RICE UNIVERSITY

Reliability and Optimization for Resource-Constrained Embedded Systems

by

Rebecca Smith

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Scott Rixner, Chair
Professor of Computer Science and Electrical & Computer Engineering

Alan L. Cox
Professor of Computer Science and Electrical & Computer Engineering

Keith Cooper
L. John & Ann H. Doerr Professor of Computational Engineering; Professor of Computer Science and Electrical & Computer Engineering

Houston, Texas
April, 2015
ABSTRACT

Reliability and Optimization for Resource-Constrained Embedded Systems

by

Rebecca Smith

Embedded systems are ubiquitous, powering countless devices ranging from cars to appliances. As the software requirements of these systems grow increasingly complex, it is necessary to develop new approaches to simplify embedded systems programming. Recently, managed run-time systems have emerged as a means of increasing the productivity of writing embedded applications. Along with increased productivity, these run-time systems bring an intrinsic structure which provides new opportunities for addressing fundamental challenges faced by resource-constrained embedded systems.

This thesis presents novel mechanisms which utilize the structure imposed by managed run-time systems to address two key challenges of embedded systems programming: reliability and memory optimization. Though a wealth of past work explores these challenges in the context of conventional computing systems, the stringent resource constraints of embedded systems demand a more economical approach. Therefore, this thesis presents new techniques designed to accommodate the unique properties of embedded systems. First, this thesis presents Phoenix, a semi-automated system for recovering from hardware peripheral failures that is integrated into the run-time system. The design of Phoenix is uniquely tailored to embedded systems, inspired by novel insights into the characteristics of these systems as they pertain to reliability. Second, this thesis presents GEM (Graphs of Embedded Memory), an
extensible framework for memory optimization and analysis that capitalizes on the structure of a managed run-time system to build and operate on a graph of memory. The versatility of this framework is highlighted by the implementation and evaluation of four use cases. Through these two systems, this thesis demonstrates the potential of managed run-time systems to improve the future of developing safe and efficient embedded applications.
I would first and foremost like to thank my advisor, Scott Rixner, for being the most amazing mentor I could possibly have asked for. Working with him has been incredibly fun and rewarding, and I know I wouldn’t be where I am without him. I’m also grateful to the other members of my thesis committee, Alan Cox and Keith Cooper, for their valuable time, insights, and advice.

Last, I’ve been fortunate to work with a great group of students on Owl: Thomas, who showed me the ropes in the early years; Lingo, who was a sounding board and source of encouragement in the middle; and most especially Karen, whose friendship and unshakable cheerfulness never failed to brighten my day in the last year.
## Contents

Abstract ........................................ ii
List of Illustrations ........................ viii
List of Tables ................................ ix

1 Introduction .................................. 1
   1.1 Contributions .............................. 5
       1.1.1 Checkpointing and Recovery .............. 5
       1.1.2 Memory Optimization and Analysis ....... 7
   1.2 Organization ................................ 8

2 Background and Related Work ............ 10
   2.1 Reliability ................................ 10
   2.2 Memory Visualization and Transformation ...................................................... 14
       2.2.1 Visualization .......................... 15
       2.2.2 Transformation ......................... 16
   2.3 Summary ................................... 20

3 Checkpointing and Recovery .......... 21
   3.1 Introduction ............................... 21
   3.2 Key Insights .............................. 23
       3.2.1 Peripheral State in Embedded Systems 23
       3.2.2 Checkpointing in Embedded Systems ... 25
   3.3 Recovery Procedure ...................... 27
       3.3.1 High-Level Steps ....................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Low-Level Steps</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>System Mechanisms</td>
<td>30</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Failure Detection</td>
<td>31</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Checkpointing</td>
<td>34</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Compilation and Code Generation</td>
<td>40</td>
</tr>
<tr>
<td>3.5</td>
<td>Programmer Mechanisms</td>
<td>42</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Peripheral Classification</td>
<td>42</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Peripheral Initialization</td>
<td>45</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Config File</td>
<td>45</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Application Code</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Microbenchmarks</td>
<td>46</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Applications</td>
<td>52</td>
</tr>
<tr>
<td>3.7</td>
<td>Discussion</td>
<td>57</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Context</td>
<td>59</td>
</tr>
<tr>
<td>4.3</td>
<td>Tool Organization</td>
<td>61</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Graph Transformation</td>
<td>63</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Parsing and Unparsing</td>
<td>68</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Memory Set and Dump</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Use Cases</td>
<td>72</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Memory Visualization</td>
<td>72</td>
</tr>
<tr>
<td>4.4.2</td>
<td>De-duplication</td>
<td>75</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Heterogeneous Compilation</td>
<td>78</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Transparent Migration</td>
<td>82</td>
</tr>
<tr>
<td>4.5</td>
<td>Evaluation</td>
<td>84</td>
</tr>
</tbody>
</table>

4 Memory Optimization and Analysis

4.1 Introduction                                      58
4.2 Context                                          59
4.3 Tool Organization                                 61
  4.3.1 Graph Transformation                           63
  4.3.2 Parsing and Unparsing                           68
  4.3.3 Memory Set and Dump                             70
4.4 Use Cases                                        72
  4.4.1 Memory Visualization                           72
  4.4.2 De-duplication                                  75
  4.4.3 Heterogeneous Compilation                      78
  4.4.4 Transparent Migration                          82
4.5 Evaluation                                        84
4.5.1 Memory Visualization .................................................. 85
4.5.2 De-duplication ............................................................. 85
4.5.3 Heterogeneous Compilation .......................................... 89
4.5.4 Transparent Migration .................................................. 90

5 Conclusions ................................................................. 95
  5.1 Future Work .............................................................. 99
    5.1.1 Reliability .......................................................... 99
    5.1.2 Memory Visualization and Transformation .................... 102

Bibliography ............................................................... 106
Illustrations

3.1 Sample Application Code ........................................ 28
3.2 Phoenix System Architecture .................................. 31
3.3 Assembly Code Snippet ........................................... 37
3.4 Peripheral Class Excerpt ........................................ 43

4.1 Unpacking a Packed Tuple ...................................... 66
4.2 Splicing Source Graph into Destination Graph .............. 67
4.3 Graphical Display for GEM’s Interactive Memory Visualizer .... 74
4.4 Valid (Flash, SRAM) Configurations ............................ 89
Tables

3.1 Benchmark Recovery Space, With and Without Failure (entries) . . . 48
3.2 Benchmark Recovery Space, With Failure (bytes) . . . . . . . . . . . 49
3.3 Benchmark Recovery Space, Without Failure (bytes) . . . . . . . . 50
3.4 Car Interval Length (ms) . . . . . . . . . . . . . . . . . . . . . . . . . . 53
3.5 Application Recovery Space, With and Without Failure (entries) . . 54
3.6 Application Recovery Space, No Failure (bytes) . . . . . . . . . . . 55
3.7 Application Recovery Space, With Failure (bytes) . . . . . . . . . . . 56

4.1 Size of Python Libraries (KB) . . . . . . . . . . . . . . . . . . . . . . . 86
4.2 Duplicate Objects in Unpacked Library Code . . . . . . . . . . . . . 87
4.3 Intra-Platform Migration, Sizes (KB) . . . . . . . . . . . . . . . . . . . 92
4.4 Inter-Platform Migration, Sizes (KB) . . . . . . . . . . . . . . . . . . . 94
Chapter 1

Introduction

Embedded systems drive the modern world, controlling devices all around us ranging from toasters to automobiles. These are complex, concurrent systems that interact with their environment by reading sensors, making decisions, and controlling actuators. Yet, despite impressive hardware and sophisticated software requirements, traditional programming environments for microcontrollers are extremely primitive.

To alleviate the burden on the programmer, managed run-time systems are increasingly being adopted within the domain of embedded systems [1, 4, 6]. Already commonplace in traditional computing systems, managed run-time systems massively increase programmer productivity by raising the level of abstraction [43]. In particular, two properties of run-time systems contribute to this increased abstraction. First, run-time systems impose structure on the program and, subsequently, on the organization of memory. Second, run-time systems serve as a common layer in which functionalities can be implemented once for the benefit of all, eliminating the need to re-implement core capabilities in each individual application.

Capitalizing on these properties, run-time systems typically increase productivity through support for high-level languages and automation of complex yet essential tasks such as thread scheduling, garbage collection, and inter-process communications. The advent of embedded run-time systems has primarily been motivated by the same aspirations of programmer productivity that inspired the development of run-time systems for traditional computing systems. Yet, the potential benefits that
run-time systems bring to embedded systems are not limited to productivity. In fact, the same two features of run-time systems that boost productivity – structured memory and support for universal enhancements – provide opportunities to address existing shortcomings of and introduce new functionalities to embedded systems. In particular, this thesis focuses on the application of run-time system techniques to two major challenges of embedded systems programming: reliability and memory optimization.

Unfortunately, existing techniques for large-scale systems cannot simply be transplanted to the domain of embedded systems, as embedded systems have unique characteristics which demand special consideration. For instance, microcontrollers operate under extremely tight resource constraints; a typical mid-range microcontroller may have only 32–256 KB of SRAM and up to 1 MB of flash. Thus, adapting techniques from large-scale systems requires approaching familiar problems through a lens of frugality. Further, embedded applications interface with the real world via hardware peripherals. These peripherals bring external state that is fundamentally different from the internal program state. Internal state is explicitly encapsulated by the program’s memory space; in contrast, external state is created by the system’s interactions with its real-world surroundings, and is typically not explicitly stored within the system. This thesis presents novel mechanisms for leveraging the structure and strength of run-time systems to increase reliability and optimize memory, even amidst the constraints and complexities of embedded systems.

Since embedded systems interact with the real world, failures can have disastrous effects. Microcontrollers often have built-in support for reliability [29, 51], and could implement additional known processor reliability techniques [8, 66, 73]. However, the sensors and actuators at the heart of these systems are extremely susceptible
to failures [12, 30]. Despite this, techniques for fault tolerance in embedded systems largely focus on handling microcontroller failures [36, 76, 80, 82], leaving management of peripheral failures entirely to the programmer.

Handling peripheral failures at the application level is a complex task for the programmer, as peripherals can fail asynchronously. The application would have to correctly restore both the internal and external system state after an arbitrary number of instructions and peripheral accesses have executed. It would be nearly impossible for a programmer to anticipate how to recover in all possible scenarios, and attempting to do so would require an enormous amount of application-level bookkeeping. Such an application would be difficult to write and even harder to test, and might still have problems if peripherals failed at unanticipated and inopportune times.

Therefore, it is essential to develop system-level mechanisms to facilitate recovery from peripheral failures in embedded systems. In many other domains, such as large-scale distributed systems, integrating fault tolerance at the system level is already a common practice [23, 34, 41, 58, 63]. However, these systems typically require the elements that can fail to actively participate in the reliability protocols. Such techniques are ill-suited to handling peripheral failures, as hardware peripherals in embedded systems are incapable of such participation. One system-level approach to fault tolerance that does befit embedded systems is checkpointing. Checkpointing enables the system to roll back to the last known correct state; then, any failures can be corrected, and the program can be re-executed from that point. However, existing checkpointing schemes for embedded systems only handle failures of the microcontroller itself, not peripheral failures [11, 76, 82].

This thesis presents Phoenix, a semi-automated system for recovering from peripheral failures comprised of interlocking compile-time and run-time mechanisms. At
the core of Phoenix is a checkpointing system lightweight enough to operate within a resource-constrained environment, yet comprehensive enough to encompass both the internal and external state. By automatically tracking and restoring all of the system state, Phoenix allows the programmer to focus on the unique aspects of the application.

The second challenge of embedded systems programming that this thesis addresses is memory configuration. The resource constraints that are characteristic of embedded systems demand careful memory management. Moreover, embedded systems are frequently composed of multiple microcontrollers with heterogeneous memory architectures; the amounts and proportions of SRAM and flash on these microcontrollers vary widely. Because memory is so scarce, it is impractical to cater to the lowest common denominator for each type of memory. This combination of resource constraints and heterogeneity makes it difficult to analyze, configure, and optimize memory utilization across the system.

Outside the domain of embedded systems, tools have been built which model memory as a graph in order to analyze the memory of a running program [7, 26, 53, 61, 64, 83]. Built for Java programs, these tools exploit the well-defined memory structure enforced by the JVM run-time system. Constructing such memory graphs for traditional embedded programs written in C is nearly impossible, due to the unstructured layout of memory and the presence of unidentifiable pointers. However, the advent of embedded run-time systems enables the application of graph-based memory analysis to embedded systems. Moreover, the memory configuration challenges faced by embedded systems motivate expanding the use of memory graphs to transform the memory space in order to better utilize scarce memory resources.

This thesis makes that advancement. It presents GEM, a framework that refash-
ions memory as a graph to aid in its transformation. Designed for maximal flexibility, GEM provides a set of basic graph transformations which can serve as building blocks for addressing a myriad of use cases. This versatility uniquely allows GEM to not only facilitate the implementation of many individual use cases, but also to enable the amalgamation of complementary functionalities.

1.1 Contributions

This thesis presents two novel systems which combine compile-time and run-time mechanisms in order to extend the capabilities of embedded systems, all while adhering to stringent resource constraints. First, this thesis introduces lightweight reliability mechanisms based on new insights into embedded systems, introducing and evaluating a semi-automated system for recovering from hardware peripheral failures. Second, this thesis proposes a new approach to memory optimization, presenting a tool that transforms memory graphs in order to optimize the layout and utilization of memory, as well as four use cases of this tool.

1.1.1 Checkpointing and Recovery

This thesis begins by presenting a series of insights into the unique characteristics of embedded systems and their ramifications on reliability. In particular, hardware peripherals introduce external state, which must be addressed by any recovery process for correctness. As part of these insights, this thesis introduces a new taxonomy for classifying hardware peripherals according to the way in which they store state and establishes guidelines for restoring each class of peripheral during recovery.

Second, this thesis outlines a three-step approach for recovering from asynchronous peripheral failures based on these insights. First, the internal state of the system
must be rolled back to the exact point of the failed peripheral access in the program. Second, the peripheral must be recovered, possibly by resetting it or switching in a hot spare. Last, the program must re-execute. To ensure that the external state is properly updated, each peripheral access during re-execution must be handled appropriately based on the peripheral’s own properties as well as its relationships to other peripherals.

Third, this thesis details the design and implementation of Phoenix, a semi-automated peripheral recovery system which utilizes lightweight checkpointing mechanisms to implement this three-step procedure. The checkpointing mechanisms in Phoenix simultaneously track internal and external state, optimize memory utilization, and minimize recovery latency. If a failure occurs, Phoenix automatically rolls back and recovers while keeping both the internal and external state consistent. For efficiency, the system only tracks state when there is a chance of failure. Checkpointing is automatically turned on when a peripheral is accessed, and turned off once all past accesses have succeeded. When checkpointing is enabled the system builds an incremental log of the internal and external state, maintaining pointers into this log corresponding to each peripheral access. This incremental approach serves three purposes. First, it only checkpoints the minimal amount of state required for rollback. Second, it allows for resource reclamation on a rolling basis as peripheral accesses succeed. Third, it enables the system to roll back to the point right before any peripheral access that fails, thereby minimizing re-execution.

Concluding the contributions to reliability, this thesis evaluates the space overheads of Phoenix on a set of microbenchmarks and applications. This evaluation demonstrates that Phoenix is space-efficient enough to operate within the resource constraints of embedded systems. Running on a microcontroller with 96 KB of SRAM,
Phoenix used on the order of 5 KB to track both the internal and external system state of the applications evaluated in this thesis, leaving the majority of the space free for use by the running program. When there was a failure, the overhead rose to 6 KB, as a small amount of additional metadata is needed during the recovery process. Further, in evaluating Phoenix on three realistic applications, this thesis characterizes three types of workloads that are typical of embedded applications. It identifies which types of workloads are amenable to use with Phoenix, revealing that the time overhead of Phoenix is imperceptible in two of the three applications studied.

1.1.2 Memory Optimization and Analysis

Furthermore, this thesis introduces a new approach to addressing the memory optimization challenges faced by embedded systems: reformulating memory as a graph and employing the properties of graphs to transform this memory. It describes the implementation of GEM, an innovative tool which builds, analyzes, and transforms a graph of the entire memory space of a program. Designed to be flexible and extensible, GEM can be employed during compile-time or run-time to achieve a variety of different functionalities.

Through GEM, this thesis presents four key contributions to memory optimization in embedded systems. First, this thesis presents GEM’s infrastructure: a versatile framework for combining low-level graph transformations to achieve high-level use cases. Second, it introduces four novel low-level transformation passes within GEM: splicing, splitting, de-duplicating, and unpacking. Third, this thesis details a mechanism for installing a transformed graph in memory, allowing the abstract graph transformations to impact the actual memory layout of the system. Fourth, it presents the design and evaluation of four high-level use cases: interactive visual-
ization, de-duplication of objects and code, compilation for heterogeneous memory architectures, and transparent migration.

Though the primary contribution of GEM is its flexible framework for applying memory transformations, each individual use case built upon GEM garnered valuable results. First, this thesis demonstrates the ability of GEM’s interactive memory visualizer to facilitate the identification of bugs and inefficiencies within both the system itself and the applications running upon it. The inefficiencies revealed by the visualizer motivated substantial design changes to the system. In particular, inefficiencies in library code storage inspired the exploration of several new library formats. From this exploration and subsequent tradeoff analysis, GEM’s second use case, de-duplication, was born. This thesis presents an overarching evaluation of the four library formats considered, as well as a thorough analysis of the redundancies eliminated by de-duplication and the resultant decrease in the memory footprint of the system.

In the third use case, GEM enables customization of the distribution of code across different regions of memory. This allows the same virtual machine to be programmed on a wide spectrum of memory architectures; the evaluation reveals the breadth of this range. Last, this thesis demonstrates that GEM’s memory graph transformations — in particular, splicing — can be employed to migrate a running program between microcontrollers, including between microcontrollers that have different contents of flash memory and/or are running different platforms.

