UNIVERSITY OF CALIFORNIA

Los Angeles

Objects to Bits:

Efficient Implementation of Object-oriented Languages on Very Small Devices

A dissertation submitted in partial satisfaction of the

requirements for the degree Doctor of Philosophy

in Computer Science

by

Benjamin Lawrence Titzer

2007

The dissertation of Benjamin Lawrence Titzer is approved.

David F. Bacon

Babak Daneshrad

Todd Millstein

Mani Srivastava

Jens Palsberg, Committee Chair

University of California, Los Angeles

2007

TABLE OF CONTENTS

TABLE OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor Jens Palsberg for giving me a tremendous number of great opportunities and good advice on how to utilize them over the past five years. He has given me a wide space of freedom to explore my interest, and there are many things that would not have been successes without his patience and guidance. I am also glad to count him as a personal friend. I would also like to thank David Bacon for giving me the opportunity to work at IBM T.J. Watson with him and numerous other talented and bright people. Thanks to Todd Millstein for comments on papers as well as advice on presentations and projects over the past three years here at UCLA. Thanks to Doug Lea for shepherding my OOPSLA paper, which led to important improvements that substantially increased its quality. Thanks to Simon Han for many discussions at UCLA, contributions to Avrora, and for porting the `MsgKernel` application to Virgil. I'd like to give special thanks to Daniel Lee, who implemented many of the initial AVR driver simulations in Avrora, and Olaf Landsiedel, who developed the energy model and framework in Avrora. Undergraduates Keith Mayoral, Adam Harmetz, Ryan Hall, and Akop Palyan all contributed to either Avrora or Virgil. Other contributors to Avrora include Evan Barnes, Jacob Everist, Thomas Gaertner, Jey Kottalam, John Regehr, Bastian Schlich, and John F. Schommer.

During my time at UCLA, I was partially supported by NSF ITR award #0427202 and a research fellowship with the Center for Embedded Network Sensing at UCLA, an NSF Science and Technology Center.

VITA

1980—Born, Hinsdale, IL.

1994—First C program created.

2001-2002—Undergraduate Research, Purdue University.

2002—Bachelor of Science in Computer Science and Mathematics, Purdue University.

2002, Summer—Research Intern, Sun Microsystems Laboratories.

2002-2003—Graduate Student/Teaching Assistant, Purdue University Computer Science.

2003, Summer—Research Intern, Sun Microsystems Laboratories.

2003, Fall—Transferred to UCLA Computer Science Graduate Program.

2003-2007—Graduate Student Researcher, UCLA.

2004—Master of Science in Computer Science, University of California, Los Angeles.

2004, Fall—Teaching Assistant, UCLA Computer Science.

2006, Spring—Research Intern, IBM T.J. Watson.

PUBLICATIONS

[1] Ben L. Titzer and Jens Palsberg. Vertical Object Layout and Compression for Fixed Heaps. In *CASES '07, the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. Salzburg, Austria. September 2007.

[2] Ben L. Titzer, Joshua Auerbach, David F. Bacon, and Jens Palsberg. The ExoVM System for Automatic VM and Application Reduction. In *PLDI '07, the ACM Conference on Programming Language Design and Implementation*. San Diego, CA. June 2007.

[3] Ben L. Titzer. Virgil: Objects on the Head of a Pin. In *OOPSLA '06, the 21st Annual Conference on Object-Oriented Systems, Languages, and Applications*. Portland, OR. October 2006.

[4] Olaf Landsiedel, Klaus Wehrle, Ben L. Titzer, and Jens Palsberg. Enabling Detailed Modeling and Analysis of Sensor Networks. *Praxis der Informationsverarbeitung und Kommunikation*, 28(2):10-15. 2005.

[5] Ben L. Titzer and Jens Palsberg. Nonintrusive Precision Instrumentation of Microcontroller Software. In *LCTES '05, the ACM SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. Chicago, IL. June 2005.

[6] Ben L. Titzer, Daniel Lee, and Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *IPSN '05, the Fourth International Conference on Information Processing in Sensor Networks*. Los Angeles, CA. April 2005.

[7] Ben L. Titzer. Avrora: The AVR Simulation and Analysis Framework. Master's Thesis. Los Angeles, CA. June 2004.

[8] Grzegorz Czajkowski, Laurent Daynes, and Ben Titzer. A Multiuser Virtual Machine. In *The USENIX 2003 Annual Technical Conference*. San Antonio, TX. June 2003.

ABSTRACT OF THE DISSERTATION

Objects to Bits:

Efficient Implementation of Object-oriented Languages on Very Small Devices

by

Benjamin Lawrence Titzer

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2007

Professor Jens Palsberg, Chair

The accelerating digital automation of the world around us has placed increased focus on the problem of developing reliable and flexible software for microcontroller-class devices. Today, microcontrollers serve as the primary or auxiliary processor in products and research systems from microwaves to sensor networks. Microcontrollers represent perhaps the most severely resource-constrained embedded processors, often with as little as a few bytes of memory and a few kilobytes of code space. Language and compiler technology has so far been unable to bring the benefits of modern object-oriented languages to such processors. In this dissertation, I will establish that *advanced language and compiler technology can bring the benefits of object-oriented programming to even the most constrained of embedded systems.*

I present two systems I have developed that significantly advance the state of the art in this area: Virgil, a new lightweight object-oriented language which I have designed with careful consideration for resource-limited domains, and the ExoVM, a system that I built for specializing the IBM J9 Java virtual machine on a per-application basis. These two systems echo three recurrent themes: *pre-initialization*, *reachability*, and *compression*. Virgil explores these themes in a pristine setting where the language, compiler, and applications can be rebuilt from the ground up. To avoid dynamic memory allocation and the resulting system complexity, Virgil introduces *initialization time*, where an application can build complex data structures during compilation. This exposes the entire heap of the program to the compiler and allows new heap-sensitive compiler optimizations: *reachable members analysis* for combined dead code and data elimination, and *reference compression* and *vertical object layout* for more memory-efficient object representation. In contrast, the ExoVM explores these three key ideas in a scenario where the primary goal is to reuse a large-scale industrial strength virtual machine, but reduce footprint by automatically removing dead code in the program and the virtual machine on a per-application basis.

INTRODUCTION

Microcontrollers are tiny, low power embedded processors deployed to monitor and control consumer products from microwaves to fuel injection systems, where often the presence of a computer is not readily apparent. Microcontrollers are gaining increased attention in the research community because they are an ideal fit for sensor networks, where programmability, physical size, and power consumption are important design criteria. A typical microcontroller integrates a central processing unit, RAM, reprogrammable flash memory, and IO devices on a single chip. They have limited computational power and memories that are often measured in hundreds of bytes to a few kilobytes, representing one of the most extreme instances of a software-programmable resource-constrained embedded system. For example, a popular microcontroller unit from Atmel, the ATMega128, has 4KB of RAM and 128KB of flash to store code and read-only data; it serves as the central processor unit in the Mica2 [52] sensor network node.

Microcontrollers allow for software programmability by storing a program's instructions in a flash memory that allows infrequent, coarse-grained updates, usually done only during testing. Software for the smallest of microcontrollers is generally written in assembly language, but medium to large microcontrollers are often programmed in C. Generally there is not enough code space or memory to fit a true operating system that provides both software isolation mechanisms and resource virtualization; thus most programs run directly on the microcontroller without any of the protection mechanisms that are common on desktop computing platforms. This complete lack of traditional software isolation mechanisms complicates the problem of developing robust software for microcontrollers, since programmers cannot rely on the familiar mechanisms by which software safety is achieved on typical desktop systems.

In this situation, static detection of errors is best. Modern programming languages offer two powerful mechanisms for increasing software robustness through static checking: strong types and software verification. Strong types offer two important advantages to programmers: the ability to *express and document* intended invariants on variables and the ability to *enforce* these invariants, limiting each variable to a known (but perhaps unbounded) set of values described by its type. When the type system is expressed as a formal system and the semantics of the programming language are formally defined, it is possible to prove a type system is *sound*, i.e. that a well-typed program will not generate any of a large class of potentially harmful software faults (often referred to as "going wrong"). The compiler will reject programs that do not pass the typechecker and generate an error message that helps the programmer to correct the problem in order to establish the invariants required by its types. Thus, successful compilation of the program indicates that it is free of a large class of potential internal errors. Software verification, however, is the process by which a verification tool such as a model checker establishes that a program meets an *external* specification.

While strong typing helps in many programming paradigms, statically typed object-oriented languages have been particularly successful in the past 15 years. Notable examples include C++ [91], Java [46], and C# [1]. More recently, blending of functional and object-oriented concepts in a statically typed context have been successful in languages such as Ocaml [5] and Scala [3][74]. Despite the demonstrable advantages that these more advanced programming languages and their associated runtime systems offer, the combination of severe resource constraints and need for low-level mechanisms have slowed their adoption in embedded systems. In particular, language implementations still tend to be designed for desktop or server scenarios and often have large, heavyweight runtime systems that include dynamic code loading, advanced JIT compilers, reflection, and serialization, all of which increase both the fixed cost of the virtual machine and the proportional cost of increasing program size. This often makes the total resource consumption of both the VM and application together unacceptable for very small devices.

It is often assumed that the steadily increasing memory capacity of computers across the spectrum will solve this problem, and all devices will eventually have enough memory to run today's software. While Moore's law holds that the transistor density on integrated circuits doubles every 18 months, allowing larger and larger scale microprocessors, microcontrollers and the size class that they represent are unlikely to become obsolete soon. First, the improvements in transistor size through more advanced process technology can be exploited on several dimensions, including transistor count, energy consumption, clock speed, and manufacturing cost. While desktop and server microprocessor manufacturers have focused primarily on transistor count and clock speed in pursuit of performance, microcontroller applications tend to push vendors to concentrate more on manufacturing cost and energy consumption. Secondly, several factors conspire to slow the apparent improvement in process technology at the low end of chip design, including industry consolidation and the increasing cost of manufacturing plants, resulting a lag time of several years before top-of-the-line process technology is employed for embedded chips such as microcontrollers. Thirdly, while microcontroller vendors continue to increase the memory capacity of their high-end models, they still manufacture and sell all of their previous lower-end models, because those chips still meet the needs of both legacy and new applications, since customers will always buy the smallest model that is capable of running their particular application.

1.1. Thesis statement

This dissertation addresses the problem of matching statically typed object-oriented languages to very small devices. My thesis can be summed up in one sentence:

Advanced language and compiler technology can bring the benefits of object-oriented programming to even the most constrained of embedded systems.

This dissertation provides evidence that this statement is true, presenting two systems that I have built which offer significant contributions to the state of the art in developing software in higher-level object-oriented languages for small devices.

Before Virgil, it was thought that building systems software in a strongly typed language was impossible without opening holes in the type system and building a specialized language runtime that underlies all programs, often including significant amounts of unsafe code. These runtime systems tended to be large enough that it was thought to be impossible to develop for microcontrollers. Virgil represents a leap forward in language technology for this domain, requiring no such loopholes in the type system, with a fixed cost so low that it rivals C, even running on the smallest of microcontrollers. Virgil programs compile directly to machine code and require no runtime system; the language provides access to hardware state sufficient to implement all device drivers necessary on a microcontroller. Key to achieving these results is a staged computational model which introduces *initialization time*, a phase during compilation in which the Virgil application can execute arbitrary code to allocate and initialize its data structures. Unlike traditional compilers that focus on code optimizations, the Virgil compiler performs an unprecedented amount of data optimizations. In this case, the strong type safety of the language and the availability of the entire program heap make optimizations easier, allowing the compiler to perform combined dead code and data removal and compression without sophisticated pointer analyses.

Before the ExoVM, it was thought that developing Java applications for embedded devices with limited memories required specialized virtual machines that were specifically designed for small memory footprint. One consequence of that assumption was that embedded VMs have tended to lag behind the state of the art in performance and feature completeness. The ExoVM challenges this assumption and presents a system for reusing an existing large, state-of-the-art virtual machine in smaller footprint scenarios without requiring intrusive modifications. The ExoVM accomplishes this by applying the main themes of this dissertation: it *pre-initializes* the virtual machine, building a complete representation of the program, applies program analysis techniques inspired by Virgil to compute the *closure* of all reachable entities, removing VM data structures and subsystems that are not used by the Java program, and then *compacts* the live data structures into a persistent image file for use at runtime.

1.2. State of the Art

This section describes the state of the art in three categories; embedded object systems, embedded languages, and program data compression technologies.

1.2.1. Embedded Object Systems

Most object-oriented languages use a virtual machine as their primary language implementation. By far the most popular language in this category is Java, but others include C#, Smalltalk, Self, Scala, Python, and Ruby. Virtual machines have important advantages for developing and distributing software, including a compact machine-independent object code format for mobile code safety, fluid integration of dynamic code loading, support for reflective language features, automatic memory management, and increasingly sophisticated dynamic optimizations. For embedded systems, virtual machines can offer a number of advantages. In fact, the original development of the Java language and virtual machine was motivated by the need to develop portable embedded software for TV set-top boxes [4]. Two advantages are that a compact bytecode format can reduce the size of the application's memory footprint and fine-grained software update can be useful in a scenario where the software on the device changes rapidly, particularly during testing and development.

In the early days of Java virtual machines, a complete implementation consisted primarily of a bytecode interpreter, class loader, and simple garbage collector. The simplicity of the system made the fixed cost of the virtual machine small enough to be suitable for devices with small memories, but this simplicity came at the expense of performance and language features. As the Java language evolved and its domain expanded, many of the original advantages were eroded by the increasing complexity of the language and virtual machine subsystems in pursuit of performance and features, usually at the expense of memory footprint. Today, a state-of-the-art virtual machine that supports dynamic class loading, JIT compilation, advanced garbage collection, and the complete Java language specification and accompanying class library spans millions of lines of code and has a memory footprint on the order of several megabytes: simply too large for many embedded devices.

This upward pressure on virtual machine implementations has led to a backlash where a number of specialized VMs with fewer language features and simplified subsystems have appeared. There have been several research systems [27][30][63][97], and a number of standard library subsets have arisen, for example, the Connected Limited Device Configuration [2]. Most of these systems target devices with tens or hundreds of kilobytes of RAM, more than an order of magnitude more than a typical microcontroller.

The Spotless System [97] was developed in 1999 at Sun Labs and later became the K Virtual Machine (KVM). It was one of the first Java virtual machine implementations developed from scratch that was specifically designed for small memory footprint scenarios; the basic RAM footprint was on the order of a few tens of kilobytes.

It achieved this small size by starting from nothing, building the virtual machine in C, and concentrating on a small (rather than full-featured) class library. It had no JIT compiler, but a simple interpreter, and had correspondingly low performance, particularly compared to modern virtual machines, but acceptable in comparison to the mainstream virtual machines of the time. Like many virtual machines that would follow, the Spotless System provided a platform for developing embedded applications, but did not address the problems of writing the actual system software, i.e. the virtual machine and operating system itself, in a safe language. The KVM [2] evolved into a language platform standard dubbed J2ME (Java 2 Micro Edition), whose focus was to define a small number of standard subsets of the Java language and libraries for embedded systems. At that time, it was simply accepted that a language subset definition was required in order to standardize the inevitable number of specialized virtual machines that would appear. The ExoVM presented in Chapter 2 directly challenges this assumption, demonstrating it is possible to reuse an existing, fully featured virtual machine in a small footprint scenario.

Resource limitations have led to a number of systems that explore moving more work, including initialization, compilation, and compression to an earlier offline phase. VM* [63] is a virtual machine construction kit that allows the automatic generation of an application-specific virtual machine, including only the parts of the interpreter and runtime system that are needed. VM* targets very small sensor network nodes, and supports a subset of the Java language. It exposes a native API for operating system and hardware services, which allows simple sensor applications to be written in Java. VM* addresses only the application level development of sensor networks; the underlying operating system services and the VM functionality are all implemented in C. Like many embedded virtual machines, a bytecode interpreter forms the core. The overhead in executing application code is between 50 (`iadd`) and 500 (`invokevirtual`) machine cycles per executed bytecode. The total RAM footprint for the system was between 500 bytes and 2 kilobytes for five applications, while code footprint was between 9 and 25 kilobytes. This makes it suitable for smaller applications on larger microcontroller models, but three problems still exist: poor performance, the virtual machine itself is implemented in C, and it cannot target the smallest microcontroller models. Later work [112] on the VM* system improved performance through precompilation and a network-attached compile server for dynamic compilation based on profiling feedback from applications on the nodes, a technique first demonstrated by Delsart, Joloboff, and Paire in [37], but the other two problems remain.

Several research systems have explored moving some of the work of class loading into an offline phase. Pugh and Weddell [80] describe a number of techniques for reducing class file sizes by compressing and removing redundant information. A more aggressive technique is to build the VM's internal view of classes offline in order to remove to need to parse the standard class file format. For example, later improvements to the KVM added support for Java Code Compact (JCC) technology, which preloads class files against the virtual machine, generating the KVM-internal representation, emitting a C file that is compiled and linked into the KVM executable. JCC also supports manually specifying which of the application classes can be removed in this step.

Courbot, Grimaud, and Vandewalle [30] present a more aggressive system. They describe *romization,* a technique that loads an application into the virtual machine, pre-initializing internal data structures, reducing them, and emitting a C file that contains the Java application with its custom virtual machine. Their system shares important similarities with the ExoVM presented in this dissertation. First, they employ pre-initialization, closure, and persistence, which are the keys to the ExoVM philosophy. Second, they share the view that customizing against a larger, fully featured JVM is superior to building customized VMs. However, they base their system on the Java in the Small [6] project, which is (as its very name implies), designed for small footprint already; they do not base their work on an existing industrial strength virtual machine as the ExoVM described in this dissertation has. Third, they do not describe their closure algorithm in detail; the ExoVM's closure constraint system sheds important light on the interconnectedness of the underlying virtual machine's implementation, teaching us larger lessons and insights into virtual machine design that are identified and described in this dissertation. And finally, the ExoVM persistence mechanisms faced important real-world challenges that are of practical importance and informative to later implementers of these ideas.

Several research projects have focused on simplifying the JIT compiler. Recent work by Gal and Franz [41] has made important progress by concentrating on compiling straight-line bytecode sequences. Their system is based around a standard bytecode interpreter that collects dynamic profile information for bytecode traces. A heuristic selects profitable bytecode sequences to compile. Optimizing these straight-line trace code sequences is far simpler than the general case, simplifying many optimizations and reducing the implementation complexity of the JIT. However, their dynamic compiler still occupies approximately 150kb of code and data, which is too large for even the largest of microcontrollers. In contrast, Manjunath and Krishnan [67] use a simple JIT compiler that compiles basic blocks by concatenating the interpreter's own code for each bytecode in the block in succession and then applying peephole optimizations. Their system occupies approximately 10 kilobytes of RAM. A detailed performance study is not provided, but it is likely the performance is somewhere between a pure interpreter and a very simple, non-optimizing compiler.

While virtual machine research has made important advancements in the state of the art for embedded systems applications, ultimately, virtual machines are an attempt to address the challenges of embedded systems for the *application* level, not the *system* level. Usually, the virtual machine itself is generally either implemented in a lower-level unsafe language like C or C++. Thus many of the problems that plague software in these languages are still present for the systems software. Some attempts have been made to use higher-level languages such as Java for systems development. However, using Java to implement virtual machines and operating system kernels, such as in Jikes RVM [15] or JX [45], generally requires a native compiler that translates Java bytecode to machine code in order to bootstrap. This often requires magic holes in the type system to be agreed on by the system and the compiler in order to implement certain language features. In the end, this provides a convenient Java environment for applications, but significant challenges still remain for building the VM or OS itself.

The Solo Operating System [49], developed in 1976 by Hansen, was perhaps the first to investigate the use of type-safe, structured programming languages for kernel and operating systems development. Solo explored applying Concurrent Pascal to target the PDP/11, developing a kernel and applications together, with a kernel of about 8KB of hand-written machine code. The compiler was part of the system; all programs had input parameter types that were checked at program invocation. Programs communicated via passing argument values, reading from buffers, and reading and writing from a very simple file system. Although extremely innovative for its time, it was a research system that languished in obscurity during UNIX's viral spread. Unlike Virgil, it was developed for an interactive console environment and thus included a fully-fledged (though simple) compiler for Pascal as part of the system to support development of new programs on the running system. Pascal, especially at that time, was a comparatively simple language, and typechecking and code generation techniques were simpler in order to be feasible on the slower machines of the day. In contrast, Virgil has an advanced type system with objects and parametric types. The Virgil compiler is not meant to run on the actual deployed hardware device, allowing it to utilize the computational power of the desktop system on which the Virgil applications are developed and apply more advanced code generation techniques.

The Singularity Project [51] at Microsoft Research is a recent attempt to build a complete desktop operating system primarily in C#, eliminating as much unsafe code as possible. In developing a modern operating system kernel with a safe language, there are several challenging problems that potentially require unsafe code. For example, implementing virtual memory requires manipulating MMU mappings, context switching requires saving and restoring processor state onto and off of program stacks, and locating certain memory-mapped devices may require scanning raw memory. The garbage collector implementation potentially requires some unsafe code and special magic tunnels under, around, or through the type system. The result is that there is still a large amount of unsafe code.

Embedded object systems can also offer embedded language features as an API or library on top of an existing virtual machine. For example, Eventrons [88] are a programming construct introduced in the context of the IBM J9 virtual machine that represent Java tasks with additional language restrictions. Specifically, an Eventron is forbidden from reading or writing mutable reference fields, allocating memory, or attempting locking operations. These tight restrictions enable the Eventron to safely preempt the garbage collector and achieve a response time that is shorter than the minimum GC pause time. A runtime verification phase (performed after the Eventron object is constructed but before execution) enforces the language restrictions. Given the fully constructed Eventron object within a Java heap, the verifier discovers the methods and objects reachable from the `run()` method, checks that each method obeys the language restrictions, and then pins the reachable objects in memory. While an important advance for responsiveness, Eventrons are too restrictive because they forbid mutation of reference fields; they also still require a large runtime system support and therefore wouldn't be suited to developing the actual VM or operating system in a safe language.

1.2.2. Embedded Languages

C++ is often cited for its suitability to writing low-level code, but the language and its implementation have a number of drawbacks [85] that make targeting a microcontroller difficult; primarily the complexity of the object model, the inefficiencies of certain language constructs, the runtime and metadata requirements, and a lack of strong safety mechanisms. C++ lacks the strong type safety guarantee given by Virgil, and thus some kinds of optimizations for C++ cannot be made sound, such as reference compression, which is described in Chapter 4 of this dissertation. Despite promises of its adherents, C++ has not succeeded for microcontroller class systems.

NesC [44] is an extension to C that adds module capabilities and a simple task model that is based on two-way interfaces which can be configured by “wiring” them together in a configuration language. NesC inherits C’s weak type system and therefore is subject to a number of potentially dangerous and difficult to diagnose program errors such as out-of-bounds memory accesses causing memory corruption, stack overflow, etc. The primary focus was to simplify configuration of programs through the module system, but these module capabilities are mostly orthogonal to the deeper language issue of safety and expressiveness, especially in regard to objects, which nesC does not provide. The task model is hardwired into the language and implemented underneath by a significant amount of TinyOS-specific C code. The core language does not provide for allocating memory, although it’s possible to link in `libc` and use `malloc()`. Applications and modules are expected to statically allocate the memory that they require, but complex initialization routines are not possible, and a significant amount of code can be required at device startup.

