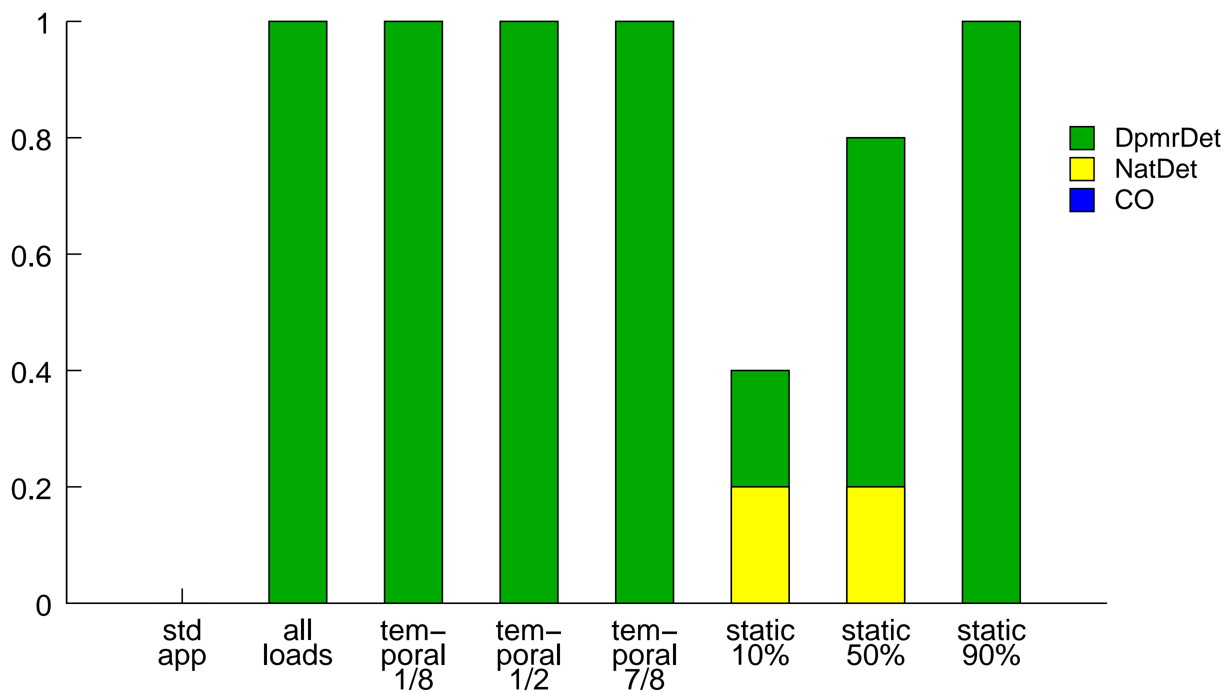

Figure 4.13: Mean MDS heap array resize conditional coverage of state comparison policies

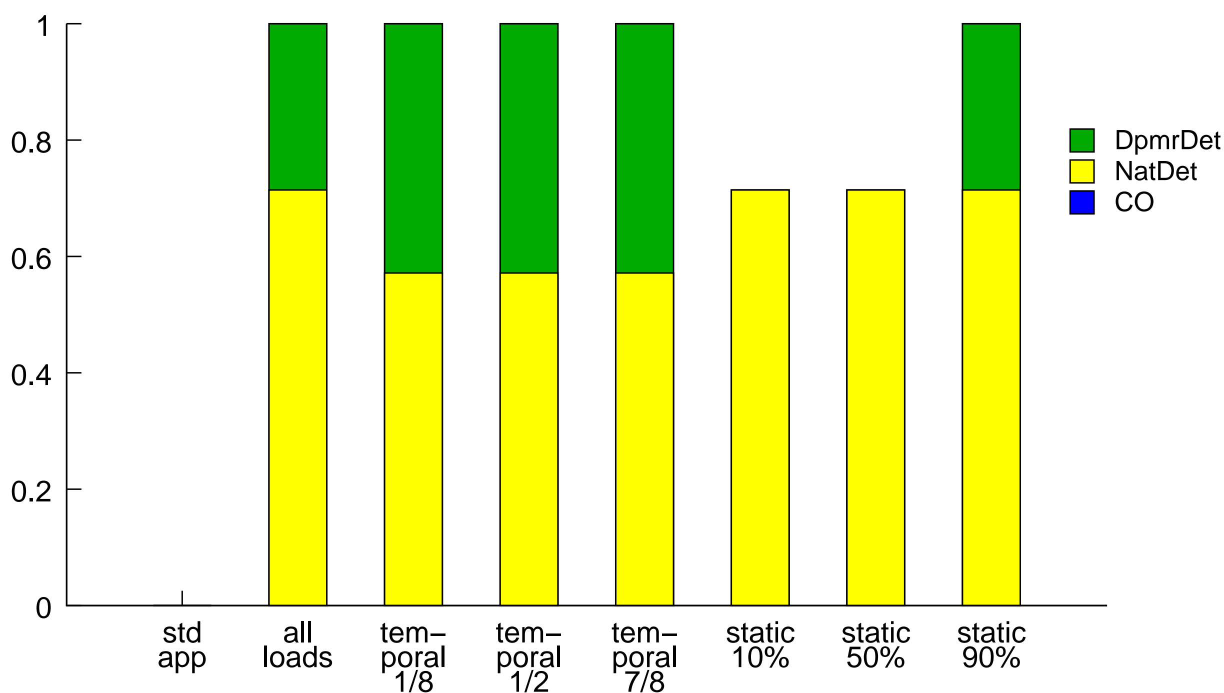

Figure 4.14: Mean MDS immediate free conditional coverage of state comparison policies

Table 4.6: Mean time to detection of state comparison policies under MDS

variant	msecs				
	art	**bzip2**	**equake**	**mcf**	
all loads	53.48	123.81	175.28	4.16	heap array resizes
temporal 1/8	96.88	270.04	182.76	4.01	
temporal 1/2	50.58	267.70	172.57	3.96	
temporal 7/8	48.64	288.11	173.50	4.15	
static 10%	42.60	183.41	196.48	4.11	
static 50%	38.24	108.71	202.00	4.00	
static 90%	45.78	103.58	176.88	4.20	
all loads	21.79	4722.31	273.40	4.25	immediate frees
temporal 1/8	84.59	7181.54	286.83	2.89	
temporal 1/2	31.00	7737.00	284.68	2.95	
temporal 7/8	31.14	8592.45	290.79	2.97	
static 10%	11.97	2997.18	211.01	5.03	
static 50%	11.93	4041.11	254.92	3.98	
static 90%	21.34	3698.74	269.41	5.75	

load-checking tends to have higher detection latencies.