1.2 Organization

The rest of this thesis proceeds as follows. Chapter 2 presents background and related work. Chapter 3 describes the design, implementation, and evaluation of the
Phoenix recovery system; Chapter 4 likewise presents the design, implementation, and evaluation of the GEM memory configuration tool. Finally, Chapter 5 concludes this thesis, summarizing the contributions and proposing avenues for future work.
Chapter 2

Background and Related Work

Reliability and memory optimization have both been explored by a large body of past work. This chapter provides an overview of this past work, setting the foundation for the work presented in this thesis. First, Section 2.1 surveys past work on reliability, from heavyweight protocols for distributed systems to simpler techniques geared towards embedded systems. Next, Section 2.2 discusses related work on memory analysis and optimization, providing background for the memory graph approach that this thesis extends, as well as for each of the individual use cases enabled via graph transformations. Last, Section 2.3 summarizes the two contributions of this thesis as they relate to this vast body of past work.

2.1 Reliability

There has been considerable past work on fault tolerance in distributed systems. The elements of these systems collaborate to provide a reliable service to external clients. In order to do so, redundancy and consensus are used to detect, mask, and recover from failures [23, 34, 47, 48, 59]. For these techniques to work, each element of the system must be able to actively participate in the reliability protocols and communicate its current status to the other elements of the system.

Peripherals in embedded systems are dedicated to a specific task and are unable to participate in specialized reliability protocols. They are more analogous to the
hardware devices attached to a general-purpose computing system, such as network interfaces or disk drives. Device drivers within the operating system interact with these devices via device-specific protocols, and are supposed to detect and survive device failure. In practice, however, device drivers are one primary cause of operating system failures [20, 67].

Although peripherals play a similar role in embedded systems to that of hardware devices in traditional computing systems, protecting a traditional computing system from device driver failure is fundamentally different from protecting an application from peripheral failure in an embedded system. The operating system has reliability features that enable it to tolerate device failures if the drivers behave properly. So, the focus has been either on hardening device drivers or protecting the interface between the operating system and the device driver [28, 38, 39, 49, 65, 74, 75].

In contrast, resource constraints dictate that embedded systems provide applications with direct access to hardware peripherals via very thin layers of software. Therefore, there is no device driver to harden against hardware failure and there is no operating system to tolerate such failures. Instead, existing techniques for embedded systems leave much of the burden of implementing fault tolerance to the programmer — despite the fact that the difficulty of exhaustively addressing all possible failure scenarios in embedded applications is widely recognized [12, 15, 50, 81]. This results in increased development costs and poor scalability [52].

Efforts have been made to raise the level of abstraction of writing fault-tolerant embedded applications through model- or template-driven development [15, 81]. Similarly, programming language primitives have been introduced to simplify the code that the programmer must write [12]. However, these software-based approaches rely on the programmer to apply the correct constructs in the right places, a non-trivial
task in the face of asynchronous failures. Ideally, fault tolerance would instead be incorporated at the system level, operating transparently to the programmer.

One recovery methodology with significant traction is rollback and re-execution [9, 16, 41, 58, 60, 63, 69]. State-of-the-art rollback relies on checkpoints, or snapshots, of the system state. Typically, checkpoints are either taken periodically by the system or inserted manually by the programmer [9, 16]. While checkpointing mechanisms have been successful in mobile and distributed systems [41, 58, 63], they are not well-suited to embedded systems. Standard checkpointing algorithms force the system to stop execution to snapshot the memory space, potentially interfering with hard and soft real-time deadlines. Further, each snapshot is typically a copy of the application’s entire memory space; in a memory-constrained environment, keeping even a single extra copy of memory is intractable.

This thesis presents a system which adapts rollback to the specific challenges faced by resource-constrained embedded systems. Instead of taking snapshots of the entire program state, Phoenix utilizes a logging technique that more closely resembles journaling filesystems [13, 25, 44, 62] and some hardware transactional memory proposals [56]. Though this adds an overhead to some store instructions, it ensures that the system only copies the subset of memory that has been changed. Additionally, automated disabling of logging significantly reduces the number of stores burdened by this overhead. Prior work has similarly adopted an incremental approach to checkpointing, maintaining a mirror copy of memory which is updated each time main memory is written [24]. Unfortunately, this approach still suffers from a large memory overhead, as two complete copies of the memory space must be maintained. In contrast, Phoenix maintains a journal that encapsulates a copy of only a small fraction of the overall memory space.
Another significant consideration in embedded systems, which are frequently event-driven, is recovery latency. Checkpointing algorithms for mobile and distributed systems have been optimized to minimize the amount of work that is re-executed by selecting only a subset of processes [60, 63], or better yet, threads [18, 41], to re-execute based on dependencies which are either implicitly established via message passing or explicitly annotated by the compiler. The Phoenix system presented in this thesis further narrows the granularity of re-execution: it has the ability roll back to exactly the point before the particular access that failed, minimizing the number of instructions and peripheral accesses to be re-executed.

The most significant drawback of existing checkpointing mechanisms is that they snapshot only the internal program state. In embedded systems, hardware peripherals introduce external state which must also be recovered. This external peripheral state is fundamentally different from the internal program state. Internal state can be restored via rollback and re-execution. In the worst case, the entire program can be re-executed with no adverse effects on the internal state. In fact, one common approach to recovery in embedded systems is to fully reboot the system [22, 54]. However, critical application state is likely to be lost on a reboot.

In contrast, peripherals interact with the real world and may modify the external state in ways that are difficult to undo. Some peripherals have transient effects, whereas others have lasting effects; these differences influence how each peripheral must be handled during recovery. Additionally, peripherals may interact with each other; the effects of one peripheral could depend on the state of another. Such dependencies determine the actions required to restore the external state. While some peripheral accesses should be replayed, in other cases it is incorrect to redo the peripheral access, so it must be skipped.
New work supplementing checkpointing with message logging acknowledged the existence of external state in the form of messages, and attempts to deal with it by skipping all message sends during re-execution [60, 69]. However, this policy leaves no room to adapt to state changes during the second execution. For instance, during re-execution the contents of a message to be sent may change, in which case this new message should be sent and processed rather than re-using the original message. In contrast, the checkpointing mechanisms presented in this thesis explicitly track all of the external peripheral state. During re-execution, each peripheral access is either replayed or skipped based on the properties of the peripheral being accessed and its dependencies on other peripherals, ensuring that the external state is properly updated.

2.2 Memory Visualization and Transformation

The most closely related work to the memory optimization techniques discussed in this thesis falls into two primary categories. First, considerable prior work has utilized memory graphs for program visualization [7, 26, 53, 61, 64, 83]. However, GEM’s memory visualizer differs from this past work due to its targeting a different audience and domain — specifically, it caters to architects of embedded systems. Section 2.2.1 presents a comprehensive discussion of the unique properties of GEM’s visualizer that are motivated by its new focus.

GEM further differentiates itself by extending the use of memory graphs to transform the actual memory layout of the system. GEM is unique in its graph-based approach to transforming memory. Still, each of the transformative capabilities built upon GEM — de-duplication, heterogeneous compilation, and transparent migration — fit into a broad body of related work that accomplishes similar feats via different
means. Section 2.2.2 places GEM’s transformative capabilities into the context of this related work.

### 2.2.1 Visualization

Existing tools for graph-based memory visualization were designed to profile Java applications running on systems with extensive resources; such applications may have over a million live objects on the heap [14]. Thus, these tools abstract away many details in an attempt to make the graph manageable. Some do not include nodes for primitive types [7, 53]; others collapse individual objects of the same type into a single node [64]. In contrast, GEM targets resource-constrained embedded systems, which have much less memory and thus far fewer live objects: on the order of hundreds to thousands. This allows GEM to provide a finer level of detail. It includes a node for every individual object on the heap, while simultaneously providing aggregated views of compound objects.

GEM’s memory visualizer further diverges from past work in audience. The primary audience of existing tools is the application developer. While GEM greatly benefits the application developer, this thesis aims to revolutionize embedded systems programming at the system level. Towards this aim, GEM primarily caters to the system designer, a choice which motivated substantially different design choices. For instance, one existing tool constructs a graph of the heap using logging data which is unavailable at boot-time [61]. While this is suitable for application profiling, it is insufficient for system profiling, as it excludes objects created during start-up that persist unmodified. Another collects the heap by recursively querying GDB from the roots of the memory graph, thereby excluding garbage and unused code [83]. In contrast, GEM profiles both of these, as they highlight opportunities for improving...
the system and toolchain.

Those memory visualizers that do collect a full snapshot of the heap perform the aforementioned abstractions. Sacrificing detail for simplicity is a reasonable choice when presenting a million-object heap to a programmer who has no interest in system-level details. However, this tradeoff conceals information that can lead to valuable system improvements. Furthermore, since the programmer has no control over the layout over the code, past work has uniformly focused on the run-time data, taking a snapshot of only the heap [7, 26, 53, 61, 64, 83]. This thesis broadens the memory graph to include both SRAM and flash, painting a more comprehensive picture of the system.

Additionally, GEM provides a mechanism by which the user can collect a snapshot at any point in execution, without specifying that point in advance. Several tools provide similar flexibility in choosing when to take a snapshot, but at the cost of either slowing execution by continuously logging [26, 61], or excluding unreachable objects [83]. Other tools sacrifice flexibility to avoid such pitfalls, and instead either automatically select points to snapshot based on memory utilization [53], or require that the application developer specify where to snapshot in advance [7].

2.2.2 Transformation

Though GEM’s memory visualizer distinguishes itself from similar tools through its unique domain and audience, the foremost novelty of this work lies in the use of memory graphs to not only inspect memory but transform it. GEM structures memory mutations as graph transformation passes, allowing these transformations to be easily combined and applied in a plethora of use cases. While GEM’s versatile framework for using graph transformations to achieve and combine memory mutations is unique,
the three transformative functionalities built upon this framework are familiar. These functionalities include de-duplication, transparent migration, and heterogeneous compilation; with the aforementioned interactive visualization, they complete the four use cases studied in this thesis. Although these three capabilities draw upon past work, their implementations upon GEM adapt this past work to the particular demands of embedded systems.

Operating systems and storage systems commonly employ de-duplication at run-time [37, 45, 46, 71, 79]. They eliminate duplicates at a coarse granularity, either at the file level or at the block level, based on the observation that large-scale duplication often arises from the storage of multiple versions of the same file. However, programs exhibit different patterns of duplication from storage systems. Large blocks are not redundant; rather, duplication occurs at the granularity of individual objects. More closely related is interning, which de-duplicates a small subset of object types at run-time, and is supported by both Java and CPython [2, 42, 33]. For efficiency, interning typically maintains a table of represented values. The space overhead of maintaining this table is often justified in a large-scale program running on a system with a vast memory space. In embedded systems, the optimal tradeoff is less clear; embedded systems not only have less space to store such a table, but also run smaller applications which have less live objects to take advantage of the benefits offered by such a table. Moreover, run-time interning misses opportunities to consolidate objects within the code. Therefore, GEM uniquely performs de-duplication at compile-time, eliminating the longest lasting redundancies while avoiding the overheads of run-time de-duplication.

Like de-duplication, migration is well-studied. A common approach to migration is checkpointing. Several checkpointing techniques have been developed for embed-
ded systems [24, 76], as discussed in Section 2.1. While these techniques effectively support basic rollback and recovery within the same device, they are not well-suited to GEM’s dual goals of preserving memory and enabling migration. Checkpointing mechanisms for embedded systems generally fall into two categories: more traditional snapshot-based techniques, and space-efficient incremental approaches. The large memory overheads incurred by the former run contrary to GEM’s efforts to save the limited memory space for use by the running program. On the other hand, incremental approaches conserve memory by producing checkpoints that are not complete snapshots of the memory. Thus, while they are sufficient for reliability, they are insufficient alone for migration.

Alternative migration techniques have been developed for mobile agents which operate only at the moment of migration, as opposed to checkpointing continuously during execution [27, 35, 70]. However, these techniques instrument the source code in order to aid with migration, inflating code size. This inflation is unsuitable for resource-constrained microcontrollers. GEM’s migration support involves no changes to the application code; instead, it simply requires the addition of a single bytecode to the run-time system’s interpreter. Furthermore, these techniques support migrating only a single thread or process at a time, whereas GEM migrates the entire run-time state, including all threads and scheduling information.

Other work has similarly expanded the unit of migration, migrating an entire virtual machine [21, 72] or operating system [31]. One advantage of migrating at a lower abstraction layer is that the state is fully encapsulated, allowing for a clean break. In contrast, extracting the state of a single thread or process is challenging, since much of the necessary state resides in lower layers such as the run-time system and operating system. On the other hand, migrating at a lower layer poses its own challenges
— among them, portability in the face of heterogeneity. Within the realm of virtual machine migration, shadow drivers have been used to transparently migrate between platforms with equivalent, but not identical, hardware devices [40]. As explained in Section 2.1, mid-range microcontrollers lack sufficient resources for the extra layer of abstraction that device drivers introduce between the application and the system. GEM instead focuses on the orthogonal task of transparent migration between systems with heterogeneous software and/or heterogeneous memory architectures. To enable migration between devices with different images in flash, GEM performs a series of off-line transformations, requiring no additional resources at run-time and capitalizing on the ease of manipulating a graph representation of memory.

The last of the four use cases built upon GEM, compilation for heterogeneous memory architectures, is a multifaceted challenge that can be tackled from a variety of angles. GEM’s graph transformation approach focuses on coordinating the layout of memory across multiple regions of memory. GEM accepts two parameters at compile-time: upper bounds on the amount of code that can be placed in SRAM and flash. Given these constraints, GEM attempts to partition the code across these two memory regions so as to achieve maximal utilization. Other compilers for embedded systems similarly accept code-size parameters [57]. However, rather than using this information to divide the code between different memory regions as GEM does, these compilers instead carefully craft code that will fit within a single memory space.

No other tool or system simultaneously supports all four use cases of GEM. At best, efforts have been made to combine two: de-duplication and migration. Prior work has integrated de-duplication into the migration process in order to minimize latency [19, 68, 77]. While these techniques greatly speed up the migration process, their impact on memory is fundamentally different from that of GEM. One
such technique requires round-trip migration in order to see any advantages of de-
duplication [77]; the others de-duplicate even during a single migration, but do so at a page granularity, rather than GEM’s finer object granularity [19, 68].

2.3 Summary

This thesis draws upon a large body of past work. Past work on fault tolerance has explored a variety of techniques for incorporating reliability at the system level, including checkpointing. Similarly, past work on memory management has utilized memory graphs to facilitate analysis and visualization. The key contributions of this thesis to these bodies of work stem from its emphasis on tackling the specific challenges faced by embedded systems. In particular, it distinguishes itself by focusing on operating within tight resource constraints. This focus has influenced the development of new checkpointing mechanisms which incur a minimal space overhead, and has likewise driven the expansion of memory graph usage from abstract analysis to actual transformation.
Chapter 3

Checkpointing and Recovery

3.1 Introduction

The development of managed run-time systems for embedded systems has unlocked new opportunities to automate reliability at the system level. Such system-level automation not only eases the task of the programmer, but prevents potentially catastrophic failures. Integrating reliability into the system is a common practice in large-scale systems; however, certain characteristics of embedded systems, including frequent interactions with the real world and stringent resource constraints, preclude the direct application of established reliability techniques to this domain.

This chapter presents Phoenix, a semi-automated peripheral recovery system built around novel checkpointing mechanisms. The design of Phoenix was motivated by a series of insights into the characteristics of embedded systems and their ramifications on recovery. Thus, while Phoenix takes advantage of the presence of a managed run-time system to build upon state-of-the-art system-level reliability techniques, it is uniquely tailored to the challenges faced by embedded systems.

The high-level insights that drove the design of Phoenix are threefold. First, peripherals introduce complexities that must be addressed by any recovery process for correctness. Traditional checkpointing mechanisms only capture the internal state, neglecting the external peripheral state, which must also be recovered. Second, simply taking a snapshot of the memory is infeasible. Embedded systems have a small
amount of memory, the majority of which is needed by the application itself. Last, time constraints motivate minimizing the latency of recovery. There may be multiple peripheral accesses in flight at any given time. Ideally, the system would be able to roll back to exactly the point before the particular access that failed, minimizing the number of instructions and peripheral accesses to be re-executed.

Based on these insights, the checkpointing mechanisms in Phoenix track all of the necessary peripheral state, minimize the space overhead of checkpointing, and enable rollback to any precise point at which a peripheral failure could occur. Upon failure detection, the Phoenix system executes a three-step recovery procedure. First, the internal program state is rolled back to the point of the failed peripheral access. Second, the failed peripheral is recovered to a working state. Third, the system re-executes the program from this point. In this final step, the system automatically decides how to handle all peripheral accesses, ensuring that the external state is restored.

This process requires minimal programmer input. In particular, the programmer need not write any code to restore the internal or external system state in the face of an arbitrary asynchronous failure. Thus, Phoenix enables the programmer to focus on the application, rather than expending significant time on reliability.

The rest of this chapter proceeds as follows. Section 3.2 presents the key insights that motivate the design of the system, and Section 3.3 presents the recovery procedure. Sections 3.4 and 3.5 describe the mechanisms used to implement the system. Next, Section 3.6 shows the experimental evaluation of Phoenix. Finally, Section 3.7 discusses how the Phoenix procedure and mechanisms could be implemented in other systems.
3.2 Key Insights

Two sets of insights into the properties of embedded systems shaped the design of Phoenix. The first set consists of broad insights into the implications of peripherals on system behavior and therefore correct system recovery. Supplementing these, a second set of insights influenced the adaptation of one particular reliability technique — checkpointing — to the domain of embedded systems.