In the 1970's, Charles Moore designed Forth [7], a minimalist interpreted language that has become popular for programming embedded systems. Forth is stack-based and untyped, exposing the stack to the programmer with its reverse polish notation. There are many free, reusable implementations available. In fact, its philosophy encourages building custom, application-specific implementations and extensions, rather than providing a one-size-fits-all library or compiler. However, its lack of type safety and constructs for information hiding, data encapsulation, and modularity limit its use in developing larger systems.

1.2.3. Program Data Compression

In addition to the previously described techniques that seek to remove or reduce the static program or virtual machine size, researchers have also investigated a number of compiler and runtime techniques for reducing the memory footprint of the program's data structures. Individual program quantities such as pointers, object headers and primitive values can be subjected to compression, reducing the average or worst case memory footprint of the heap. Compression of the heap can sometimes help performance as well. For example, Mogul et al. [68] observed in 1995 that pointer sizes could affect performance significantly on a 64-bit computer because larger pointers occupy more space, putting greater stress on the memory system, affecting cache hit ratios and paging frequency.

Some compiler techniques are purely static. Cooprider and Regehr [31] use static analysis to analyze C programs with statically allocated heaps. Their technique uses abstract interpretation and a simple pointer analysis available with the CIL framework [72] to compute bounds on the range of values for every variable in the program. A source-to-source transformation packs scalars, pointers, structures, and arrays into fewer bits using a compression-table scheme. Lattner and Adve [65] use static analysis to convert and compress 64-bit pointers to 32 bits on a per-data structure basis. Their technique extends their earlier work on static pool allocation [64] that uses sophisticated context-sensitive, field-sensitive pointer analysis to assign data structures to pools.

While traditional static compilers do not have the complete heap, dynamic techniques implemented in the language run-time system can track all objects that have been created and use the information to dynamically compress pointers. Some research systems exist that employ dynamic techniques, sometimes assisted by hardware. The simplest is perhaps that described by Adl-Tabatabai et al. [4], which represents 64-bit pointers as 32-bit unsigned offsets from a known base; the result is a significant performance improvement, despite additional instructions.

Many virtual machines employ compression techniques as well, usually on object headers. In Java, object headers contain such data as type information, a hash code, and a lock, which can often occupy two or three words of space. Bacon, Fink, and Grove [20] presented compression techniques that allow most Java objects to have a single-word object header. Full object compression is possible as well. Chen et al [28] study dynamic heap compression techniques in an embedded Java VM setting, based on the KVM [97] reference implementation of the Connected Limited Device Configuration (CLDC) [2], which is intended for use on devices that have at least 192KB of RAM. The center of their system is an enhanced garbage collector system that compresses objects when a traditional compacting collector cannot recover enough space for new objects. Their collector employs several techniques for tuning the tradeoff in execution time versus memory size. It uses a simple and fast compression algorithm that operates on the raw memory values of objects, without considering the types. Dynamic checks are required so that compressed objects are decompressed upon their first use.

The hardware and operating system can also assist in reducing program data size, particularly pointers. For example, Zhang and Gupta [115] use special hardware instructions to help compress pointers and integers on the fly; they use profiling information to guide what data should be compressed and when compression should be done. Wright, Seidl, and Wolczko [114] present a memory architecture with hardware support for mapping object identifiers to physical addresses, thereby enabling new techniques for parallel and concurrent garbage collection; such an architecture could support compression of pointers as well. Wilson [111] supports large address spaces with modest word sizes by using pointer swizzling at page fault time to translate large pointers into fewer bits.

Ananian and Rinard [17] present a suite of both static and dynamic techniques to reduce data size for Java programs. They propose *field reduction*, a whole-program static analysis that bounds the ranges of values that primitive fields may contain over any execution in order to reduce their size, *unread and constant field elimination*, that removes unused and constant fields, *static specialization*, which eliminates fields that are constant by subclass, *externalization*, which removes frequently constant fields and puts them in a hash table, and *class pointer compression*, which is essentially a single compression table for object headers. Most of these techniques are subsumed by or are complementary to techniques described in this dissertation.

Tip et al [98] present Jax, a system that employs a number of static techniques for reducing Java programs, particularly their static footprint in terms of code and metadata size. Among these are *name compression*, which replaces long class and method names with shorter ones, *class merging*, which reduces the total number of classes, *dead code elimination,* which removes unreachable methods, *dead field elimination*, which removes unused and write-only fields, and devirtualization of method calls for removal of associated metadata. Name compression is unnecessary for Virgil because class and method names are not included in the compiled program. Class merging is also unnecessary because the Virgil compiler will not generate any metadata (including meta-objects) for classes that have no objects remaining in the live heap after initialization. RMA also removes all dead methods and dead or write-only fields, inlining constant fields with their actual values (which are available after initialization, but may not necessarily be available to Jax). Like Jax, it also performs devirtualization and removes all metadata for methods that are completely devirtualized. However, Jax employs a form of flow analysis in its call graph construction algorithm, which is potentially more precise, whereas RMA uses a type-based approach with heap liveness information.

Agesen and Ungar [10] in 1994 described a system for extracting Self applications from their environment. Their system extends previous work by Agesen on type inference for Self by adding the ability to remove dead slots from objects and persist them to disk, solving a similar problem to RMA. However, because Self is dynamically typed, their extractor requires type inference in order to give a sound approximation of possibly accessed slots of objects. RMA, however, simply uses the static, declared types of objects. Also, unlike RMA, the three phases of type inference, computation of live object slots and selection of live objects are distinct and are not iterative, therefore it is possible that type inference could become more precise after the first iteration. In contrast, RMA discovers reachable code, objects, and methods on demand in a single phase. Their experimental results showed their system could reduce image sizes by a factor of 10, but the resulting images are still more than 300kb in size.

1.3. Three themes

Though Virgil and the ExoVM have different views of code reuse and runtime infrastructure, they nevertheless echo three recurrent themes: *pre-initialization*, *closure*, and *compression*.

The first key theme is *pre-initialization*, where data structures are constructed and initialized in an offline phase, which exposes them to compiler analysis and optimization. Unlike traditional compiler problems where the unavailability of the heap and the inevitability of the halting problem forces analyses to consider approximations of the possible runtime data structures of the program, pre-initialization exposes a rich new realm where analyses can use not only the code of the program with an approximation of the heap, but actual data *instances*, in its optimization. This theme of pre-initialization is expressed in Virgil directly in the concept of initialization time, whereby an application allocates its entire heap during compilation. The further restriction of forbidding dynamic allocation means that the compiler has a complete picture of the *entire runtime heap* during compilation. The ExoVM system clearly echoes the pre-initialization theme with its approach of first loading the application program into the unmodified virtual machine, causing the VM to build its internal representation of the program, then resolving inter-class references, and then further initializing the application itself by executing the static initializers of its classes. This pre-initialization phase of the ExoVM allows the further analysis phases to have a complete picture of the VM's data structures and an initial picture of the application's heap.

The second key theme that is echoed in these two systems is the concept of *closure*, or computing an approximation of the set of program entities reachable over execution, both code and data. Virgil introduced this idea in the *reachable members analysis (RMA)* optimization that simultaneously computes a closure over live objects in the pre-allocated heap and the code of the program. This optimization idea appears again in the ExoVM, where closure over the VM's initialized data structures, application objects, VM code, and application code is computed simultaneously. We can consider the ExoVM closure process to be an extension of RMA that works over not only application entities, but also virtual machine entities. This idea, while implemented in separate systems, shares the same core reachability algorithm that can be posed succinctly as subset constraints.

The third key theme that is explored in this dissertation is *compression*, where the typical representation of an application quantity is replaced by a more space efficient (but perhaps slower) representation in order to reduce the total memory footprint. The Virgil compiler performs compression on object references using two techniques: table-based compression and compressed vertical object layout. Both techniques exploit the type-safe nature of Virgil to represent references in fewer bits than would be required in the familiar pointer representation of object references. While the ExoVM system does not employ compression directly, it does however *compact* the virtual machine's internal data structures, which are normally allocated in pools and arenas that can be subject to both internal and external fragmentation. Compaction in the ExoVM reduces the internal fragmentation and eliminates external fragmentation, thereby saving overall memory footprint.

1.4. Two Systems, Two Perspectives on Reuse

The exploration of these two systems in this dissertation offers complementary views of the three main themes from two distinct perspectives on code reuse. The ExoVM takes the perspective that we would like to reuse as much infrastructure as possible that has been invested in developing state-of-the-art language implementations. In the case of Java, this means reusing an industrial strength virtual machine that includes advanced compilation and garbage collection technology. Such systems can be millions of lines of code and represent hundreds of man-years of effort, which should be reused if possible.

1.4.1. The ExoVM

As discussed previously, the pressure for more language features, libraries, and performance for Java has led to a vast increase in the size and complexity of virtual machines with a concomitant increase in memory footprint. The traditional approach has been to build customized virtual machines, often entirely from scratch, that trade off feature completeness or performance for memory footprint.

This traditional approach has important disadvantages. First, though a small VM is comparatively less engineering effort than a fully featured one, software development and maintenance effort is inevitably duplicated as engineers spend valuable resources implementing and tuning subsystems that could likely be reused if they were originally designed to be suitably modular. Secondly, improvements in the state of the art in implementation technology, which usually first appear in mainline virtual machines, cannot be automatically utilized in the custom VM. Thirdly, evaluations of research ideas and implementation techniques inevitably have narrower scope because results are not immediately comparable across domains that do not share a common virtual machine infrastructure.

Unlike all previous work on state-of-the-art embedded virtual machines, The ExoVM approach is to *reuse* an existing industrial strength virtual machine and specialize it to a particular application by *pre-initializing* the virtual machine with the target program. In order to make the system more amenable to automatic specialization, the ExoVM limits the dynamism of both the program and the virtual machine, making them as *static* and predictable as possible. This makes it easier for an automatic program analysis to *include* only what is necessary from the virtual machine on a per-program basis.

The starting point of the ExoVM system is a research configuration of the J9 Java virtual machine from IBM, a complete industrial-strength JVM implementation and its associated Java class library. The ExoVM assumes a closed world scenario, which allows the program and its VM implementation to be viewed in a more static way. Loading the entire application into the unmodified virtual machine allows the first step, pre-initialization, where all of the internal virtual machine data structures are created offline. The next step is to employ *feature analysis*, a program analysis technique that computes the closure, or the set of reachable application and virtual machine entities that may be accessed over any execution of the program. The reachable entities are then copied and relocated to an image file that serves as a completely pre-initialized snapshot of the virtual machine with respect to that particular program. At runtime, the image file can be loaded into a specialized booter VM that is derived from the mainline VM by removing the dynamic resolution mechanisms and unnecessary subsystems such as the class loader and byte code verifier.

The ideas embodied in the ExoVM and its prototype implementation represent a significant step forward in virtual machine construction in general, and the application of virtual machine technology in embedded systems specifically. The ability to measure the modularity of the virtual machine with respect to the source language using feature analysis opens up new possibilities for building more modular virtual machines in the future. The ExoVM has the potential of closing the gap between embedded virtual machines and mainline virtual machines by allowing a single infrastructure to span both domains, reusing important VM subsystems such as the JIT compiler and garbage collector.

1.4.2. Virgil

Virgil takes a radically different perspective on code reuse and considers the implications of redesigning the language and compiler from the ground up, free from the restrictions of any particular language feature or implementation. In contrast to the ExoVM, which attempts to reuse an existing virtual machine, Virgil approaches the problem of matching objects to microcontrollers from the language and compiler level, which allows the additional freedom to add or remove language features in order to meet the constraints of the microcontroller domain.

Unlike previous work on embedded virtual machines, which only address application-level programming on embedded systems, Virgil is intended to address the challenges of developing systems software at the lowest layer, without requiring system designers to bootstrap a state-of-the-art virtual machine. Virgil is intended to allow both applications and operating systems to be developed in one language, without any supporting software. This is appropriate for the microcontroller domain, where applications are generally written to run in a standalone manner. Here, the simplicity of the hardware devices, the lack of virtual memory, and the one-stack, single-process model allows Virgil programs to be implemented completely in safe code and compiled directly to the bare hardware with no runtime system.

Virgil represents a unique point in the design space of languages for the embedded domain because it is both freed from the burden of legacy operating system and driver code yet constrained by the limitations of microcontroller devices. Virgil starts with the assumption that it will be used to build entire software systems from the ground up, in one language, without the need to interface to existing code in unsafe languages like C. Having the entire system in one safe language affords more complete control to the compiler to make efficient representation choices for objects and object references since it is not constrained by any external code and is only constrained by the hardware. This lack of legacy code also eliminates the need for a mechanism to interface between the two realms and potentially increases the robustness of the system.

While the potential language design space is large, memory limitations are so severe on microcontrollers that Virgil is forced to avoid those language features that have large runtime cost in order to fit on the tiniest of devices. In fact, Virgil is designed to require no runtime system or library of intrinsic classes. It jettisons many language features that are common in object-oriented languages such as first-class metadata ("class" objects), a universal super-class (`Object` class), and reflection because their space costs are too high. All of Virgil's features are designed to make the proportional cost of using Virgil's features as low as possible so that a small change in the program size produces a small change in its footprint in a predictable manner, which increases programmers' ability to make intelligent resource tradeoffs.

Memory allocation during runtime on the device is forbidden, removing the need for a garbage collector with its associated performance and footprint overheads. Instead, Virgil introduces initialization time, where programs can allocate and initialize objects during compilation for use at runtime. After initialization time, the entire runtime heap of the program is available to the compiler before generating code. This allows the compiler to ensure that memory is not exhausted and provides an opportunity for novel heap-sensitive optimizations. The Virgil compiler employs reachable members analysis, a sophisticated dead code and data elimination, and several techniques for reference compression that represent object references in a more compact way in order to save RAM.

1.5. Organization

The remainder of this dissertation is organized into three main chapters that describe in detail the systems I have built and their contribution to the field. Chapter 2 gives the technical development of the ExoVM system for J9 and corresponds closely to the paper published in PLDI 2007 [102]. Chapter 3 discusses the design of the Virgil language and the reachable members analysis optimization, containing most of the material from the OOPSLA 2006 paper [101] with added sections on raw and parametric types, as well as a more thorough experimental evaluation with new benchmarks drawn from Virgil driver code. Chapter 4 focuses on compression optimizations and includes some material from the OOPSLA 2006 paper as well as all of the material in the recent

CASES 2007 paper [104], which introduced vertical object layout. Chapter 4’s experimental evaluation mirrors that of Chapter 3, using the same benchmarks from the recently completed Virgil drivers. Chapter 5 gives the overall conclusion.

2. THE EXOVM

This chapter describes the details of the ExoVM system, which exists in a scenario where we would like to reuse the investment in a state-of-the-art virtual machine for an embedded device with a smaller memory footprint. The ExoVM establishes part of the thesis by showing that *advanced virtual machine technology can be reused in more resource-constrained embedded systems.* This chapter shows that the main themes of this dissertation, particularly pre-initialization and closure, help to achieve this goal when applied to an unmodified virtual machine.

The Java programming language and platform offer compelling advantages for a large class of embedded systems applications, including cross-platform compatibility, mobile code safety, automatic memory management, and a compact code distribution format. However, embedded Java virtual machines lag significantly behind the state of the art, particular in terms of performance and garbage collection technology. One of the primary reasons for this is that embedded virtual machines are often developed independently of standard "mainline" implementations and focus on small memory footprint rather than performance or feature completeness. Most embedded virtual machines have a completely separate source code base that is maintained independently, although some larger virtual machines have clumsy and ad-hoc configuration mechanisms that allow coarse-grained removal and replacement of subsystems.

The ideal solution to the disparity would be a single VM implementation that spanned both domains. In theory, such a system would be able to seamlessly adapt between server situations with long-running, massively parallel and enormously memoryintensive tasks, desktop situations with focus on user interactivity and media, and embedded scenarios with limited footprint and even real-time requirements. We believe this is a grand challenge in virtual machine construction that has not yet been solved, indicated by the proliferation of single-domain systems which each have their particular strengths and weaknesses.

The ExoVM approaches the embedded virtual machine problem from a new perspective and contributes positively to this grand challenge by applying the three main themes of this dissertation. Instead of building a new, specially designed virtual machine, we attempt to reuse an existing industrial strength virtual machine (IBM's J9 virtual machine as the case is here), thus benefiting from the development effort already invested in building and tuning the "mainline" virtual machine. Our work highlights important deficiencies in the system we decided to adapt to our purposes and gives important lessons for virtual machine design in the future. Our approach is to apply *pre-initialization* to build the initial program representation within the VM and resolve references, *closure* to compute the reachable entities over any execution of the program, and *persistence* to compactly store the live data structures for use at runtime. The result is a vast reduction in the fixed cost of the virtual machine and sizeable footprint savings.

We based the ExoVM implementation on the CLDC 1.1 MT version of J9 and additionally included some minimal Java reflection support that is required to implement ExoVM pre-initialization and closure computation, thus our configuration does not precisely correspond to any particular IBM product. We studied two variants of this VM:

one using the CLDC class library (j9cldc, approximately 190kb), and another using a much larger class library that approximates the J2SE 1.4 (j9max, approximately 1.6mb).

2.1. Overview of the Technique

The first step of the ExoVM is to *pre-initialize* the unmodified virtual machine by loading the target application using the virtual machine's own internal class loading mechanisms. This does not require intrusive modifications and causes the virtual machine to initialize itself and build its own internal data structures as well as data structures that represent the program, including its classes, methods, threads, etc. Part of this step is to resolve all internal and inter-class references that would normally be resolved dynamically while executing the program. If we assume a closed world scenario where all the code of the application is available, the pre-initialization phase can resolve all references statically and build a representation of the entire program, allowing the metadata that is associated with lazy resolution of references (as well as the mechanisms themselves) to later be removed.

The second step of the ExoVM is to compute a closure over this pre-initialized virtual machine, including entities from both the internals of the VM and from the Java program itself. This closure may contain live VM data structure instances, Java objects, methods, and VM code. This echoes the second main theme of this dissertation, to use program analysis techniques to compute the set of reachable entities over any execution of the program. The computation of the closure relies on an approximation of the runtime behavior of the program, since the closure must identify all entities that might be accessed during any execution. In particular, the analysis needs to relate Java-level operations and entities to VM-level operations and entities. To accomplish this, we apply *feature analysis*, which uses a constraint system that relates Java language features to their implementation in the J9 virtual machine.

The third step of the ExoVM is to persist the closure of data structures and Java code that was identified in the feature analysis step into a compact, ready-to-use image. This closure may contain Java methods, classes, and objects, as well as virtual machine data structure instances and native methods. We developed a system that understands the layout and composition of all important VM data structures and is able to copy and relocate actual data structure instances from inside the virtual machine's internal memory to a specialized image file. The image file contains a complete, ready-to-go virtual machine snapshot that has the target program loaded into it and contains only the data structures necessary to execute that particular program.

The last step, runtime, is accomplished by using a customized "booter" VM that is derived from the unmodified virtual machine by simply removing the dynamic loading and resolution mechanisms, virtual machine initialization routines, as well as other unnecessary subsystems like the bytecode verifier. When mated with an image file that contains all of the data structures necessary, ready and initialized, the booter VM can simply memory map the image file and begin executing the Java program beginning at `main()`.

2.2. Fixed and Proportional Costs

The dynamic memory footprint of a Java application is comprised not only of its own code and data, but also that of the virtual machine and class libraries. We can classify the memory usage into two main quantities: a *fixed cost* and a *proportional cost*. The fixed cost corresponds to the code and data of the VM that is independent of the application, such as a garbage collector, runtime class loading mechanism, interpreter, JIT compiler, etc. The proportional cost corresponds to program's code and heap—e.g. the internal representation of its classes, bytecodes, dispatch tables, compiled code, object type information, method exception tables, Java objects, etc. Like the VM, the Java class library has both proportional and fixed costs, since many core classes are needed for any program and others are only loaded as necessary, though the exact breakdown is not clearly delineated or typically well-understood.

For many embedded applications, the fixed cost of the JVM runtime system and its data structures may dwarf the size of the application. For example, the j9cldc VM executable has more than 600kb of native code, 40kb of static data, and 190kb of Java classes, while none of the 6 EEMBC benchmarks (Section 6) requires more than 120kb for its own class representations, and 5 of 6 execute successfully with a heap less of just 128kb. In this case, it is most important to reduce the fixed cost of the virtual machine. However, for larger applications, the fixed cost of the VM becomes amortized, and eventually the proportional cost will dominate. If we were to envision an ideal solution, we would want the fixed cost to be as small as possible to avoid penalizing small programs, and we would want the proportional cost to be directly related to the size and characteristics of the application so that simplifications and reductions of large programs produce predictable reductions in total footprint.

Our insight is that the virtual machine can be divided into more fine-grained pieces of functionality that can be related to features in the Java programming language, and that the fixed cost of a state-of-the-art virtual machine is not as fixed as previously thought. Dividing the VM along feature lines allows costs that were previously fixed to become proportional to the feature usage of the program. Automated program analysis can then produce the set of features used in a particular application and therefore allow a customized Java VM with a smaller fixed cost to run the application.

2.3. Pre-Initialization

Many large programs have complex initialization routines that build data structures for use throughout the life of the program. In the case of a Java virtual machine, there are data structures to represent and manage the program and the program's state, including threads, Java classes and methods, locks, the garbage collector, JIT compiler, the Java heap, etc. The insight of pre-initialization is that these complex, often long-lived data structures that are normally built at the beginning of the program execution can instead be built offline and saved for use when the program begins execution. If the data structures are pre-built, then the code for the initialization routines can simply be removed, saving both in memory footprint and startup time.

We began studying our unmodified virtual machine's startup routines with the intention of building a separate pre-initialization phase that would build the initial data structures offline and save them. However, we soon discovered that the mechanisms that build and maintain internal data structures both at startup and throughout the execution of the program (e.g. resolving and loading a class) were simply too complex to replicate for our purposes. A far more elegant solution is to simply reuse the existing initialization routines by running them without modification until they reach a consistent state, and then taking a snapshot of the resulting data structures.

The ExoVM system implements this solution by loading the program into the fully featured virtual machine using the standard startup and loading routines already built into the internal API. This naturally and nonintrusively causes the virtual machine to initialize itself to a state that is ready to begin executing the program. In particular, the VM has already built the internal representation of the first of the program's classes and methods as well as parts of the class library. The initial threads data structures are allocated, and some Java objects have been created as a side effect of resolving some string constants. Important Java classes needed in the internal implementation of certain language features are already resolved. Thus the ExoVM analysis system has a complete picture of the initial data structures that are required to begin executing the program. Normally, the virtual machine would dynamically resolve and load new classes, but the closed world assumption of pre-initialization allows the VM to load all of the application classes over any execution. Therefore the VM has built the internal representation of all of the application classes and the class loading mechanisms are no longer necessary at runtime.

2.3.1. Class Initializers

In Java, a class may define an optional *class initializer* (also called a static initializer), a static method that is executed upon the first use of the class while the program is executing. While lazy initialization gives rises to some semantic problems such as nondeterminism in initialization, exceptions in initializers, cyclic dependencies, and dynamic incompatible class change exceptions, the dynamic resolution of class, method, and field references in Java code has definite implementation costs. First, it requires that the constant pool references include the metadata needed for dynamic resolution, including the string names of methods, fields and classes. Second, dynamic resolution may trigger class loading and initialization. Third, the VM must also maintain more metadata for every declared class, field, and method in anticipation of new references to them in the future. Fourth, resolution mechanisms inevitably include hash tables and other such fast search data structures that consume space.