Chapter 5

Scope Expansion Through Static Analysis

In this chapter, we aim to eliminate most of the restrictions on input programs under the MDS-based DPMR approach. We do so by reasoning about program memory using a context-sensitive, field-sensitive points-to analysis called Data Structure Analysis [63]. Section 5.1 describes the features of Data Structure Analysis that are most important for DPMR. In Section 5.2 we examine the restrictions imposed by MDS and why they exist. In Section 5.3 we propose algorithms, based on Data Structure Analysis, that eliminate most of those restrictions. In Section 5.4 we examine how external code support libraries can exist alongside those algorithms. Section 5.5 discusses behavior that Data Structure Analysis considers unknown. Finally, Section 5.6 considers issues that remain.

5.1 Data Structure Analysis

Data Structure Analysis (DSA) is a context-sensitive, field-sensitive pointer/memory analysis. Context sensitivity is achieved by distinguishing heap objects by (acyclic) call paths, rather than by allocation sites. DSA also maintains field sensitivity when memory is used in a type-homogeneous manner, which is of particular importance for DPMR. Field sensitivity allows DPMR to distinguish between pointers and non-pointers in memory. The version of Data Structure Analysis that we use is based on the latest developed version and contains some features not present in the version detailed in [63]. However, all of the features we use are described here.

DSA produces a per-function DS graph that can be queried to determine properties about memory pointed to by virtual registers in the associated function. A DS graph is a directed graph defined for a function F by the tuple (N, V, N_{call}, E, E_V), where N, V, and N_{call} are

sets of nodes in the graph, and E and E_V are sets of edges in the graph.[1] N is a set of nodes called *DS nodes*, each of which represents some set of memory objects that are visible to F. As discussed below, DS nodes are composed of a set of data fields, among other things. V is the set of virtual registers in the scope of F, including registers for F's arguments, F's return value, and global variables. As stated at the beginning of Chapter 2, global variables are treated as pointers to memory. N_{call} is a set of *call nodes* that represent call sites in F. Like DS nodes, call nodes are composed of a set of fields, the meaning of which is discussed below. E is a set of edges, each of which maps either a DS node-field pair $\langle n_{s1}, f_{s1}\rangle$ or a call node-field pair $\langle n_{s2}, f_{s2}\rangle$ to a DS node-field pair $\langle n_d, f_d\rangle$, such that $n_{s1}, n_d \in N$, $n_{s2} \in N_{call}$, $f_{s1} \in \text{fields}(n_{s1})$, $f_{s2} \in \text{fields}(n_{s2})$, and $f_d \in \text{fields}(n_d)$. Edges in E are constrained such that at most one edge $e \in E$ may have $\langle n, f\rangle$ as a source, for a given $n \in (N \cup N_{call})$ and $f \in \text{fields}(n)$. Finally, E_V is a set of edges mapping a virtual register $v \in V$ to a DS node-field pair $\langle n, f\rangle$.

We now discuss the components of DS nodes. DS nodes are composed of four components. One component is a list of types that the memory represented by a node may take. Another component is a list of global objects represented by the node. Functions are global objects, so a DS node may represent a function. The third component is a set of fields. The fields represent a partitioning of the corresponding memory object into different data types. Field sensitivity is maintained only if the object is used in a type-homogeneous manner. If a field is used as a pointer, i.e., it is used to perform a load, store, or indexing operation, then the field will have an outgoing edge that is an element of E. The last component of a DS node is a set of flags, each of which we describe below.

- *complete and incomplete flags (C, I)*: Complete nodes are nodes for which all information has been processed. Complete nodes may not alias with any other nodes (with the exception of unknown nodes, which we discuss later). Once a node has been marked complete, its properties will never change, and it will never be merged with another node. On the other hand, incomplete nodes represent objects for which we have partial information. Information may be partial because pointers to the DS node escape to

[1]Note that the DS graph definition given here is slightly different from that given in [63].

```
1 int32* x = (int32*)malloc([3xint32]);
2 int64 y = (int64)x;
3 y = y+4;
4 int32* z = (int32*)y;
5
```

(a) Pointer-to-int and int-to-pointer casts

```
1 int32** x = malloc(int32*);
2 int32* y = (int32*)malloc([2xint32]);
3 *x = y;
4 int64* xc = (int64*)x;
5 int64 yval = *xc;
```

(b) Layered pointer-to-int and int-to-pointer casts

Figure 5.1: Examples of pointer-to-int and int-to-pointer behavior

external functions. The properties of an incomplete node may change in the future, and the node may be merged with other nodes. Conservatively, we must assume that any two incomplete nodes can alias.

- *memory segment flags (H, S, G)*: DS nodes are marked with H, S, and G flags indicating that the memory represented by the node may reside on the heap, in the stack, or in global variables, respectively. One DS node may possess multiple memory segment flags, indicating that the memory it represents may reside in any of the flagged segments.

- *array flag (A)*: The array flag indicates that the memory represented by a DS node includes one or more array objects. If a DS node has fields f_0 to f_n and is marked as an array, then it is assumed that each element in the array has fields f_0 through f_n.

- *collapsed flag (O)*: If the fields of a DS node are used in a non-type-homogeneous way, then the fields of the node are collapsed into a single field, and the collapsed flag is set. When the fields are collapsed, the outgoing edges of those fields are collapsed into a single edge through merging of the nodes to which those pointers point. Finally, the node is marked with the array flag, and its type is set to a byte array.