3.2.1 Peripheral State in Embedded Systems

Embedded systems are inherently event-driven, interacting with their surroundings through sensors and actuators. These peripherals have unique characteristics which manifest themselves in subtle and complex ways, and which must be addressed in order to correctly recover from a failure. Five key insights about these hardware peripherals drove the design of the recovery mechanisms presented in this paper.

First, hardware peripherals introduce external state in addition to the internal state. Therefore, to guarantee correct recovery from failures, Phoenix restores both types of state.

Second, the way that peripherals affect external state varies. Peripherals can be classified into four categories based on how they impact state: stateless, ephemeral, persistent, or historical. The first category includes simple sensors such as accelerometers which cannot affect their surroundings. Ephemeral peripherals do affect the external state, albeit fleetingly; these include, for example, buzzers. In contrast, the effects of persistent peripherals are maintained until overwritten. The state of a persistent peripheral is entirely determined by the last write to it. For example, setting the speed of a motor causes a state change that persists until a new speed is set. Historical peripherals likewise have lasting state, but the state of a historical
peripheral is an aggregation of a series of prior writes. The Phoenix system builds in this notion of peripheral classification, selecting the recovery actions to perform for a given peripheral based on the classification it was registered with upon initialization.

Third, peripherals do not operate in isolation; the state of one may impact the behavior of another. Such dependencies are often specific to the context of a particular application. There are two primary types of dependencies between peripherals: behavioral and situational. A peripheral P1 has a behavioral dependency on a peripheral P2 if P2 failing results in P1 not having its intended effect on the external state. As an example, consider an autonomous car that uses a motor and steering servo to drive. The servo will rotate the car’s wheels regardless of whether the motor is functioning. However, if the servo’s goal in this application is turning the car itself, the motor failing will prevent it from accomplishing its goal. Thus, the servo has a behavioral dependency on the motor in this application. Unlike behavioral dependencies, which span, at a minimum, an entire application, situational dependencies exist at a finer grain. Consider again the car, but with a light that turns on when the car reaches a certain speed. At other points in the program the light may be used in ways that are unrelated to the motor. However, this particular write to the light is inaccurate if the motor has failed. Situational dependencies capture such temporary relationships between peripherals. The Phoenix system allows for the specification of both types of dependencies.

Fourth, not all peripheral accesses can be replayed. Consider a historical peripheral such as a message passing interface between two devices. If a message is read from this interface, but an unrelated peripheral fails before the message is processed, re-executing the read is incorrect for two reasons. First, the program must process the original message. Moreover, the state of the message passing interface is not only
external but shared; depending on the higher-level messaging protocol being used, the device on the other end of the connection may not re-send, so a re-read may time out. Instead, this read must be rematerialized: skipped during re-execution, in favor of reusing the original return value. On the other hand, accesses to peripherals that depended on the failed peripheral must be re-executed. When a peripheral fails, the Phoenix system uses the dependency information to populate an initial set of peripherals to redo; accesses to peripherals not in this set are rematerialized. As re-execution proceeds, this set is updated as necessary to adapt to changes in state.

Finally, the lasting effects of persistent peripherals demand additional mechanisms for restoring their state. Correct restoration requires three steps. First, prior to recovering from the failure, all persistent peripherals must be put into a safe state; otherwise, they may continue operating in an erroneous state. For instance, a safe state for a motor could be a speed of zero. Next, all peripherals selected for re-execution must be restored to the state that they held prior to the failed access. That way, when re-execution begins, they will be in the same state that they were at this point in the original execution. Last, once re-execution is complete, all persistent peripherals whose accesses were rematerialized must be set to the last state that was rematerialized, so that they are in the correct state going forward. Phoenix automatically performs each of these three steps, ensuring a consistent state throughout the duration of the recovery process.

3.2.2 Checkpointing in Embedded Systems

Checkpointing is often used to provide fault-tolerance in domains such as mobile and distributed systems [41, 58, 63]. At a high level, the concept of tracking and restoring a known consistent state extends well to other domains. However, embedded systems,
which are both resource-constrained and event-driven by nature, present three unique challenges to adapting existing checkpointing mechanisms.

First, all of the insights introduced in Section 3.2.1 must be considered when designing any recovery mechanism for embedded systems; checkpointing is no exception. In the context of checkpointing, this means that this external peripheral state must be logged and rolled back just like the internal program state. Phoenix therefore logs every peripheral access that is performed. During re-execution, it decides whether to re-execute or skip on the granularity of an individual peripheral access.

Second, embedded systems are burdened by tight memory constraints. For example, a typical ARM Cortex-M3 microcontroller, the Texas Instruments Stellaris LM3S9B92 [78], has only 96 KB of SRAM and 256 KB of flash memory. Phoenix addresses the lack of memory space by using a logging technique that resembles journaling filesystems [13, 25, 44, 62]. Rather than preemptively checkpointing some or all of the memory, Phoenix only copies memory that has actually been changed. Moreover, Phoenix takes advantage of the fact that when there are no outstanding peripheral accesses before a given point in the program’s execution, there is no possibility of rolling back past that point. Thus, Phoenix automatically frees checkpointing metadata incrementally as peripherals are acked, disabling logging entirely when there are no more outstanding accesses. Logging is re-enabled only when another peripheral access is issued.

Finally, embedded systems face time constraints as well. The LM3S9B92 operates at 50 MHz; at this relatively slow frequency, minimizing the latency of rollback is crucial to maintaining performance. This is magnified by the fact that embedded systems are inherently event-driven, and often face deadlines. Phoenix guarantees that when a peripheral fails, there will be sufficient checkpointing data to roll back
to the exact point of failure, avoiding unnecessary re-execution. It achieves this by maintaining one checkpoint corresponding to each outstanding peripheral access. Yet, Phoenix does not require multiple copies of memory to accomplish this. Instead, each checkpoint is identified by pointers into the checkpointing structures, which themselves contain only a small subset of the memory at any given time.

3.3 Recovery Procedure

This section presents a semi-automated recovery procedure for peripheral failures that builds on the insights from Section 3.2. This procedure was implemented and evaluated in Owl \[4, 10\], an embedded run-time system including a Python bytecode interpreter that runs directly on the bare metal. Owl uses a very thin C library to access the peripheral interfaces on the microcontroller. While some implementation details were tailored to Owl, Section 3.7 will show that this procedure can be applied to a wide spectrum of other run-time systems.

When a peripheral access fails, it must be re-executed in order to achieve correct program behavior. However, re-executing this access at the moment that its failure is detected may be insufficient or incorrect, for three reasons. First, many bytecodes may have been executed between the peripheral access and the detection of its failure. These bytecodes could have changed the program state. If the peripheral access interacts with the internal program state at all, such as through its parameters, then in order to re-execute it correctly the state of the program must be restored to the moment at which the access originally occurred. Second, if the hardware failed permanently, naively re-executing the same peripheral access will result in another failure. Thus, the failed peripheral must be recovered prior to re-execution. Last, some of the operations performed after the peripheral access was issued but before
its failure was detected may have depended on the success of the failed access; in this case, those operations must be re-executed as well.

3.3.1 High-Level Steps

Motivated by these facts, the Phoenix system implements a three-step recovery process which automatically executes upon detection of failure. First, the internal state is rolled back to the failed access. Second, the failed peripheral is recovered. Third, the correct external state is reached via redo mode execution. Within redo mode, the system re-executes the failed access, as well as all accesses to dependent peripherals, but rematerializes accesses to unrelated peripherals. Once execution reaches the point of failure detection, the system seamlessly exits redo mode and resumes normal execution.

As an example, consider Figure 3.1, which is sample code for an autonomous car that uses a motor and steering servo to drive and an SD card to record its movements.

```python
# Run the motor
speed = 100
motor.run(speed)
SD.write("set motor to 100")
speed += 100

# Turn the wheels
servo.set_servo(LEFT)
...
```

Figure 3.1 : Sample Application Code
Assume that the write to the motor in line 3 fails, and the system receives notification of this failure after executing line 8.

Upon detecting this failure, the system will first put the motor and servo into safe states. It will then restore the internal state as it was immediately prior to line 3, resetting the value of the variable `speed` to 100. The system will recover the motor by invoking a programmer-provided recovery function, and then will enter redo mode. During redo mode, line 3 will be re-executed, since it failed initially. However, line 4 will be rematerialized, as there is no dependency between the motor and the SD card. This rematerialization is a requirement for correctness. Effectively, redo mode restores the peripheral state that would have been reached had there been no failure at all. It would be inaccurate to replay this write and record the motor as having been set to 100 twice, when in fact this write was only successfully executed once. Line 5 will be re-executed, because it affects the internal state, which was rolled back. Last, the servo depends on the motor, so line 8 will also be re-executed. Finally, normal execution will resume.

### 3.3.2 Low-Level Steps

The Phoenix toolchain and system collaborate to achieve the behavior described in Section 3.3.1. The procedures they follow are driven by the fact that checkpointing must be performed during execution in order to enable rollback and correct re-execution.

During compilation, the toolchain takes the following steps to enable the run-time checkpointing system:

A1. Parse the application to collect peripheral metadata.

A2. Inject code to enable checkpointing prior to each peripheral access.
A3. Tag each peripheral access function.

A4. Auto-generate code to log the arguments and return value of each peripheral access.

A5. Build structures in the run-time system encoding the peripheral metadata.

Then, when a failure is detected at run-time, the system automatically performs the following steps:

B1. Identify which peripheral failed.

B2. Throw a rollback exception to the interpreter.

B3. Determine the rollback point based on the identity of the failed peripheral.

B4. Put all peripherals in a safe state.

B5. Restore the internal state via rollback.

B6. Recover the failed peripheral.

B7. Enter redo mode and restore the external state.

B8. Exit redo mode and resume normal execution.

Together, these procedures restore a consistent internal and external state. Each step is achieved through a combination of interlocking mechanisms, which work together to address the complexities described in Sections 3.2 and 3.3.1. Sections 3.4–3.5 present these mechanisms and demonstrate how they fit into the context of these two overarching procedures.

### 3.4 System Mechanisms

The procedures presented in Section 3.3.2 are implemented by mechanisms built into the compiler and run-time system. None of these mechanisms alone are sufficient
for recovery. But, taken as a whole they form a complete recovery system which facilitates every step from failure detection to rollback, recovery, and re-execution. Figure 3.2 presents an overview of the Phoenix system architecture, illustrating how the various components of the Phoenix system, in white, fit into the Owl system, in grey.

### 3.4.1 Failure Detection

To enable peripheral access, microcontrollers offer a wide variety of external interfaces, which notify the processor of events via interrupts. The LM3S9B92 includes universal asynchronous receiver/transmitters (UART), serial peripheral interfaces (SPI), inter-integrated circuit interfaces (I2C), an Ethernet interface, and pulse-width modulators (PWM).

Phoenix provides mechanisms to interact with these external interfaces in order to detect failure and identify the failed peripheral (Step B1). These mechanisms set the
rest of the procedure in motion by throwing an exception to the interpreter (Step B2). There are two system mechanisms involved in this process: interrupt handlers and lightweight software shims for external interfaces that do not provide the appropriate type of interrupts.

As these mechanisms are the link between the peripherals and the rest of the Phoenix system, Phoenix automatically recovers from all failures that can be detected by these mechanisms. These primarily consist of communication failures. In particular, Phoenix provides fault tolerance for fail-stop peripheral failures, violations in the communications protocols, and interrupt storms or other spurious interrupts. Such failures are primarily detected via interrupt handlers. However, the software detection mechanisms described in Section 3.4.1 are used to provide fail-stop fault tolerance in cases where the interrupt handlers do not suffice. Furthermore, these same software detection mechanisms could be augmented to perform additional validation that the peripheral is operating correctly — thereby extending fault coverage beyond communication failures — without requiring any changes to the rest of the system.

**Hardware Detection**

The interrupt handlers in Phoenix manage both successful and unsuccessful peripheral accesses. Each interrupt handler begins with Step B1: pinpointing the peripheral whose access generated the interrupt. During initialization (see Section 3.5.2), peripherals are registered with an (interface, interface details) pair. The required details for each interface satisfy two properties: they uniquely identify a single hardware peripheral, and the interrupt handler has access to the details for the peripheral that generated the interrupt. Thus, the interrupt handler simply searches the set of regis-
tered peripherals for a match against the known (interface, interface details) pair. As an example, for an I2C device this pair would look like (I2C, (master address, slave address)). While Phoenix allows for optimistic execution of arbitrary Python code as well as accesses to other peripherals, it waits between successive writes to the same peripheral, ensuring that the interrupts can always be traced back to a particular peripheral access.

Next, the interrupt handler checks the status flags to determine whether the access succeeded. If the access succeeded, the interrupt handler acks it by decrementing the appropriate counter of outstanding reads or writes for this peripheral. If it failed, the interrupt handler throws a rollback exception to the interpreter (Step B2).

At a high level, this is accomplished via an exception mechanism that uses `longjmp` to return control to the interpreter, passing an error code as the return value; the interpreter then checks this error code and enters a special block that performs Steps B3–B8. However, it is not safe to `longjmp` from within an interrupt handler. Therefore, if the interrupt handler detects an error, it instead reaches into the stack and modifies the program counter that was stored on the stack by the hardware upon entering the interrupt handler. It updates this program counter to point to a trampoline function which executes a `longjmp` to the jump buffer of the currently-executing native frame. It also modifies r0, the first argument to this trampoline function, to contain the error code. Thus, on exiting the interrupt handler, control is given to the trampoline function, which then jumps to the interpreter.

**Software Detection**

Not all peripheral interfaces provide a convenient interrupt mechanism to detect success and failure. For example, the interrupts for the UART interface on the LM3S9B92
are designed to allow asynchronous transmission and reception of data. Interrupts correspond to whether or not the device is ready for a subsequent transfer, not whether previous transfers have succeeded or failed.

To address this limitation, the system was augmented with a small (64 entry) buffer to hold UART data. Each entry is two bytes, consisting of one byte that was received by the UART and associated error bits. As the hardware receives data, the interrupt handler transfers the data, and any error bits, to this software buffer. UART reads made by the application then access the software buffer. When a read occurs on a byte that was successfully transferred, the peripheral access is acked, as if there had been a “success” interrupt. When a read occurs on a byte that experienced a transmission error, the recovery procedure is triggered as if a “failure” interrupt had occurred.

This is a general software mechanism that can be used for any peripheral for which the peripheral interface does not provide interrupts that enable the system to easily determine the success or failure of each access.

3.4.2 Checkpointing

To enable rollback and re-execution, the Phoenix system performs checkpointing whenever there is a chance of peripheral failure. Checkpointing is automatically enabled prior to each peripheral access. Once all past accesses have succeeded, there is no possibility of rolling back past the current point. In this case, the system turns checkpointing off until the next peripheral access is issued. While checkpointing is enabled, Phoenix maintains three structures: a journal, a control flow queue, and a set of rematerialization queues. These structures are kept on a second heap which is not itself checkpointed, allowing them to persist past rollback. They are freed on a
rolling basis as accesses succeed. Once all past accesses have succeeded, rollback past the current point is impossible, and so Phoenix turns checkpointing off until the next access is issued.

Software Journal

In Owl, all of the program state is stored on the Python heap. So, when checkpointing is enabled, the history of this heap must be tracked to enable Step B5, rollback of the internal program state. When an assembly language store instruction to the heap occurs, Phoenix first records the memory address to be stored and its current contents within a journal. This journal only needs space proportional to the number of stores to the heap since checkpointing was last enabled, rather than the entire size of the heap. Upon failure, the run-time system can roll back the state by replaying the journal backwards. For each entry, the value in the journal is written to the corresponding address.

Microcontrollers do not include conventional memory management units (MMUs), so the prototype implementation of Phoenix keeps the journal using software that exploits existing hardware mechanisms defined by the ARM ISA. In particular, Phoenix uses the ARM Cortex-M3’s memory protection unit (MPU), flash remap hardware, and system call instruction (svc) to keep the journal. These are the only hardware requirements of Phoenix.

The journaling process proceeds as follows. First, the Python heap must be protected by the MPU. The MPU supports dividing the memory into multiple regions, and each region’s attributes and access permissions can be set independently. When checkpointing is enabled, the regions of memory covering the heap are set to be read-only, so that writing to the heap will cause a memory management fault. The memory
management fault handler receives both the address of the instruction that caused the fault and the memory address of the faulting access. Based on the type of instruction (ARM has “store multiple” instructions), it determines how many journal entries to populate. Each entry contains a faulting address and the contents of that address.

If adding an entry fills the journal, logging cannot continue. Therefore, an exception is thrown to the system telling it to wait for all prior peripheral accesses to succeed. Once all accesses have succeeded, the system clears the journal and disables logging. Execution can then resume.

After the store instruction completes, memory protection must be turned back on. So, before returning to the program, the fault handler remaps the instruction that follows the store to become an svc system call instruction. This ensures that only the store instruction will execute and then a system call will be made immediately afterwards. Note, however, that the program is executing from flash memory, so the instruction cannot be overwritten. Instead, the Cortex-M3 flash patch and breakpoint support is used. Small regions of flash can be “patched” by remapping them to SRAM. When the system call executes, the memory region is turned back on in the MPU. The flash remapping is turned off, and the return from the svc is routed back to the instruction following the store. Execution then continues as usual.

To better understand this process, consider the assembly code snippet in Figure 3.3. Assume that memory protection is currently on. As the microcontroller executes the movw and ldr instructions at addresses 0x9340 and 0x9344, respectively, everything will proceed as normal. Then, the str instruction at address 0x9346 will cause a memory management fault. The address of the str instruction (0x9346) and the memory address causing the fault (sp + 0) will be available to the handler. It will read the value pointed to by sp and store that value and its address in the
Address 0x9348 will then be remapped to a svc instruction and memory protection will be turned off. The fault handler will then return to address 0x9346. The str instruction will execute, followed by the remapped svc instruction at address 0x9348. The svc handler will restore the memory protection, disable the flash remapping, and return to the real 0x9348 so that the ldr instruction can execute.