While dynamically loading application classes may reduce the average case footprint for some applications, we consider dynamic resolution and initialization of classes as unwarranted complexity and resource consumption, leading us to explore the implications of changing the model according to our original design philosophy of making the program more static. Therefore, the ExoVM aggressively executes all class initializers for the live classes of the program and resolves all constant pool references to classes and methods as part of the pre-initialization phase.

Changing the model has advantages as well as disadvantages. First, it ensures that class initializers will not need to be executed at runtime, which allows their code to be removed. Second, no dynamic resolution of class, method, or field references will occur, so the metadata that is needed for dynamic resolution can be removed, and the mechanism can be removed from the VM. Third, this allows a program written with the model in mind to pre-allocate needed data structures in its static initialization routines, which are discarded before runtime, yielding a staged computation model closer to Virgil's initialization time, which is discussed in Chapter 3.

One disadvantage of this approach is that it subtly alters the semantics of Java's class initializers, which some programs may depend on. Also, eager initialization could trigger the execution of routines that might not be triggered at runtime, which might allocate large data structures that waste space, destroy the state of other classes, and generally interact in unintended and unpredictable ways. However, we believe that most programs for this domain do not depend on the order or laziness of initialization. For example, in the EEMBC benchmark suite, only one program, `Parallel`, appears to do significant computation in its class initializers. This initializer does not depend on other classes, but simply allocates and initializes a static matrix of data that is used during the benchmark. Moreover, we believe that the closure technique described in the next section will automatically remove many data structures that are allocated by the initialization phase but are unused at runtime.

2.4. Closure and Feature Analysis

To ensure the smallest possible program footprint, we would like to automatically compute the smallest set of classes and methods that are reachable over any execution of the program. There are a number of whole-program techniques to address this problem, including RTA [21], CHA [32], RMA [101], and flow analyses such as 0-CFA, as well as whole-module analyses such as that used in Jax [98]. All of these techniques share a common conceptual approach to the problem, beginning at some entrypoint method(s) in the program and building a static call graph that approximates the reachable code in the program. Typically a closed world assumption is made, allowing code that is not reachable to be safely removed, but if an open world is assumed, constraints can be added to prevent unsafe removal of possibly live code while still allowing for some dead code to be removed (e.g. unused private members).

In the ExoVM system, we must compute reachability over not only classes and methods in the Java program, but over the initial Java heap as well as the data structures and code in the virtual machine. Our analysis builds on both RTA and RMA and extends the class of whole-program, closed world techniques that include live heap objects in the analysis. While RMA, which is described in Chapter 3, computes closure over a complete Virgil program, which does not require a runtime system, the ExoVM requires three new types of constraints that relate entities at the Java level to runtime entities at the virtual machine level.

2.4.1. Feature Analysis

Feature analysis extends the traditional approach of analysis over program entities to include analysis of entities that are the explicit implementation of language features within the virtual machine. We will use the term *entity* to refer to a single data structure instance, Java object instance, Java method, string constant, or VM native method that consumes either code or data space. Our analysis makes entities in the virtual machine explicitly analyzable and will only include entities in the final program image if they are used over any execution of the program. In discussions of programming languages, *feature* is perhaps the most loosely used term and most poorly defined concept. In order to be more precise in our discussion, we will use the term *feature* to refer to the members of or operations on entities.

Once we restrict our attention to entities and features that have an isolatable implementation in the virtual machine, we can reason more concretely about the language in terms of these implementation artifacts. For example, a large, coarse-grained service might be garbage collection. By studying its implementation, we can break this service down into a small, well-defined set of entities and features that require metadata about classes, objects, methods, and threads. Another example might be the `getClass()` method in `java.lang.Object`, which allows inspection of the run-time type of an object. This feature also has an identifiable implementation which accesses the object header and exposes a representation of the class to the program when it calls this method, both of which can be modeled as features. Another example is the use of the `Class.forName()` static method; this method's implementation requires the VM to have a mapping between string names and class representations, as well as the ability to search for a class if it is not already loaded. If the program does not invoke this method at any point, then the data structures corresponding to implementing this feature can be removed. Other, finer-grained examples are floating point arithmetic, explicit casts, synchronization operations, weak references, JNI, reference arrays, static initializers, and exceptions.

Many features correspond almost directly to Java bytecodes (and therefore source-level Java language features), and some correspond to Java library methods and classes. But internal virtual machine features become apparent after some study of the VM implementation, such as the ability to search for a method by its name in a particular class, or to resolve constant pool entries. Most such internal features do not have a direct language expression but are demanded by the implementation of other features. For example, the ability to search for a class by its name is necessary for the VM to resolve some internal Java classes such as those representing language-level runtime exceptions.

The key idea behind feature analysis is that by exposing all of these VM data structures as first class entities in the closure process, just like Java classes, objects and methods, the analysis of language features can be expressed as relations on members and features of these entities. The problem of computing reachability over entities then becomes analogous to the familiar notion of reachability over heap objects; an entity is only reachable if it is referred to by another reachable entity through a feature. If an entity is not reachable through a chain of feature uses in the program and the virtual machine, then it is not used during any execution of the program and can be safely removed.

2.4.2. Constraint-based Analysis

Constraint-based program analyses separate the specification of a correct solution to a program analysis problem from the implementation of the algorithm that computes the best solution [11]. For example, in a program analysis problem such as flow analysis or pointer analysis, the primary goal is to compute sets of program quantities, such as *"what variables may this pointer refer to over any execution of the program?"* or *"what* *method implementations are reachable at this call site in the program?"*. Constraint-based analyses usually have the property that there is always a default, correct, but overly conservative solution such as *"this pointer might point to anything"*. The art of getting a good and verifiably correct solution to the analysis problem is deriving a rule set that describes the minimal properties of a correct solution. Typically, the analysis inspects the program once and generates a complete constraint system that is fed as input to a general constraint solver. The constraint solver then computes the least solution to the constraints, giving the most precise answer.

2.4.3. Entities and VM Types

The overall goal of our analysis is to compute the set of all live entities needed to implement the program, both at the Java level and at the VM level. In our analysis, each entity has an associated type, each type has an associated live entity set, and an entity and is considered live if it is contained in its type's set of live entities. The overall solution is the union over all types of live entities. Our analysis models Java-level entities such as methods, classes and objects in a manner that is similar to RMA. To simplify the constraints, Java methods with implementations have type `method`, classes have type `Class`, and each object instance's type is its dynamic Java type. Note that each of these Java-level entities may have one or more associated VM-level entities, not all of which may be ultimately considered live.

In addition to the Java types from the program, our implementation models 24 different VM data structure types that are listed in Figure 2.1. Among these types are: `VMNative`, which models the native code implementations of Java methods such as

`Object.hashCode()`; `VMClass`, the in-memory representation of a Java class; `VMMethod`, the in-memory representation of a method; `VMROMClass`, the on-disk and in-memory representation of the read-only portion of a Java class such as string names, the constant pool, declared methods; `VMThread`, a representation of a Java thread; and the all-important `VMJavaVM` data structure, which contains pointers to important classes, the heap, collections of classes, threads, and at least a dozen other subsystems.

```
VMNative                VMStackWalkState
VMJavaVM                VMHashTable
VMClass                 VMMemorySegment
VMArrayClass            VMMemorySegmentList
VMClassLoader           VMPortLibrary
VMROMClass              VMThreadMonitor
VMMethod                VMJavaLangString
VMROMMethod             VMJavaLangThread
VMConstantPool          VMInternalVMFunctiona
VMROMConstantPool       VMMemoryManagerFunctions
VMITable                VMInternalVMLabels
```

Figure 2.1: List of modeled VM Types.
Native data structures are modeled in the analysis, and each has its own set of features. For example, the VMNative entity type models implementations of Java native methods from the class library that are supplied by the VM.

Each pointer field within a native data structure is modeled as a feature. This allows fine-grained precision in the analysis of the data structures of the VM. Our analysis models dozens of features for these types; space limitations preclude a complete list.

2.4.4. Constraint Sets

Our constraint formulation uses two kinds of sets. The first kind of set, an $\mathbf{E}$ (entity) set, contains live entities such as Java objects, VM data structure instances, or Java method implementations. For example, for a Java class `C`, the set $\mathbf{E_C}$ represents the set of all reachable objects of exact dynamic type `C` in the initial heap.

The second kind of set is an $\mathbf{F}$ (feature) set, which contains the used features of a particular type. The set $\mathbf{F_C}$ for a Java class `C` contains the declared fields and methods of

C that have been used explicitly within the program. Similarly, the $\mathbf{F_T}$ set for a VM type T contains the declared fields of T that are used by the program and the VM. Consider the VMMethod type. It has declared fields name and signature that reference UTF8 strings. These fields are modeled as features of the VMMethod type, and if the fields (features) are used, then they will be added to the $\mathbf{F_{VMMethod}}$ set. Further constraints will ensure that the strings to which these fields refer will be included in the closure.

There is one $\mathbf{E_C}$ set and one $\mathbf{F_C}$ set for every Java class C in the program and one $\mathbf{E_T}$ set and one $\mathbf{F_T}$ set for every type T of VM data structure types. To simplify the number of different types of constraints, our analysis models a Java method implementation (i.e. a method that contains code) as an entity of type method, and the set of all reachable method implementations with $\mathbf{E_{method}}$.

2.4.5. Constraint Forms in Feature Analysis

Our analysis generates 8 forms of constraints. Most of these constraint forms should be familiar to readers who have prior experience with analyzing Java code with constraints.

(1) Base case for entities: expresses initially reachable entities. If an entity e of type T is present at the beginning of the program execution, for example the main method, then e is reachable.

$$e \in E_T$$

(2) Call site: analyzes call sites in the code of reachable methods in the program. For each method M and each call site e.p() in the code of M, where the static type of e is C, we have the constraint:

$$M \in E_{method} \Rightarrow p \in F_C$$

(3) New object: analyzes allocation sites in the code of reachable methods in the program. We use $dummy_C$ to denote a dummy entity of type C. For each method M and each new C() in the code of M, we have the constraint:

$$M \in E_{method} \Rightarrow dummy_C \in E_C$$

(4) Feature use: approximates the result of using a feature of a type by using the feature on *all live instances* of that type. Specifically, if the entity e_0 of type S is live, and the feature f of type S is live, then the entity referred to by e.f is also live:

$$f \in F_S \wedge e_0 \in E_S \Rightarrow e_0.f \in E_{typeof(e0.f)}$$

(5) Subtyping: establishes the relationship between used features in a supertype to the used features in a subtype. Specifically, for types S and T in the Java program, where S is a subtype of T, we have the constraint:

$$F_T \subseteq F_S$$

(6) Feature implication: expresses cases where the use of a feature entails that some other feature is also used. Specifically, for a type S with feature f, and a type T with feature g we may have a constraint of the form:

$$f \in F_S \Rightarrow g \in F_T$$

(7) Entity implication: expresses cases where the reachability of one entity implies the reachability of some other entity. Specifically, for an entity `d` of type `S`, and another entity `e` of type `T`, we can have constraints of the form:

$$d \in E_S \Rightarrow e \in E_T$$

(8) Entity implies feature: expresses cases where the reachability of one entity entails the use of a feature of some other type. Specifically, for an entity `e` of type `S`, and for a type `T` with feature `f`, we may have the constraint:

$$e \in E_S \Rightarrow f \in F_T$$

The constraints (1), (2), and (3) are basically equivalent to rapid type analysis, which maintains a set of possibly instantiated classes RTA_C and a set of reachable method implementations RTA_M. We can take this view if we consider the existence of $dummy_C$ in E_C is equivalent to `C` being in the live set RTA_C maintained in RTA. However, constraints (4) and (5) extend this basic view with live entity sets that are similar to those maintained in the RMA analysis. The key insight is that the new constraints (6), (7), and (8) extend the power of the analysis even further, allowing us to specify per-language and per-VM constraints that relate Java entities to their implementation and vice versa.

Figure 2.2 gives examples of some constraints that handle native method implementations in the class library. These constraints model the fact that native methods can trigger Java-level features such as creating new Java objects and arrays, as well as directly manipulating the VM's internal data structures.

(a) $\texttt{fillInStackTrace} \in E_{\texttt{VMNative}}$
    $\Rightarrow \texttt{dummy}_{\texttt{[I}} \in E_{\texttt{[I}}$
(b) $\texttt{fillInStackTrace} \in E_{\texttt{VMNative}}$
    $\Rightarrow \texttt{classSegmentList} \in F_{\texttt{VMJavaVM}}$
(c) $\texttt{startThread} \in E_{\texttt{VMNative}}$
    $\Rightarrow \texttt{run} \in F_{\texttt{java.lang.Runnable}}$
(d) $\texttt{startThread} \in E_{\texttt{VMNative}}$
    $\Rightarrow \texttt{J9VMInternals.threadCleanup} \in E_{\texttt{method}}$
(e) $\texttt{forName} \in E_{\texttt{VMNative}}$
    $\Rightarrow \texttt{classTable} \in F_{\texttt{VMClassLoader}}$
(f) $\texttt{indexOf} \in E_{\texttt{VMNative}}$
    $\Rightarrow \texttt{bytes} \in F_{\texttt{java.lang.String}}$
(g) $\texttt{javaVM} \in E_{\texttt{VMJavaVM}}$
(h) $\texttt{e} \in E_{\texttt{VMJavaVM}}$
    $\Rightarrow \texttt{mainThread} \in F_{\texttt{VMJavaVM}}$
(i) $\texttt{m} \in E_{\texttt{method}}$
    $\Rightarrow \texttt{repof(m)} \in E_{\texttt{VMMethod}}$

Figure 2.2: Example VM-specific constraints
Natives can (a) allocate new Java objects (b) use features of VM structures (c) invoke Java virtual methods, (d) invoke Java static methods (f) use fields of Java objects. Default constraints assert certain entities (g) and features (h) to be live. The constraint (i) ensures that if a Java method implementation is live, then its representation in the VM is live.

Consider the example constraint (e) in Figure 2.2, which models the need for the class table, a hashtable that maps strings to class representations in implementing the `Class.forName` Java native method. If this native method is never called (i.e. it never is added to the set $E_{\texttt{VMNative}}$), then the `classTable` pointer need not be analyzed, and consequently, this data structure can be removed.

2.4.6. Granularity and Natives vs. Sanity

In our experience, writing the constraints for all of Java's bytecodes was comparatively little effort, as this problem is generally well understood and has already been explored in many previous analysis techniques. If we make the assumption that the constant pool entries are resolved and that classes are loaded and initialized, then most bytecodes amount to little more than manipulating Java objects and the stack and performing calls to some simple VM services such as the allocator. At the bytecode level, it is easy to have confidence that our analysis constraints for each bytecode will force the inclusion of the necessary data structures into the image, and that “pure Java” programs will execute without problems on the ExoVM.

However, the bulk of Java—its class library—is not so simple. Java has dozens of classes in its standard library that are wormholes into the VM; many have native methods that manipulate internal VM data structures directly. In the case of J9, the VM and the native code that implements the class library are developed separately but significantly interdependent. In the j9cldc class library, there are 75 such native methods, many of which are implemented in assembly code. In the J2SE (j9max) class library, there are more than 200. Some use JNI or internal services to call back into Java code or allocate Java objects. Each of these methods requires constraints that trigger the inclusion of Java code and VM structures that are needed to implement them. We were able to derive constraints for many of the most important ones. For some we simply coarsen the granularity of the analysis of data structures and conservatively include some possibly unreachable data structures. Otherwise, we forbid native methods that we do not yet support by dynamically trapping calls to them.

An example of tuning the analysis between fine-grained and coarse-grained is the idea of modeling every pointer in every data structure in the virtual machine as a feature that is only used when certain constraints are triggered, such as the use of a particular native method or VM service. While the most fine-grained approach is attractive because it allows the maximum possible reduction of data structures, only including them under the most specific circumstances, the VM is complex enough that determining the most specific constraints for each pointer becomes infeasible. For many pointers, we were forced to simply assert them either dead or live. Asserting them live is always conservative and correct, provided that the data structure that they refer to is correctly identified and copied into the image, but of course this may include many unnecessary data structures. However, asserting a pointer to be dead is too aggressive if the associated language feature or service is needed at runtime, in which case the virtual machine or native libraries will crash due to the missing data structures.

Our approach has taken the middle of the road, asserting many pointers to be dead that correspond to VM features that we do not intend to support, such as dynamic class loading, and asserting some pointers live and always copying the referred data structures because the right constraints may be elusive. Some data structures are always necessary, such as the `VMJavaVM` data structure and the `VMThread` structure for the main Java thread. To guide our effort, we developed a suite of micro-programs that target individual features, including the basic bytecode set and specific native methods. This proved to greatly expedite testing and debugging, allowing us to pinpoint the usage of many pointers of VM structures and relate them directly to language features. For more complex correctness validation including native methods, we rely on running larger benchmark programs and verifying that each program computes the same results as it does on the complete JVM. An industrial scale, feature-complete implementation of our technique would have to test against the Java language compliance kit, since we do not believe that it is possible to directly prove the correctness of the analysis technique due to the sheer size and complexity of the virtual machine.

2.5. Persistence

Persistence is the process of taking a snapshot of the fully initialized virtual machine, including the data structures that represent the program and the program's state, and saving it to an image file or other persistent store to be loaded later. Persistence has been studied widely in programming languages and database systems [18] and has a number of compelling advantages for programming systems. Key issues are the transparency and efficiency of the persistence mechanism, as well as data evolution and versioning.

In our system, we perform imaging of the VM only once as part of an offline analysis, so the efficiency considerations do not apply, and we do not support data evolution simply because the kinds of data we are saving are heavily tied to one particular VM implementation. As such, our persistence framework, which we refer to as the *imager*, need not be as general as that in previous systems. After the closure process has computed a set of reachable Java methods, classes, objects, and VM data structures, the imager copies and relocates the data structures that exist inside the virtual machine to a special region of memory which is then saved to the disk. This image file is a compacted snapshot of the VM data structures that represents only the reachable parts of the program. The image file contains essentially a complete ready-to-go VM that can be used immediately by simply mapping it into memory.

2.5.1. Persisting C-based Data Structures

Once the closure process has computed the set of reachable data structures of the VM that are needed to correctly execute the program, the imager must copy and relocate these data structures to persistent store. These data structures are declared in C but are manipulated by C, C++, and assembly code. The imager therefore needs to persist C data structures in a way that preserves the invariants that are implicit in the code that manipulates them. We began studying the layout of these data structures and the code that manipulates them, discovering that many were more complex than we initially thought and had many implicit constraints. This manual process represents a particularly unromantic but significant amount of our development time, approximately 3-5 man-months. From our efforts we were able to develop a description of each important data structure: its layout, address alignment constraints, contents, and its pointers to other data structures. The imager uses the description to determine how to copy and relocate VM data structures of each type, which includes computing the size and layout of a particular instance and where pointers to other data structures lie within the structure. This is similar to the description of a Java object that a garbage collector needs in order to scan a Java object for references to other objects, but can be considerably more complicated. We discovered a number of implicit constraints on data structures. Two constraints of note are implicit adjacency/layout requirements, and strangely encoded pointers.

Many kinds of data structures are segregated into segments, which allows mass allocation and deallocation as well as fast traversal over all data structures of a given type. The dependence on this layout is buried deep in the assembly and C code of the

VM; to reuse this code without modification requires preserving the invariants it expects. This requires the imager to collect certain structures into new segments during the copy process.

Some data structures have grown very complex as they evolved over time. For example, the J9 representation of a class has numerous adjacent, embedded members of variable size; code throughout the VM relies on being able to find known structures at computed offsets from the beginning of the structure. Worse, other data structures throughout the VM point into the middle of the class structure. A correct description of this data structure for the imager required tedious manual analysis of the code to determine its undocumented layout and implicit constraints.

Many virtual machine techniques pack extra information into pointers in the high or low-order bits, such as in implementation tricks for monitors [73], virtual tables, and object headers, etc. These pointers are assumed to point to structures aligned on addresses that are particular powers of two (most often 8, 16, and 256 bytes), which allows the lower bits to be reused. To address this common undocumented tendency, the description of each data structure in the imager contains alignment constraints that are used when the imager chooses a new address for a data structure, making the undocumented constraints explicit. Similarly, pointers that contain extra information bits have special types that instruct the imager to preserve the appropriate low-order bits; the type makes it obvious that the pointer contains extra information.

Another problematic feature of the system is the use of *self-relative pointers* within some data structures; a self-relative pointer stores an offset instead of an actual address; instead, code that uses the pointer computes the actual address of the target by adding the pointer's value to the pointer's location. This allows some data structures to be copied to and from disk and shared across processes without relocation, as long as the entire data structure containing the self-relative pointers are moved as a unit. Because our imager may move pieces of these data structures around independently, it must encode and decode self-relative pointers correctly. Like pointers with extra bits, self-relative pointers have a special type in the data structure description that documents this fact and allows the imager to handle these pointers with equal ease as normal pointers.

2.5.2. Compilation

By completely initializing the VM before imaging, the system can also save any compiled code of the application that has been produced by the JIT. In fact, because of the offline nature of the imaging process, we can simply compile all of the reachable methods with the JIT compiler ahead of time. The JIT and its data structures can then be removed completely from the ExoVM, effectively turning the original VM into a static compiler – albeit one which may generate superior code because all classes are resolved and initialization code has already been executed. Because the compilation takes place in a closed-world scenario, there is no need to invalidate code and recompile. In most cases the bytecode of compiled methods can be discarded, though sometimes looking up exception handlers and generating a source-level stack trace requires bytecode-level information.

In our case, supporting pre-compilation required some small modifications to the JIT compiler. First, we had to direct it to generate code into the image, rather than into internal code buffers, and we had to disable mechanisms that trigger recompilation of methods. Also, the JIT often writes the absolute address of data structures and functions that it assumes do not move into the compiled code; the imager must make sure that these pointers are found and relocated before the image is finished. We instrumented the JIT to record where it writes absolute addresses into the compiled code, and then patch the addresses at image load time. With this approach, there is no need to alter the machine code that the JIT generates. This feature was fully operational in an earlier version of the ExoVM but our experimental results here use the interpreter and do not include any compiled code. Of course, the size of the compiled code depends on the quality of the JIT compiler and the total amount of reachable code of the program. Our early experience was that the size of the compiled code was roughly comparable to the size of the class representations.

2.6. Loading a VM Image

Although the imager is capable of producing an image that contains a complete collection of data structures that represent the program and the VM needed to run the program, the imager is not capable of actually copying the machine code of the VM into the image because the linking model of C and C++ precludes this. For example computed jumps and branches within machine code cannot be supported without linking information. Our approach to this problem is to separate the data structures (which are stored in the image file) from the *boot VM*, a specialized offline build of the fully featured VM that contains little or no internal data structures. The boot VM lacks the normal VM initialization routines that build these internal data structures, as well as mechanisms such as the JIT compiler and dynamic class loader, but instead only contains VM subsystems that will be needed at runtime for each application, such as the interpreter, garbage collector, natives of the class library, etc. The boot VM loads all of the needed data structures from the image.

Our imager produces image files that are intentionally not relocatable; i.e. all of the internal data structures and code within an image file contain absolute pointers to each other that assume the image starts at a fixed memory address. This simplifies both the imager and the boot VM, allowing the boot VM to simply memory map the image from the file to the specific address and thus begin using the image in memory without relocating any internal pointers. Additionally, the image header contains pointers to the main class, the main method of the program, and to important global VM data structures so that the boot VM need not search the image for where to begin execution.