- *pointer-to-int and int-to-pointer flags (P, 2)*: The latest version of DSA uses pointer-to-int and int-to-pointer flags. There are two types of pointer-to-int and int-to-pointer behavior that DSA captures. The first type is standard pointer-to-int and int-to-pointer casts as illustrated in lines 2 and 4 of Figure 5.1(a), respectively. The code in Figure 5.1(a) causes the DS node to which `x` points to be marked with a pointer-to-int flag. Meanwhile, the node to which `z` points is marked with an int-to-pointer flag.

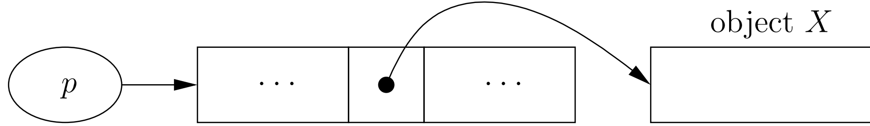

Figure 5.2: Reachable object

Note that in the example in Figure 5.1(a), `x` and `z` point to the same DS node. The second type of behavior that DSA captures is layered pointer-to-int and int-to-pointer casts, i.e., casts that are buried under at least one pointer. Line 4 of Figure 5.1(b) contains a layered pointer-to-int cast. In line 5, the pointer that `x` points to is loaded as an integer. At that point, DSA marks the memory to which `x` points as pointer-to-int and int-to-pointer.

- *unknown flag (U)*: The unknown flag is used for nodes whose allocation source is unrecognized. Since DSA does not track pointers through integers, nodes marked int-to-pointer are also marked as unknown. Conservatively, it must be assumed that unknown nodes can alias with any other node, including complete nodes.

Call nodes are composed of a set of fields, just like DS nodes; however, the other DS node components do not apply to call nodes. The fields of a call node represent the return value for, the called function pointer for, and parameter values passed to the call site represented by the call node. If the call site's return value or a parameter of the call site is used as a pointer, then the corresponding field in the call node will have an outgoing edge that is a member of E. A call node's function pointer field points to a DS node representing the function (or collection of functions) that may be called by the corresponding call site.

DS graphs are constructed using a three-phase process. The first phase, the *local phase*, constructs a DS graph for each function in the input program. The graph that is constructed for a function F considers only F's instructions. All DS nodes in F's DS graph are initially marked incomplete. During the local phase, those nodes will be marked complete only if the memory they represent is not reachable via potential callers of F and potential callees of F. The reachability property includes reachability through pointers stored in memory. For example, in Figure 5.2, if p is passed to a function G via a function call, then object X is reachable from G.

The second phase is the *bottom-up phase*. During the bottom-up phase, information from callees is propagated to callers. That is done by cloning of the graphs of callees into callers. After the bottom-up phase, some incomplete nodes in a DS graph may be marked complete. In particular, if an incomplete node in function F's DS graph is not reachable by callers of F, or external callees, then the node is marked complete. The final phase is the *top-down phase*. The top-down phase propagates analysis from callers to callees by merging the graphs of callers into callees. Upon completion of the top-down phase, the nodes in function F's DS graph will be marked complete unless they are reachable by unresolved external callers or callees.

5.2 Behavior Forbidden by Mirrored Data Structures

Section 4.4 lists the restrictions imposed by MDS on input programs. In this section, we discuss in detail the problems that lead to those restrictions so that we can handle them using DSA.

- *load comparison problem*: When DPMR loads a value from memory, it needs to be able to determine if the value is a pointer value. If the value could be a pointer value, then we need to omit any comparisons, or the result could be a false positive.
- *int-to-pointer casts*: MDS cannot tolerate int-to-pointer casts, because such casts would prevent it from computing an appropriate ROP.
- *stores of pointers masquerading as integers*: When a pointer masquerading as an integer is stored to memory, MDS stores an application pointer to both application memory and replica memory, as illustrated in Figure 5.3. That can lead to two problems. First, it can lead to a memory leak. If the only pointer to an object is stored to memory as an integer and then goes out of scope, we will lose the only pointer to its replica. The second problem that can occur is the omission of a replica update. Consider the memory layout in Figure 5.4. Storing a pointer to X in A that is masquerading as an integer, causes the same pointer to be stored to A_r. If the pointer stored to A is later

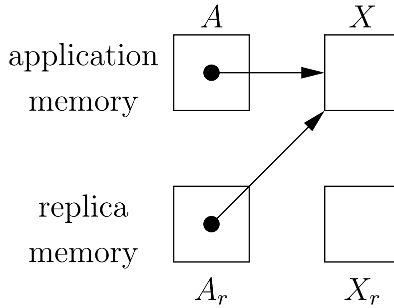

Figure 5.3: Result of storing a pointer masquerading as an integer

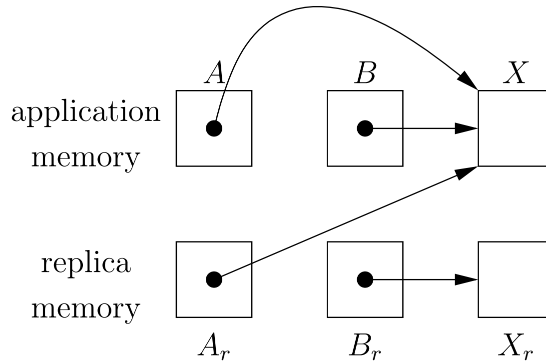

Figure 5.4: Situation that could lead to an update omission

used to update X, X will be updated, but X_r will not be updated. If the pointers to X and X_r stored in B and B_r, respectively, are used to read X and X_r, we will detect a discrepancy between the values of X and X_r.