The software journal is effective, but it incurs an overhead for each protected store. As two exception handlers must run for each protected store to the heap, there is a minimum overhead of 48 cycles (12 cycles to enter and 12 cycles to exit each exception handler), plus any pipeline flushes. In practice, the handlers themselves execute for hundreds of additional cycles. As will be discussed in Section 3.6, the overhead per protected store averages around 308 cycles.

**Hardware Journal**

For systems where the software journal overhead is too high, the microcontroller could be augmented with a simple hardware journal. This would involve minimal changes to the ARM ISA. It would require MPU support for an additional “log” mode, and a circular buffer with entries that can hold a memory location and a 32-bit value to serve as the journal. An extra pair of registers would be needed to store head and
tail pointers into the journal. The entire journal would need to be memory mapped such that the microcontroller can read or write the head/tail pointers and any of the entries. Software could then initialize the journal by setting the head and tail pointers appropriately and by setting the log attribute in the protection bits for the Python heap.

During execution, the hardware journal would operate as follows. When a store instruction references a memory address within a region whose log attribute is set, the hardware first stores the address and contents at that address in the journal, incrementing the tail pointer modulo the size of the journal. The original store then completes. If the tail pointer now equals the head pointer, the journal is full, and an interrupt must be generated notifying the run-time system of this fact and prompting a wait, as described in Section 3.4.2.

Hardware journaling would increase the latency of a store instruction by two cycles. If the store accesses a region that has the log attribute set, the microcontroller would have to abort the actual write to the SRAM. It would then read the contents of the SRAM at the address, which would take a single cycle. Then, the actual write to the SRAM could occur in parallel with the write to the journal, taking another cycle. This is two orders of magnitude faster than the software journal, which would yield perceptible improvements in applications in which the journal is frequently active.

Rematerialization Queues

The rematerialization queues serve to checkpoint the external state. Specifically, they fulfill two purposes: keeping track of the precise point to which to roll back should an access fail (Step B3), and storing the arguments and return value of each access for use during redo mode (Step B7). Each peripheral has a queue, with one entry per
access.

On failure, the system selects the right checkpoint based on the identity of the failed peripheral, as determined by the detection mechanisms from Section 3.4.1. Each peripheral access corresponds to a checkpoint, and each peripheral access has its own rematerialization queue entry. Therefore, Phoenix stores all of the metadata for a given checkpoint within the corresponding queue entry. This includes indices into the journal and control flow queue, as well as pointers into the other rematerialization queues.

After rollback, the system automatically constructs a set of peripherals to redo based on programmer-specified dependencies. Each time a peripheral access function call is reached during redo mode, Phoenix checks to see if the peripheral is in the redo set. If so, the function is called as usual. However, if it is not in the redo set, Phoenix compares the new arguments against those in the rematerialization queue entry. If they match, the function call is skipped; the system pops the new arguments from the stack and pushes the old return value. If they do not match, this peripheral is added to the redo set, and then the function call is executed anew.

A given rematerialization queue entry is freed once the access it corresponds to, plus all accesses that were issued earlier, are acked. This is because once all of these have been acked, the system is guaranteed to never roll back to (or past) this entry. The contents of the journal up to the index stored within this rematerialization queue entry are freed at the same time.

Control Flow Queue

When checkpointing is enabled, the control flow queue keeps track of the instruction pointer of each Python bytecode that is executed. During redo mode, this queue is
used to determine when to resume normal execution (Step B8). Redo mode is exited either when the point at which failure was detected is reached once more, or when control flow diverges from the original path. For instance, a re-executed peripheral read may return a different value the second time through and cause a different path to be taken at a branch. In such a case, exiting redo mode early allows the application to adapt to changes in the external state that have occurred since the initial execution.

3.4.3 Compilation and Code Generation

During compilation, the Phoenix toolchain performs Steps A1–A5 in order to provide the system with all of the information it needs to perform Steps B1–B8 at run-time.

The bulk of these compile-time transformations are applied to the application code, with two primary goals. When the run-time system detects a peripheral failure, it must select the precise instruction to which to roll back so as to incur minimum latency. Then, when execution reaches a function call during redo mode, the system must check whether this function accesses a peripheral; if so, the system must use the corresponding rematerialization queue entry to properly handle the function call. The Phoenix toolchain transforms the application in three ways to equip the system to make these decisions.

First, to enable rollback, Phoenix records the journal and control flow queue indices at each checkpoint. Checkpoints correspond to peripheral access function calls — in Python, `CALL_FUNCTION` bytecodes. However, it is insufficient to re-execute a failed function call with the same parameters, as they may have changed during recovery. For instance, an access function for an I2C device may take the master address as a parameter. If this address was originally passed as a variable, and this variable was changed by the recovery function, the new value must be loaded prior
to re-execution. As such, the true checkpoint is not the CALL_FUNCTION bytecode but the bytecode that loads the first parameter for this call. The exact locations of these checkpoints are difficult to identify at run-time without compiler support. Thus, Phoenix extends Python with a new bytecode, JOURNAL_STORE, and adds a custom AST rewriter to the Python compiler which inserts a JOURNAL_STORE bytecode prior to loading the parameters for each peripheral access (Step A2). On encountering this bytecode during run-time, the interpreter stores the current journal and control flow indices and enables checkpointing.

Second, Phoenix augments each code object with a peripheral action flag (Step A3). Code objects for peripheral accesses are tagged as PA_READ or PA_WRITE; all others are tagged as PA_NONE. When the run-time system reaches a CALL_FUNCTION bytecode during redo mode, it checks this flag. If the flag is not PA_NONE, the system applies the methodology from Section 3.4.2 to decide whether or not to rematerialize. Otherwise, it replays the function call.

Third, the Owl toolchain includes an autowrapping tool which makes native C functions callable from Python by generating wrapper functions that convert the arguments from Python types to C types and back [10]. Phoenix modifies this tool to insert recovery support code into the wrappers of the peripheral access functions (Step A4). Prior to a peripheral access, the autowrapper adds code to create a rematerialization queue entry. After the call, it inserts code to increment the number of outstanding reads or writes (as appropriate) to this peripheral and set the new rematerialization queue entry’s return value. Allocating and initializing the queue entries enables rematerialization, should a peripheral access fail; tracking the outstanding accesses allows the system to automatically free the checkpointing structures and disable checkpointing as accesses succeed.
In addition to making these three changes to the application code, the Phoenix toolchain auto-generates code in the run-time system itself to build structures into the system containing peripheral metadata (Step A5), which was collected during parsing (Step A1). In particular, it encodes the programmer-provided classification and behavioral dependencies of each peripheral, which the run-time system uses to correctly restore the peripheral state post-failure.

3.5 Programmer Mechanisms

To utilize Phoenix’s recovery capabilities, the programmer must follow a few rules during development. At a high level, a recoverable application must consist of three pieces: the peripheral code, a config file, and the application code.

3.5.1 Peripheral Classification

Each peripheral must extend one of four provided peripheral classes, which mirror the categories described in Section 3.2.1: StatelessPeripheral, EphemeralPeripheral, PersistentPeripheral, and HistoricalPeripheral. The programmer must define access functions to allow the peripheral to interact with its environment. These must be written in C, so that the autowrapper can inject code to support checkpointing at run-time, as described in Section 3.4.3; this is the only C code that the programmer must write. Additionally, these should encapsulate an atomic unit of work, since rollback and re-execution occur at the granularity of a single call to a peripheral access function.

Further, each peripheral class must support two operations: recovery and restoration. For the former, the programmer must define a recover function, which the system will automatically call after rollback (Step B6). This gives the programmer
class Motor(PersistentPeripheral):
    def __init__(self):
        # Initialize primary device
        self.init(PRIMARY)

    def recover(self):
        # Switch to backup device
        self.init(BACKUP)

    def safe_state(self):
        # Stop the motor
        self.set_speed(0)

    def last_state(self, *args):
        native_write(*args)

Figure 3.4: Peripheral Class Excerpt

complete flexibility in determining how to handle a failed peripheral, such as by resetting it, switching in a hot spare, or even signaling for user intervention. Yet, the programmer need never worry about the complex bookkeeping and control flow logic to determine when to call this function. Furthermore, all applications utilizing the same peripheral can reuse the same peripheral class.

Restoration is more subtle; it involves placing peripherals in appropriate states for rollback and re-execution to proceed, and what this entails varies by classification. Since stateless and ephemeral peripherals hold no lasting state, restoration is a no-op for them. However, for persistent peripherals the programmer must define two restoration functions, motivated by the final insight in Section 3.2.1: safe state
and `last_state`. During Step B4, Phoenix automatically invokes all `safe_state` functions to put the peripherals into states that will have minimal effects during subsequent recovery steps. For example, a motor might be set to a speed of zero. As with `recover`, the definition of `safe_state` is up to the programmer.

In contrast, `last_state` follows the same template for all peripherals: it takes a variable number of arguments and performs a write with these arguments. This allows the system to invoke `last_state` by simply passing the arguments extracted from a rematerialization queue entry, regardless of the write function’s signature. During redo mode (Step B7), the system calls `last_state` in two places. Peripherals whose accesses must be replayed are set to their last state at the beginning of redo mode; peripherals whose accesses were not replayed during redo mode are set to their last state prior to resuming normal execution. An example of recovery and restoration code for a persistent peripheral is shown in Figure 3.4.

Restoring a historical peripheral is challenging, as its state is an aggregation of multiple past writes. The naive solution is to re-execute every past write. However, this would prevent freeing the rematerialization queue entries, which is untenable given limited memory; developing a better policy is an area of future work. While Phoenix does not include specialized mechanisms for historical peripherals, this does not preclude its use with all applications involving historical peripherals. Consider a TFT display. Many applications will update a display by completely redrawing its contents, which would work seamlessly with Phoenix. More generally, a finite number of pixels comprise the display’s state. Their values can be stored in a buffer program-matically and restored by `recover`, requiring no extra support from Phoenix. This approach effectively allows Phoenix to treat a historical peripheral like a persistent peripheral — albeit one with a much larger amount of state.
3.5.2 Peripheral Initialization

To enable Step B1, failure identification, the peripheral class must do two things upon initialization: enable interrupts and register the new instance. First, interrupts must be enabled prior to any accesses to the peripheral, as they are used to ack and detect failure. Second, Phoenix provides a peripheral_register function which must be invoked with the interface and interface details for the peripheral, as described in Section 3.4.1. Registration must likewise occur prior to any accesses, as the interface information is used by the interrupt handlers to identify the peripheral that generated the interrupt. However, registration and interrupt information can be updated at any time to reflect changes to the hardware configuration. In particular, if the recovery function switches to a backup device, it should also re-register the peripheral.

3.5.3 Config File

For each application, the programmer must provide a config file with metadata about the peripherals used in that application: specifically, the behavioral dependencies, and the number of interrupts that the peripherals generate. Because the system treats peripheral access functions as atomic units, the programmer must specify the number of interrupts generated per call as a C expression. This expression is used by the autowrapper (Section 3.4.3) to track how many interrupts must occur before a peripheral access is deemed successful. Writing the config file proved trivial for the three applications studied in this thesis.

3.5.4 Application Code

Recoverable application code is nearly identical to unrecoverable application code, with just two small differences. First, recoverable applications may only access hard-
ware peripherals through peripheral classes which satisfy the requirements laid out in Section 3.5.1. Second, the programmer may specify situational dependencies between peripherals. Phoenix provides a wrapper for this which takes two parameters: the dependent function call, and a tuple of peripheral instances upon which this function call depends. For the light and motor example, this might look like:

\[ x = \text{dependency(light.on(), (motor,))} \]

This sets the situational dependencies of all peripheral accesses within light.on(), executes light.on(), and assigns the return value of light.on() to x.

### 3.6 Evaluation

This section will first introduce the benchmarks and applications used to evaluate the Phoenix system. It will then assess Phoenix, showing that its space overhead is minimal and its time overhead is completely hidden in realistic workloads.

#### 3.6.1 Microbenchmarks

The benchmarks use a gyroscope and compass as representative peripherals, and follow a common structure. Each one is named in the form \(<\text{peripherals}>, <\text{actions}>\), where \(<\text{peripherals}/callback\) is a subset of \{gyro, compass\} and \(<\text{actions}/callback\) is a subset of \{r, w, c\}, for read, write, and compute. After the peripherals are initialized, the actions are performed in a loop in the order in which they are named. For benchmarks with two peripherals, each action is done for both peripherals before moving on to the next action.

Even in the absence of a failure, checkpointing necessarily incurs a time overhead. This is primarily due to the cost of journaling. Across the benchmarks listed in Ta-
ble 3.1, the weighted average cost of adding an entry to the software journal was 6.2µs. Given a 20ns cycle length, this indicates that one journaled store takes, on average, 308 cycles. A hardware journal, requiring only 2 cycles per store, would dramatically decrease the journaling overhead. A single failure incurred a relatively small additional overhead of 12–143ms. The overhead did not exceed 45ms for benchmarks with a single peripheral; for gyro_compass_wr, it is larger (143ms) due to additional safe.state and last.state calls.

This overhead is incurred in support of speculative execution. Another approach would be to wait for each access to be acked before proceeding. However, the length of the control flow queue reveals that Phoenix takes advantage of significant opportunities to make progress where a stop-and-wait system would not. As shown in Tables 3.1 and 3.5, the control flow queue reached lengths of 9–18 during the period in which a wait-based system would be stalling. This is non-trivial; a single Python bytecode may, for example, invoke a native function call.

Just as the checkpointing process takes time, the structures storing the internal and external state require space. There are three main checkpointing structures: the journal (JNL), control flow queue (CFQ), and rematerialization queues (RMQ). A single journal entry requires 6 bytes, as the software journal is optimized to store an offset into the 64 KB heap rather than the full address. A control flow queue entry requires 4 bytes, and a rematerialization queue entry requires a minimum of \((36 + (72 \times n))\) bytes, where \(n\) is the number of hardware peripherals. The rematerialization queues may consume extra space if the peripheral access calls take arguments or if there are situational dependencies. Owl's best-fit memory allocator also introduces a small amount of variance in the size of the rematerialization queues.

A long-running program may generate many entries within these checkpointing
structures over the course of its execution. However, since past entries are discarded as peripheral accesses are acked, only a small number of entries are live at any given point in time. Table 3.1 shows the maximum number of live entries for each of the three checkpointing structures in each of the benchmarks, both with and without a peripheral failure. None of the benchmarks required more than 301 journal entries, 14 control flow queue entries, or 3 rematerialization queue entries per peripheral.

Tables 3.2 and 3.3 show the total space overhead when the benchmarks are run with no failures and with a single failure. The primary source of this overhead is the checkpointing structures. Phoenix uses a fixed-size journal of 512 entries, and a fixed-size control flow queue of 64 entries. As seen in Table 3.1, these sizes proved more

---

**Table 3.1: Benchmark Recovery Space, With and Without Failure (entries)**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JNL Max Live</th>
<th>CFQ Max Live</th>
<th>RMQ Max Live</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Failure</td>
<td>Failure</td>
<td>No Failure</td>
</tr>
<tr>
<td>compass_r</td>
<td>144</td>
<td>169</td>
<td>6</td>
</tr>
<tr>
<td>compass_w</td>
<td>192</td>
<td>226</td>
<td>9</td>
</tr>
<tr>
<td>compass_wr</td>
<td>192</td>
<td>211</td>
<td>9</td>
</tr>
<tr>
<td>compass_wcr</td>
<td>190</td>
<td>219</td>
<td>9</td>
</tr>
<tr>
<td>compass_wrc</td>
<td>190</td>
<td>211</td>
<td>9</td>
</tr>
<tr>
<td>compass_wcrc</td>
<td>192</td>
<td>211</td>
<td>9</td>
</tr>
<tr>
<td>gyro_r</td>
<td>220</td>
<td>220</td>
<td>9</td>
</tr>
<tr>
<td>gyro_w</td>
<td>165</td>
<td>182</td>
<td>7</td>
</tr>
<tr>
<td>gyro_wr</td>
<td>220</td>
<td>220</td>
<td>9</td>
</tr>
<tr>
<td>gyro_compass_wr</td>
<td>220</td>
<td>301</td>
<td>9</td>
</tr>
</tbody>
</table>
Table 3.2: Benchmark Recovery Space, With Failure (bytes)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>compass_r</th>
<th>compass_w</th>
<th>compass_wr</th>
<th>compass_wcrc</th>
<th>gyro_r</th>
<th>gyro_w</th>
<th>gyro_wr</th>
<th>gyro_compass_wr</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNL</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
</tr>
<tr>
<td>CFQ</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
</tr>
<tr>
<td>RMQ</td>
<td>664</td>
<td>656</td>
<td>656</td>
<td>656</td>
<td>656</td>
<td>656</td>
<td>656</td>
<td>984</td>
</tr>
<tr>
<td>Peripheral Metadata</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>384</td>
</tr>
<tr>
<td>Recovery Metadata</td>
<td>304</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>304</td>
<td>304</td>
<td>304</td>
<td>300</td>
</tr>
<tr>
<td>Total</td>
<td>4436</td>
<td>4636</td>
<td>4636</td>
<td>4636</td>
<td>4632</td>
<td>4632</td>
<td>4632</td>
<td>5072</td>
</tr>
</tbody>
</table>
Table 3.3: Benchmark Recovery Space, Without Failure (bytes)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>compass_r</th>
<th>compass_w</th>
<th>compass_wr</th>
<th>compass_wrc</th>
<th>gyro_r</th>
<th>gyro_w</th>
<th>gyro_wr</th>
<th>gyro_compass_wr</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNL</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
</tr>
<tr>
<td>CFQ</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
<td>284</td>
</tr>
<tr>
<td>RMQ</td>
<td>276</td>
<td>472</td>
<td>472</td>
<td>472</td>
<td>484</td>
<td>484</td>
<td>484</td>
<td>752</td>
</tr>
<tr>
<td>Peripheral Metadata</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>268</td>
<td>384</td>
</tr>
<tr>
<td>Recovery Metadata</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>3972</td>
<td>4168</td>
<td>4168</td>
<td>4168</td>
<td>4180</td>
<td>4180</td>
<td>4180</td>
<td>4564</td>
</tr>
</tbody>
</table>
than sufficient, never exceeding 58.8% or 21.9% capacity, respectively. Section 3.6.2 will show that these sizes also fit the applications comfortably. Since rematerialization queue entries are both larger and fewer than the journal and control flow queue entries, the rematerialization queues are implemented as linked lists.