2.6.1. Patching and Rebuilding

The separation between the code and data of a VM instance is not perfectly clean, and many internal data structures that are saved in the image contain pointers to internal VM functions that do not exist in the image. The boot VM must supply the implementation of these functions by patching these pointers when the image is loaded into memory. For example, a `VMMethod` instance contains a pointer to code that implements the calling convention for that method. An interpreted method contains a pointer to machine code in the interpreter to set up the interpreter state, while a synchronized method has a pointer to code that obtains the lock on its receiver object before executing the method, and so on. When the imager copies a data structure and encounters pointers to VM machine code or a C function, it uses a table of known VM routines to identify the target routine. At load time the boot VM loads the image and replaces these pointers with pointers to its implementation of the corresponding routines.

One further complication with the imaging process is that not all internal data structures can be persisted. In particular, the VM has data structures that correspond to operating-system level resources such as threads that are not transferable from one process to the next. The boot VM rebuilds certain data structures as necessary when it loads the image into memory.

2.7. Experimental Results

2.7.1. Footprint

We have implemented pre-initialization, closure, and persistence in a J9-based virtual machine with the j9cldc and j9max class libraries to investigate the memory footprint of the VM and the application in an embedded scenario. These numbers are obtained on the x86 build of J9 running on Linux 2.6. We did not specifically measure the execution time for the imaging process, but even with our completely untuned implementation written mostly in Java and running in interpreted mode, the entire loading, initialization, closure, and copying process of the ExoVM took less than 5 seconds on a fast Pentium IV workstation for all our benchmarks.

To evaluate the effectiveness of the ExoVM approach, we measured a number of footprint factors for our benchmark programs. First, we evaluate the fixed cost of the VM in terms of the VM's static code and data footprint for the two original VM configurations and the ExoVM specialized boot VM. The j9cldc configuration consists of 600kb of compiled VM code and natives, 260k of read-only data (of which 190kb is the class library compiled into the executable), 20kb of initialized data and 17kb of uninitialized data. The j9max configuration consists of 750kb of compiled VM code and natives, 90kb of read only data, 25kb or initialized data, and 17kb of uninitialized data. To reduce the size of the boot VM, we statically compiled out some subsystems, including the JIT compiler, bytecode parser and verifier, zip library support, and some initialization routines, saving about 200kb of compiled code. We believe that there is more code that can be removed from this specialized VM, but linking issues and time constraints limited our ability to explore this.

	CLIB	**ROCL**	**RWCL**	**INH**	**NHA**	**IHEAP**	**total**
Chess	0	597	162	0	557	0	1316
-exo	0	619	172	27	10	171	999
Crypto	0	595	163	0	591	0	1349
-exo	0	615	173	26	10	195	1019
kXML	0	588	159	0	646	0	1393
-exo	0	610	169	26	11	204	1020
Parallel	0	574	147	0	549	0	1270
-exo	0	0	0	0	0	0	0
PNG	0	549	148	0	504	0	1201
-exo	0	577	160	25	11	181	954
RegExp	0	571	156	0	518	0	1245
-exo	0	598	168	26	11	173	976

Figure 2.3: j9max Memory Footprint
Dynamic non-heap memory footprint for six benchmarks on the j9cldc configuration. Each benchmark has two rows: one for its footprint in the standard VM, and the next row for its footprint using the ExoVM system.

	CLIB	**ROCL**	**RWCL**	**INH**	**NHA**	**IHEAP**	**total**
Chess	188	74	60	0	394	0	716
-exo	0	113	42	33	10	14	212
Crypto	188	70	62	0	466	0	786
-exo	0	114	46	25	10	41	236
kXML	188	56	58	0	483	0	785
-exo	0	113	45	27	11	50	246
Parallel	188	49	45	0	415	0	697
-exo	0	87	26	30	89	33	265
PNG	188	26	48	0	383	0	645
-exo	0	74	32	23	11	30	170
RegExp	188	47	55	0	389	0	679
-exo	0	98	41	26	11	21	197

Figure 2.4: j9cldc Memory Footprint
Dynamic non-heap memory footprint for six benchmarks on the j9cldc configuration. Each benchmark has two rows: one for its footprint in the standard VM, and the next row for its footprint using the ExoVM system.

In Figures 2.3 and 2.4, we compare the dynamic memory footprint measurements for the data structures and loaded classes across our benchmarks for the j9cldc and j9max configurations. The first row of each benchmark contains the measurements of several footprint factors on the unmodified VM running the applications with the corresponding class library. These footprint factors are CLIB; the size of the class library which is compiled into the binary executable (applicable only to j9cldc); ROCL, or read-only portions of the application classes (`VMROMClasses`); RWCL, or the read-write portions of these same application classes (`VMClasses`); NHA, or non-heap allocations, which are data structures allocated by the VM that are not Java objects and thus not part of the heap. Each of these numbers is given in kilobytes. The two remaining footprint factors apply only to ExoVM images. These are INH, or imaged non-heap data structures, which are non-heap data structures that were allocated during pre-initialization and have been persisted; and IHEAP, which is the initial heap of Java objects, consisting of everything from string constants to application objects that have been determined to be reachable by the closure process. Note that we do not measure the dynamic heap of the program here; we were able to successfully execute the benchmarks with just 128kb of heap (except kXML, which required 512kb), which makes the VM data structures by far the dominating factor.

These measurements show the effectiveness of pre-initializing the virtual machine and the application. With a completely built image, the ExoVM has no need of an external class library (CLIB). Feature analysis detects that a number of classes are unused and removes them, showing a moderate reduction in the size of the read-write class representations (RWCL). The size of the initial heap (IHEAP) generated by running the class initializers in the virtual machine is relatively small. But by far the biggest factor is the reduction of the VM's dynamic non-heap memory allocations. This shows that pre-initialization of the VM and feature analysis allow the ExoVM to remove the dominant factor of space consumption in these benchmarks. The reduction of nonheap memory allocations is between 62 and 73% for these six benchmark applications.

Figure 2.3 shows the result of the same experiment with the j9max configuration, which consists of the same VM, but a more complex, fully featured class library. In this scenario, the class library is much larger and not compiled directly into the virtual machine's binary. However, we can see that the dominant cost is now the size of loaded classes, because the more fully featured class library has many more interdependencies that force many classes to be loaded and initialized.

The most surprising result is that running the feature analysis to produce an image for each of these programs does not yield a smaller ROM or RAM class footprint. We investigated the reason for this and discovered that the j9max's `Class.getName()` implementation uses a `HashMap` that maps a class representation to its `String` name. Because our analysis is partly written in Java and runs on this underlying class library to compute the closure, if the program being analyzed calls the `Class.getName()` method, then the analyzer will discover that this `HashMap` is reachable, and begin analyzing its contents. Because these classes are reachable through Java references, it therefore concludes that all loaded classes are live, and none are removed from the image.

We were not able to successfully run the Parallel benchmark on the ExoVM because the larger class library demanded an implementation of protection domains which were beyond our resources to support. This highlights another problem with a larger class library. Adding a security layer tends to demand reflective features from the VM that vastly increase the complexity of the virtual machine.

2.7.2. *Feature study*

During the course of developing the ExoVM system and testing feature analysis for correctness, we wrote a large number of Java micro-programs that each uses a specific language feature, such as virtual dispatch, throwing an exception, calling API methods, running threads, etc. While primarily intended for our internal use in testing correctness, they had the side effect of exposing just how much of the class library and VM is tied to a particular language feature. Though we don't claim that our micro-program suite is fully comprehensive of the Java language, it did highlight important issues.

	cldc	max
empty	5	5
arrays	38	225
checkcast	42	228
constructors	13	31
floating point	7	7
nullptr	13	31
.getClass()	30	872
refarray	40	226
Hello world	75	872

Figure 2.5: Microprograms
Resulting image sizes for several different microprograms, where each stresses a single language feature.

Our micro-programs were all less than 25 lines of code and primarily target a single language-level feature. We found a good approximation of the cost of a feature to be the size of the image generated by our analysis, which includes not only VM data structures, but also persisted classes and objects. As a starting point, we tested how small an image our system could generate for the empty program; i.e. a single static main method that just returns. On both j9cldc and j9max, our system generates a 5kb image that contains the main class (1kb), `java.lang.Object` (1kb), the `VMJavaVM` structure (1.3kb), a thread (0.6kb), and a small number of other data structures. This is enough to reuse the existing VM code unmodified and execute successfully.

From this starting point, we investigated the incremental cost of supporting individual languages features; Figure 2.5 shows several microprograms and the resulting image size for the j9cldc and j9max configurations. From the table, we can see that several of the programs that generate small images on the j9cldc configuration have large images on j9max.

We were able to pinpoint the problems that cause this phenomenon of "feature explosion" in j9max by using these unit feature tests. Our analysis revealed that the larger class library contains a small number of "precarious" dependencies, such as the `HashMap` in the `Class.getName()` implementation mentioned previously. When one such dependency is triggered, it tends to pull in a large subset of the class library as a whole. This can be seen in the tests that construct and print exceptions: they tend to pull in a large portion of the class library, which ultimately dwarfs their small size. Our conclusion from this study is that future design of class libraries and careful implementations should strive for modularity in features so as to avoid penalizing small programs and avoid precarious dependencies. Another approach might be to embed more special knowledge into analysis about the Java-level entities that implement Java features, such as introducing a special case for the `Class.getName()`'s internal data structures. This remains as future work.

2.8. Experience

In our experience developing the ExoVM system in J9, we learned important specific lessons about its implementation and virtual machine design in general that we think are valuable to others. The first is that complex and arcane data structures frustrate automated imaging techniques, and judging from the implementation complexity that seems to replicate itself over and over throughout the virtual machine, we simply do not believe they are worth whatever gain they intend. By far most of our manual effort was inferring implicit constraints of data structures and fixing problems with pointers and layout tricks, working backwards from VM crashes. Although certain techniques have advantages for performance or space usage, our overwhelming sense after studying the code is that the most complicated data structures have evolved by accretion and they survive because their deep entanglement with the VM makes them particularly dangerous to migrate or refactor. We think that our work shows the value of persisting the internal VM data structures for an embedded domain, and simpler, more regular data structures make this technique far easier.

The second lesson we learned from our experience is that there appears to be more modularity to source-level language features than previously thought. This dimension of modularity does not seem to be borne out in current virtual machine design and class library implementations, including J9 and those with which the authors have previous experience. We believe that this dimension of modularity has important applications in the embedded domain, and that valuing it more highly in the design of new virtual machines will have positive consequences for the ability to scale from small devices to server class machines.

The third lesson that we learned is that the implementation technology of the virtual machine itself matters considerably. We cannot achieve our ultimate goal of total automatic VM specialization given J9's current implementation technology, in particular the static linking model inherent in C and C++ applications. A large amount of our development effort has been spent in recovering implicit usage patterns of data structures in the virtual machine, which is difficult to automate in these languages. Given our experience in program analysis for large applications written in higher-level, statically typed languages like Java, we believe that much analysis can be streamlined, if not automated completely, if the VM itself were implemented in a language that is more amenable to disciplined program analysis.

Not surprisingly, we found that complexity of the class library makes an important difference to the footprint of an application, especially with the implementation of the basic language features such as exceptions. The CLDC implementation of the class library contains not only fewer classes over all, but the implementation of basic classes such as exceptions has fewer dependencies, resulting in smaller image sizes. The implementation of exceptions and I/O, particularly international formatting of strings, is significantly more complex in the j9max library, which results in many more live classes and consequently more used language features. For this technique to work well on such class libraries, more modularity in these implementations seems to be necessary, or the analysis must be improved.

Java's dynamic invocation of class initializers may work well for a bigger domain, but our results with pre-initialization of the classes in an image tends to suggest that for this domain, significant gains can be made by changing the model.

3. VIRGIL

This chapter describes the Virgil programming language and compiler system. Unlike the ExoVM, which seeks to reuse an existing virtual machine, the Virgil language and compiler are developed in a scenario where we have the freedom to rethink the language syntax and semantics and have the opportunity to rebuild all of the software, from the drivers to the operating system. Virgil establishes part of the thesis by showing that *advanced language and compiler technology can bring the benefits of object-oriented programming to developing microcontroller software*.

Microcontroller-class devices represent an extreme setting for the challenges inherent in building embedded systems. These challenges include not only resource constraints such as code space, data space, and CPU performance, but also the lack of supporting software and hardware mechanisms to enforce safety, the need to access low-level hardware state directly, and the concurrency introduced by handling hardware interrupts. This chapter considers the question of how object technology can benefit developing software in this domain when not constrained by legacy code or backwards compatibility. Objects have much to offer embedded systems software where events, queues, packets, messages, and many other concepts exist that lend themselves naturally to being expressed with object concepts. Unfortunately the domain constraints have thus far limited the adoption of object-oriented languages.

The Virgil programming language addresses the challenge of bringing object technology to microcontrollers through language and compiler techniques. The most important design consideration when taking this approach is the space overhead that language features add to the program implementation. This space overhead can be divided into two categories: *runtime*, which consists of libraries, routines and subsystems needed to implement language features like garbage collection, class loading, reflection, dynamic compilation, and serialization; and *metadata*, which consists of data structures added to the program such as dispatch tables, string constants, type signatures, and constant tables. Virgil avoids heavyweight features that require a runtime system or significant metadata and selects features that admit a straightforward, low-overhead, constant-time implementation that is both clear to programmers and can be accomplished without sophisticated compiler analyses. The lack of supporting hardware and software mechanisms for enforcing safety is overcome by enforcing strong type-safety at the language level with some dynamic checks. Finally, Virgil's compilation model allows for complex application initialization at compile time and enables three new aggressive optimizations that further increase application efficiency.

3.1. Design Principles and Constraints

This section explains the design principles that have guided the design of Virgil language as well the design constraints that the domain imposes. While the overall goal is to ease the development of modular and robust programs, design principles translate the goal into a set of concrete, desirable properties that the language should have. Virgil's four main design principles are:

i. **Objects are a good fit.** The object-oriented programming paradigm has successfully led to better designed programs that are more modular, flexible, and robust. Embedded software often uses events, queues, packets, and messages; objects are a natural fit to represent such entities.

ii. Static detection of errors is best. Strong static type systems catch a large class of errors that are still embarrassingly prevalent in embedded systems software. The weak type systems in languages like C and C++ fail to catch an avoidable class of bugs in the interest of allowing direct control over data representations, manual memory management, and access to hardware state for software at the lowest level. Ironically, these kinds of systems have the greatest need for static checking, because errors are the hardest to find and the most damaging. Strong static safety guarantees in this domain are paramount.

iii. Objects are not always perfect. Although object-oriented concepts are a good fit for many tasks, new expressiveness problems continually stress object-oriented constructs. For some problems, functional or procedural programming styles still have important advantages that should not be overlooked. The language should afford programmers some degree of flexibility to seek elegant solutions.

iv. Some dynamic checks are OK. An object-oriented language cannot usually avoid all potential safety problems statically, particularly when indexable arrays, null references, or type casts are allowed. In this case the language must fall back on some dynamic checks that may generate language-level exceptions. Although microcontrollers often lack hardware safety checks and thus require explicit checks to be inserted by the compiler, modern compiler optimizations are now advanced enough that this overhead is usually acceptably small.

Design principles outline desirable properties that Virgil should have, while the limitations of microcontrollers impose an important set of constraints. The resource challenges of an embedded system require a systematic design approach that avoids introducing unacceptable resource consumption in implementing the basic language and libraries. One of the primary efficiency considerations for Virgil is to ensure that overheads introduced by the language are small and proportional to usage in the program. This affords programmers control over resource consumption by avoiding uncontrollable, fixed costs like a large runtime system. Where and when language feature overheads occur will be apparent to moderately skilled programmers and therefore can be reduced or avoided by restructuring the program if needed. This leads to the imposition of the following design constraints on the language and compiler:

i. **No runtime.** First, any large, fixed cost that is beyond the control of the programmer should be avoided. Secondly, Virgil programs will run as the lowest layer of software, so the notion of a language runtime underlying a Virgil program is problematic; the runtime is often implemented "under the language" and is typically not under control of the application or system programmer. Microcontroller programmers usually need to have control over all of the code that will end up on the device.

ii. No intrinsics. Intrinsics are library code and types, other than primitives, that are needed to implement basic language features and are generally established by a language standard. For example, in Java, the entire java.lang.* set of classes are needed by both the compiler and runtime system to implement a Java program. Transitively, these classes pull in a nontrivial part of the JDK. Despite the positive effect that standardizing basic libraries can have, like a runtime, the implementation of intrinsics isn't supplied by the programmer, and thus represents yet another uncontrollable source of resource consumption.

iii. No dynamic memory allocation. Manual memory management, aside from concurrency, is perhaps the most error-prone part of software. Modern languages employ automatic memory management, often in the form of a general-purpose garbage collector, which eliminates most of the problem. But even the best garbage collectors impose significant space overhead on programs, often requiring two to three times as much heap space in order to achieve good performance, as well as significant metadata to support precise collection. In the microcontroller domain, static pre-allocation of all necessary data structures is very common, in fact, one of the nesC [44] language's primary design criteria was that dynamic memory allocation is unnecessary for the targeted class of systems.

iv. Minimal metadata. Metadata associated with a program, such as runtime type information, virtual dispatch tables, and object headers should be small and proportional to the program's complexity. This allows the programmer to trade some space for better language features at his or her discretion, provided the overhead is acceptably small and readily apparent during implementation and tuning.

3.2. Virgil Language Features

In this section, we examine the features of the Virgil programming language, both those features selected for inclusion and those rejected. In this design space there is significant tension between expressiveness and its runtime cost, with RAM usually the scarcest resource. For example, embedded programmers have often felt the need for explicit control of data representations in order to save space, while to save execution time and code space, they often shy away from language constructs that appear inefficient. The common rule of thumb in C++ is "you get what you pay for," which leads programmers concerned about efficiency to avoid exceptions, runtime type information, templates, and many other language features. Most microcontroller programmers avoid higher-level languages altogether, preferring C because developing a standalone program is relatively easy, and C is perceived as an inherently efficient language because it is very low-level. Worse yet, some microcontrollers are so tiny they are still developed primarily in assembly language.

Virgil balances this design tension at a unique point, carefully selecting features according to the design principles and constraints, increasing expressiveness while retaining an efficient implementation that builds programmer trust. Each feature is considered carefully against the efficiency and straightforwardness of its implementation. This will allow a programmer to trust that a basic compiler will implement objects efficiently. Advanced optimizations presented later in this chapter and in Chapter 4 will further reduce program footprint, lightening the burden on the programmer, leading to higher productivity and more robust systems.

3.2.1. Inheritance Model

Virgil's inheritance model is motivated primarily by the need to allow a straightforward and very efficient object implementation with minimal metadata, while retaining strong type safety. Because programmers in this domain often face tension between program flexibility and implementation efficiency, Virgil makes the efficiency tradeoff more explicit and controllable.

Virgil is a class-based language that is most closely related to Java, C++ and C#. Like Java, Virgil provides single inheritance only, with all methods virtual by default, except those declared `private`, and objects are always passed by reference, never by value. However, like C++ and unlike Java, Virgil has no universal super-class akin to `java.lang.Object` from which all classes ultimately inherit. But Virgil differs from C++ in two important ways; it is strongly typed, which forces explicit downcasts of object references to be checked dynamically, and it does not provide pointer types such as `void*`. The implications of lacking an `Object` class are explored in the next subsection.

Java provides limited support for multiple inheritance through the use of interfaces, which increase the flexibility of object classes. However, the implementation efficiency of interfaces can be troublesome, particularly in terms of the metadata needed for interface dispatch. In some cases, altering a single class to implement a new interface can result in a significant increase in the size of dispatch tables across multiple types. Alpern et al [4] discuss efficient implementation techniques for Java interface dispatch in the Jikes RVM; their technique uses a hashing scheme that works well in practice, but can require generating code stubs that perform method lookup. In general, most interface dispatch techniques are either constant-time (e.g. two or three levels of indirection), or space-efficient (e.g. linear search, hashing, caching), but not both. Because of these limitations, Virgil does not support interfaces.

Restricting Virgil classes to single inheritance and removing features such as interfaces and monitors reduces the amount of metadata needed for each class and each object instance. A Virgil object requires only a single-word header that is a pointer to the meta-object of its class. Because Virgil has no universal super-class, a root class inherits nothing and its meta-object contains only a type id and the slots for virtual methods declared in the class (although Chapter 4 discusses more advanced and efficient object layouts). Further, because Virgil meta-objects have no mutable state, they can be stored in ROM in order to save precious RAM space.

Single inheritance also allows subtype tests to be implemented by using the well-known range-check technique where each class is assigned a type id and range of type ids that contains its subclasses. A dynamic type test of object **O** against type **T** is implemented as a check of **O**'s type id against the range of type ids for **T**. For leaf types **T**, only one comparison is necessary. This approach, first presented in [86], is more efficient than dynamically searching **O**'s list of parent types, but requires the availability of the complete inheritance hierarchy. This technique is a good fit for Virgil; it guarantees every cast is a constant-time operation, regardless of the depth of the class hierarchy, and requires at most one integer type id per meta-object. Virgil's compilation model ensures the entire class hierarchy is available at compile time.

3.2.2. To Object or Not to Object

One important design choice in Virgil is the lack of a universal super-class such as `Object` that all classes implicitly extend. In Java, `Object` includes a host of features including monitors, first-class meta-objects (the `getClass()` method), hashing, equality, etc. A number of these services require space in each object header, in addition to mark bits needed by the garbage collector. Bacon et al [20] discuss in detail the challenges inherent in implementing the Java object model efficiently. Even in high performance virtual machines, two or more words of space are usually needed for object headers. Class-based inheritance requires the meta-object for a class to be at least as large as that of its super-class. In Java 5, `Object` contains 11 virtual methods, forcing every meta-object in the program to be at least as large.

As an alternative to the Java model, one could consider an empty `Object` that contains no methods and no capabilities. An empty `Object` that is the root of the hierarchy would prevent bloating of all meta-objects and allow generic collections such as a list to hold any kind of objects, at the cost of forgoing the convenience of default functionality. At first, this seems like a reasonable tradeoff. However, this still forces all objects to retain a header that contains type information because objects can be implicitly cast to `Object` and then later explicitly downcast, which requires type information for a dynamic safety check.

The decision to eliminate the universal super-class in Virgil allows some objects to be implemented without any metadata. Virgil programmers can write what are termed *orphan classes*: classes that have no parent class and no children classes. An instance of an orphan class is a degenerate case of an object; it can be represented as a record without any object header, like a C `struct`, since it is unrelated to any other classes. Because the Virgil type system rejects casts between unrelated classes (as in Java), an object of an orphan class never escapes to a point where its exact type is not known. The compiler can also statically resolve method calls on orphan objects, removing the need for a virtual dispatch table.

Orphan classes can arise intentionally and unintentionally in a Virgil program; a programmer need not restrict a class to be an orphan explicitly. In fact, each class is an orphan by default, unless it extends some other class, in which case neither class is an orphan. Personal experience with large applications in Java gives tends to suggest that a substantial number of classes tend to be orphans without purposeful contemplation. The Virgil compiler extends this tendency to a guarantee that orphan instances will be represented without an object header. Orphans therefore give the careful programmer a way to extract maximum efficiency, at some cost to the program's flexibility.