- *pointer-to-int casts and loads of pointers masquerading as integers*: Pointer-to-int casts are not a problem; neither are loads of pointers masquerading as integers, as long as such loads are not involved in a load comparison. However, pointer-to-int casts and loads of pointers masquerading as integers often lead to int-to-pointer casts and stores of pointers masquerading as integers, which are problems, as we have discussed.

- *loads and stores of integers masquerading as pointers*: We wish to point out that loads and stores of integers masquerading as pointers are not actually problems, however bizarre such behavior may seem.

- *external code*: In the MDS design discussed in Chapter 4, external code is a problem, because it can lead to the omission of DPMR behavior, such as replica allocation,

deallocation, and updates.

5.3 Eliminating Limitations

Using DSA, we can eliminate most restrictions on input programs. Before presenting the techniques for doing so, we stipulate that after DSA analyzes an input program, any DS nodes that are marked as unknown must have been caused by int-to-pointer behavior whose source is pointer-to-int behavior. Such a stipulation is more than reasonable for most applications. Some exceptions do exist, such as device drivers, which may require writes to a particular range of memory addresses. However, Section 5.5 discusses methods for getting around the new stipulation. It should be noted that the stipulation is a subset of the restrictions given in Section 5.2.

The technique for handling int-to-pointer casts and stores of pointers masquerading as integers is discussed first. The approach we take is to eliminate the allocation of replica memory corresponding to an application object O, whose references could be involved in an int-to-pointer cast or stored as pointers masquerading as integers. At such allocation sites, the corresponding ROP is set to point to O, the allocated application object. In addition, at int-to-pointer casts, the corresponding ROP receives the same value as the resulting application pointer. The result is that an ROP corresponding to a pointer that points to O will also point to O. Such a procedure renders some load comparisons ineffective, but it is safe, as those load comparisons will never falsely detect errors.

When stores of pointers masquerading as integers occur, we store to replica memory the same integer value that is stored to application memory, which is the precise behavior we desire, given the above modification to our allocation strategy. Furthermore, the two problems that result from stores of pointers masquerading as integers, as discussed in the previous section, are no longer issues. A replica memory leak and an omitted replica update can no longer occur because there is no replica.

The technique just discussed causes us to require two additional changes. First, we must eliminate replica allocations for application memory that is reachable from unreplicated memory. We must do so because unreplicated memory does not provide a place to store ROPs

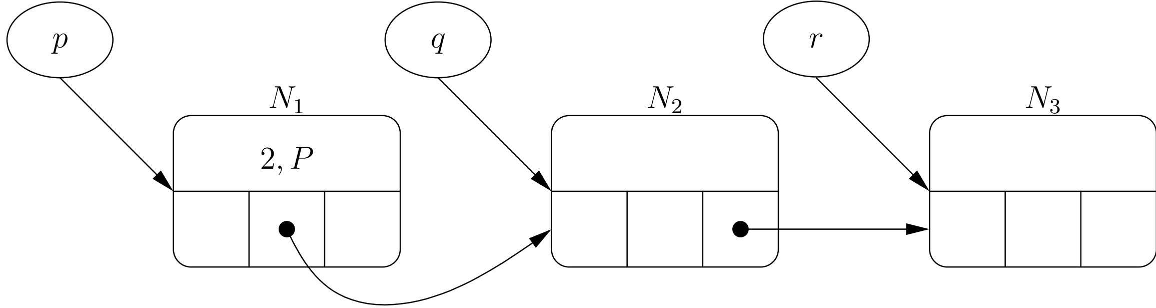

Figure 5.5: DS graph for $F1$

for objects directly reachable from the unreplicated memory. Second, at heap deallocation sites that may involve unreplicated memory, we must insert a check into the transformed program that eliminates the deallocation of replica objects if the pointers to them are equal to the corresponding application pointers.

When determining if an allocation site should be replicated, we eliminate an allocation site $p \leftarrow allocate(\tau)$ from consideration if the DS node to which p points is marked pointer-to-int, int-to-pointer, or incomplete. Allocation sites pointing to DS nodes marked pointer-to-int or int-to-pointer are not replicated because pointers to those nodes may have been stored as integers. In addition, nodes marked int-to-pointer include nodes whose pointers may have been involved in int-to-pointer cast instructions. Allocation sites pointing to incomplete nodes should not be replicated because we do not have enough information to know if they should be marked with pointer-to-int or int-to-pointer flags.

Those conditions may seem sufficient for determining if an allocation site should be replicated. However, they are not. As stated earlier, we need to eliminate the replication of memory that is reachable from other unreplicated memory. While it is true that a node reachable from an incomplete node will be incomplete, that type of reachability property does not extend to pointer-to-int and int-to-pointer flags. For example, if a node is marked int-to-pointer, that does not imply that the nodes it can reach will be marked int-to-pointer. Furthermore, DS graphs are insufficient to determine that reachability. Consider the DS graph in Figure 5.5, for a function $F1$. $N1$ is marked pointer-to-int and int-to-pointer, and one of its fields points to $N2$. In addition, one of $N2$'s fields points to $N3$. Imagine the case where q is passed to a callee $F2$. When $F1$'s DS graph is cloned into $F2$'s DS graph,

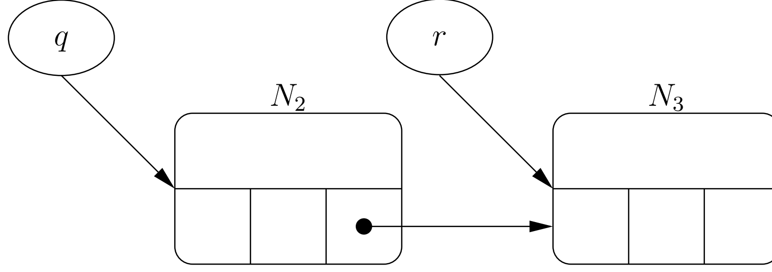

Figure 5.6: DS graph for $F2$

Input $G = (N, V, N_{call}, E, E_V)$: DS graph
markX(G)
1: $WorkingSet \leftarrow \varnothing$
2: $ProcessedSet \leftarrow \varnothing$
3: $\forall n \in N$
4: **if**($P \in$ flags(n) $\vee$ $2 \in$ flags(n) $\vee$ $X \in$ flags(n))
5: $WorkingSet \leftarrow WorkingSet \cup \{n\}$
6: **while**($\exists n \in WorkingSet$)
7: $WorkingSet \leftarrow WorkingSet - \{n\}$
8: $ProcessedSet \leftarrow ProcessedSet \cup \{n\}$
9: flags(n) $\leftarrow$ flags(n) $\cup \{X\}$
10: $\forall f \in$ fields(n)
11: **if**($E(\langle n, f\rangle) = \langle m, g\rangle : m \in N, g \in$ fields(m))
12: **if**($m \notin WorkingSet \wedge m \notin ProcessedSet$)
13: $WorkingSet \cup WorkingSet\{m\}$