In addition to the checkpointing structures, Phoenix maintains a small amount of metadata. Peripheral metadata keeps track of the active peripherals in the system, including their interface details, and thus grows with the number of peripherals. Some of this metadata is generated at boot-time based on the configuration of the run-time system; the rest is created when a peripheral is initialized at run-time. The same build of the run-time system was used for all benchmarks; thus, there is two peripherals’ worth of boot-time metadata in all of the benchmarks, and the peripheral metadata does not double in size on adding a second active peripheral.

Recovery metadata is populated upon a failure with the information needed to perform rollback and re-execution, including which peripheral failed and which peripherals to redo. The size of this metadata is initially negligible, and grows proportional to the number of failures. Even considering both types of metadata, the metadata space overhead is insignificant. In the benchmarks without failure, it never amounted to more than 408 B (gyro_compass_wr); with failure, it reached a maximum of 684 B (gyro_compass_wr).

Furthermore, the overall space overhead of Phoenix was relatively consistent across all benchmarks. With no failures, it began at 3.9 KB for the case where a single peripheral is being read but not written (compass_r), as this minimizes the number of rematerialization queue entries live at any given point in time. Introducing writes (compass_w, compass_wr, compass_wcrc) added an additional 196 B, as a second live rematerialization queue entry is required to support last_state. The gyro bench-
marks do not show this same increase upon adding writes, as the gyro requires a single write during configuration, and therefore even gyro_r holds a second rematerialization queue entry. The largest increase occurs when a second active peripheral is introduced, as in gyro_compass_wr. This is due to two factors: extra rematerialization queue entries and extra peripheral metadata.

These space requirements do not change drastically in the case of failure; only the rematerialization queue entries and the recovery metadata consume additional space. The expansion of the rematerialization queues is due to the fact that the system must maintain segments of the queues from the original execution in order to replay accesses. At the same time, replaying these accesses results in additional entries being generated. Thus, during redo mode there are brief periods in which two rematerialization queue entries are live for the same peripheral access.

Overall, the space overhead of the Phoenix system is quite small. Across all benchmarks, the total heap space consumed by the recovery data structures never exceeded 4.5 KB when no failure occurred, nor did it exceed 5.0 KB when a failure occurred. Using a mere 4.0–5.2% of SRAM, Phoenix leaves most of the space available for the running program while still maintaining multiple simultaneous checkpoints — all of which are positioned at precisely the latest possible location to which rollback could occur in order to recover.

3.6.2 Applications

The Phoenix system was evaluated on three applications, each of which is representative of a different pattern of peripheral accesses. The first application is an autonomous RC car whose electronics were replaced with an LM3S9B92. The microcontroller is attached to three types of peripherals: a motor, a steering servo, and
two gyroscopes. An event loop controls the car’s movements by periodically reading from the gyro and writing to the motor and steering servo. The second application is an obstacle tracker which periodically logs the distance to the nearest obstacle by reading from one of two range finders and writing to a TFT display. The final application uses two types of peripherals, two compasses and a TFT display, to draw a virtual compass pointing towards magnetic North. The range finder is a stateless peripherals; the compass, gyro, steering servo, and motor are persistent peripherals; and the TFT display is a historical peripheral. Each application was evaluated for ten seconds.

The autonomous car code uses a proportional-integral control loop to query the gyro and steer. It attempts to hit a specific period (30ms) between updates, where an update consists of reading the gyro and setting the steering servo. Table 3.4 shows the minimum, maximum, and average interval lengths, which were identical across all three configurations — Owl with no Phoenix, Phoenix with no failure, and Phoenix with a single failure. On average, the car hit its soft real-time deadline of 30ms. Yet, in all three cases, the maximum was 44ms. Clearly, this delay was not due to Phoenix. Rather, it was due to ill-timed runs of Owl’s mark-and-sweep garbage collector.

The obstacle tracker reads the range finder and displays the distance to the nearest obstacle. It does not aim for a set period between updates. Instead, it sleeps for a

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Phoenix</td>
<td>30</td>
<td>44</td>
<td>30</td>
</tr>
<tr>
<td>Phoenix (No Failure)</td>
<td>30</td>
<td>44</td>
<td>30</td>
</tr>
<tr>
<td>Phoenix (With Failure)</td>
<td>30</td>
<td>44</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 3.4 : Car Interval Length (ms)
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>JNL Max Live</th>
<th>CFQ Max Live</th>
<th>RMQ Max Live</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Failure</td>
<td>Failure</td>
<td>No Failure</td>
</tr>
<tr>
<td>autonomous car</td>
<td>220</td>
<td>220</td>
<td>9</td>
</tr>
<tr>
<td>obstacle tracker</td>
<td>346</td>
<td>346</td>
<td>18</td>
</tr>
<tr>
<td>virtual compass</td>
<td>346</td>
<td>346</td>
<td>18</td>
</tr>
</tbody>
</table>
fixed amount of time (one second) between iterations. As a result, sleep dominates the workload, and all three configurations completed the same number of iterations in the allotted time.

In contrast to the other applications, the virtual compass is peripheral access-intensive. It attempts to update as frequently as possible; on each iteration, it reads the compass and draws an arrow on the display. Without Phoenix’s recovery capabilities, each iteration took, on average 1005ms; with Phoenix enabled, an iteration averaged 1862ms. The main reason for this slowdown is that the peripheral accesses are so frequent that checkpointing, and therefore memory protection, was nearly always enabled.

While the performance of the applications varied due to their disparate update patterns, the space overhead was consistently small. As seen in Table 3.5, all of the applications fit easily within the 512-slot journal and 64-slot control flow queue, and the rematerialization queue length never exceeded 3. The total space overhead was comparable to that of the benchmarks, requiring a maximum of 5.0 KB (virtual

<table>
<thead>
<tr>
<th>Application</th>
<th>autonomous car</th>
<th>obstacle tracker</th>
<th>virtual compass</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNL</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
</tr>
<tr>
<td>CFQ</td>
<td>284</td>
<td>284</td>
<td>284</td>
</tr>
<tr>
<td>RMQ</td>
<td>876</td>
<td>1184</td>
<td>1360</td>
</tr>
<tr>
<td>Peripheral Metadata</td>
<td>476</td>
<td>352</td>
<td>368</td>
</tr>
<tr>
<td>Recovery Metadata</td>
<td>36</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>Total</td>
<td>4792</td>
<td>4964</td>
<td>5156</td>
</tr>
</tbody>
</table>
Table 3.7: Application Recovery Space, With Failure (bytes)

<table>
<thead>
<tr>
<th>Application</th>
<th>autonomous</th>
<th>obstacle</th>
<th>virtual</th>
</tr>
</thead>
<tbody>
<tr>
<td>JNL</td>
<td>3120</td>
<td>3120</td>
<td>3120</td>
</tr>
<tr>
<td>CFQ</td>
<td>284</td>
<td>284</td>
<td>284</td>
</tr>
<tr>
<td>RMQ</td>
<td>1068</td>
<td>1084</td>
<td>1584</td>
</tr>
<tr>
<td>Peripheral Metadata</td>
<td>476</td>
<td>352</td>
<td>368</td>
</tr>
<tr>
<td>Recovery Metadata</td>
<td>336</td>
<td>294</td>
<td>312</td>
</tr>
<tr>
<td>Total</td>
<td>5280</td>
<td>5134</td>
<td>5668</td>
</tr>
</tbody>
</table>

compass) when no failure occurred and a maximum of 5.5 KB (virtual compass) when a single failure occurred.

Reliability cannot come for free. It demands tradeoffs of time and space, both of which are scarce in embedded systems. Phoenix optimizes for both, saving time by enabling rollback to the exact point of failure and saving space by logging only that which is absolutely necessary. Still, a time overhead is perceivable in peripheral-intensive workloads. In such workloads, the costs of improved reliability may outweigh the benefits if time is of the essence; yet, if reliability is critical, this tradeoff may well be worthwhile. Further, this overhead could be largely eliminated by employing a hardware journal.

However, Phoenix is well-suited to applications that access peripherals periodically; evaluation of Phoenix in these applications revealed no observable delays. Such access patterns are more characteristic of typical embedded workloads, which periodically monitor their surroundings via sensors and react to discrete events. Finally, Phoenix’s lightweight checkpointing mechanisms proved uniformly amenable to the
space constraints of embedded systems, even when running realistic applications with multiple peripherals.

3.7 Discussion

While Phoenix was implemented within the Owl Python run-time system, the concepts and procedures at its core transcend this context. Phoenix could easily be implemented within the interpreter for a different high-level language. The run-time mechanisms would be the same; the compile-time mechanisms would simply need to be ported for the new language.

Phoenix could also be implemented in an embedded run-time system for a lower-level language such as C, and the system mechanisms would remain largely the same. For instance, failure detection could be managed by the same C interrupt handlers. Checkpointing would also be extremely similar. Owl stores all program state on the Python heap, so the only internal state to be logged is the Python heap. In contrast, an implementation for C would need to protect both the C heap and stack. As with any language, the compile-time mechanisms would need to be ported. The programmer mechanisms would likewise require some superficial transformations; for instance, rather than using classes, the system could enforce a requirement that all code for a single peripheral be encapsulated within a single file.
Chapter 4

Memory Optimization and Analysis

4.1 Introduction

Just as managed run-time systems provide new opportunities to advance the state-of-the-art of reliability in embedded systems, so too do they present new opportunities for innovations in memory optimization and analysis. In particular, the structure that run-time systems impose on memory enables the construction of graphs of embedded memory, thereby facilitating a variety of memory optimizations.

The power of graphs lies in the ease of their analysis, visualization, and transformation. In particular, a graph structure provides a clear and intuitive representation of the relationships between objects, greatly facilitating transformations across multiple nodes. This property is a hugely valuable asset in memory transformation, as inter-object references make it nearly impossible to achieve meaningful transformations by mutating a single object in isolation. Yet, existing tools build memory graphs solely for visualization and analysis, transforming these graphs only insofar as is necessary to display them in a meaningful way.

Embedded systems face challenges that incentivize extending the use of memory graphs to impact the actual layout of memory on the microcontroller. First, memory is small and precious, motivating careful memory allocation. Second, embedded systems often consist of many interconnected, heterogeneous microcontrollers with varied quantities of SRAM and flash. This further complicates memory allocation, as
a one-size-fits-all approach will squander valuable memory.

This chapter details GEM (Graphs of Embedded Memory), an offline memory configuration tool for embedded run-time systems designed to address the aforementioned challenges through the transformation of memory graphs. Because the infrastructure of GEM is flexible and extensible, GEM can be applied in a plethora of different use cases. As a proof of concept, this thesis evaluates the application of GEM to four use cases: interactive memory visualization, de-duplication of objects and code, compilation for heterogeneous memory architectures, and transparent migration.

The rest of this chapter is organized as follows. Section 4.2 describes the structural properties that the system must impose on memory in order to support GEM’s capabilities. Section 4.3 describes the infrastructure of GEM, including the graph transformations fundamental to GEM’s functionalities. Next, Section 4.4 explains the workflow that GEM follows for each of the four use cases. Last, Section 4.5 presents an evaluation of GEM across all four use cases.

### 4.2 Context

Memory analysis and optimization tools are especially valuable at the intersection of embedded systems and managed run-time systems. Embedded systems have much to gain from memory optimization; run-time systems impose structure on memory that facilitates these gains. Thus, GEM targets embedded run-time systems.

A major obstacle to building tools to manage the scarce, heterogeneous memory on these microcontrollers is the unstructured memory layout of traditional embedded systems. Managed run-time systems establish order within the system memory space. They typically store the type within each object explicitly, and support the allocation of objects belonging to these well-defined types. This structure enables the creation
of tools such as GEM that can analyze and manipulate the entire memory space of a program. In particular, GEM requires a memory organization that has two properties. First, all data must be stored in objects that have well-defined types known to the run-time system. Second, all objects must include either explicit or implicit type and size identifiers.

The fundamental concepts behind GEM are generalizable, and could be implemented for any run-time system which satisfies these two requirements, such as the Oracle JVM or CPython. A prototype was built for Owl [4, 10], an embedded Python run-time system representative of managed run-time systems for embedded microcontrollers. Owl stores the entirety of the Python program, including the code, as Python objects. Each object type is defined as a C struct with a 4 B object descriptor specifying its type and size, and has fields containing data and references to other objects.

These Python objects are distributed across SRAM and flash. All of the run-time state, including threads and stack frames (activation records), is allocated on the Python heap, which resides in SRAM. At compile-time, modules containing Python library code are stored in flash. Owl first compiles the modules using the standard CPython compiler, generating one Python code object for each module. Each code object includes the code itself — a series of bytecodes, in the form of a string of bytes — as well as metadata such as the filename and variable names. Then, the Owl toolchain converts the compiled modules into a “packed” format, a compressed format similar to the CDR coding data representation used in Lisp [32]. To support this format, Owl supplements the standard CPython types with two additional types: packed code objects and packed tuples. Packed objects do not reference other objects, instead storing constituent objects internally as a contiguous array. Packing saves space that would otherwise be consumed by references. Further, it eliminates the
need for a dynamic linker, as each module is completely self-contained.

Finally, at boot-time, Owl allocates a global object on the heap to hold the run-time system state, constructs a set of “module paths” consisting of the addresses of the code objects corresponding to modules, and adds a reference from the global object to these module paths. The module paths serve two purposes. First, they facilitate importing and executing these modules at run-time. Second, they ensure that all of the Python objects in flash are reachable from the heap. This allows GEM to collect all Python objects in flash simply by following references from the heap.

Because Owl stores Python objects in both SRAM and flash, the memory graphs that GEM builds have two components, one for each memory region. To support systems with more complex memory hierarchies, these graphs could trivially be expanded to include components for additional memory regions.

### 4.3 Tool Organization

Due to resource constraints, GEM operates offline. This allows GEM to build and manipulate a graph representation of memory, without having to reserve any of the microcontroller’s memory space for this functionality. This requires at least one controller with sufficient resources to run GEM. However, a single controller can serve an entire system composed of many microcontrollers, and could be connected either physically or via a network. Further, this controller need not be dedicated to GEM; the same controller could be used to perform other deployment and management tasks already required by the system.

GEM is composed of three layers. Foremost is the graph transformation layer, consisting of a series of passes that operate on a memory graph. Next to this layer is the parser/unparser layer, which converts between two representations of memory: a
graph format and a raw hex string format. Last is the memory access layer, which transports memory contents to and from the microcontroller.

Across these three layers, GEM uses two custom representations of memory. First, GEM includes an intermediate representation of the microcontroller’s entire memory space: a PyMem. A PyMem is a simple container used by the parser to aggregate objects prior to building the graph. It consists of one mapping for each region of memory — in Owl, SRAM (heap) and flash (library code objects). These mappings associate memory addresses with PyObject, which are simple, generic representations of Python objects. Each PyObject \( o \) has a type, a size, and two name-to-value mappings of its fields: \( \text{data}(o) : \text{data.names}(o) \rightarrow \text{data.vals}(o) \), for primitive types, and \( \text{pyptrs}(o) : \text{pyptr.names}(o) \rightarrow \text{pyptr.vals}(o) \), for pointers to other Python objects.

Additionally, frame objects include a third mapping: \( \text{cptrs}(o) : \text{cptr.names}(o) \rightarrow \text{cptr.vals}(o) \), which includes all pointers that are not base addresses of Python objects. Though arbitrary, ill-defined pointers complicate graph construction, the cptrs contained in frames are well-defined pointers into other Python objects. In particular, there are three: the instruction pointer, stack pointer, and pointer to the last stack slot. The former points into a separate bytecode object; the latter two point into this same frame object.

Ultimately, GEM’s graph transformation layer operates on a MemGraph. A MemGraph is a directed graph of memory which, like the PyMem, encompasses multiple regions of memory. Each node corresponds to a PyObject, and edges run from a given PyObject \( o \) to the nodes of each of the objects in pyptr.vals\( (o) \). Formally, a MemGraph is a graph \( G = (V, E) \) such that \( V(G) = \{ u \mid u \text{ is a PyObject} \} \) and \( E(G) = \{ (u, v) \mid \text{address}(v) \in \text{pyptr.vals}(u) \} \).
4.3.1 Graph Transformation

Converting the memory space into a graph unlocks a multitude of possible transformations. This section presents seven fundamental transformations supported by GEM. Strategic composition of these rudimentary transformations enables use cases including, but by no means limited to, the four that will be presented in Section 4.4.

Updating References

Moving even a single object in memory requires updating all references to it to reflect its new address. GEM provides two different means of updating references. First, given a mapping of old to new addresses, GEM can relocate all objects to their new addresses. After changing the addresses of the objects themselves, GEM iterates over the `pyptr_vals` and `cptr_vals` of each object. If it encounters an address from `pyptr_vals` that is a key in the old-to-new mapping, it updates the reference to point to the corresponding new address.