The advantages of this lack of a universal super-class and the special support for orphans include:

i. **Removes the need for intrinsics.** There is no need for a special root class that is built into the language. Such special built-ins have a tendency toward feature bloat, which reduces the programmer's ability to make efficient implementation decisions and goes against the design criteria of Virgil.

ii. **Orphan objects are very efficient.** Orphan instances require no object header and no meta-object that contains runtime type information for the class. A programmer can use objects like structures without penalty in a way that is statically type-safe.

iii. **Improves type-based analysis**. Several compiler analyses use the static type information as an approximation of aliasing and flow information [35][77] and lose precision when references are typed `Object`. Such analyses get a precision boost by the virtue that objects cannot escape beyond their ultimate root class.

iv. **Lightweight confinement.** Virgil's strong type system affords a kind of lightweight confinement. By introducing a new class hierarchy unrelated to the rest of the program, the programmer can confine objects to a region of code, because such objects cannot escape through implicit casting to super-classes. Confinement, in addition to security benefits, helps modularize program reasoning for programmers and tools [108][116].

v. **Documentation and understanding.** Static types of fields and parameters provide valuable documentation to programmers. When finding uses of a class in a Virgil program, the programmer need only consider places where the class is mentioned by name and need not reason about objects escaping through subsumption.

vi. **Reference compression.** Covered in detail in Chapter 4, the compiler can exploit the confinement properties of disparate class hierarchies to compress object references in order to save RAM.

On the other hand, the lack of a unifying super-class has important disadvantages if the language lacks other mechanisms for polymorphism. First, it is difficult to write generic collections and data structures such as lists, maps, and sets that work with any kind of objects. A library might address this problem by reintroducing a base class for "collectible" classes that client code must extend in order to use the functionality—its own `Object` class, for example. Classes that choose to extend this `Object` class would forgo the efficiency benefit of orphans. A second problem is that as different libraries emerge, competing versions of `Object` could complicate programs that use multiple libraries. Another approach is to employ the Adapter [43] pattern by writing wrapper classes to adapt their classes to the API of various libraries. Delegates (covered later in this section) reduce this problem by allowing limited functional programming. The best solution overall is parametric types [22], which are discussed next.

3.2.3. Parametric Types

Many reusable data structures are type-agnostic, meaning that they are intended to store program data without regard to its type. For example, a linked list simply stores a sequence of data items, and its fundamental structure does not depend on the representation size or operations that the data type supports. Similarly, a hash table that maps values of one type (a "key" type) to another type (a "value" type) typically does not depend on any aspect of the value type, and only requires hash and equality routines for the key type. Maximizing code reuse for these basic data structures removes the need for programmers to re-implement each data structure for every combination of types.

The traditional object-oriented way to solve this problem is to implement the linked list or hash table using an overly general "top" type (such as `Object` in the case of Java). Because this type is the root super-class of all classes, the data structure can store any type of objects, but not primitives. Worse, the information about which data type a particular collection instance contains is lost; a list of strings has the same type as a list of integers because both use the same implementation that internally uses the `Object` type. It is then the responsibility of the programmer to remember which data types are in which collection. In a statically typed language like Java, the programmer is forced to insert a cast to the expected type when retrieving data from the collection. The cast is dynamically checked; it may fail at runtime if the programmer has made a mistake. In a dynamically typed language, no explicit cast is required, but the implicit dynamic type test can still fail. The underlying problem is that the use of the collection causes a *loss of static type information*.

Functional languages beginning with ML recognized and solved this problem with a technique known as *parametric types*. With parametric types, declarations of program entities such as classes and methods can be parameterized over a type. The type parameter declaration introduces a *type variable* that names a type for use within the scope of the class or method, without specifying exactly which actual type the variable refers to. Instead of declaring and implementing a list of `Object`, the programmer can declare and implement a list of `X` (written in Java, C#, and C++ as `List<X>`). The list implementation can be reused for many different types by substituting in the actual type in the place of the type parameter at the usage site.

The general trend is that statically typed object-oriented languages are transitioning away from the `Object`-based polymorphism of traditional implementations towards polymorphism based on type parameterization. In the object-oriented world, parametric types are known as *generic types* or simply *generics*. Virgil uses the term parametric types because it is more evocative of the underlying concept, and early implementations of generics had severe handicaps that carry a negative connotation.

The need for parametric types in Virgil is especially great because there is no universal super-class such as `Object`, giving rise to a number of problems as discussed in the previous section.

Java 5 allows classes and methods to have type parameters. However, there are a number of pitfalls that arise because the implementation focuses on backward compatibility with existing Java virtual machines. The Java compiler erases all generic type information after the typechecking phase, replacing all references to type parameters with their upper bound [70], usually `java.lang.Object`, and inserting type casts into the bytecode where necessary. Such casts never fail for correctly written generic code, but nevertheless impact the runtime performance. Worse, type erasure dictates the source-level semantics; the loss of generic type information at runtime leads to certain operations with generic types being either forbidden (e.g. allocating an array of a generic type) or unchecked (casting to a generic type). But the insuperable limitation of this approach in the context of Virgil is the automatic boxing of primitive types (`int`, `char`, `short`, `boolean`, `long`, etc), which requires memory allocation. Boxing is necessary with type erasure because the one (type-erased) implementation must have a single data representation for variables of the parametric type. Because Java primitive types are not classes and have different data representations than object references, the implementation requires boxing to ensure that all values have the same machine representation.

C++ uses *templates* to parameterize a section of code, either a class or method, over a type parameter. The code is duplicated for each instantiation of the type parameter. Because typechecking of the duplicated code happens for each instantiation individually, C++ templates are not typically considered a generic or parametric type system, but more of a macro or code-duplication mechanism. The advantages of C++ templates include the ability to write very terse code due to the use of operator overloading within the template, as well as good performance due to implicit and explicit inlining, the lack of dynamic type tests or boxing, and template meta-programming techniques [8]. However, the downsides are that type errors in the template code can manifest themselves at usage sites and the possibility of exponential code explosion due to the aggressive duplication of code.

C#'s parametric type system does not perform type erasure, but preserves all parametric type information in the bytecode [61]. All objects and meta-objects carry all of the type information at runtime, meaning that every type is *reified*, or has a runtime representation. This allows all operations on parametric types to be supported, including allocating arrays of a parametric type and casting to a parametric type. C# also allows type parameters to be bound to any type, including primitive types, which means that the data representation and therefore object layout might vary, potentially requiring multiple versions of the code. Code duplication is performed by the virtual machine on-demand as new instances of a parameterized class are allocated or new instantiations of a parameterized method are called, with some sharing if the generated code is identical. Runtime duplication of the code means that the number of runtime types can potentially be unbounded; for example, for any type `T`, the program can allocate a new `List<T>`, leading to new types such as `List<List<T>>`, `List<List<List<T>>>`, and so on.

Scala [74] is a relatively new object-oriented language that offers parametric types as well as some functional programming constructs. Scala compiles to Java bytecode; Scala programs and their language runtime execute together on an unmodified Java virtual machine. Because of the implementation platform, Scala implements parametric types through type erasure, like Java 5. However, unlike Java, every Scala value is an object and there is one universal supertype called `scala.Any`, which avoids some of the problems with Java's primitive types. This also means that all values are object references and therefore have a common machine representation, which makes a single, type-erased implementation of generic methods and classes feasible. However, this still requires dynamic memory allocation, which is infeasible given the Virgil design constraints.

```
class Tree<K, V> {
    field key: K;
    field val: V;
    field left: Tree<K, V>;
    field right: Tree<K, V>;
    method add(k: K, v: V) { . . . }
    method find(k: K): V { . . . }
}
```

Figure 3.1: Virgil Type Parameters
This example shows a binary tree implementation using Virgil's parametric type system.

The design of Virgil's parametric type system is closest to C#. However, instead of dynamic instantiation and specialization of types by a virtual machine, the Virgil compiler performs specialization statically. The number of types is bounded because Virgil does not allow dynamic memory allocation; the only types that can exist at runtime are the types of those objects in the live heap allocated by the initialization phase. The availability of the complete heap after initialization also reduces the amount of code duplication, since only the code of reachable live objects needs to be duplicated. Virgil does not allow wildcards in parametric types, which forbids heterogenous collections such as a list of lists of any type. The lack of wildcard types also means that the multiple-inheritance problem that arises when duplicating class hierarchies does not arise (see [14] for details). Currently, the Virgil compiler does not maximize sharing of duplicated code, and there is some room for improvement in the implementation.

3.2.4. Components

In addition to classes and simple inheritance, Virgil contains a singleton mechanism called a *component* that serves to encapsulate global variables and methods. While Java allows static members inside of classes, all members in Virgil class are instance members. Components are used to encapsulate those members that would be declared `static` in Java. This provides for global state and procedural style programming, but within modules. This explicit separation of static and instance concepts reduces problems of incomplete abstraction (e.g. hidden static state in classes), and makes the separation apparent to both programmers and program reasoning tools. Components require no metadata to implement, since they are not first-class values. They cannot have type parameters like classes, although their methods can. Components also serve an important purpose that will be explored more in Section 3.3: they encapsulate the initialization portion of the program and their fields serve as the roots of the live object graph.

3.2.5. Delegates

Purely class-based languages have one important drawback that design patterns such as the Adapter, Observer, and Visitor [43] attempt to address; for different modules to communicate, they must agree not only on the types of data interchanged, but the *names* of the operations (methods). This is manifest in the proliferation of interfaces that serve to name both types and methods for interchange between modules. Some view this as a language flaw that can lead to needlessly complicating applications and libraries with interfaces. Parametric types are

```
class List {
    field head: Link;
    method add(i: Item) { . . . }
    method apply(f: function(Item)) {
        local pos = head;
        while ( pos != null ) {
            f(pos.item);
            pos = pos.next;
        }
    }
}
component K {
    method printAll(l: List) {
        l.apply(print);
    }
    method append(src: List, dst: List) {
        src.apply(dst.add);
    }
    method print(i: Item) { . . . }
}
```

Figure 3.2: Components and Delegates
Example code in Virgil that demonstrates the use of components and delegates. Component `K` contains static members and data. The `List` class provides an `apply()` method that accepts a delegate, which `K` uses to implement `printAll()` and `append()`.

only a partial solution to this problem. Functional programming paradigms have a more elegant solution to this problem and allow first-class functions to be used throughout the program based only on type signatures. Unfortunately, implementing higher-order functions in general can require allocating closures on the heap, and Virgil does not allow any dynamic memory allocation.

Virgil makes a compromise between the functional paradigm and the object paradigm by borrowing from C# the *delegate* concept, which is a first class value that represents a reference to a method [1]. Delegates in Virgil are denoted by the types of their arguments and their return type, in contrast to C# where in addition to the argument and return types, a delegate type must be explicitly declared and given a name before use. Thus a delegate in Virgil is more like a first-class function in any statically typed functional language than an object concept as it is in C#. A delegate in Virgil may be bound to a component method or to an instance method of a particular object; either kind can be used interchangeably provided the argument and return types match.

Delegate uses in Virgil do not require any special syntactic form for their use. Rather, delegate syntax generalizes the common `expr.method(args)` notation for instance method calls, by allowing `expr.method` to denote a delegate value and `expr(args)` to denote applying the delegate expression `expr` to the arguments. This retains the familiar method call syntax of Java, but allows delegates to be created by simply referring to the method name as if it were a field. See Figure 3.1 for an example.

The Virgil compiler implements all delegate operations, including creating, assigning, and applying delegates as efficient, constant-time operations that do not require allocating memory. At the implementation level, a delegate is represented as a tuple of an object pointer and a function pointer. A delegate tuple is not allocated on the heap, but is represented as two scalar variables or two single-word fields, depending on where it occurs. When the programmer uses an object's method as a delegate, the receiver method is resolved dynamically as in a virtual dispatch, and the object reference and the resolved method pointer constitute the delegate tuple. Referring to a component method as a delegate creates a tuple with `null` as the object. Invoking a delegate with its argument values is implemented as a simple indirect function call, passing the bound object reference as the hidden `this` parameter. Since method resolution takes place at creation time rather than invocation time, delegate invocations actually require one fewer memory access than a virtual dispatch, and require no memory accesses if both the receiver object and method are in registers. Further, the scalar variables representing the object reference and the method reference of a delegate tuple can be subjected to standard compiler optimizations such as constant/copy propagation, code motion, etc.

In contrast, delegates in C# are compiled to an *intrinsic* `Delegate` class supplied by the compiler; using delegates requires both dynamic memory allocation and reflection mechanisms in the runtime system. However, C# also supports multi-cast delegates, where a delegate can refer to multiple methods and invoking it invokes all methods. Virgil does not support multi-cast delegates.

3.2.6. Raw Types

Low-level code such as device drivers often has to manipulate data that is encoded in specific patterns of bits. For example, often a hardware register for controlling a device will be divided into several subfields, where some bits select the operating mode, another bit enables the device, etc. Recent work by Diatchki, Jones and Leslie [33] has explored adding facilities for specifying bit-level representations of data types in functional languages. Bacon [19] designed a language where the basic building blocks are bits and all other data types are derived. However, mainstream languages such as C and Java still force programmers to express such bit-level operations with masks and shifts on integers, often mixed with hexadecimal constants representing bitmasks. Such code is tedious to write and get correct; it is also very often obscure and ugly to read.

Virgil defines a family of types that correspond to bit-level quantities that are inspired by work done by Redwine and Ramsey [83]. The types `1`, `2`, `3`, to `64` define fixed-width bit quantities called *raw types*. Raw types are primitive value types like integers and booleans. Assignment and promotion rules are defined naturally to capture the essence of working with bits: i.) a smaller raw type can be assigned to or used in the place of a larger raw type, with promotions always filling the upper bits with zero, and ii.) assigning a larger raw type to a smaller raw type requires an explicit conversion which discards the upper bits. Integers, booleans, and characters can be implicitly converted to their raw representations, but conversion from raw types back requires an explicit conversion. References can never be converted to bits or vice versa.

A programmer can write hexadecimal, octal, and binary literals in Virgil programs. The length of the literal determines its size in bits, and therefore the resulting raw type. For example, a binary literal `0b1000` has four bits, and therefore raw type `4`, while a hexadecimal constant `0xf4c` has three hex characters, therefore it has raw type `12`. Unlike other languages that define the bitwise operators such as *exclusive-or* and shifts on integral types, bitwise operators in Virgil apply only to raw types. Each operator defines its result type naturally from the types of its operands, which helps the programmer ensure that the expressions compute the intended result and that the resulting storage location has enough bits. The shift operators (<< and >>) are defined to operate within a window that is the same size as the raw type being shifted; this makes the operation's semantics independent of the width of any particular machine's registers. Virgil also overloads the array subscript operator `[]` to allow the programmer to access and update individual bits within raw values. This helps to improve the terseness and readability of bit-level code.

3.2.7. Hardware Registers and Interrupts

Microcontrollers typically define a particular region of memory for communicating with and configuring on-chip devices such as an analog-to-digital converter (ADC), timer, or USART. Each individual device defines a set of registers that lie at known addresses within this region of memory. For example, the Timer0 device on the ATMega128 defines an 8-bit register named TCNT0 that contains the current counter value, as well as a TCCR0 register that is used for configuration. A device driver written in C or assembly typically uses explicit pointers the known memory addresses in order to access individual registers. Of course, a software device driver usually hides the details of these hardware registers and offers a simplified interface to higher levels of software.

Virgil has support for directly accessing these hardware I/O registers in a controlled way, without having to resort to calls to native methods, indirect accesses through pointers, VM tricks, or other magic holes in the type system. Instead, the hardware registers with fixed memory addresses in the I/O space are exposed to the program as named fields of a special component named `device` that can be read or written with raw types only. The names and locations of these registers are defined by a machine specification that is distributed with the compiler and selected by the programmer when targeting a specific device. The compiler will arrange the heap in memory so that objects and data structures do not overlay the I/O space. Accesses to these registers are always direct, by name, and thus the program cannot inadvertently alter the contents of the heap through indirect pointers.

On-chip devices can also generate interrupts that must be handled by device drivers. Virgil allows the programmer to specify individual component methods that are connected to particular hardware interrupts, allowing a complete hardware device driver to be written entirely in Virgil, without any underlying unsafe code.

3.2.8. Concurrency Model

On a desktop or server system with a true operating system, preemptive multitasking is normally provided by the kernel, which manages stacks and multiplexes processes or tasks on the CPU (or CPUs) to provide concurrency because each underlying CPU offers only a one-stack execution model with hardware interrupts and traps. Although Virgil does not currently have a formal concurrency model, it mirrors the hardware's one stack model and exposes the hardware interrupts as entrypoints. Virgil does not offer synchronization primitives, but allows access to the hardware state that enables and disables interrupts. Therefore, the task of providing mutual exclusion is currently left to the programmer, e.g. by disabling all interrupts or a specific interrupt within critical sections.

Incorrectly synchronized programs can have unpredictable results, which is why some languages such as nesC offer synchronization primitives built into the language and a phase which performs race condition checking, warning about possible synchronization violations. In the future, Virgil will offer an atomic region construct similar to nesC's `atomic` statements and offer a similar verification phase.

3.2.9. Virgil Anti-Features

There are a number of language features available in modern object-oriented languages that have important expressiveness benefits but nevertheless cannot be comfortably supported given the design constraints. Section 3.1 has already discussed the Virgil inheritance model that allows efficient object implementations by removing features such as interfaces, but the design constraints have led Virgil to omit a number of features that entail large metadata and runtime overheads, such as:

i. **Locks.** Synchronization primitives require runtime support in the form of locking and unlocking operations. This includes natively implemented atomic instruction sequences and spin loops, but most importantly queues, which consume memory. Wait queues also assume a threading model; a microcontroller is generally a one-stack system without real threads.

ii. **Class loading.** Dynamically loading new classes into the program is generally not needed for the types of programs that are written for microcontrollers. Additionally, dynamic class loading requires attaching significant metadata to the classes so that the host system can integrate the code into its current view of the program's type system. This requires a significant runtime support structure. Additionally, dynamic loading can invalidate almost any interprocedural compiler optimization, which forces a static compiler to be overly conservative.

iii. **Reflection.** The ability to reflectively inspect the members of objects and modify them by name requires a substantial runtime support system that carries significant metadata with the program. Large cost aside, the development model of microcontroller programs would tend to suggest that runtime reflection and dynamic configuration techniques should rather be replaced with static configuration mechanisms.

iv. **Garbage collection.** Garbage collection is simply unnecessary because no dynamic memory allocation is allowed. Instead, programs in Virgil must statically allocate all of their needed memory during compilation.

v. **Method Overloading.** C++, Java, and C# all allow overloading methods by their parameter types. Although overloading is a purely syntactic form of polymorphism and thus has no inherent runtime cost, it ruins the simplicity of Virgil's delegate mechanism. Because Virgil supports using a method as a delegate by simply referring to it by name, overloading would introduce ambiguity and require a clumsy resolution mechanism.

What remains in Virgil is a simple but elegant set of object-oriented, procedural, and functional concepts that all require very little metadata, no runtime support, and all support strong type checking, with minimal dynamic safety checks. The dynamic checks required in Virgil are inserted automatically by the compiler and optimized where possible. These are explicit null checks, array bounds checks, subtype tests for explicit downcasts, and division by zero.

3.3. Program Initialization

Many embedded and real-time programs have a natural separation between application start up, where global data structures are allocated and initialized, and steady state execution where events are handled and the main computation is carried out. For example, an operating system allocates data structures associated with process tables, memory management, device management, caches, and drivers once when it boots and then reuses them through its lifetime.

```
class List<T> {
    field head: Link<T>;
    method add(i: T) { . . . }
}
component K {
   field a: List<int> = new List<int>();
   field b: List<int>;
   constructor() {
      b = new List<int>();
      add(a, 0);
      add(b, 1);
   }
   method add(l: List<int>, i: int){
      l.add(i);
   }
}
```

Figure 3.3: Program Initialization
Example initialization code in Virgil that demonstrates the use of component constructors. Component field initializers and the `constructor()` method are run inside the compiler before generating code.

Because the core Virgil language has been carefully designed to allow applications to execute on the bare hardware without any supporting software or language runtime, it provides an explicit separation between *initialization time*, where data structures are allocated and initialized to a consistent state, and *run-time*, where data structures will be manipulated but no longer created or destroyed.

Each component in a Virgil program can optionally contain a *constructor*, much like an object's constructor, that contains code that initializes the component. The Virgil compiler contains an interpreter for the complete language and provides an *initialization* environment for this constructor that is richer than the run-time environment. Constructors execute *inside* the Virgil compiler, before any code is generated. The initialization environment allows unrestricted computation using all the language features; in particular the constructor may access other component's fields, allocate and initialize objects and arrays, call component and object methods, create delegates, etc. Because the initialization phase represents Turing-complete computation, it is of course undecidable to determine whether the constructors will terminate, and the Virgil compiler leaves this to the programmer. In the future, a timeout option could be provided along with other debugging facilities to examine the operation of the program's initialization phase if it goes awry.

In Virgil, initialization is considered an inseparable part of the compilation process for a program. Initialization requires the entire program to be available, since initialization code can transitively reference any part of the program. The assumption of whole-program compilation is justified in this domain because when building a standalone program for an embedded device there is always a point, traditionally link time, where the complete binary is put together. The Virgil compilation model recognizes this as inevitable and makes it an integral part of the compilation process.

3.3.1. Initialization Determinism

Initalization of a Virgil program is always deterministic and determined by the program alone. This avoids one significant drawback of previous persistent systems such as Smalltalk, where replicating the initialization environment for a particular program could be nontrivial. The order in which component constructors are executed is given by the order in which the program files are specified on the compiler command line. However, dependencies between components can force initialization to happen earlier. For example, if the field of an unconstructed component `K` is accessed during the initialization of an earlier component `J`, then `K`'s constructor is invoked before the field operation completes. A cycle in constructor invocations cannot occur because a component is marked as constructed just before executing its constructor. Fields not explicitly given an initialization value, or fields that have not yet been initialized because of a cycle in dependent initialization, have a default value given by their type (e.g. `0` for `int`; `null` for arrays and objects).

3.3.2. Initialization Garbage

The built-in interpreter utilizes a general-purpose garbage collector so that any unreachable objects allocated throughout initialization are reclaimed. Upon termination of the application initialization phase, the compiler traces from the component fields through objects and object fields to discover the graph of objects that are transitively reachable from the roots. All unreachable objects are discarded, and only the code and metadata associated with live objects are included in the final program binary.

3.3.3. Code Generation and Runtime

After the identifying the reachable heap, the Virgil compiler will compile both the code and the heap of the program together into a single binary that can be loaded onto the device or executed in a simulator. When the program begins execution on the device, the entire initialized heap is available in memory and the program can manipulate these objects normally, reading or writing fields, invoking methods, creating delegates, etc. However, the program will not be allowed to allocate new objects, which eliminates the need for a runtime memory manager or a garbage collector.

3.4. Optimization

Careful adherence to the design constraints allows Virgil to be implemented straightforwardly and efficiently without a language runtime and with minimal metadata. In addition to the base efficiency of the straightforward implementation, basic optimization techniques can be applied. For example, the Virgil compiler will employ class hierarchy analysis [32] to devirtualize calls and delegate uses, as well as to identify degenerate orphan classes to be represented without object headers.

The availability of the complete program heap enables an advanced Virgil compiler to substantially improve on the base implementation with three new optimizations. The first, *reachable members analysis*, removes code, objects, and fields of objects that are unused in the program and is described in this section. *Reference compression*, covered in Chapter 4, exploits the language's type safety to represent object references in a compact way, and *ROM-ization* reorganizes object layouts to move read-only fields into the larger ROM memory. All three optimizations exploit the type-safe nature of the Virgil language and are made possible by the availability of the program heap at compile time.