Figure 5.7: Algorithm for markX()

$N1$ will not be included because it is not reachable from $F2$, as illustrated in Figure 5.6. Now, imagine that an allocation instruction A in $F2$ can allocate $N3$. When we examine the DS node to which A points, i.e., $N3$, $F2$'s graph will not be enough to tell us that a node marked pointer-to-int or int-to-pointer could point to $N3$.

One solution to the above problem is to add a new flag X to DSA. If a node is marked with X, then it is reachable by a pointer-to-int or int-to-pointer node. It is possible to propagate X by applying the algorithm in Figure 5.7 to each DS graph after each phase of DSA. The algorithm starts by building a working set of all nodes that have pointer-to-int, int-to-pointer, or X flags. For each node in the working set, we add the X flag to it. We also add the node's children to the working set, if they are not already in the working set and have not already been processed. The result allows us to declare, at the conclusion of DSA,

that a node should not be replicated if it has either the I or X flag.

We now tackle the load comparison problem. Given our new constraints on replicating memory, the load comparison problem becomes very simple. All we need to do is check if the loaded value has ever been used as a pointer. For a load $x \leftarrow *p$, if p points to $\langle n, f \rangle$, such that $n \in N$ and $f \in \text{flags}(n)$, then we should omit a comparison if $E(\langle n, f \rangle) \neq \varnothing$. If $E(\langle n, f \rangle) \neq \varnothing$, then x has been used as a pointer. If $E(\langle n, f \rangle) = \varnothing$, it is still possible that x is a pointer. That case can arise only if n has incomplete, int-to-pointer, or pointer-to-int behavior. However, if n has incomplete, int-to-pointer, or pointer-to-int behavior, then it will not be replicated, in which case we will be comparing a pointer to itself, which is safe.

An additional benefit of the proposed approach is that it allows us to process input programs that use external code for which DPMR does not have support. Any memory that originates in unsupported external code or is passed to unsupported external code is not replicated. At first, it might seem that we would not be able to detect errors in such unreplicated memory. However, that is not necessarily the case. Our detection mechanism is not based solely on examining memory from which an error originated. It is also based on examining replicated memory in which one member is corrupted by an error that originated elsewhere. Our results using static load-checking suggested that it is not necessarily a requirement that we perform load checks on the memory from which an error originated, lending further credibility to the techniques proposed in this chapter.

5.4 Using Data Structure Analysis with External Code Support Libraries

The techniques suggested in this chapter lead to the question of whether we can still use external code support libraries to try to strengthen dependability. The answer is yes. In order to use an external function wrapper for an external function F, we require that F be modeled in DSA. Since external function wrappers emulate the DPMR behavior that would be applied to an external function if it were not external, we need the external function to be modeled by DSA so that we can make decisions on what memory to replicate and which loads to check. The current DSA implementation has behavior modeled for most

`libc` functions.

A wrapper for F that is used at a call site C must now meet the following necessary conditions. (1) If F allocates memory, and at the call site C, according to our algorithms, that memory should not be replicated, then the wrapper must not replicate it. (2) If F loads a value from memory that should not be compared to a replica, then the wrapper must not perform the load comparison. (3) If F deallocates memory that might not be replicated, according to our algorithms, then an ROP for that memory must not be deallocated if it equals the corresponding application pointer, at runtime. That leads to the existence of multiple wrappers for F, all with different replication and load comparison behaviors. Fortunately, there is a default safe wrapper behavior that can be used if one does not wish to generate wrappers for all possible replication and load-checking behaviors, of which there could be an arbitrarily large number for a function like `memcpy()`. The default safe behavior is never to replicate allocations, never to perform load comparisons, and to check that ROPs do not equal their corresponding application pointers before deallocating them.

5.5 Unknown DSA Behavior

As discussed in the previous section, we have one remaining restriction. Specifically, we require that unknown nodes in DSA must result from int-to-pointer behavior whose source is pointer-to-int behavior. Thus, programs are forbidden to hard-code addresses, instead of going through the standard channels of `malloc`, `alloca`, and global variables.[2] If a program violates our restriction, one of three things can occur. (1) If unknown behavior results in a pointer to memory M that does not alias with the memory allocated by `malloc`, `alloca`, and global variables, then everything will work fine. M will be like any other unreplicated memory, and pointers resulting from int-to-pointer instructions will result in ROPs with the same pointer value, as they should. Therefore, device driver code that requires hard-coded addresses to memory that is not allocated through standard channels will operate correctly

[2]Other standard channels of memory allocation exist, such as `mmap()` and application-specific allocators. Such allocation channels are not handled by the current implementation of DPMR; however, they will work with the algorithms proposed in this chapter.

under DPMR. (2) If unknown behavior results in pointers to unreplicated memory allocated by `malloc`, `alloca`, or global variables, then a similar scenario will result. (3) Finally, if unknown behavior results in pointers to replicated memory, omitted replica updates, memory leaks, and false positives in DPMR detection could occur.

If a program wishes to replicate memory that must reside at hard-coded addresses and is not allocated using `malloc`, `alloca`, or global variables, the easiest solution is to include an external function in the input program that produces the address. Then a wrapper can be written for the external function that allocates replica memory, most likely on the heap. The model of the external function that would be used by DSA would treat the function as if it allocated a block of memory on the heap.