Updating `cptr_vals` is slightly more complicated; while these references may not exactly match any of the keys in the old-to-new mapping, they might point into an object whose base address is a key in the mapping. To handle stack pointers, GEM checks if the containing frame object is in the mapping, and if so shifts the stack pointers by the difference between the frame’s new and old addresses. To adjust instruction pointers, GEM first iterates over the bytecodes and builds a set of [start, end) address pairs encoding the range of addresses spanned by these bytecodes. For each instruction pointer GEM finds the start address of the range in which the pointer falls, checks whether this start address is a key in the mapping, and if so increments the pointer by the difference between the bytecode’s old and new addresses.

The second way in which GEM can update references is by accepting an offset by
which to shift the entire heap. GEM first shifts the address of every object in the heap component, and then shifts all intra-heap `pyptr_vals` and `cptr_vals`.

**Allocation and Garbage Collection**

GEM requires a memory allocator and garbage collector. These need not use the same algorithms as the run-time system, so long as they maintain the proper structure of the free list. For simplicity, both the memory allocator and the garbage collector within GEM mimic those used by the run-time system. This uniformity allows GEM to leverage the existing structure of the free list, and also ensures that no additional work is needed to transform the free list into a format compatible with the run-time system once GEM’s work is done.

The memory allocator is best fit with arbitrary fracturing: it traverses the free list, looking for the smallest free block that is larger than the requested size. If this block exceeds the requested size by more than the system’s minimum allowed chunk size, GEM carves off the extra and adds it to the free list. Otherwise, the entire block is used for allocation.

Owl’s garbage collector performs two-pass mark-and-sweep collection, converting unreachable objects into free blocks. GEM supplements this with an optional compacting pass, effectively providing mark-compact collection. Compaction moves all live objects into contiguous addresses, aided by the first reference updating technique, and then coalesces the free list.

**Finding and Eliminating Duplicates**

Discovering and eliminating duplicate objects is instrumental to achieving a space-efficient memory layout. To identify duplicates, GEM builds a set of object pools using
a deep equality checker. This equality checker takes a pair of PyObjects, directly
compares their data and cptrs mappings, and recursively compares their pyptrs
mappings. Objects deemed equal are placed in the same pool. After every object
has been assigned to a pool, pools of size one — representing unique objects — are
filtered out, leaving only pools of duplicates.

Having found these pools of duplicates, GEM can free large chunks of memory
by eliminating redundancies. For each pool, GEM chooses one representative object,
and then uses the first reference-updating technique to change all references to other
objects in that pool to point to the representative. Last, it uses the compacting
garbage collector to free duplicates that were not chosen as representatives.

While GEM can consolidate all types of objects, in practice it only does so for
immutable objects, as combining mutable objects that were originally distinct would
violate the semantics of Python. Despite this limitation, GEM finds significant op-
portunities for de-duplication, as Python defines a large number of immutable types
including booleans, integers, floating point values, strings, and tuples.

Unpacking Objects

As described in Section 4.2, Owl contains packed types which, rather than referencing
independent objects, embed their constituents within themselves. During the unpars-
ing phase which constructs the memory graph (Section 4.3.2), GEM automatically
generates individual nodes for the top-level object and each constituent, and then
adds edges from the top-level object to each constituent.

Unpacking a packed object consists of allocating an equivalent unpacked object
and freeing the original packed object. First, GEM allocates the top-level unpacked
object using its memory allocator. Second, GEM handles the constituents by cre-
Figure 4.1: Unpacking a Packed Tuple

Before:

<table>
<thead>
<tr>
<th>type: PTP</th>
<th>length: 3</th>
<th>size: 44</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>type: INT</th>
<th>value: 1</th>
<th>size: 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: INT</td>
<td>value: 2</td>
<td>size: 12</td>
</tr>
<tr>
<td>type: INT</td>
<td>value: 3</td>
<td>size: 12</td>
</tr>
</tbody>
</table>

items: (, , , )

After:

<table>
<thead>
<tr>
<th>type: TUP</th>
<th>length: 3</th>
<th>size: 20</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>type: INT</th>
<th>value: 1</th>
<th>size: 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>type: INT</td>
<td>value: 2</td>
<td>size: 12</td>
</tr>
<tr>
<td>type: INT</td>
<td>value: 3</td>
<td>size: 12</td>
</tr>
</tbody>
</table>

items: (, , , )

ating references from the unpacked top-level object to the existing nodes for these constituents. Constituents which are themselves packed are unpacked recursively.

Taken alone, unpacking packed objects consumes additional space, as the extra layer of indirection adds an overhead. Consider Figure 4.1: unpacking the packed tuple in this example increased the total size of the tuple and its constituents from 44 B to 56 B. However, unpacking presents additional opportunities for fine-grained de-duplication. Thus, unpacking can be a valuable transformation when taken in conjunction with de-duplication, as will be shown in Section 4.5.2.

Splicing and Splitting

Splicing takes two graphs, a source and a destination, and splices one or more components of the source into the destination. To do so, GEM first performs an equivalence search to identify objects that are present in the source, but missing from the destination. This is achieved via an inter-graph duplicate search, using the duplicate
discovery technique. Next, GEM uses its memory allocator to allocate each of these missing objects within the SRAM portion of the destination graph. During the allocation phase, GEM constructs a map that relates the original addresses of the missing objects to the addresses of the newly-allocated equivalent objects. It then updates all of the intra-graph references according to this mapping, using the first reference-updating technique. Figure 4.2 demonstrates the splicing of one source graph into another destination graph, in which nodes a and c from the source’s SRAM and e and f from the source’s flash are allocated in SRAM at the destination. Note that b and d need not be allocated, as they are already present in the destination’s flash.

Splitting takes a single graph and partitions its objects into multiple components. GEM’s splitting framework takes as an input an algorithm for partitioning the objects. This algorithm must take in a single graph $G = (V, E)$ and return a partition in the form of two sets of objects $S1$ and $S2$ such that the objects in $S1$ and $S2$ are disjoint and the union of $S1$ and $S2$ is $V$.

While this design choice was made to maximize flexibility by allowing the user to customize the partitioning to new use cases, several different partitioning algorithms
are already built into GEM; one such algorithm will be presented in Section 4.4.3. GEM uses the input algorithm to partition the objects, and then constructs new graphs representing each set. To eliminate any gaps introduced by partitioning the objects, GEM applies its garbage collector’s compaction pass to each new graph.

4.3.2 Parsing and Unparsing

While graphs are very amenable to analysis and manipulation, they are not compact enough to be used as the memory format on the microcontroller. Instead, Owl lays out the Python objects as a contiguous array of bytes. Thus, GEM includes a parser and unparsers to convert between a hex string representation of the raw byte format on the microcontroller, and the MemGraph format on which the transformations from Section 4.3.1 operate.

The parser and unparsers include top-level parse and unparse functions which take in a hex string and graph, respectively. Additionally, they include parse_<type> and unparse_<type> functions for each type, including free blocks. The top-level functions first process each individual object by dispatching a call to the appropriate type-specific function, and then aggregate the results.

In more detail, parse takes in the base address of the heap and a hex string of its contents, and outputs a MemGraph. First, it builds an intermediate PyMem representation as follows. It sets the current address to the heap’s base address and begins reading the hex string. While it has not reached the end, it interprets the next four bytes as an object descriptor and extracts the size and type. It then calls the appropriate parse function based on the type, which creates a PyObject or free block. The parser maps this object’s address to the newly-created PyObject, increments the current address by the size of the just-parsed object, and attempts to read the next
object descriptor. Once the entire string has been processed, the parser builds the MemGraph by iterating over the PyObjects, inserting them as nodes, and adding edges for each of their pyptrs.

Conversely, unparse receives a MemGraph and outputs a hex string. It first builds a sorted list of the addresses of all heap nodes in the graph. None of GEM’s transformations introduce gaps in the address space, so these addresses will be contiguous assuming that GEM was given a valid image when it initially built the graph. It then unparses each object in order by invoking the correct unparse function, again based on type, to get a hex string representation of that object. Last, it concatenates the hex strings for the individual objects to get a complete representation of the heap.

GEM auto-generates all of the type-specific parse and unparse functions at compile-time. First, the header files containing the type definitions are passed through pycparser [5]. GEM uses a custom visitor function to process each type definition, outputting metadata to an intermediate file. Specifically, the parser generator outputs a list of tuples of (field name, field type, field size) for each type. Next, the parser generator uses this metadata to auto-generate the parse_<type> and unparse_<type> functions. The parse functions classify each field as belonging to data, pyptrs, or cptrs according to its type, and advance the index into the hex string based on the field’s size and the amount of padding between fields. The unparse functions simply fetch and unparse each field in order, concatenating the results together. As needed, the unparse functions insert padding, both between fields for memory alignment as well as at the end of the object to round out the size in the case that the memory allocator did not yield a chunk of perfect fit.

This auto-generation adds robustness to the parser and unparsers. As new types are introduced to the run-time system, or fields are added or removed from existing
types, no maintenance is required; the parser and unpars will be re-generated upon
compilation to reflect these changes. Similarly, Owl accepts configuration options,
specified as C defines, which may affect the type definitions. GEM passes this list of
defines to `pycparser` so that the parser and unpars are customized to be compatible
with the specified configuration.

Further, this auto-generation makes GEM extensible to other systems. With mi-
nor modifications to the parser generator, GEM could generate a parser and unpars
for any system whose memory layout adheres to the principles outlined in Section 4.2.
In particular, slight changes would need to be made to accommodate different formats
for object descriptors and type definitions.

### 4.3.3 Memory Set and Dump

To escape the microcontroller’s resource constraints during graph transformation,
GEM operates on a desktop “host” which is connected to the microcontroller. GEM
provides two commands, `dump` and `memset`, which transfer a raw hex string rep-
resentation of the microcontroller’s memory space to and from the host machine,
respectively. These operations allow the memory transformations made by GEM to
affect the actual layout of memory on the microcontroller.

The first of these, `dump` consists of three stages. In each stage, GEM compiles
and sends over the connection a call to a function which, upon execution by the
interpreter running on the microcontroller, causes the microcontroller to send data
back over to GEM. GEM first queries for metadata regarding the heap, including
its base address and roots. It then requests the contents of the heap, beginning at
this base address. After this stage, the heap is parsed according to the procedure
described in Section 4.3.2. Last, GEM fetches all of the Python objects in flash. As
explained in Section 4.2, this consists of the code objects for the Python modules.

The reverse of dump is memset, which overwrites the current heap on the microcontroller. The source and destination devices need not be the same. However, if they are different, the base addresses of the heaps and the contents of flash may also be different. Therefore, memset does not blindly replace the heap. Instead, GEM first dumps the memory of the destination device and builds a graph of it. It then constructs a new graph consisting of the SRAM (heap) component of the source and the flash component of the destination. In doing so, it updates all SRAM-to-flash references from the source to point to equivalent objects in the destination flash, if such equivalents exist, using the technique from Section 4.3.1. Next, it uses the splicing technique described in Section 4.3.1 to allocate, within the SRAM component of the new graph, copies of all objects that were present in the flash of the source but not present in the flash of the destination. After splicing is complete, GEM shifts the intra-heap references by the difference between the old and new base addresses, using the second reference-updating technique. Last, GEM unparses this new graph and uses it to overwrite the heap of the destination device.

GEM initiates this final step by compiling and sending a special MEMSET bytecode to the microcontroller. Unlike dump, memset cannot be implemented as a function call, as overwriting the heap obliterates the call stack. GEM then sends the heap contents and metadata over the connection in a raw byte format. In executing the MEMSET bytecode, the microcontroller reads the heap contents and metadata, overwriting both. Once the MEMSET bytecode completes, the program automatically resumes execution from the exact point at which its memory was dumped. This is because all of the Python program context, including the instruction pointer and stack pointers, is stored on the heap.
4.4 Use Cases

The mechanisms and transformations described in Section 4.3 serve as fundamental building blocks which can be combined to support a breadth of use cases. To illustrate the versatility of GEM, this section presents four sample use cases. First, GEM can be augmented with a GUI to visualize the memory space. Second, it can be utilized to de-duplicate constants and code at compile-time, substantially decreasing the code size. Also at compile-time, GEM can be employed to customize the distribution of code across SRAM and flash, allowing the same run-time system to be programmed on a wide spectrum of memory architectures. Fourth, GEM can be used at run-time to transparently migrate a running program.

4.4.1 Memory Visualization

Extracting meaningful information from a raw memory dump is extremely challenging. Thus, GEM supports a graphical visualizer which provides an organized view of the entire memory space of the microcontroller. It allows the user to view how objects are laid out in memory, see which objects are live at a given point in execution, and identify which objects are consuming the most space. Moreover, it uses techniques from Section 4.3.1 to find and display duplicate objects and unreachable objects. GEM’s visualizer is highly interactive, allowing the user to sort, search, navigate, and inspect objects.

User Interface

The GUI for GEM’s visualizer is divided into three vertical panes, as shown in Figure 4.3. The left pane is a glossary of all objects in the graph. It contains two sub-panes, one for SRAM and one for flash. The objects in each sub-pane can be
sorted by address, size, type, or value. A search bar enables quick lookup by address, and menu options allow for moving sequentially to the previous or next address. Selecting an object in the left pane will update several object-specific displays in the middle and right panes.

The middle pane offers six different display options. Three of these display options are specific to the selected object: textual “parent” and “child” displays, plus a graphical “ringschart” display. An object’s parent set consists of all objects that reference it; its child set contains those objects that it references. The graphical display is based on the open-source Linux Graphical Disk Usage Analyzer [3]. It displays a subset of the graph as a “ringschart” centered around the selected item. The center of the ringschart shows the combined size in bytes of the weakly-connected component rooted in the selected object, broken down into SRAM and flash consumption. Hovering over an object will highlight that object and spawn tooltips from each child of that object, annotating each child with its type and size. If there are aliases, only one reference to the aliased object will be displayed in color, with the rest denoted by grey; hovering over an aliased object will highlight all of its aliases.

Rather than displaying the entirety of the component rooted in the selected object, the graphical display limits the depth of the rings displayed to a manageable number — which can be configured via “zoom in” and “zoom out” options in the dropdown menu — and indicates which branches of the ringschart are deeper than can be displayed. More of the graph can be explored by clicking on an object’s segment in the ringschart, which has the same result as selecting it from the left pane: the ringschart is re-centered around the selected object, and all other object-specific panes are re-rendered.

The last three display options in the middle pane highlight general properties
Figure 4.3: Graphical Display for GEM’s Interactive Memory Visualizer

- Objects in SRAM
  - For a selected object O1,
    1. Graphical view
    2. Objects referenced by O1
    3. Objects that reference O1

- Objects in Flash

- Object-specific Options

- Aggregated View
  - Type-specific aggregation for a selected object O1

- General Graph Options
  1. Roots of the graph
  2. Unreachable objects
  3. Duplicated objects

- Graphical Visualization
of the graph: roots, garbage, and duplicates. The roots display lists the addresses and types of the graph’s roots, which correspond to global objects in memory. The garbage display lists all objects that are unreachable from the roots. Last, the duplicates display organizes all duplicate objects — found using the duplicate discovery technique from Section 4.3.1 — by type, with an entry for each pool of size greater than one. This entry lists the addresses of all objects in that pool.

The contents of the right pane are specific to the currently-selected object. This pane serves as an aggregated display for compound objects. It has special display modes for lists, tuples, sets, dictionaries, frames, threads, and code objects. For instance, in Figure 4.3 the selected object is a thread, and the right pane displays the stack of frames running in that thread. Each frame is tagged not only with the name of the function to which it corresponds, but also its attributes, global variables, and local variables. Variables’ names are displayed, along with their address and current value.

4.4.2 De-duplication

Python code is fraught with duplicates. These duplicates are generated by the standard CPython compiler, and are relatively harmless on a desktop machine with vast resources. However, in an embedded system where memory is a precious commodity, eliminating duplicate objects within the Python code is critically important. GEM uniquely applies de-duplication during compilation, and as such its effects are completely orthogonal to any interning the system may do at run-time.

For correctness, GEM limits de-duplication to immutable objects. In reality, this is no limitation at all, as all of the constants referenced by the Python code objects belong to immutable types. Due to the prevalence of duplicates, GEM’s de-duplication
yields substantial memory gains, as Section 4.5.2 will demonstrate.

**Workflow**

GEM performs de-duplication at compile-time by sequentially applying graph transformations described in Section 4.3.1. At a high-level, de-duplication proceeds as follows:

1. Build a graph of the library code, in Owl's default packed format, using the parser described in Section 4.3.2.

2. Unpack the library using the unpacking technique described in Section 4.3.1.

3. Remove duplicates using the duplicate discovery and elimination technique described in Section 4.3.1.

4. Shift objects to eliminate gaps created by garbage collection, using the first reference-updating technique described in Section 4.3.1.

5. Unparse the graph using the unparsers described in Section 4.3.2.

6. Output the unparsed image to a C file, which will then be compiled into the run-time system by the Owl toolchain.

7. Post-compilation, parse the image into a graph once more, again using the parser described in Section 4.3.2.

8. Find the base run-time address of the image and shift all references within the graph by this base address, using the second reference-updating technique described in Section 4.3.1.
9. Unparse the reference-shifted graph using the unparser described in Section 4.3.2 and overwrite the ELF file with this new image.

In greater detail, de-duplication begins immediately after Owl’s image creator has built a binary representation of the Python library code. GEM converts this binary representation into a hex string and then parses it into a graph, using a base address of 0 since the location of the library in memory has not yet been determined.

The packed objects that Owl’s image creator produces by default cannot be de-duplicated, since they directly contain their constituent objects. Therefore, the next step is to unpack these packed objects. GEM provides two levels of unpacking. First, it can indiscriminately unpack all packed objects. However, while the conversion to unpacked objects enables de-duplication, it also introduces an overhead by reinstating references. Thus, GEM also provides a more sophisticated hybrid approach: selectively unpacking only those objects for which the projected space savings due to de-duplication exceed the projected overhead due to reference reinstatement. Section 4.5.2 will show that either level of unpacking, in conjunction with de-duplication, yields significant savings; however, the latter approach consistently surpasses the former.