3.4.1. Reachable Members Analysis

Initialization time allows a Virgil program to build complex data structures such as lists, queues, pools, maps, and trees during compilation for use at runtime. Garbage collection following program initialization uses the standard notion of transitive reachability through object references to discover the reachable heap and discard temporary objects. However, libraries or drivers used by a Virgil program may create data structures that are reachable through object references but are not actually used at runtime by the program.

This can arise in a number of scenarios. For example, a software device driver may create data structures that are only used if the hardware device is used by the program. Imagine a timer driver with an event queue used to trigger application events at specific future times; the queue is only necessary if the application actually uses this feature of the timer. Another example is when a device with many different modes of operation is used in only one particular mode. In other situations, an application may only use a subset of the functionality provided by a complex data structure; a doubly linked list that is only traversed forward will never use the back pointers, or a tree that is only searched and not modified may not need parent pointers in its nodes.

A compiler may remove objects and fields from the program, provided they are never accessed upon any execution. This is especially important when compiling an application that reuses drivers, modules, and data structures that provide more functionality than is needed for the program. The compiler need only generate the code and include live data structures, reducing the total memory footprint of the program.

There are numerous techniques for dead code elimination and data structure reduction [95][100], but the consideration of initialization code leads to overly conservative approximations. In general, removal of dead code requires computing a sound set of reachable methods and requires approximating the possible receiver methods of dynamic dispatches in the program. Unlike all previous work, the explicit separation of initialization time and run time in Virgil eliminates the need to consider initialization code: the availability of the complete program heap provides access to all of the objects that will be manipulated by the program at run time.

Now we are ready to state the reachable members problem and begin exploring possible solutions.

Reachable Members Problem: Given (**P** a Virgil program, **R** a set of initialized root fields, **H** the initial heap of objects, and **E** initial methods representing entrypoints into the program), which methods in **P** and which fields **F** of object instances in **H** might be accessed on some execution of **P**? As stated, the problem is clearly undecidable, reducible to the halting problem. So we will consider sound approximations that are less precise.

3.4.2. Classical approaches

Let's first sketch a general idea of how a compiler might approach this problem. The classical solution would be to begin analyzing the code of the entrypoint methods **E** and build a call graph that represents the set of reachable methods. At virtual method and delegate invocation sites in the program, the algorithm would use some conservative approximation of possible receiver methods, leading to a conservative approximation of the reachable methods that may include some methods that are dead. After computing a set of all live methods in the program, the compiler would analyze the code of each method for accesses of root fields **R** and instance fields of objects. Then, the compiler would remove unused root fields as well as unused fields in objects in the heap.

Following this approach, what approximation is appropriate at each invocation site? We might use a simple analysis such as CHA, which considers the class hierarchy of the program and the static type of the object reference at the call site to determine a set of reachable method implementations. However, this approximation may be too conservative because CHA considers all the code of all classes declared in the program, including ones that may not have instances in the heap **H**. Another approach might be to only consider the classes of objects that have live object instances in the heap **H**. This would be similar to Rapid Type Analysis [21], which maintains a set of possibly live types during analysis by inspecting the object allocation points of the program. This second approach is more precise than CHA because only method implementations corresponding to live objects in the heap are considered. However, simply using the existence of any object of a particular class in the initial heap may be too imprecise, because after removing dead objects, the set of live types might also be reduced. Another iteration of the algorithm may be able to further reduce the set of reachable methods because the approximation of each call site may become more precise. In general, the algorithm might need to iterate to a fixpoint to get the least solution.

There are some situations where even the fixpoint will not give the best result. For example, a *liveness cycle* can arise where a class has a method that contains the only use of a root field, and that root field is the only path by which objects of that class are reachable in the heap. In this case, the existence of the object in the heap forces consideration of the method, which forces the root field to appear live, which forces the object to be considered live, even if the field is not used elsewhere in the program. Iterating the RTA analysis will not discover the field, and therefore the method, is dead.

Figure 3.4 contains an example program for analysis that illustrates this liveness cycle problem. Note that the component field initializers are run in the compiler, and by the time analysis begins, these fields refer to actual live object instances in the heap, which we will call object `A1`, `B1`, and `C1`. The initial assumption of CHA is to ignore the heap and assume that the call to `m()` in `Main.entry()` can reach any of the three implementations, considering them live; however it correctly discovers that field `Main.h` is unused because there are no references to it in any of the code. Now consider RTA, where the first iteration assumes that `A.m`, `B.m`, and `C.m` are reachable because objects of those types exist in the heap; RTA therefore concludes that `Main.g` is used because it is used in `B.m`. After the first iteration, RTA can eliminate field `Main.h` and object `C1.` Upon beginning the second iteration, `C.m` is no longer live because `C` has no live instances in the heap; however, RTA still considers the code in `B.m` live and therefore `Main.g` is still live.

```
component Main {
    field f: A = new A();
    field g: A = new B();
    field h: A = new C();
    method entry() {
        while ( true ) f = f.m();
    }
}
class A {
    method m(): A { return this; }
}
class B extends A {
    method m(): A { return Main.g; }
}
class C extends A {
    method m(): A { . . . }
}
```

Analysis	Methods	Fields	Objects
CHA	`Main.entry` `A.m` `B.m` `C.m`	`Main.f` `Main.g`	`A1` `B1`
RTA (1)	`Main.entry` `A.m` `B.m` `C.m`	`Main.f` `Main.g`	`A1` `B1`
RTA (2)	`Main.entry` `A.m` `B.m`	`Main.f` `Main.g`	`A1` `B1`
RMA	`Main.entry` `A.m`	`Main.f`	`A1`

Figure 3.4: Analysis Comparison
Example Virgil program used to compare analysis precision. A liveness cycle exists involving the method `B.m` and the field `Main.g` preventing CHA and RTA from computing the most precise result. The table on the right gives the analysis results for CHA, two iterations of RTA, and RMA.

The core imprecision of classical approaches to this problem is that they are not *data-sensitive*, meaning they do not operate in the context of the live object instances in the heap. The main weakness of CHA is that it doesn't consider live objects at all. RTA, however, is too imprecise because in each pass a class's method implementation is considered live if at least one instance of the class exists in the heap, even if the object is later considered unreachable.

3.4.3. Reachable Members Analysis

Reachable members analysis addresses the imprecision of classical approaches by analyzing code and objects together as they become reachable from the entry points of the program. RMA is an optimistic algorithm and initially assumes that nothing is reachable. By pulling in objects, methods, and fields in an on-demand fashion, it avoids the imprecision inherent in the CHA and RTA analyses. Before beginning the detailed algorithm, consider a conceptual outline. RMA begins at the entrypoint methods analyzing the code of each method by inspecting reads of root and object fields. For a use of a new root field, it considers the field to be live and puts the object referenced by the field into the live object set. For a use of a new object field, RMA considers that field live for every object of that type; for every object in the live set, it transitively pulls in objects reachable through the new field. For a new method invocation, it

```
info: Map<Type, {members: Set<MemberName>,
                 subtypes: Set<Type>,
                 instances: Set<Object>}>
methods: Set<Method>

(1) analyze(Program p) =
    foreach( Method m in p.entrypoints )
        post(m)
    while( !empty(worklist) )
        analyze(dequeue(worklist))
(2) analyze(Method m) =
    methods.add(m)

    foreach ( Expr e in m.body )
       if ( e = read(C.f) ) post(C, m)
       if ( e = read(e.f) ) post(type(e), m)
       if ( e = call(C.m) ) post(C, m)
       if ( e = call(e.m) ) post(type(e), m)

(3) analyze(Type t) =
    info[t].subtypes.add(t);
    foreach( Type p in parents(t) )
        info[p].subtypes.add(t)
    let pm = info[parent(t)].members
    foreach( Member m in pm )
        post(t, m)
(4) analyze(Type t, Field f) =
    info[t].members.add(f)

    foreach( Object o in info[t].instances )
        post(value(o.f))
    foreach( Type s in info[t].subtypes )
        post(s, f)
(5) analyze(Type t, Method m) =
    info[t].members.add(m)

    foreach( Type s in info[t].subtypes )
        post(resolve(s, m))
(6) analyze(Object o) =
    post(type(o))

    info[type(o)].instances.add(o)
    foreach( Field f in info[t].members )
        post(value(o.f))
```

Figure 3.5: RMA Algorithm
Data structures and analysis rules for each type of work unit. The `post()` method produces a new work unit of the corresponding type and inserts it into the worklist if the unit of work has not already been performed.

considers only method implementations corresponding to classes that have instance objects in the current live set. The algorithm iterates until there are no new method implementations or objects to analyze.

Figure 3.4 contains the core of the RMA algorithm. The two central data structures used in the RMA algorithm are `info`, a map from a class or component type to a set of used members, instantiated subtypes, and object instances; and `methods`, a set of the currently reachable methods.

The `info` data structure is initialized for every type in the program with an empty entry, and the `methods` set is initially empty. The analysis is organized into five different units of work that are all inserted and removed from a central work list. The work list is processed in order, and each kind of unit of work may produce new units of work to be inserted in the list and performed later. One can view the algorithm as recursive, with the work list implementing memoization for termination. The five types of work units are:

i. **New Method.** This unit represents a previously unseen method that contains new code to analyze.

ii. **New Type.** This unit represents a new instantiated type that has not been encountered before.

iii. **New Field Access.** This unit represents a previously unseen field access of a class or component.

iv. **New Method Access.** This unit represents a new access to a method of a class or component.

v. **New Object Instance.** This unit represents a new object instance that has been discovered to be reachable in the heap.

When a new unit of work is available, the `post()` method is called with that unit. The post method is analogous to the `analyze()` method, and is overloaded for each type of work unit. The `post()` implements a form of memoization; it always checks to see whether the unit of work has already been performed or is already pending before placing the unit in the work list.

Let's examine the work units in detail. Imagine that we are running the analysis algorithm by starting at (1), and initially begin processing a work unit of type (2) on the entry method of the program. At this point there are no objects yet considered reachable, and nothing in the main data structures. The work unit (2) iterates over the statements in the method; if the program reads a component field, the analysis posts a new unit of work of type (4) to analyze the component field later. Similarly if (2) detects a read of an object field, then a work unit (4) is posted on the type of the expression and the field name. The analysis treats component and object field accesses are together in (4) by considering a component to be a class with a single instance in its `instances` list. Work unit (4) analyzes the new field for all live objects in the `instances` list, posting the objects those fields reference into the work list, and then recursively posts a work unit (4) on each of the instantiated subtypes with the same field. For a virtual method call, the work unit (5) resolves the method implementation for the static type and posts the method to be analyzed later by (2). To process a new object instance, the work unit (6) first posts a work unit on the object's type (3), which integrates the type into the lists of its parents and posts any fields or method accesses performed on the parent on the new type, and then analyzes the fields of the new object.

RMA's worse case complexity is quadratic in the number of declared fields in the program, but this only occurs for pathological inheritance scenarios. The source of nonlinearity is the repeated posting of field members from a super-class to its instantiated subtypes (4), which happens at most once per field per subtype, which in the worst case is quadratic. For simple hierarchies, the algorithm runs in linear time. RMA analyzes the code of each reachable method at most once, since it need only glean from the body the static types of field and method accesses. Secondly, each object instance that the analysis considers is added to exactly one `instances` list, since each object has exactly one dynamic type. The `instances` list for a type may be processed multiple times, but at most once per new field encountered, thus each field of each reachable object is inspected at most once, either when the object instance is first encountered, or when a new field read is encountered in the program. A less precise result could be obtained by only keeping field access information in the type where the field was declared. This would reduce the worst-case complexity, but would reduce precision.

The algorithm as presented can be used to compute the necessary information for the *pull members down* optimization that moves fields from a super-class to its children classes if it is unused in the super-class, which saves space in instances of the super-class.

This transformation originally appeared in automated refactoring tools, but admits a small opportunity for space savings here. Tyma in [106] describes field percolation, where members are pulled up into super-classes when possible. This reduces the meta-data per class, but potentially increases the size of objects if the super-class is instantiated.

3.4.4. ROM-ization

Reachable members analysis can also be used to statically determine an approximation of which component fields and object fields the program may modify. For example, if no writes to a particular component field exist in the program, then that field will remain constant throughout any execution and the compiler will simply replace accesses to this field with its value and remove the field. For object fields, if no writes exist to a particular object field, then for all instances of the object in the program, the corresponding field will not change value over the execution of the program. These fields can be factored out of the object and stored in the ROM.

There are various techniques to represent the constant. If it is the same value across all object instances, the compiler can inline it as a constant wherever reads occur. If it is constant by subclass [17], the compiler can move it to the meta-object, and otherwise, the compiler can store the field in ROM. It may choose to split the object into a read-only and a read-write portion; either a hash table or a pointer from one to the other can associate the two halves of the object. More techniques for ROMization are discussed in Chapter 4.

3.4.5. Metadata Optimizations

In addition to optimizing the layout of objects within the heap, the compiler can also perform a number of optimizations on metadata, including the meta-objects and the object headers. First, the compiler can use the results from reachable members analysis to optimize the meta-object itself. The reachable members analysis computes which virtual methods are used within the program and the unused entries in the meta-object can be removed. Similarly, the slots in the meta-object that correspond to methods that have been fully devirtualized can be removed. The compiler can also compress the object header, which normally contains a direct pointer to the meta-object, replacing the pointer with an index into a meta-object table. This can allow the object header to be compressed to only a few bits. Third, since the meta-objects are read-only throughout the life of the program, they can be stored in the ROM to save precious RAM space.

3.5. Experience

We have implemented a prototype compiler that supports the complete Virgil language, including a front-end that parses and typechecks the program, an interpreter that runs the initialization phase to obtain the complete program heap, a middle portion that implements reachable members analysis and performs optimizations, and a backend that produces C source code. The compiler totals approximately 48,000 lines of Java code including all comments and documentation. It transforms the Virgil program and emits all of the code into a single C source file, including the live objects in the heap, their metadata, and all the reachable code. This C program includes all code necessary to run on the bare device, and does not require the use of any libraries, including `libc`, the C language runtime. Of course, this intermediary C code generation step is not intrinsic in the language compilation, runtime, or linking model; a production Virgil compiler would output native code directly.

The Virgil compiler is open source. In May 2007 the third version of the Virgil compiler system was released, which includes more extensive documentation and a suite of example programs. The source code is covered under a BSD-like license, which grants copying and redistribution rights provided the copyright notice is left intact. The entire system is available for download at:

```
http://compilers.cs.ucla.edu/virgil
```

3.5.1. AVR Driver Libraries

Based on my example driver for Timer0 device on the ATMega128 microcontroller, undergraduate students Akop Palyan and Ryan Hall developed software drivers for most of the on-chip AVR devices during their Winter 2007 quarter project. Both students had intermediate experience with Java, but neither student had prior experience with Virgil. Both were able to learn the language very quickly and had their first working drivers within three weeks. Their rapid absorption of the language may be partly due to the total lack of any APIs or standard libraries that they needed to learn which allowed them to start from scratch and build a completely standalone world.

The device driver suite is written entirely in Virgil, without any underlying unsafe C or assembly code, and offers a simplified interface to the hardware devices based on queues and events. Development and testing of the drivers was done with the Avrora [105] cycle-accurate AVR emulator that we built in 2004 and 2005, as well as on actual Mica2 sensor network nodes in our lab. Avrora provides detailed instrumentation and measurement capabilities [103] that proved to be invaluable during development. Based on the students' feedback, a simple Avrora monitor was developed that generates a source-level stacktrace when a Virgil program throws an exception while running in the simulator. Virgil's static type system backed with dynamic checks proved to be extremely useful to both students in diagnosing program errors that would have otherwise manifested themselves in mysterious crashes and resets had the programs been written in C or assembly code.

Available device drivers include the analog-to-digital converter (ADC), serial driver (USART), and the serial peripheral interface (SPI). Each driver has an associated test program that can be used to test the driver in the simulator and on the hardware. A driver for the CC1000 external radio chip is now partially working, which will allow further layers of software to implement a protocol stack so that applications can communicate with other sensor nodes, including those that may be running another operating system such as TinyOS or SOS.

3.5.2. Benchmark Programs

This section uses 13 Virgil programs that are drawn from several disparate sources. `Blink` is a simple test of the timer driver, toggling the green LED twice per second; `LinkedList` is a simple program that creates and manipulates linked lists;

`TestADC` repeatedly samples the analog to digital converter device; `TestUSART` transmits and receives data from the serial port; `TestSPI` stresses the serial peripheral interface driver; `TestRadio` initializes the CC1000 radio and sends some pre-computed packets; `MsgKernel` is an SOS excerpt that sends messages between modules; `Fannkuch` is adapted from the Programming Language Shootout Benchmarks and permutes arrays; `Decoder` is a bit pattern recognizer and is discussed in more depth in the next section; `Bubblesort` sorts arrays; `PolyTree` is a binary tree implementation that uses parametric types; `and BinaryTree` is the same tree implementation but uses boxed values.

3.5.3. Exploiting Initialization Time - Decoder

After some experience writing code in Virgil, the initialization time concept has proved to be quite versatile. An application can use initialization time not only to initialize its state and allocate pools of objects, but it can build complex data structures, balance them, and run test cases on its own code. This can be especially useful when building complex data structures such as trees and maps that need only be constructed once and then repeatedly reused throughout the lifetime of the program. In this case, the program can perform data structure tuning at compile time to get the most efficient data structures possible.

One illustration of the flexibility that this mechanism provides is in the `Decoder` application. The `Decoder` application builds a b-tree which represents an efficient bit pattern recognizer that can be used to differentiate patterns of bits such as machine instructions, commands, network packets, etc. It is tedious to write the b-tree by hand, especially if it is encoded with several levels of switch statements. Instead, efficient algorithms exist to build a decision tree from a list of bit patterns in time linear in the number of patterns. The tree can even be reduced using techniques similar to those for BDDs [cite], yielding a directed acyclic graph. The `Decoder` application runs this algorithm during its initialization phase to produce and optimize the decoder data structure. Specifically, it creates a `DecoderBuilder` object in the constructor of the main application and inserts its specific set of bit patterns and then calls the `DecoderBuilder.build()` method. This method constructs the tree from the bit patterns, optimizes the tree, allocates the node objects, and then connects the nodes together, returning a reference to the completed decoder graph. The main program stores the data structure for use at runtime. After initialization terminates, the `DecoderBuilder` and its data structures are garbage collected automatically by the compiler. The program retains only the decoder data structure, which only contains only a few nodes that can be optimized by the compiler. Reachable members analysis will remove all the code and data structures that are unreachable from the entry point, so the complex initialization code for the `DecoderBuilder` is discarded.

3.6. Experimental Results

This section provides experimental results that demonstrate the space savings achieved by reachable members analysis. In addition to removing dead code and data, the prototype compiler uses RMA to inline the values of read-only fields where possible, reduce the size of meta-objects, and devirtualize method calls where possible. The impact of these optimizations on footprint and execution time are evaluated for the AVR architecture using the Avrora simulator [105]. The Virgil compiler emits C code that is compiled to AVR machine code using `avr-gcc` version 4.0.3 with an optimization level of `-O2`. The ROM-ization optimization described previously in this chapter is not currently implemented in the prototype compiler.

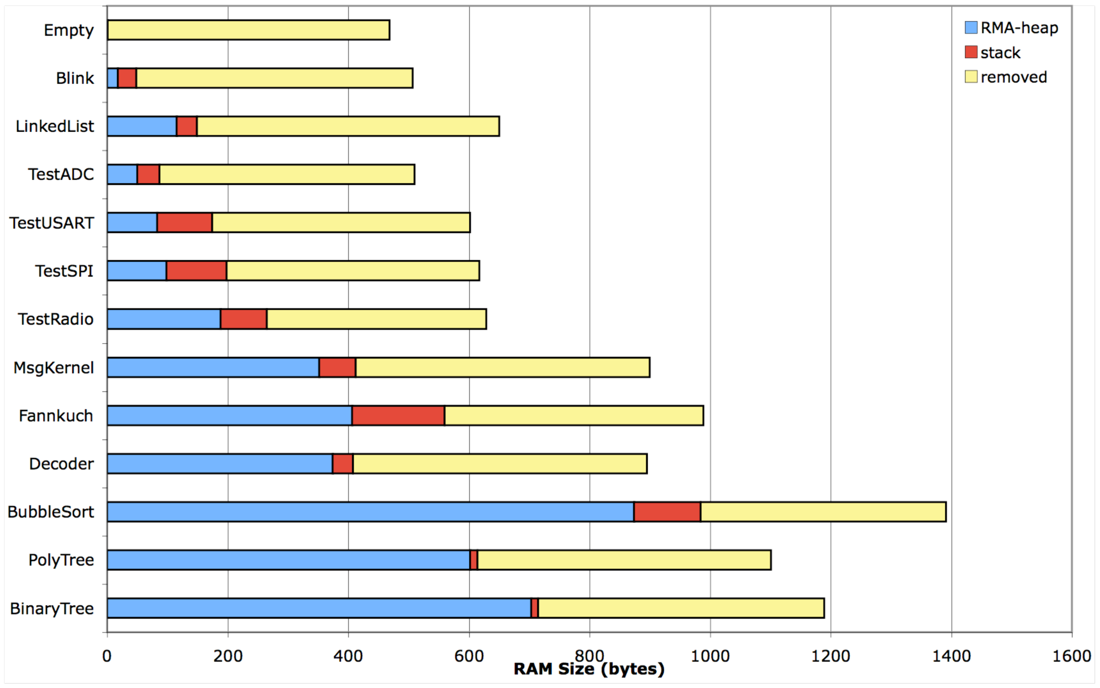

Figure 3.6: RAM Reduction

RAM reduction by applying the RMA optimization. The first section (blue) of each bar represents the heap that is live after applying RMA. The second section (red) represents the size of the dynamic program stack (obtained through instrumentation and unaffected by RMA). The last section (light yellow) is the size of the heap removed by RMA.

Figure 3.6 shows the reduction in RAM usage of the benchmark programs before and after applying the RMA optimization. The first section (blue) of each bar represents the heap size after RMA has been applied; the second section (red) represents the size of the runtime stack (included only for comparison, because RMA does not affect stack size); and the third section (yellow) of each bar represents the size of heap data removed by RMA. Thus, the total length of the bar represents the total RAM consumption if RMA had not been applied. First, notice that the empty program requires no RAM whatsoever, which means the fixed RAM cost of using Virgil is zero. Secondly, all applications fit comfortably in less than 1000 bytes of RAM; the larger heaps actually have more than 100 objects. The smaller applications have heaps that fit in less than 100 or 200 bytes of RAM, even with drivers that include large arrays, queues, and callbacks, demonstrating that it is feasible to build software for even the smallest of microcontroller models in Virgil.

This figure also illustrates how effective program slicing has altered the way Virgil applications are built. Instead of the developer tediously specifying which code in which files are required for each application, all programs in this suite are simply compiled against the entire driver library for AVR, regardless of what they actually need. These drivers internally require storage for numerous configuration fields, data structures, but RMA works so well that all of this is removed automatically. Some of these programs, of course, do use parts of the drivers, and the necessary data structures remain. Nevertheless, all applications benefit substantially from RMA—so much so that the application build process has changed and application programmers at the moment needn't even bother removing unused drivers their programs use in order to save space.