5.6 Remaining Issues

Earlier in this chapter, we presented techniques to remove the restrictions on input programs under MDS. However, there are a few remaining issues that deserve discussion. The first is the precision of DSA. As previously noted, a DS node represents a group of memory objects. If a node is marked incomplete, pointer-to-int, or int-to-pointer, that does not necessarily mean that every memory object represented by the node escapes to external code, is involved in pointer-to-int behavior, or is involved in int-to-pointer behavior, respectively. One way to distinguish the objects represented by a single DS node is to utilize function inlining. Consider the code in Figure 5.8. Assume that pointers in the target architecture are 64 bits in length. In addition, assume that `f()` and `g()` are the only functions that ever call `copy128()`. After DSA runs, the pointers `p` and `q`, in function `f()`, will point to DS nodes that are marked complete, while `o1` and `o2`, in function `g()`, will point to DS nodes that are marked incomplete. `o1` and `o2` are marked incomplete because they escape to the external `getObject()` function. As a result, the DS nodes pointed to by `s` and `d`, from `copy128()`, will be marked incomplete. In `copy128()`, there is no way to distinguish between the objects coming from `f()` and the objects coming from `g()`, so their behavior is merged in the analysis of `copy128()`. If `copy128()` were inlined into `f()` and `g()`, then we would be able to distinguish between the two different behaviors.

```
1 typedef struct{
2   int8[]* x;
3   int64 y;
4 } Object;
5
6 extern Object* getObject(void);
7
8 void copy128(void* d, void* s){
9   int64[]* a = (int64*)s;
10  int64[]* b = (int64*)d;
11  a[0] = b[0];
12  a[1] = b[1];
13 }
14
15 void f(){
16  int64[]* p = malloc([2xint64]);
17  p[0] = 1;
18  p[1] = 2;
19  int64[]* q = malloc([2xint64]);
20  copy128(q, p);
21 }
22
23 void g(){
24  Object* o1 = getObject(void);
25  o1->x = malloc([10xint8]);
26  o1->y = 10;
27  Object* o2 = getObject(void);
28  copy128(o2, o1);
29 }
```

Figure 5.8: Candidate for inlining

Another issue that we have not yet addressed is concurrency. Given our assumption that global variables are pointers to statically allocated memory, we can assume that the only information shared between threads is shared through memory. To ensure consistency, we must ensure that reads to application and replica memory, as well as writes, are atomic. Memory allocations of application and replica memory need not be atomic as long as they occur one after the other before any users of either allocation are allowed to proceed in the corresponding thread of execution. Memory deallocation need not be atomic either, although the reasoning is more complicated. Not requiring deallocation to be atomic could lead to arbitrary behavior if another thread of execution accessed one object in an application object-replica pair, when one of the objects was deallocated and one was not. However, such behavior could only result from a race condition, which is erroneous behavior. Therefore, the expected results are already arbitrary.

Chapter 6

Future Possibilities

This dissertation has proposed and evaluated many ideas centered around DPMR; however, much more research could be done. We discuss five possibilities below.

- *application of DPR in other contexts*: As discussed in Section 1.2, DPMR falls into a broader category of Diverse Partial Replication techniques. In that section, we discussed how diverse replicated execution might be used to detect race conditions. One path of new research would be to explore the ability of DPR to detect concurrency errors.
- *using DSA with SDS*: In Chapter 5, we discussed applications of DSA to remove restrictions on the MDS-based DPMR design. DSA could be used to do the same for the SDS-based design. One of the key differences between the restrictions of SDS and those of MDS is that SDS requires more extensive knowledge of memory types. Fortunately, DSA tracks memory types and discerns when memory is used in a type-inconsistent manner. Using a strategy similar to the one discussed in Chapter 5, SDS could use DSA to disable the replication of memory whose type is not used in the required manner.
- *selective pool replication*: One goal of DPMR that could be further explored is tunability. One approach for achieving greater tunability is to logically segregate application memory into independent pools for which replication can be enabled and disabled. A similar concept is applied in Chapter 5. In that chapter, we describe work in which we segregated memory into two pools and disabled replication for one of the pools. However, that idea could be extended using concepts from the compiler transformation

Pool Allocation [64]. In the Pool Allocation work, the heap is physically partitioned; however, the framework is there to distinguish pools of memory logically.

- *runtime tunability*: Another potential direction of future work would be development of techniques for runtime tunability. We already explored one strategy that could potentially be used for runtime tunability, i.e., temporal load-checking. The frequency of runtime checks could be tuned at runtime under temporal load-checking. Unfortunately, we saw that temporal load-checking was unable to fulfill its goal of reducing overhead. However, in Section 3.8 we discussed how the periodic nature of temporal load-checking could be exploited to achieve a more efficient implementation. Although such a technique requires knowledge of the checking frequency, a program, at runtime, could switch between multiple versions of a code segment, each with a different checking frequency. If selective pool replication were developed, it might also be possible to provide tunability by allowing replication of memory pools to be enabled and disabled at runtime.

- *memory performance optimization*: Clearly, if replica memory is required, there will always be overhead associated with it. However, one method for minimizing that overhead is to hide latencies associated with fetching memory pages that hold replica and shadow objects. We propose two ways to do that. First, for objects that are significantly smaller than a page size, we could force application objects, replica objects, and shadow objects (if using SDS) to be placed on the same memory page. When doing so, we would still need to ensure sufficient diversity for detection purposes. Second, load checks could be transformed so that replica objects are fetched at the time application objects are loaded, but the load checks are delayed, so that execution is not stalled waiting for replica objects. Prefetching instructions exist in the x86 SIMD instruction set that could be used for that purpose.