After unpacking, GEM finds and consolidates duplicates, using the compacting garbage collector to regain a contiguous sequence of objects. GEM then determines the module paths — the addresses of the top-level modules in the library. The graph now reflects the exact structure in which the Python code will be stored in flash. However, the base address is still 0. Therefore, GEM stores the de-duplicated graph and its module paths for later adjustment once the C linker has placed the image. It then passes unparsed versions of the graph and its module paths back to the image creator.
For now, GEM’s job is done, and normal compilation resumes. The Owl toolchain auto-generates a C file containing the unparsed library image and module paths and compiles this file into the run-time system. At this point, the base run-time address of the library has been determined, so GEM can adjust the intra-library references. It uses `readelf` to find this base address, reloads the de-duplicated graph that was stored previously, and shifts all references by an offset equal to the base run-time address. It likewise increments each module path by the base address. Finally, GEM unparses the library and the module paths back into binary representations and overwrites the ELF file with the adjusted versions.

### 4.4.3 Heterogeneous Compilation

Designing a run-time system that fits within the wide spectrum of available microcontroller memory architectures is extremely valuable, as it eases the construction of cooperative heterogeneous embedded systems. GEM is unique in enabling customization of the run-time system for a particular memory architecture by partitioning the Python library code amongst SRAM and flash, vastly expanding the range of memory hierarchies on which the run-time system can operate. The amount of SRAM and flash available for the Python library code are simply specified at compile-time. Given these constraints, GEM will then either successfully compile the run-time system, or output an error message indicating that the specified capacity is insufficient.

**Workflow**

To increase the likelihood of successfully accommodating the specified memory constraints, heterogeneous compilation is bracketed by the de-duplication procedure outlined in Section 4.4.2. Thus, its compile-time workflow is as follows:
1. Build a graph of the library code, in Owl's default packed format, using the parser described in Section 4.3.2.

2. Unpack the library using the unpacking technique described in Section 4.3.1.

3. Remove duplicates using the duplicate discovery and elimination technique described in Section 4.3.1.

4. Partition the library code between SRAM and flash, using the splitting technique described in Section 4.3.1.

5. Unparse both graphs using the unparser described in Section 4.3.2.

6. Output the unparsed flash image to a C file, which will then be compiled into the run-time system by the Owl toolchain.

7. Post-compilation, parse the flash image into a graph once more, again using the parser described in Section 4.3.2.

8. Find the base run-time address of the flash image and shift all references within the graph by this base address, using the second reference-updating technique described in Section 4.3.1.

9. Unparse the reference-shifted graph using the unparser described in Section 4.3.2 and overwrite the ELF file with the new image.

In addition, the objects designated for SRAM are installed in the heap at run-time using a six-step process, which the user can initiate with a single command:

1. Boot the run-time system.
2. Dump the memory using the technique described in Section 4.3.3.

3. Parse the memory into a graph using the parser described in Section 4.3.2.

4. Splice the SRAM library code into the current memory graph using the splicing technique described in Section 4.3.1.

5. Unparse the augmented graph using the unparsing described in Section 4.3.2.

6. Memset the augmented graph using the technique described in Section 4.3.3.

As with de-duplication, GEM enters the compilation process just after Owl’s image creator has built the binary representation of the packed Python library. GEM then performs the same parsing, graph construction, unpacking, and duplicate elimination as it did for de-duplication. However, rather than compacting the objects into a single contiguous chunk of memory, it partitions these objects into two components, one for flash and one for SRAM.

For this particular use case, the partitioning algorithm passed to GEM’s splitting framework must adhere to three constraints. First, it must respect the upper bound on space for each memory region, as specified at compile-time. Second, since the SRAM library code will not be loaded until the end of boot-time, it must ensure that all modules needed to boot the run-time system are relegated to flash. Third, it must guarantee that nothing placed in flash references anything assigned to SRAM. This is because the final SRAM addresses are not known prior to boot time. Some microcontrollers cannot overwrite flash memory during execution, and it is expensive for those that can; additionally, references from flash into SRAM would become dangling references upon reboot, potentially causing the run-time system to crash.
GEM currently uses a simple greedy algorithm, which the evaluation in Section 4.5.3 proves effective. This algorithm begins by placing the minimal set of objects required to boot the run-time system in flash and assuming that all other objects are in SRAM. To satisfy the third constraint, this algorithm promotes objects to flash on the granularity of components, where each candidate component consists of a root object plus all of the objects that are still in SRAM that are reachable from that root. It greedily chooses the largest component in SRAM that will fit in flash without exceeding capacity, terminating when even the smallest component left in SRAM is larger than the remaining capacity. If the size of the remaining SRAM component exceeds the specified bounds, the algorithm reports failure; otherwise, GEM proceeds with compilation.

After executing this algorithm GEM builds and compacts two new graphs, one for each memory space. The flash graph is given a temporary base address of 0; the SRAM graph is given a temporary base address equal to the size of the flash graph. This prevents ambiguity in interpreting references, since although nothing in flash references SRAM, objects in SRAM may reference flash. GEM stores both graphs and their module paths for later reference adjustment and then resumes the normal compilation process. As in de-duplication, it goes back and adjusts the references to objects in flash — this time for both graphs — once the run-time address of the flash library has been established.

Finally, the SRAM library code must be allocated on the heap. The run-time system is first booted into a special SRAM installation mode which imports only those modules which are needed to perform dump and memset. The current memory space is then dumped and parsed into a graph format, using the techniques from Sections 4.3.3 and 4.3.2. Next, the SRAM graph built during compilation is loaded
and spliced into the current memory graph. During this splicing process, GEM keeps track of the addresses assigned to any top-level modules. Last, GEM unparses and memsets the graph, overwriting the current heap and augmenting the module paths with the addresses of the modules in the SRAM library code.

While this installation process is the same across all platforms, the mechanism that triggers this process varies. The Owl toolchain for the desktop version is configured to automatically boot the run-time system post-compilation and display an interactive prompt. In this case, GEM hooks into the boot process to install the SRAM library code after the run-time system is initialized, but before the prompt is displayed to the user. In contrast, the Stellaris and STM32 platforms cause the microcontroller to wait in a loop for one of four different commands: connect to the desktop via the interactive prompt, program an application in flash, run an application that has been programmed in flash, or compile and run a program in SRAM. GEM adds a fifth command which triggers SRAM installation.

4.4.4 Transparent Migration

Transparent migration, which preserves the run-time state of the program, is valuable in many scenarios. For instance, if one device fails, migrating a pre-crash checkpoint to another device prevents complete loss of work. Likewise, a system with substantial startup delays may benefit from migration of a pre-booted image [55]. While many existing systems support transparent migration, the value of GEM’s approach lies in the ease with which it harnesses graph transformations — in particular, its novel splicing technique — to enable migration between devices with disparate hardware and software.
Workflow

At a high level, a program running upon a run-time system can be migrated by transplanting the run-time system’s heap. However, as described in Section 4.3.3, the base address of the heap is not guaranteed to be identical across all instances of the run-time system. Further, the contents of flash may not be the same at the source and destination, which is problematic as objects on the heap may reference flash. GEM’s \texttt{memset} command automatically corrects for these differences by adjusting all intra-heap references and allocating all missing flash objects in SRAM — since, as mentioned in Section 4.4.3, many microcontrollers do not support mutating flash at run-time. Since \texttt{memset} automatically addresses these challenges, GEM’s migration process is quite simple:

1. Dump the source memory using the technique described in Section 4.3.3.

2. Parse the memory into a graph using the parser described in Section 4.3.2.

3. Optionally perform any desired transformations such as de-duplication or free list compaction.

4. Write the snapshot of the transformed graph to a file.

5. Memset the source snapshot onto the destination using the technique described in Section 4.3.3, which dumps the destination memory and splices the source graph into the destination’s flash component.

In the prototype implementation, the \texttt{dump} and \texttt{memset} are initiated by the user at the interactive prompt, though they could be automated. To allow the user to initiate a \texttt{dump} or \texttt{memset}, GEM includes two mechanisms by which the user can pause the program and access the prompt. First, if the programmer knows the point
in the program at which he or she wishes to migrate in advance, he or she can insert a call to a built-in function which pauses execution.

However, in many cases the programmer may not anticipate wishing to migrate. Thus, GEM extends Owl’s interpreter to support pausing execution at any time by simply pressing a button on the microcontroller. Once the program is paused, the user can connect to the microcontroller via the interactive prompt and perform migration. Furthermore, the user can press this button a second time to resume the program on the original microcontroller, if desired. While manual activation serves as an effective proof-of-concept, the same pause/resume logic could be invoked by an internal error-recovery mechanism, or combined with Owl’s message-passing capabilities to enable remote triggering.

4.5 Evaluation

This section evaluates the four use cases described in Section 4.4 on a series of benchmarks. Each benchmark is a snapshot of a specific workload running on a specific platform, named in the form `<platform>`-`<workload>`. GEM was evaluated on three platforms: a Stellaris LM3S9B92 microcontroller (“Stellaris”), an STM32F4-Discovery microcontroller (“STM32”), and a desktop machine (“Desktop”). The LM3S9B92 has 96 KB of SRAM and 256 KB of flash; the F4-Discovery board has 192 KB of SRAM and 1 MB of flash. The workloads include two snapshots that are not application-specific, collected at compile-time (“compile”) and boot-time (“boot”), as well as a run-time snapshot of an application that uses an accelerometer and TFT display to present an artificial horizon display (“ahd”).
4.5.1 Memory Visualization

GEM’s memory visualizer has proven extremely useful for exposing inefficiencies and bugs within Owl. For example, GEM’s visualizer exposed a sub-classing bug during the development of Owl. Initially, applications that used sub-classing would inexplicably crash. When GEM was employed to analyze the problematic programs, the root cause was easily found and fixed: the bases pointers of the class objects were being garbage collected due to a small bug in the mark phase of Owl’s garbage collector.

Further, GEM’s visualizer has exposed opportunities for design changes to improve system efficiency. In particular, it inspired the proposal of two improvements to the format used to store the Python library code in flash. Originally, Owl used the same Python object format as CPython. However, using GEM to analyze the contents of flash revealed much wasted space. Therefore, two changes to the library format were proposed in order to conserve flash: a packed format, which bypasses the overhead of references, and the duplicate-free format introduced in Section 4.4.2. A comparison of the original and packed formats, which will be presented in Section 4.5.2, validates the intuition that the references in the unpacked version consume an exorbitant amount of space. Yet, further analysis by GEM’s memory visualizer exposed many duplicates within these packed code objects, as shown in Table 4.2. GEM indicated that the gains to be had by eliminating duplicates would outweigh the losses associated with adding references. Both of the proposed library formats were accepted, and Owl now allows the user to choose which format is used at compile-time.

4.5.2 De-duplication

As de-duplication occurs at compile-time, it was evaluated on the Stellaris_compile, STM32_compile, and Desktop_compile benchmarks. For each benchmark, the Python
<table>
<thead>
<tr>
<th>Platform</th>
<th>Unpacked (KB)</th>
<th>Packed (KB)</th>
<th>De-duplicated (KB)</th>
<th>Hybrid (KB)</th>
<th>Unpacked to Hybrid (% Decrease)</th>
<th>Packed to Hybrid (% Decrease)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop</td>
<td>50.4</td>
<td>41.1</td>
<td>31.5</td>
<td>31.3</td>
<td>37.9%</td>
<td>23.8%</td>
</tr>
<tr>
<td>Stellaris</td>
<td>86.4</td>
<td>71.4</td>
<td>62.7</td>
<td>60.3</td>
<td>30.2%</td>
<td>15.5%</td>
</tr>
<tr>
<td>STM32</td>
<td>97.1</td>
<td>80.9</td>
<td>70.3</td>
<td>68.9</td>
<td>29.0%</td>
<td>14.8%</td>
</tr>
</tbody>
</table>
Table 4.2: Duplicate Objects in Unpacked Library Code

<table>
<thead>
<tr>
<th>Type</th>
<th>All Objects</th>
<th>Intra-Module Duplicates</th>
<th>Inter-Module Duplicates</th>
<th>All Duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count</td>
<td>Combined Size (B)</td>
<td>Count</td>
<td>Combined Size (B)</td>
</tr>
<tr>
<td>None</td>
<td>132</td>
<td>528</td>
<td>108</td>
<td>432</td>
</tr>
<tr>
<td>Integer</td>
<td>111</td>
<td>888</td>
<td>17</td>
<td>136</td>
</tr>
<tr>
<td>Float</td>
<td>5</td>
<td>40</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>String</td>
<td>1418</td>
<td>23156</td>
<td>711</td>
<td>11212</td>
</tr>
<tr>
<td>Tuple</td>
<td>604</td>
<td>11340</td>
<td>284</td>
<td>2708</td>
</tr>
<tr>
<td>Bytecode</td>
<td>111</td>
<td>7876</td>
<td>37</td>
<td>852</td>
</tr>
<tr>
<td>Code Object</td>
<td>148</td>
<td>5920</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Native Object</td>
<td>121</td>
<td>968</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>2687</td>
<td>51568</td>
<td>1158</td>
<td>15348</td>
</tr>
</tbody>
</table>
library code was stored in four formats. The first format, unpacked without de-
duplication, is the format output by the CPython compiler; the second is the packed
format adopted by Owl. The last two formats are de-duplicated using the proce-
dure described in Section 4.4.2, first with naive unpacking and second with selective
unpacking based on a heuristic that approximates the potential savings. Table 4.1
shows the space consumed by the library code in each of these four formats. The
modules included in the Python library vary across platforms. Many modules are
platform-independent, such as `math`, `time`, and the built-in types. The desktop plat-
form is smallest, as it consists solely of these platform-independent modules. Stellaris
and STM32 include additional platform-specific modules which support hardware
peripheral access.

Across all three platforms, GEM’s de-duplication consistently saved memory.
Compared to the unpacked format used by CPython, the savings amounted to
19.1 KB, 26.1 KB, and 28.6 KB for Desktop, Stellaris, and STM32, respectively;
in each case this was more than a 29% decrease in space. Table 4.2 presents a break-
down of the de-duplication performed by GEM by type. Strings accounted for over
72% of the de-duplication savings; the next largest sources of redundancies were tu-
uples and bytecodes, respectively. Note that these results indicate a significant amount
of duplication even amongst deep objects such as tuples.

Even as compared to the more compact packed format, de-duplication yielded
savings of 9.8 KB, 11.1 KB, and 12.0 KB for the three platforms: improvements of
14.8–23.8%. The percent improvement was greatest for the desktop version. However,
Stellaris and STM32 yielded better absolute decreases. This dichotomy is due to the
fact that the extra platform-specific modules within Stellaris and STM32 primarily
contain constants such as register addresses, and thus a very small proportion of
duplicate objects.

The presence of modules with a low proportion of duplicate objects motivated the creation of the hybrid format described in Section 4.4.2. The difference in size between the third and fourth formats in Table 4.1 indicates that GEM’s hybrid approach elected to leave some low-redundancy modules packed, as it found insufficient duplicates to offset the overhead of unpacking. This selectivity proved profitable for each platform, saving up to an additional 2.4 KB.

### 4.5.3 Heterogeneous Compilation

By default, the Owl toolchain attempts to place all Python library code in flash. In contrast, GEM allows the placement of this code to be tailored to the particular memory shape of a system. Figure 4.4 presents examples of valid (flash, SRAM) divisions of the Python libraries supported by GEM for each platform. Note that the lower bound on flash is approximately 10 KB, since a small set of modules needed to
boot the virtual machine must be placed in flash.

The total amount of space consumed by a given platform’s library code is effectively constant regardless of the flash/SRAM breakdown. This is because GEM consolidates duplicate objects not only within flash, but across SRAM and flash. Therefore, there is no net disadvantage to moving code to SRAM for microcontrollers with limited flash space. Further, collecting the data in Table 4.4 revealed that the greedy algorithm described in Section 4.4.3 successfully partitions the objects without wasting flash memory. Out of the 30 datapoints, flash was always filled to within 4 B of capacity, and was filled exactly to capacity 83.3% of the time.

4.5.4 Transparent Migration

GEM allows for migration across different memory architectures and across different platforms. Such heterogeneity complicates migration; since the contents of flash may be different at the source and destination, blindly migrating the SRAM may result in missing or duplicated objects. As described in Section 4.4.4, GEM handles missing objects at the destination by allocating these missing objects within the SRAM prior to overwriting the heap. Redundancies can optionally be eliminated immediately prior to `memset`, using the de-duplication technique described in Section 4.4.2.

Table 4.3 shows the results of migration across the same platform (Stellaris), but between different memory architectures. It presents the memory distribution of the artificial horizon display application before and after migration between a system with sufficient space to fit all of the library and application code in flash memory (denoted by a flash cap of $\infty$) and a system with only 80 KB of flash available for the library code. Further, it shows the resultant memory layout with and without de-duplication. For these experiments, the core libraries were already de-duplicated at compile-time.
to isolate the effects of migration: all savings due to de-duplication during migration result from eliminating migration overhead and consolidating run-time structures.

Migration from a system with more flash to a system with less flash added effectively no overhead (0.3%) to the overall memory footprint of the program. Objects which were in flash on the source but were absent from flash at the destination were simply placed in SRAM at the destination; thus, while the memory breakdown at the destination is different, the total consumption is roughly the same. The negligible overhead comes from a small amount of build-specific information. Since this overhead was trivial to begin with, de-duplication was not crucial. However, it did yield 2.9 KB of savings, more than compensating for the minimal overhead.