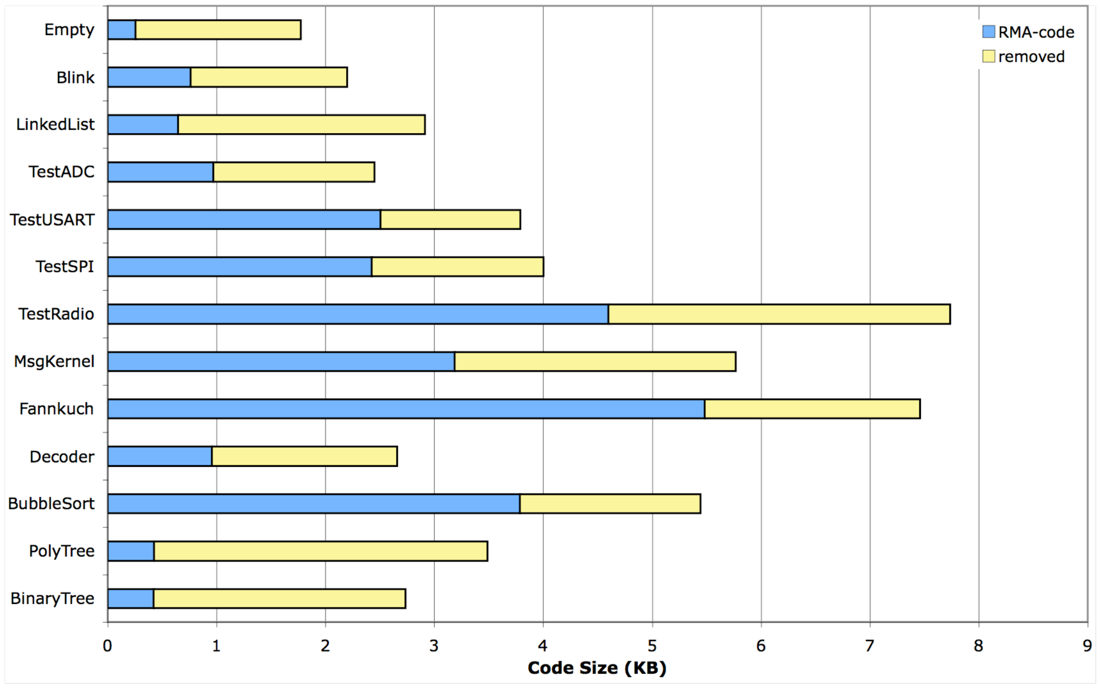

Figure 3.7: Code Size Reduction
Code Size reduction by applying the RMA optimization. The first section (blue) of each bar represents the size of the code that is live after applying RMA. The last section (light yellow) is the size of the code removed by RMA.

Figure 3.7 shows a similar comparison for code size. The first section of each bar (blue) represents the code size after RMA has been performed, and the second section (yellow) represents the code removed. After optimization, all applications fit in less than 6 kilobytes of code space, with 11 of 13 in less than 4KB and 6 of 13 fitting in less than 1KB. The `Empty` program requires just 262 bytes of code; this includes the interrupt table for the microcontroller (approximately 120 bytes) and some boilerplate code that is generated by `avr-gcc`. Here again we see the large amount of dead code from the driver library that is automatically removed.

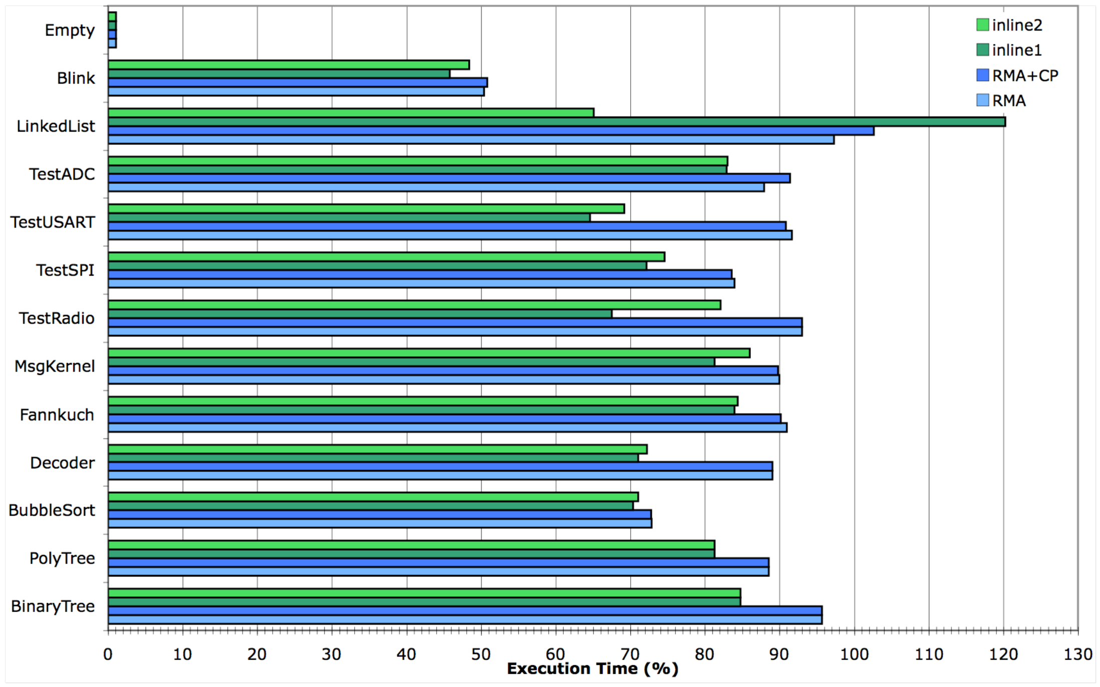

Figure 3.8: Normalized Execution Time

Execution time of four different optimization configurations, normalized to the `base` configuration with no optimization (i.e. `base` = 100%).

Figure 3.8 gives a comparison of execution time of the benchmarks over four Virgil compiler configurations. The time for each benchmark is normalized to the time of the base configuration (i.e. `base` = 100%). This chart compares four configurations: RMA: devirtualization with RMA; RMA+CP, devirtualization with RMA, followed by propagation of constant field values; inline1, devirtualization with RMA followed by some inlining; and inline2, devirtualization with inlining and further optimizations. RMA affects execution time because later passes in the compiler use the analysis results to devirtualize call sites and inline the values of read-only fields. The `Empty` program is a degenerate case and is just shown here for comparison; its execution time is dominated by the bootstrap code that loads the heap from ROM into memory—after RMA, there is no heap. `Blink` benefits highly because its main computation is a simple interrupt handler that after inlining becomes very trivial updating of hardware registers. Most other programs have between 10-15% performance improvement from RMA alone and about 20-30% performance improvement total when combined with inlining. Currently, little work has gone into improving inlining heuristics in the prototype compiler, which indicates there may be more room for even better results in the future.

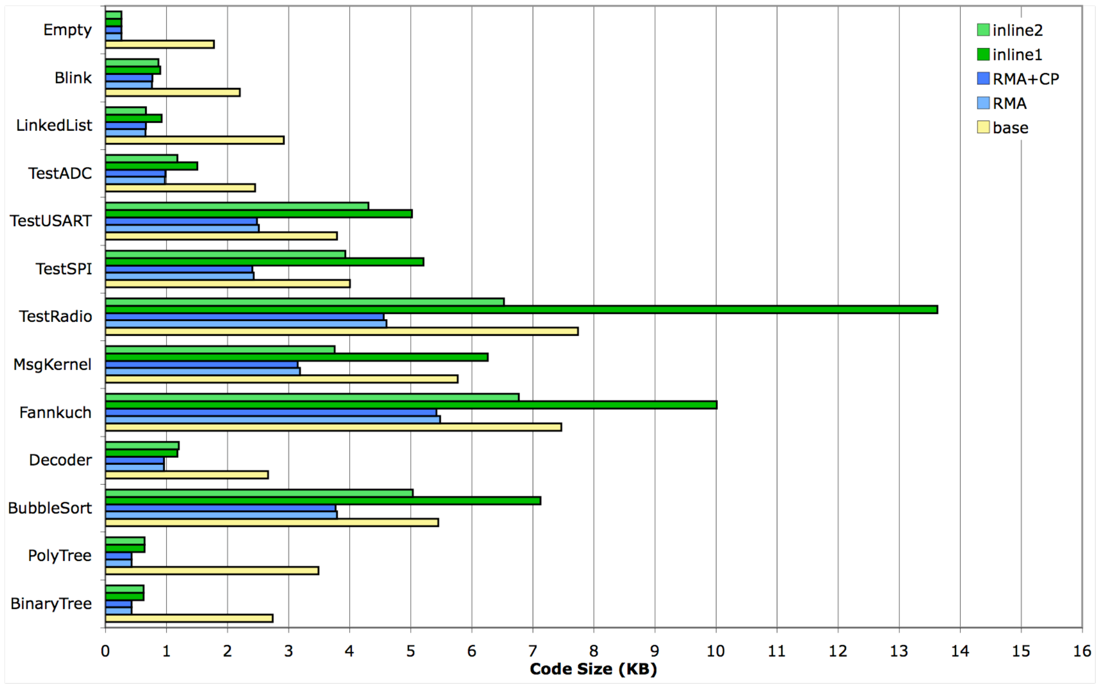

Figure 3.9: Absolute Code Size
Absolute code size of five configurations in kilobytes. The four configurations are the same as in Figure 3.8.

Figure 3.9 compares code size for all five of the compiler configurations mentioned in this section, including all of the results from Figure 3.6 and adding results for the two inlining configurations. Here we can see the unsurprising result that inlining increases code size, sometimes substantially. But the most interesting thing about this graph is that the code size of the `inline2` configuration is typically less than code size of the `base` configuration (i.e. without RMA). Thus, we can take the view that RMA removes enough dead code that it gives the compiler a “budget” for achieving better performance through inlining optimizations. This is a positive step towards a compiler than can make intelligent resource tradeoffs.

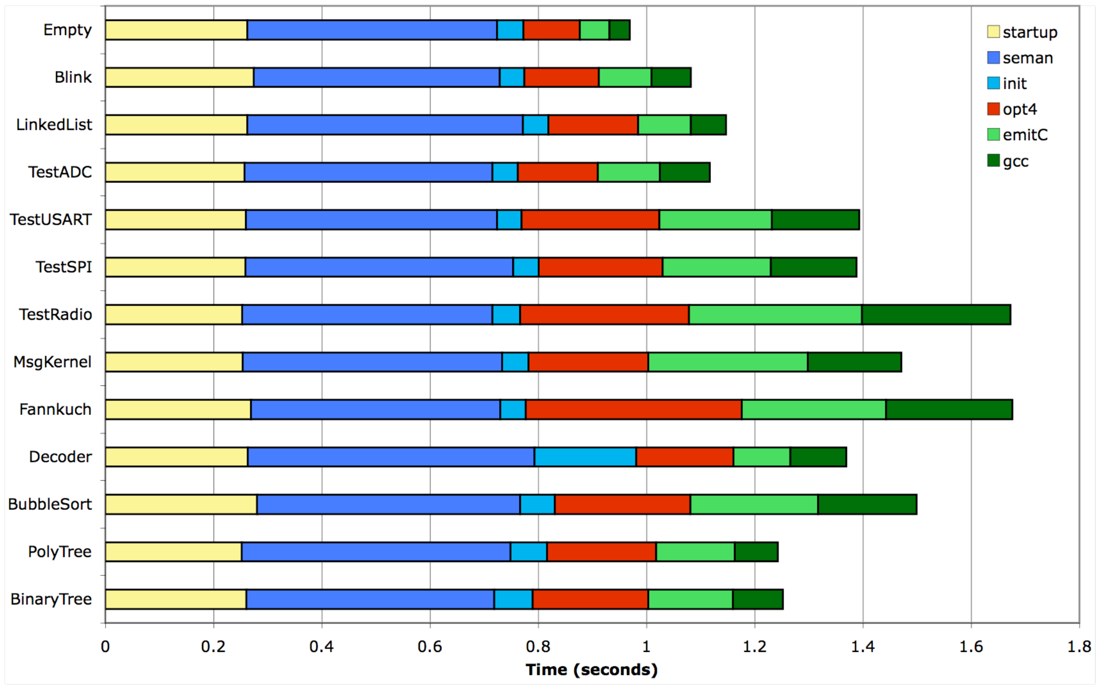

Figure 3.10: Compilation Time
This figure shows the compilation time of all applications, broken down into distinct phases, including parsing/typechecking, optimization, emission of C code, and compilation of C code to machine code.

Figure 3.10 shows compilation time for each of these benchmarks. The Sun Java 1.5.0 virtual machine is used to run the Virgil compiler on our Linux 2.6 server with two 3.06ghz Xeon processors and 4GB of memory. No application requires more than 1.7

seconds of compilation time. By far the dominant cost is parsing and type checking (second section, blue), followed by the compiler and JVM startup time (first section, yellow) and optimizations (fourth section, red). Initialization time cost (third section, light blue), where the compiler interprets the program's initialization code, is small for all applications, though proportionally larger for `Decoder`, which contains a complex initialization routine for building its main data structure. Emission of C code and `gcc` compilation time vary the most, since these are proportional to the size of the program coming out (as opposed to going in); these two are considerable for large programs, but negligible for small programs. Overall, the compiler is fast enough that whole-program compilation isn't even noticeable, yet it remains to be seen whether compilation time will become an issue for larger applications.

3.6.1. Effect of Safety Checks

The Virgil compiler inserts safety checks in the program to detect program errors such as null dereferences, array bounds violations, and failed type casts. While language runtime systems usually trap null dereferences using virtual memory techniques, the Virgil compiler must insert explicit null checks because the AVR microcontroller has no hardware support for null or bounds checks. To study the effect on code size and execution time, the Virgil compiler supports an option to disable these safety checks. This option is only meant for tuning of the compiler, and not for application programmers. Figure 3.11 gives the code size comparison for two new configurations. The `basenn` configuration is the same as the `base` configuration from the previous figures, but with safety checks disabled, while the `inline2nn` configuration is the same as the previous `inline2` configuration, but also with safety checks disabled. This figure gives the *respectively* normalized code sizes, where `basenn` is normalized against `base`, and the `inline2nn` is normalized against `inline2`.

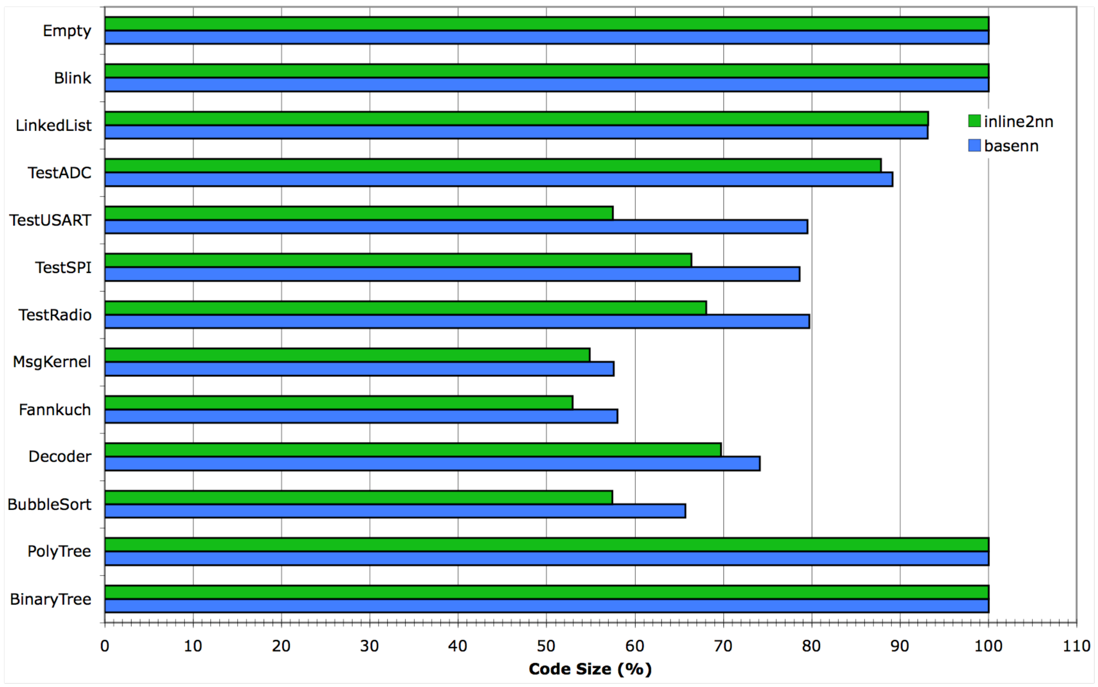

Figure 3.11: Normalized Code Size w/o Safety Checks
This figure shows the resulting code size when safety checks are disabled. All results are normalized to the RMA configuration.

Here we can see that there are four applications whose code size is not affected by safety checks. `Empty` contains no object operations, and hence no safety checks; `Blink` contains object operations that are optimized to remove safety checks and inline constants; both `PolyTree` and `BinaryTree` include null checks as part of their main logic, thus the extra null checks inserted by the Virgil compiler are actually redundant and optimized away by `avr-gcc`. Further, there are several applications with substantial code size reductions from disabling safety checks (up to 45%), particularly bounds checks. Currently, the Virgil compiler does not perform any safety check optimizations; techniques such as [59] and [23] could likely reduce the disparity significantly.

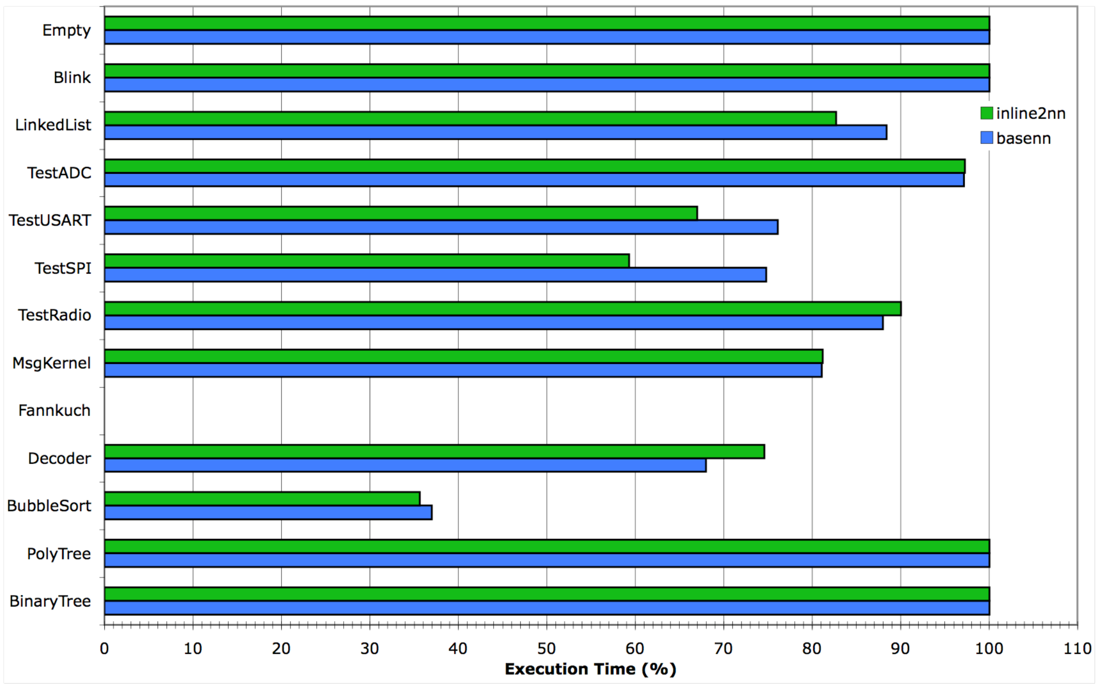

Figure 3.12: Normalized Execution Time w/o Safety Checks
This figure shows the resulting execution time when safety checks are disabled. Each result is normalized to its respective configuration with safety checks enabled.

Dynamic safety checks also impose some runtime overhead on the program as well. Figure 3.12 shows the impact of disabling safety checks on the runtime of the programs. As in Figure 3.11, each bar is normalized to its respective configuration with safety checks. Here we can see that the same four applications are not affected, but some applications, particularly the array-intensive `BubbleSort` and `Decoder` applications, are severely affected. This is in line with literature on array bounds optimizations [23], which typically reports speedups as high as 2-3x for some programs. As can be seen from the results, it will be important to optimize these safety checks in the future. The

`Fannkuch` results are an anomaly; although the program does not actually trigger any safety check violations during execution, a compiler bug in `avr-gcc` triggers an incorrect optimization that causes the program to fail to terminate when the checks are not present in the code.

3.6.2. Virgil versus C

For programming microcontrollers, C is still the dominant language due to its low-level nature with small fixed and proportional costs. With `avr-gcc`, there is little to no built-in runtime system required, allowing very small C programs to be created for the AVR. These properties make it an attractive choice for most microcontroller programmers. They also make it an attractive choice for the Virgil backend target,

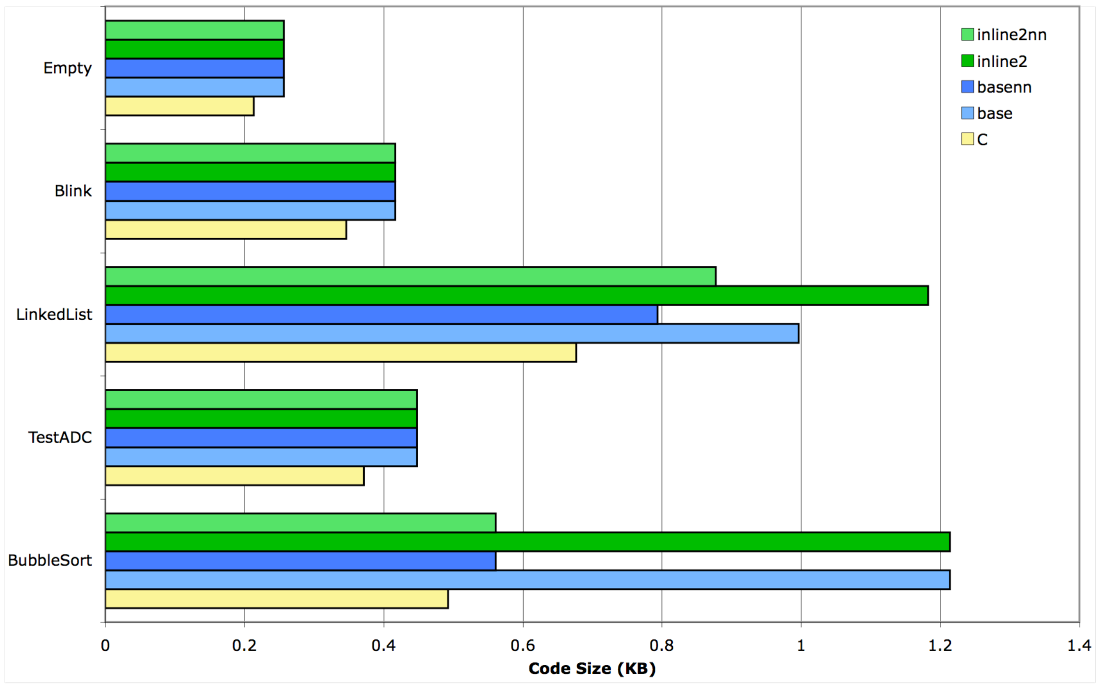

Figure 3.13: Code Size (C vs. Virgil)
This figure compares code sizes for Virgil and C programs over several different Virgil compiler configurations, with and without safety checks.

allowing the Virgil compiler to concentrate on high-level data and code optimizations while leaving low-level optimizations such as instruction selection and register allocation to the C compiler. Nevertheless, we want to measure the overhead introduced in writing code in Virgil, with its safety and expressiveness benefits, as opposed to writing the code directly in C.