Chapter 7

Conclusion

At the beginning of this dissertation, we hypothesized that diversity and replicated runtime execution could be utilized in an automated way to detect software memory errors in an effective and tunable manner. The analysis throughout this dissertation justifies that hypothesis. In particular, we proposed the DPMR approach, in Chapter 2, to detect spatial and temporal memory errors in all segments of data memory. In doing so, we defined the partial replication concept and discussed its application to memory. We also proposed a solution to the problem of handling pointers stored to memory. Our solution involved the use of shadow data structures and was chosen to maximize error detection. We then developed compiler techniques that automatically transform an input program to use partial memory replication. The development of those transformations was followed by the development of transformations for diversifying memory layouts. We also designed tunable policies for comparing the states of partial memory replicas.

In Chapter 3, we discussed the construction of a DPMR compiler for the C programming language. That discussion involved many practical details specific to the application of DPMR to C. That chapter also contained an extensive experimental evaluation of DPMR. The evaluation included fault-injection experiments to evaluate the dependability properties of DPMR, as well as non-fault-injection experiments to ascertain the overhead imposed by DPMR. Results showed that the proposed rearrange-heap transformation detected all of the faults we injected. When evaluating state comparison policies, we found out that the temporal load-checking implementation was insufficient for reducing overhead, while static load-checking was able to reduce overhead up to one-third while still achieving high coverage. Based on static load-checking results, we conjectured that coverage was spatially robust, as coverage did not drop until we dramatically spatially reduced the checking frequency.

Chapter 4 proposed another memory layout as an alternative to the SDS-based layout that was initially proposed. The alternative layout, MDS, has three advantages over SDS. First, its memory footprint is two times the size of the original application's, while the memory footprint for SDS is two to four times the size of the original application's. Second, MDS imposes fewer restrictions on input programs than SDS does. Third, MDS has the potential to impose less overhead than SDS. Indeed, experimental results demonstrated modest overhead improvements for two of the applications under study.

Although the MDS approach relaxed the restrictions of SDS, it still imposed important restrictions. Chapter 5 aimed to remove nearly all of those restrictions, including requirements on external code. The approach for relaxing the restrictions is to use a memory analysis, called Data Structure Analysis, to identify the program behavior about which we can adequately reason. We then eliminate the application of DPMR to behavior for which we cannot adequately reason. Although that has the potential to reduce coverage, it does not mean that we will not be able to detect software memory errors in the unreplicated components. Our approach detects errors when they corrupt replicated memory; however, the errors may originate in replicated or unreplicated memory.

We feel that DPMR is an important new approach to detecting software memory errors. It showcases the power of diverse replication as an approach for detecting software errors. Although further contributions can be made to improve tunability and reduce overhead, we feel that this dissertation has contributed a solid framework on which to build.

References

[1] CERT, "CERT statistics," 2009. [Online]. Available: http://www.cert.org/stats/

[2] R. Charette, "Why software fails," *IEEE Spectrum*, vol. 42, no. 9, pp. 42–49, September 2005.

[3] NIST: National Institute of Standards and Technology, "Software errors cost U.S. economy $59.5 billion annually: NIST assesses technical needs of industry to improve software-testing," June 2002. [Online]. Available: http://web.archive.org/web/20080602124623/http://www.nist.gov/public_affairs/releases/n02-10.htm

[4] D. Scott, "Making smart investments to reduce unplanned downtime," Gartner Group, Tactical Guidelines TG-07-4033, March 1999.

[5] J. Nelißen, "Buffer overflows for dummies," May 2002, SANS Institute. [Online]. Available: http://www.sans.org/reading_room/whitepapers/threats/buffer-overflows-dummies_481

[6] CERT, "CERT advisory CA-2003-02 double-free bug in CVS server," March 2003. [Online]. Available: http://www.cert.org/advisories/CA-2003-02.html

[7] CERT, "Vulnerability note VU#350792 MIT Kerberos krb524d insecurely deallocates memory (double-free)," September 2004. [Online]. Available: http://www.kb.cert.org/vuls/id/350792

[8] iDefense Labs, "Opera Software Opera web browser BitTorrent dangling pointer vulnerability," July 2007. [Online]. Available: http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=564

[9] J. Afek and A. Sharabani, "Dangling pointer: Smashing the pointer for fun and profit," August 2007, Watchfire. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

[10] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[11] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the USENIX 2005 Technical Conference*, April 2005, pp. 17–30.

[12] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the Winter USENIX Conference*, January 1992, pp. 125–136.

[13] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 290–301, June 1994.

[14] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Software: Practice and Experience*, vol. 27, no. 1, pp. 87–110, January 1997.

[15] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2004, pp. 117–126.

[16] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the 22nd ACM Symposium on Operating System Principles*, October 2009, pp. 45–58.

[17] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 3, pp. 477–526, May 2005.

[18] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006, pp. 144–157.

[19] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proceedings of the 28th International Conference on Software Engineering*, May 2006, pp. 162–171.

[20] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks*, June 2006, pp. 269–280.

[21] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *Proceedings of the Third International Workshop on Automatic Debugging*, May 1997, pp. 13–26.

[22] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004, pp. 159–169.

[23] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006, pp. 158–168.

[24] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: Trading address space for reliability and security," in *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, March 2008, pp. 115–124.

[25] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, October 2010, pp. 573–584.

[26] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," in *Proceedings of the 12th International Symposium on Architecture Support for Programming Languages and Operating Systems*, October 2006, pp. 207–218.

[27] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: Automatically correcting memory errors with high probability," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007, pp. 1–11.

[28] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies – a safe method to survive software failures," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005, pp. 235–248.

[29] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: Protecting critical data in unsafe languages," in *Proceedings of the 3rd ACM European Conference on Computer Systems*, April 2008, pp. 219–232.

[30] A. Avižienis, "The methodology of N-version programming," in *Software Fault Tolerance*, M. R. Lyu, Ed. Wiley, 1995, pp. 23–46.

[31] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *Proceedings of the 15th USENIX Security Symposium*, August 2006, pp. 105–120.

[32] B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space," in *Proceedings of the 4th ACM European Conference on Computer Systems*, April 2009, pp. 33–46.