In contrast, migration from a system with less flash to a system with more flash resulted in a 21.2% overhead without de-duplication. This is because the complete contents of the source heap were placed at the destination, with no regard for the fact that much of what was in SRAM on the flash-constrained source was already in flash at the destination. This was easily solved via de-duplication, which eliminated the extraneous modules as well as some redundant run-time structures, reclaiming over 30 KB for a net gain upon migration.

Migration of the artificial horizon display application between the Stellaris and STM32 platforms highlights GEM’s cross-platform capabilities. Chosen for its realistic workload, this application requires hardware peripherals at the source and destination. However, the peripherals at the source and destination need not be identical, so long as they are compatible with the application. For these experiments the same model of TFT display was used for both Stellaris and STM32, but the Stellaris board was connected to an external accelerometer whereas the STM32 board utilized its on-board accelerometer. Table 4.4 presents the results of this cross-platform mi-
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Flash Cap</th>
<th>Flash Cap</th>
<th>De-duplicated?</th>
<th>SRAM</th>
<th>Flash</th>
<th>Total</th>
<th>SRAM</th>
<th>Flash</th>
<th>Total</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stellaris</td>
<td>∞</td>
<td>80.0</td>
<td>No</td>
<td>34.2</td>
<td>109.4</td>
<td>143.7</td>
<td>64.1</td>
<td>80.0</td>
<td>144.1</td>
<td>0.3%</td>
</tr>
<tr>
<td>Stellaris</td>
<td>∞</td>
<td>80.0</td>
<td>Yes</td>
<td>34.2</td>
<td>109.4</td>
<td>143.7</td>
<td>61.2</td>
<td>80.0</td>
<td>141.2</td>
<td>-1.7%</td>
</tr>
<tr>
<td>Stellaris</td>
<td>80.0</td>
<td>∞</td>
<td>No</td>
<td>60.5</td>
<td>80.0</td>
<td>140.5</td>
<td>60.8</td>
<td>109.4</td>
<td>170.2</td>
<td>21.2%</td>
</tr>
<tr>
<td>Stellaris</td>
<td>80.0</td>
<td>∞</td>
<td>Yes</td>
<td>60.5</td>
<td>80.0</td>
<td>140.5</td>
<td>30.7</td>
<td>109.4</td>
<td>140.1</td>
<td>-0.3%</td>
</tr>
</tbody>
</table>
gration with and without de-duplication. Without de-duplication, migration added an overhead of approximately 1–2%; however, de-duplication once again more than compensated for the overhead.
<table>
<thead>
<tr>
<th>Migration</th>
<th>De-duplicated?</th>
<th>Size Before</th>
<th></th>
<th>Size After</th>
<th></th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmark → Platform</td>
<td></td>
<td>SRAM</td>
<td>Flash</td>
<td>Total</td>
<td>SRAM</td>
<td>Flash</td>
</tr>
<tr>
<td>Stellaris,ahd → STM32</td>
<td>No</td>
<td>34.4</td>
<td>128.2</td>
<td>162.5</td>
<td>37.2</td>
<td>128.5</td>
</tr>
<tr>
<td>Stellaris,ahd → STM32</td>
<td>Yes</td>
<td>34.4</td>
<td>128.2</td>
<td>162.5</td>
<td>33.0</td>
<td>128.5</td>
</tr>
<tr>
<td>STM32,ahd → Stellaris</td>
<td>No</td>
<td>34.3</td>
<td>128.5</td>
<td>162.8</td>
<td>37.4</td>
<td>128.2</td>
</tr>
<tr>
<td>STM32,ahd → Stellaris</td>
<td>Yes</td>
<td>34.3</td>
<td>128.5</td>
<td>162.8</td>
<td>33.2</td>
<td>128.2</td>
</tr>
</tbody>
</table>
Chapter 5

Conclusions

Embedded systems have long suffered from primitive development environments, making it difficult for programmers to keep up with increasingly complex software demands. However, the recent emergence of embedded run-time systems has greatly advanced the productivity of writing embedded applications. These managed run-time systems raise the abstraction level of embedded systems programming by supporting high-level languages and assuming responsibility for critical tasks such as thread scheduling and inter-process communication.

Productivity gains alone ensure that managed run-time systems will have a place in the future of embedded systems programming. Yet, the advantages of managed run-time systems go far beyond productivity. Leveraging the inherent structure and uniformity of managed run-time systems to their full potential requires exploring their application to a broader range of challenges in embedded systems programming. This thesis investigated the application of run-time system techniques to two such challenges: reliability and memory optimization.

Automating reliability and memory optimization at the system-level is a formidable task in any computing system. In embedded systems, several unique concerns further complicate this task. First, these systems operate under extreme memory constraints. Second, they typically are event-driven, and therefore require some degree of reactivity to real-world events. Last, the sensors and actuators which empower these real-world interactions bring external state to the system that cannot
be ignored.

This thesis contributed two systems, Phoenix and GEM, designed to simultaneously meet the demands of embedded systems and harness the structure of managed run-time systems. These systems thereby advance reliability and memory management, respectively, within the domain of embedded systems.

First, this thesis presented the design and implementation of Phoenix, a novel system for surviving peripheral failures. Phoenix is composed of a set of integrated compiler-, run-time system-, and program-level mechanisms, which work together to efficiently record the system state and automatically recover from asynchronous peripheral failures. Several new insights motivated the design of Phoenix. In particular, peripherals interact with the real world and with each other in ways that are substantively different than internal program interactions. Based on this, one of the key innovations of Phoenix is that it implements a novel, light-weight checkpointing system to efficiently track both the internal and external state. After a failure, this enables Phoenix to not only reset the internal system state, but also to restore the external peripheral state by determining whether each peripheral access must be re-executed or rematerialized.

Furthermore, the design of Phoenix was motivated by the fact that time and space are at a premium in embedded systems. Upon failure, Phoenix guarantees rollback to precisely the point at which a failed peripheral access occurred, re-executing only the minimal necessary set of actions during recovery. Additionally, for two of the three applications on which Phoenix was evaluated there was no perceivable overhead during normal system operation. Perhaps most importantly, Phoenix used no more than 6 KB to log both the internal and external state for the applications evaluated in this thesis, allowing the running program to utilize the vast majority of the memory.
Second, this thesis presented GEM, a memory configuration tool which models memory as a graph in order to facilitate a variety of memory transformations. No such tool exists for any other embedded system; in fact, prior to the advent of embedded run-time systems, building such a tool would have been nearly impossible due to the lack of organization to memory. At a high level, GEM captures the entire memory space of a program, refashions it as a graph, transforms that graph, and then replaces the memory space — of the same or a different device — with an unparsed version of that transformed graph. This same workflow can be applied to numerous use cases, of which four are implemented and evaluated in this thesis: memory visualization, de-duplication, heterogeneous compilation, and transparent migration.

These use cases illustrate the power and versatility of GEM. First, they exemplify the value that GEM brings to a wide range of substantive challenges in managing embedded memory. This thesis has demonstrated how GEM’s interactive visualizer facilitates the identification of bugs and inefficiencies; how its de-duplication capabilities prevent the squandering of precious resources by reclaiming up to 24% of the space consumed by the Python library code; how its heterogeneous compilation widens the range of memory architectures within which the Python virtual machine can fit; and how its migration capabilities allow for the transparent transport of a running program, even amidst incongruities between the source and destination. Second, these use cases demonstrate the ease with which GEM can be used to uniquely synthesize distinct yet complementary functionalities such as migration and de-duplication.

As a proof of concept, both Phoenix and GEM were implemented within Owl, an existing Python run-time system. However, the insights, approaches, and mechanisms presented in this thesis transcend this context. The insights underlying Phoenix characterize fundamental properties of embedded systems and their hardware periph-
erals, independent of the specific system. Its procedure and mechanisms are likewise broadly applicable: apart from minor implementation details, nothing in Phoenix relies upon the Python language, and so the same recovery mechanisms could be implemented within other run-time systems. Similarly, the benefits of modeling memory as a graph in order to apply powerful graph transformations go beyond the specific challenges faced by Owl, and even beyond the challenges faced by embedded systems. In recognition of this, GEM was designed to be portable. Its underlying graph representation is extremely general; moreover, the parser and unparsers which translate between the system-specific object model and this generic graph representation are auto-generated. With only minor modifications to the parser/unparser generator, GEM could easily be adapted to any other managed run-time system with a well-defined structure to memory.

Embedded systems face many challenges, challenges which are amplified by extreme resource constraints and implicit external state. This thesis has utilized managed run-time systems as a vehicle for addressing these challenges at the system-level, thereby alleviating the burden on the programmer. By providing a complete recovery process that addresses the unique challenges of resource-constrained embedded systems, the Phoenix system presented in this thesis is an important step towards improving the future of writing reliable embedded applications. Likewise, by providing a versatile framework for improving the layout of memory via graph transformations, the GEM tool substantially eases the task of memory optimization in embedded systems.
5.1 Future Work

This thesis is an important step towards harnessing the power of managed run-time systems within the resource constraints of embedded systems. It has demonstrated the potential of managed run-time systems to serve as a foundation for solving two particular challenges faced by embedded systems: reliability and memory optimization. Yet, these are by no means the only challenges of embedded systems programming. Best practices for security, concurrency management, energy management, inter-process synchronization, and even debugging are still open questions for embedded systems researchers. In light of the success of leveraging run-time systems to advance reliability and memory management, the most imperative, albeit high level, future research avenue is the design of run-time system techniques to address these other challenging aspects of embedded systems programming.

Further, even the two challenges addressed by this thesis have subtle intricacies and tradeoffs that merit future investigation. Thus, Section 5.1.1 outlines prospective research that builds upon the reliability mechanisms presented by this thesis; Section 5.1.2 discusses potential extensions of the memory management technique proposed by this thesis.

5.1.1 Reliability

There are many opportunities to build upon the reliability mechanisms introduced by this thesis to enable their utilization in a broader range of applications. In particular, this thesis lays the foundation for three categories of future work: conceptualizing and implementing recovery support for additional classes of peripherals, upgrading the run-time recovery mechanisms to address more complex failure scenarios, and developing supplementary microarchitectural mechanisms for improved efficiency.
The optimal policy for restoring historical peripherals is still an open question. The current implementation of Phoenix does not contain specialized mechanisms for restoring these peripherals. Despite this, Phoenix may still be used with many applications involving historical peripherals. For instance, the obstacle tracker and virtual compass applications presented in Section 3.6.2 both utilize a historical peripheral — the TFT display — and work seamlessly with Phoenix. Both applications periodically reset the contents of the display as part of their update process. This ensures that the display will end up in a consistent state despite the lack of special attention.

However, this eventual consistency is an artifact of these applications, and cannot be relied upon in all applications that utilize historical peripherals. Ideally, future work would develop a general approach to managing these historical peripherals that is comprehensive, yet economical. As discussed in Section 3.5.1, the brute force approach to restoring a historical peripheral involves re-executing all past writes — an intractable task for a long-running program, not only in time but in the space required to store a complete history of peripheral accesses.

One possible compromise between the current laissez faire approach and this brute-force approach is a multi-level solution that divides the work between the run-time system and the peripheral access functions. For instance, consider the TFT display once more. The complete current state of the display could be cached in software, independent of the writes that were executed to reach that state. Assuming the ability to write each pixel independently, this would impose a finite bound on the number of writes required to restore the state of the peripheral. The software cache could be written by the peripheral access functions, and later read and reinstated by the run-time system. This approach could be generalized to other historical peripherals, though it may prove more challenging for peripherals that store a larger amount of
state in a less straightforward way, such as a filesystem.

Additional future work could be done at the system level to support especially complex failure scenarios. Though the Phoenix system can, over time, handle a large number of peripheral failures — constrained only by memory and hardware availability — it currently assumes that there will only be one outstanding failure at any given point in time. There are many facets of the recovery process that could be re-examined in the context of multiple simultaneous failures. These include determining the point or points to roll back to, the appropriate time and order in which to restore each peripheral, and the subset of peripherals to maintain in the redo set at each point in the re-execution.

Another potential source of intricate failure scenarios is multi-threading. The infrastructure of Phoenix has already been designed for extensibility to multi-threaded applications; all of the peripheral and recovery metadata is stored on a per-thread basis. However, many of the algorithms within Phoenix do not optimize for multi-threaded programs.

In particular, scheduling would benefit from improvements designed to accommodate multi-threading. Upon detecting a peripheral failure, the current implementation of Phoenix immediately interrupts the execution of the current thread to perform the recovery procedure. In the case of a single-threaded program this is ideal, since the failure must have been triggered by the current thread; however, asynchronicity undermines this guarantee in the multi-threaded case. Future research would be required to develop more sophisticated scheduling algorithms to optimally allocate time between the threads and the recovery procedure. For instance, if the failed peripheral is not used by the current thread, it may be fairer to delay the recovery procedure until control is regained by a thread that uses that peripheral — including but not
limited to the thread that triggered the failure.

The journal brings an additional layer of complexity to scheduling. As the journal has a finite amount of space, the current implementation of Phoenix waits until all outstanding peripheral writes have been acked prior to executing any additional bytecodes; in a multi-threaded program, a better solution would be to switch to a thread that has no dependencies on outstanding peripheral accesses, and therefore could proceed immediately without requiring journal space.

Last, the checkpointing mechanisms presented in this thesis motivate the development of additional architectural support for reliability. The incremental approach to checkpointing presented in this thesis, chosen for its modest space overhead in comparison to a full snapshot approach, is streamlined at the expense of burdening each journaled store with a non-negligible time overhead. As explained in Section 3.4.2, a hardware journal would substantially reduce the overhead of each journaled write. The expected two orders of magnitude decrease in overhead would allow for the use of journal-based checkpointing with a broader spectrum of applications, including those that are more peripheral-intensive.

5.1.2 Memory Visualization and Transformation

The graph transformation framework presented in this thesis could serve as a basis for considerable future work. These primarily fall into three different categories: building new functionalities on top of GEM, expanding the existing functionalities presented in Section 4.4, and applying GEM to new systems.

GEM is designed for extensibility. The four use cases presented in this thesis, though beneficial in their own right, are just examples of the capabilities that can be built upon GEM. Therefore, one potential avenue for future work is building
additional functionalities upon GEM. As an example, GEM could be employed to
patch faulty code at run-time without having to restart execution from the beginning
of the program.

Furthermore, several of the use cases that have already been implemented would
benefit from future work. For instance, the ability to compare snapshots from different
points in a program’s execution would be a valuable asset in memory visualization.
Differences between snapshots could be identified using graph isomorphism analysis,
offering greater insight into trends in memory allocation and the lifespans of objects.

The migration system built upon GEM likewise presents opportunities for further
development. Several of the current limitations on GEM’s migration capabilities arise
from the native function table, an array of pointers to native functions stored in flash.
While GEM can overcome many differences in the contents of flash via its splicing
and splitting techniques, it currently requires that a given native function appear at
the same index at the source and destination, because the compiler encodes native
function calls with an index into this table. Future work could investigate obviating
this requirement. One possible approach is to store the native function table as a
Python object on the heap, so that it can be updated or swapped out upon migration.
Similarly, Owl can be customized by enabling or disabling a variety of different lan-
guage features; however, these features affect the object model, and therefore GEM
currently requires identical language features at the source and destination for com-
patibility. This is an overly strict approach. In many cases, moving to a less featureful
platform could be accomplished by dropping the defunct fields prior to unparsing and
memsetting.

Additionally, several of the insights from Section 3.2.1 regarding external state
apply not only to recovery, but also to migration. In particular, migrating a running
application that uses hardware peripherals would ideally maintain as much of the external state as possible — subject to the physical configuration of the destination device. To accomplish this fact, GEM could be combined with a subset of Phoenix's logging capabilities. In particular, GEM would need to know the identities of all of the peripherals in the system, and the last state each persistent peripheral was put in prior to migrating. Therefore, GEM would require the rematerialization queues. The journal and control flow queue could be elided, as they are only needed in the case of failure.

In Phoenix, the rematerialization queues are stored in a second heap so that they persist past rollback. For migration purposes, they could be stored on the standard Python heap, as rollback is not a factor. This would cause them to be migrated along with the internal program state. However, simply migrating these queues is insufficient; each peripheral would need to be actively restored post-migration. This would require an extension of the interpreter's handler for the \texttt{memset} bytecode. While many of the hardware peripherals may have been initialized and used on the source microcontroller prior to migration, the same may not be true on the destination. Each rematerialization queue holds a reference to the peripheral instance to which it belongs, allowing it to access methods on that peripheral, as needed. Therefore, \texttt{memset} could perform a pass over the rematerialization queues, invoking the initialization function of each peripheral, to ensure that all of the hardware is in a working state. Persistent peripherals would require a second restoration step: for each persistent peripheral, \texttt{last\_state} would need to be invoked with the arguments from the latest rematerialization queue entry.

Finally, GEM's parser and unparsers could be ported to other run-time systems, embedded or otherwise. These systems would likely benefit from many of the same
capabilities built upon GEM for Owl. For instance, the CPython compiler outputs an unpacked object format that could benefit greatly from the de-duplication capabilities presented in Section 4.4.2. Porting GEM to different systems would also potentially facilitate migration between different systems. For instance, Owl and CPython largely share the same object model, with a few differences such as Owl’s addition of packed types. Future work could explore adding an additional conversion layer between parsing and unparsing to reconcile these differences and enable such cross-system migration.

More compelling still is the potential for adapting GEM to modular systems, including those composed of multiple connected microcontrollers as well as distributed systems — embedded and otherwise. While this thesis discusses the use of GEM to optimize the memory of a single microcontroller, GEM could be extended to cooperatively manage and optimize memory across a multi-node system. The domain of embedded systems alone provides numerous applications for such a technology, as many embedded systems encompass multiple microcontrollers; for instance, a typical car contains dozens of microcontrollers [17]. Naturally, expansion to a broader class of distributed systems opens still more possibilities for improved interoperability.
Bibliography


[51] W. Lyons, “Enabling increased safety with fault robustness in microcontroller applications.”