[33] T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Transactions on Computer Systems*, vol. 28, no. 2, pp. 1–54, July 2010.

[34] L. Chen and A. Avižienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proceedings of the 8th International Fault-Tolerant Computing Symposium*, June 1978, pp. 3–9.

[35] D. F. McAllister and M. A. Vouk, "Fault-tolerant software reliability engineering," in *Handbook of Software Reliability Engineering*, M. R. Lyu, Ed. McGraw-Hill, 1996, pp. 567–614.

[36] D. F. McAllister, C.-E. Sun, and M. A. Vouk, “Reliability of voting in fault-tolerant software systems for small output spaces,” *IEEE Transactions on Reliability*, vol. 39, no. 5, pp. 524–534, December 1990.

[37] Y.-W. Leung, “Maximum likelihood voting for fault-tolerant software with finite output-space,” *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 419–427, September 1995.

[38] J. Xu and B. Randell, “Software fault tolerance: $t/(n-1)$-variant programming,” *IEEE Transactions on Reliability*, vol. 46, no. 1, pp. 60–68, March 1997.

[39] B. Randell and J. Xu, “The evolution of the recovery block concept,” in *Software Fault Tolerance*, M. R. Lyu, Ed. Wiley, 1995, pp. 1–22.

[40] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, “A program structure for error detection and recovery,” in *Proceedings of an International Symposium on Operating Systems*, April 1974, pp. 171–187.

[41] B. Randell, “System structure for software fault tolerance,” *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, June 1975.

[42] J.-C. Laprie, J. Arlat, C. Beounes, K. Kanoun, and C. Hourtolle, “Hardware- and software-fault tolerance: Definition and analysis of architectural solutions,” in *Proceedings of the 17th International Fault-Tolerant Computing Symposium*, July 1987, pp. 116–121.

[43] J. C. Knight, N. G. Leveson, and L. D. St. Jean, “A large scale experiment in N-version programming,” in *Proceedings of the 15th International Fault-Tolerant Computing Symposium*, June 1985, pp. 135–139.

[44] J. C. Knight and N. G. Leveson, “An empirical study of failure probabilities in multi-version software,” in *Proceedings of the 16th International Fault-Tolerant Computing Symposium*, July 1986, pp. 165–170.

[45] S. S. Brilliant, J. C. Knight, and N. G. Leveson, “Analysis of faults in an N-version software experiment,” *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 238–247, February 1990.

[46] J. C. Knight and N. G. Leveson, “A reply to the criticisms of the Knight & Leveson experiment,” *SIGSOFT Software Engineering Notes*, vol. 15, no. 1, pp. 24–35, 1990.

[47] M. Castro, R. Rodrigues, and B. Liskov, “Base: Using abstraction to improve fault tolerance,” *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 236–269, August 2003.

[48] I. Gashi, P. Popov, and L. Strigini, “Fault diversity among off-the-shelf SQL database servers,” in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, June 2004, pp. 389–398.

[49] P. Koopman and J. DeVale, “Comparing the robustness of POSIX operating systems,” in *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, June 1999, pp. 30–37.

[50] A. Nguyen-Tuong, D. Evans, J. C. Knight, B. Cox, and J. W. Davidson, “Security through redundant data diversity,” in *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2008, pp. 187–196.

[51] V. Strumpen, “Compiler technology for portable checkpoints,” 1998, Massachusetts Institute of Technology. [Online]. Available: http://theory.lcs.mit.edu/~strumpen/porch.ps.gz

[52] K.-F. Ssu and W. K. Fuchs, “PREACHES – Portable recovery and checkpointing in heterogeneous systems,” in *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, June 1998, pp. 38–47.

[53] S. Forrest, A. Somayaji, and D. Ackley, “Building diverse computer systems,” in *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997, pp. 67–72.

[54] M. Chew and D. Song, “Mitigating buffer overflows by operating system randomization,” Department of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-02-197, December 2002.

[55] J. Xu, Z. Kalbarczyk, and R. K. Iyer, “Transparent runtime randomization for security,” in *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*, October 2003, pp. 260–269.

[56] S. Bhatkar, D. C. DuVarney, and R. Secar, “Address obfuscation: An efficient approach to combat a broad range of memory error exploits,” in *Proceedings of the 12th USENIX Security Symposium*, August 2003, pp. 105–120.

[57] The PaX Team, “Documentation for the PaX project,” December 2009. [Online]. Available: http://pax.grsecurity.net/docs/index.html

[58] G. S. Kc, A. D. Keromytis, and V. Prevelakis, “Countering code-injection attacks with instruction-set randomization,” in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003, pp. 272–280.

[59] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, “Randomized instruction set emulation,” *ACM Transactions on Information and System Security*, vol. 8, no. 1, pp. 3–40, February 2005.

[60] R. M. Lefever, V. S. Adve, and W. H. Sanders, “Diverse partial memory replication,” in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2010, pp. 71–80.

[61] “The LLVM compiler infrastructure,” November 2009. [Online]. Available: http://www.llvm.org/

[62] "SPEC CPU2000 V1.3," February 2007. [Online]. Available: http://www.spec.org/cpu2000/

[63] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007, pp. 278–289.

[64] C. Lattner and V. Adve, "Automatic pool allocation: Improving performance by controlling data structure layout in the heap," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005, pp. 129–142.

Author's Biography

Ryan M. Lefever received a Bachelor of Science degree with highest honors in Computer Engineering from the University of Illinois at Urbana-Champaign in May 1999. He remained at the University of Illinois to pursue graduate work in the PERFORM research group (`http://www.perform.csl.illinois.edu`), headed by Professor William H. Sanders. He received his Master of Science degree in Electrical and Computer Engineering in May 2003, having performed research that involved the experimental evaluation of distributed systems using fault injection. He then began his doctoral research at the University of Illinois, being co-advised by Professors Vikram S. Adve and William H. Sanders. He completed his Ph.D. research in December 2010, and in January 2011 began working for NetApp (`http://www.netapp.com`) in their Advanced Technology Group.