For this experiment, we rewrote five of the example programs directly in C, eliding the object-oriented features and the driver library, cutting the programs down to bare essentials. Then, to make the comparison as fair as possible, these C programs were translated back into Virgil with a one-to-one function-to-method and struct-to-class mapping. This ensures, as close as can be, that the comparison focuses the language overheads rather than the design decisions made by libraries or driver code. Figure 3.13 gives the code size comparison between the five C programs and their Virgil equivalents, with four Virgil compiler configurations. First, we can see that the Virgil compiler configurations with safety checks enabled (`base`, `inline2`) can be significantly larger in the `BubbleSort` and `LinkedList` cases, while the other three programs have are not affected by safety checks. Second, when safety checks are disabled (`basenn`, `inline2nn`), the Virgil code size is only slightly larger than the corresponding C program.

Figure 3.14 compares the execution time for the four Virgil compiler configurations, normalized against the execution time for the respective C program. Here, we can see that only `BubbleSort` is affected by the safety checks, because the other programs either require no safety checks, or they are redundant when combined with the program's logic. Three programs have 5% or less execution overhead with respect to the C program, with 29% for `LinkedList` without inlining. The `avr-gcc` compiler does not perform aggressive inlining, which means the Virgil compiler actually has an opportunity to outperform the C compiler more aggressive optimizations. It has been long known that inlining is key for good performance in object-oriented and functional programming languages. We saw earlier in Figure 3.7 that inlining with the Virgil

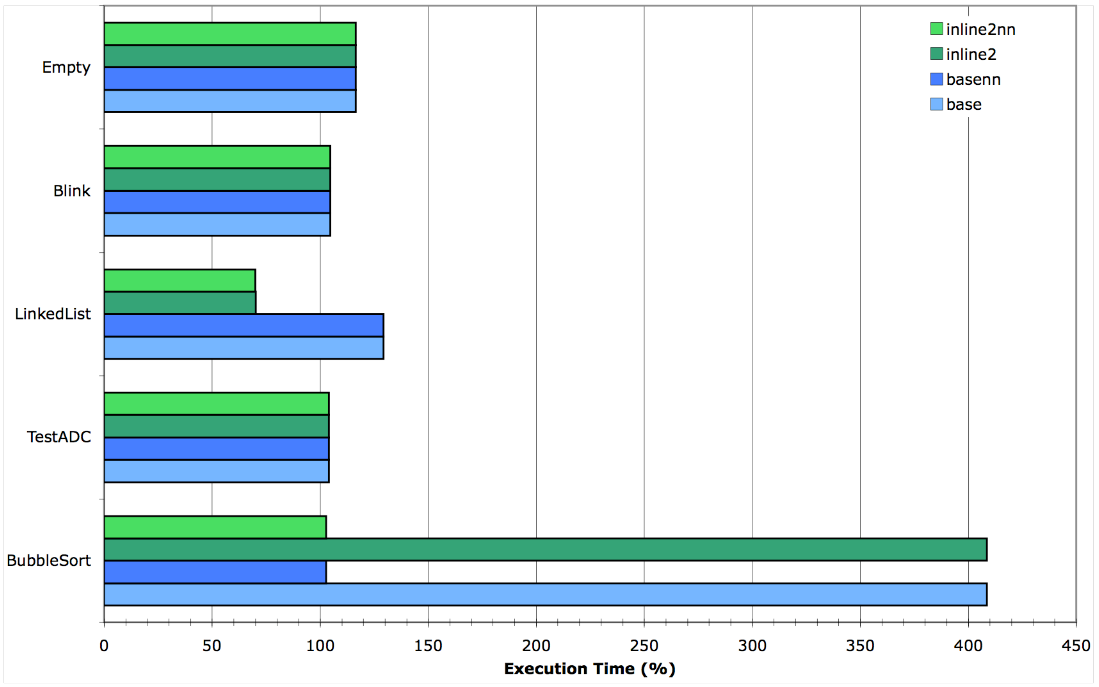

Figure 3.14: Execution Time (C vs. Virgil)
This figure compares execution time for Virgil and C programs over several different Virgil compiler configurations, with and without safety checks. The execution times are normalized to the execution time for the C version.

compiler significantly improves performance; in this case, the `LinkedList` program actually runs over 30% faster than the C program. At a higher level, if we view performance as an item that can be budgeted, then inlining and aggressive optimization buy us the necessary performance to support the higher-level features of Virgil.

The data sizes for these five programs are also very close. `Empty`, `Blink`, and `TestADC` require no heap whatsoever, in either the C or Virgil versions. The `BubbleSort` program is dominated by a large array of integers. It requires 800 bytes of memory in the C implementation and 805 bytes in the Virgil implementation. `LinkedList` requires 124 bytes in the C implementation and 125 bytes in the Virgil implementation. Both the C and Virgil `LinkedList` programs have a static array that is a pool of link nodes that is used to build the main list at startup; the main list is used throughout the rest of the benchmark. In the C implementation, the pool is an array of `Link` structures rather than array of pointers to `Link` structures, while in the Virgil version, it is an array of references to `Link` objects, which increases the heap size by one pointer per pool entry. During optimization, the Virgil compiler detects that the `Link` objects are orphans and removes their object headers. Then the Virgil compiler's RMA optimization detects that the backward pointers are unused and removes them. This results in `Link` objects that are actually smaller than the corresponding C `Link` structs, balancing out the cost of the extra pointers in the pool. Interestingly, if we apply reference compression techniques from the next chapter, the heap size of the `LinkedList` program can be reduced to 97 bytes, which is actually 22% smaller than the C version.

Overall, these results clearly show that Virgil is a very close competitor to raw C. We can see that the safety checks impose significant performance and code size costs in some cases, but this can likely be removed in the future with well-known array bounds and null check optimizations. Heap size results are very promising. Orphan classes provide the programmer with low-cost data structures that do not require metadata, and RMA removes unused data structures and fields. Compression techniques described in the next chapter serve to further reduce the heap size.

4. COMPRESSION

This chapter describes the compression optimizations that the Virgil compiler employs in order to reduce the RAM consumption of applications. These optimizations exploit the type safe nature of Virgil code in order to represent program quantities in a more space efficient way. These compression techniques establish part of the thesis by showing that *advanced compiler technology can reduce the resource consumption of programs written in Virgil for microcontroller devices*.

Often the most important resource constraint of a microcontroller is the RAM space available to store the program heap and runtime stack, while makes reducing RAM consumption of paramount importance. Reducing RAM consumption allows larger applications to be built and deployed on the same microcontroller model, while optimizing a single application also allows a smaller, cheaper microcontroller to accomplish the same task. There are several general techniques to reduce RAM consumption. One is to simply remove unused data structures through compiler or manual analysis, such as the reachable members analysis described in the previous chapter. Another technique is to reduce the average footprint of a program by moving infrequently used data to larger, slower storage such as disk, e.g. with virtual memory mechanisms. A third technique is to compress infrequently used data and dynamically decompress it as it is accessed. A fourth technique is to statically compress program quantities so that dynamic decompression is unnecessary.

This chapter evaluates two offline heap compression techniques implemented in the Virgil compiler. Both techniques exploit the type-safety of Virgil and the availability of the entire program heap at compile time to encode references in a more compact way. Unlike previous approaches [31][65], this technique is based on the type safety of the language and does not require sophisticated program analysis. The first technique represents object references as object handles instead of direct pointers, allowing them to be represented with fewer bits. Because the entire heap is available at compile time, the compiler can introduce a compression table stored in ROM that contains the actual memory address of each object. This adds a level of indirection, as object operations require first loading the actual memory address from the table using the object handle. The second technique is a novel object layout model that we call *vertical object layout*. Vertical object layout represents objects in a more compact way by viewing the heap as a collection of field arrays that are indexed by object number, rather than the traditional approach of a collection of objects that are accessed via pointers. This object layout technique represents object references with integer identifiers that can be used as indices into each individual field array, requiring no extra indirection. A special numbering system for identifiers ensures that each field array can be represented compactly without wasting space, even in the presence of subclassing.

Our experimental results show that vertical object layout has better execution time and code size than the table-based compression scheme on nearly all benchmarks, while achieving similar RAM size savings. Relative to the standard object layout strategy, the code size increase from vertical layout is less than 10% for most programs, and less than 15% for all programs, while the execution time overhead is less than 10% for 7 of 12 programs and less than 20% for 9. Interestingly, compressed vertical layout actually improves execution time over the standard object model for two programs that use dynamic casts intensively, because type casts are implemented more efficiently.

4.1. Pointer Waste

On microcontroller architectures with between 256 bytes and 64 kilobytes of RAM, pointers into the memory are typically represented with a 16-bit integer byte address. In a weakly typed language like C, a pointer is not constrained to point to values of any particular type and can conceivably hold any value. In fact, pointer arithmetic relies on the fact that pointers are represented as integers and allows operations such as increment, addition, subtraction, and conversion between types. Worse, C allows pointers to be converted to integers, manipulated, and converted back to pointers.

However, in a strongly typed language such as Virgil, each reference has an associated static type, and the type checker enforces that every reference may only refer to heap entities of the correct type. For example, object references of declared type `A` must only refer to objects of type `A` or one of its subtypes. In Virgil, the representation of object references is entirely opaque to the program; references cannot be converted to or from primitive types, and their machine width is not exposed. Recall that after initialization time, a Virgil program has already allocated all the objects that will ever exist in the heap and no further objects can be allocated at runtime. The compiler can exploit this combination of type safety and static allocation to encode references in a more compact way, rather than simply using pointers to an object's address in memory.

In order to reduce the total memory space consumed by the heap, we would like use as little space to store each reference field as possible. We will refer to the compact representation of a reference stored in a field as the *compressed reference*, and refer to the actual address of the object in memory simply as the address. Reference fields may be written during the execution of the program; thus a sound compression scheme must approximate the set of objects that could be referenced by each field over any execution of the program. We will refer this approximation as the *referencible set*. The compression scheme must therefore ensure that each compressed reference can represent all objects in its referencible set. A simple and intuitive approximation is to use the declared type of the field as an approximation of the referencible set.

Consider a program that has allocated some number `K` of objects of type `A` during its initialization phase. Type safety ensures that every reference of declared type `A` may only refer to one of these `K` objects (or possibly `null`), over any execution of the program. We can therefore use the static type of a reference as a simple and conservative approximation of the possible set of objects to which it may refer. This means that only `log(K+1)` bits are required to distinguish between all of the possible referent objects. Because the approximation is conservative and the representation is opaque to the program, the compiler will never need to dynamically compress and decompress the reference representation. This is in contrast to [31], which attempts to compress C data values whose representation is not opaque to the program and therefore sometimes requires both dynamic decompression and dynamic compression.

4.2. Heap Layout

The dedication to complete type safety and opaqueness of references and object layout issues gives the Virgil compiler complete control over the arrangement of the heap in memory. For example, it may elect to place objects of the same type next to each other in memory, reorder objects and fields for cache locality, etc. The compression techniques presented here do not assume any particular assignment of addresses to objects or arrangement of fields within objects. Locality is not an issue here because microcontrollers typical lack any memory cache.

It is important to note that other compression schemes are possible if the compiler chooses a heap layout with particular properties. For example, if the heap layout algorithm places all objects of a particular referencible set into the same region of memory starting at a known location, the offset of an object's address from the starting location of the region could be used as the compressed reference representation. In this scheme, direct addresses could be used throughout the program, with field reads being decompressed by adding the starting address of the first object and field writes subtracting the starting address before storing the field. While an offset may require more bits to store than an index into a compression table, the indexed address scheme does not require any compression tables in ROM.

4.3. Table-based compression

The most straightforward way to implement reference compression is to use a compression table where each compressed reference is an object handle: an integer index into a table that contains the actual addresses of each object. Because Virgil has disjoint inheritance hierarchies, the compiler can compress each reference by creating a compression table for its associated root class, with one entry in the table for each object whose type is a subclass of that root. The number of bits needed to represent the integer index is therefore the logarithm of the table size. For example, if the table has 15 live objects plus `null`, we could use a 4-bit integer index, a savings of 75% over using a 16-bit address. Because there is no garbage collector which may move objects at runtime, object addresses do not change during runtime, which allows the compiler to store the table in ROM or flash, which is considerably larger than RAM, though usually slightly slower to access. Figure 4.1 gives an illustration of the table-based compression scheme.

The table adds a level of indirection to all object operations. Reads and writes on object fields require first reading the object address from the compression table and then performing the operation as before. For frequently access objects, especially within loops, the compiler may be able to avoid the cost of the indirection by using standard code motion optimizations to cache the actual address. When compressing fields in the heap, accesses may be slower if the fields are bit-packed in memory and require masks and shifts, but can be faster if the field requires only one byte of storage instead of two. Thus table-based compression represents a classic space/time tradeoff: it consumes some ROM space for the tables and may reduce performance, but saves RAM.

It is important to note that table-based compression can sometimes save

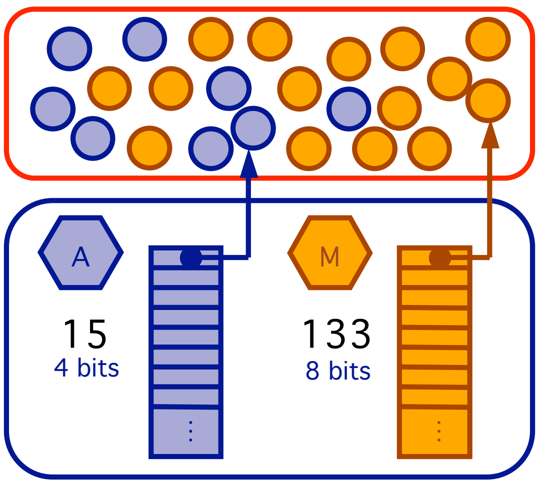

Figure 4.1: Table-based Compression
Each hexagon represents a complete class hierarchy labeled with its root class. The reference table is stored in ROM (bottom box) and stores the addresses of the actual objects in RAM (top box). The representation size for a reference is the logarithm of the table size.

RAM space even if the compression tables themselves are also stored in RAM. This is because for a table of size `K` and a pointer size of `P` bits, the cost of the table is `K*P` bits while the savings is `N*(P - log(K+1))` bits, where `N` is the number of references compressed. `N` is always larger than `K` because every object must have at least one reference to it to be considered live. If `N` is large enough, `N*(P — log(K+1))` is larger than `K*P`. We don't expect this case to be common; our implementation always stores compression tables in ROM for maximum RAM savings.

4.4. Vertical Object Layout

In traditional high-performance object-oriented systems, each object is represented in memory as a contiguous region of words that contain the values for each of the object's fields. An object reference is represented as a single-word pointer to this contiguous memory region, and the different fields of a single object are located at fixed offsets from this base address. Advanced features such as mix-ins, multiple inheritance, etc may be implemented by indirection to further contiguous memory blocks. This layout strategy has the best performance in a scenario where objects are created, moved, or reclaimed dynamically. An object allocation operation amounts to little more than an acquisition of a small contiguous region of memory, often simply bumping a top-of-heap pointer by a fixed amount. Field accesses in this model are implemented straightforwardly as a read or write of a memory address that is a small fixed offset from the object pointer; nearly all architectures allow this operation to be implemented with a single instruction. We will refer to this implementation strategy as the standard or *horizontal* layout, for reasons that will become obvious in this section.

In Virgil, the compiler has maximum freedom to layout objects in any way that respects the program's semantics. Our basic insight is that Virgil's initialization time model gives rise to a scenario where objects are not created, moved, or reclaimed dynamically; this means that objects need not be laid out as contiguous regions of memory words in order to simplify these operations.

Imagine the heap of the program after initialization has completed. The program has allocated some number of objects of various types, and each object has values for all of its declared and inherited fields. If we view the objects as a matrix, each object corresponds to a row in the matrix, and each declared field in the program corresponds to a column, with each entry in the matrix storing the value for the field for the corresponding object. In the standard layout, an object reference is represented by a pointer to a row of the matrix, where the elements of a single row are adjacent in memory. In a sense, the standard layout arranges the matrix in memory *horizontally*. But one can also explore the implications of arranging this matrix in memory *vertically*, where an entire column has its elements adjacent in memory.

Consider the example in Figure 4.2. The classes `A`, `B` and `C` have declared fields `f`, `g`, and `h`, respectively. Suppose now that we collect all the objects in the initialized heap of these types and number them so that all the objects of exact type `A` are first, `B` second, and `C` third. Then if we put these objects into a table such that the columns are the fields `f`, `g`, and `h`, we can see that each column has a contiguous range of indices for which the field is valid corresponding to the indices of the class in which the field was declared. If we represent an object reference as an index from 0 to 9 (with -1 representing

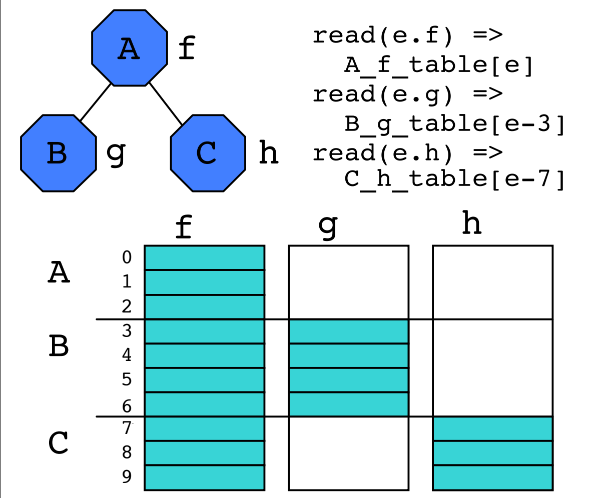

Figure 4.2: Vertical Layout
For classes `A`, `B`, and `C`, and declared fields `f`, `g`, and `h`, we can represent each of the fields by a table indexed by preorder object number. Preorder numbering ensures objects of exact type `A` have ids 0-2, `B` have 3-6, and `C` have 7-9. Only the occupied portion of each table is stored and the object id is adjusted upon access.

a `null` reference), and represent the field `f` as an array `A_f_array`, we can read and write the field by simply indexing into `A_f_array` by the object number.

An access of the field `g` in the program requires the receiver object to be of type `B`; therefore we know statically that accesses of field `g` must use indices in the valid range for `B` objects. While we could represent the field `g` as an array over the entire index range 0 to 9, we can avoid wasting space by instead rebasing the array so that element 0 of the array corresponds to index 3, the first valid index for `B`. Then, an access of the field `g` for a type `B` would simply adjust by subtracting 3 from the object index before accessing the array. While these seems slower, it is equivalent to a base 0 array if the compiler constant-folds the known fixed address of the array and the subtraction adjustment; the compiler will just use a known fixed address corresponding to where the array would have started in memory if it had been based at 0.

It is simple to generalize from the example. For any inheritance tree, we simply assign object identifiers using a pre-order tree traversal. Figure 4.3 gives the algorithm. The output of the algorithm is an interval of valid indices for each class and an object id for every object. By employing preorder traversal of the inheritance tree, the final assignment guarantees that each class has a contiguous range of indices corresponding to all objects of that type or one of its subtypes. Therefore the array that represents that field in the vertical object layout can be compact, avoiding wasted space. This algorithm chooses to restart the object id at zero for each root class in the hierarchy, which means that an object id is unique within its inheritance hierarchy, but not necessarily globally unique.

We can use the same technique to represent meta-objects vertically as well. In Virgil, meta-objects store only a type identifier that is used for dynamically checking down casts and a dispatch table that is used for virtual dispatch. We can use the same algorithm to number the meta-objects according to the inheritance hierarchy and then represent each method slot in the dispatch table vertically. A virtual dispatch then amounts to two vertical field accesses (as opposed to two

```
void assignAll(Program p) {
    for ( ClassInfo cl : p.getRootClasses() )
        assignIndices(0, cl);
}
int assignIndices(int min, ClassInfo cl) {
    int max = min;
    // assign the indices for objects of this type
    for (ObjectInfo o : cl.instances) o.index = max++;
    // recursively assign id's for all the children
    for (ClassInfo child : cl.getChildren)
        max = assignIndices(max, child);

    // remember the interval for this class
    cl.indices = new Interval(min, max);
    return max;
}
```

Figure 4.3: Object and Class Numbering
Algorithm to compute object indices by pre-order traversal of inheritance tree. For each class, `ClassInfo` stores a list of the child classes and an interval representing the valid indices for objects of this class and subclasses. For each object, `ObjectInfo` stores the object id (index) assigned to the object.

horizontal field accesses in the traditional approach). The first vertical field access retrieves the meta-object id by indexing into the meta-object id array using the object index. The retrieved meta-object id is then used to index into the virtual method array for the specified method to retrieve a direct pointer to the code of the appropriate method.

This numbering technique also has another advantage in that the contiguousness of the object identifiers makes dynamic type tests extremely cheap, because the object identifier actually encodes all the type information needed for the cast. The algorithm assigns object identifiers so that every class has an interval of valid indices that correspond to all objects of that type. Thus, given a reference `R` that is represented by an object index and a cast to a class `C`, we can simply check that the index `R` is within the interval for the class `C`. This requires only two comparisons against two constants; no indirections and no memory loads are required.

Reference compression becomes trivial with vertical object layout. Because each object reference is now represented as an index that is bounded by the number of objects in its inheritance hierarchy, like table-based compression, it can be compressed to a smaller bit quantity. Thus, wherever the reference is stored in the heap (e.g. in the fields of other objects), it consumes less space. However, the field arrays may not be completed packed at the bit level. If the field is compressed to fewer than 8 bits, the indexing operation is more efficient if the field array is a byte array rather than packed at the bit-level because memory is usually not bit-addressable. Our implementation does not compress references in the vertical layout to be smaller than a byte.

Vertical layout also potentially saves memory by eliminating the need to pad fields in order to align their addresses on word boundaries, which is sometimes needed in the horizontal layout. Padding is unnecessary in vertical object layout as long as each field array is aligned at the appropriate boundary for its type, ensuring that each element in the array is aligned by virtue of being of uniform size. (However, padding and memory alignment is not generally an issue on 8-bit microcontrollers.)

4.5. Experimental Results

In this section we evaluate the impact that reference compression and the vertical object model have on three program factors: code size, heap size, and execution time. We use the same benchmark programs from Chapter 3, omitting the `Empty` program, which has no heap. As before, these applications target the popular Mica2 sensor node, and we use `avr-gcc` version 4.0.3 to compile the C code emitted by the Virgil compiler to AVR machine code. Precise performance numbers are obtained by using the program instrumentation capabilities [101] of the Avrora cycle-accurate AVR emulator.

We tested five configurations including the standard horizontal object layout; the four new configurations are normalized against the results of the standard layout to show relative increase and decrease in code size, data size, and execution time. The three main configurations are: `hlrc`, which is the standard horizontal layout with table-based compression; `vl`, which is the vertical object layout without compression; and `vlrc`, which is the vertical layout with compression applied to object indices. The last configuration, `hlrcram`, is only shown for code size and execution time comparison; it corresponds to horizontal layout with reference compression, but instead of storing the compression tables in ROM, they are stored in RAM in order to compare the cost of accessing ROM versus accessing RAM.

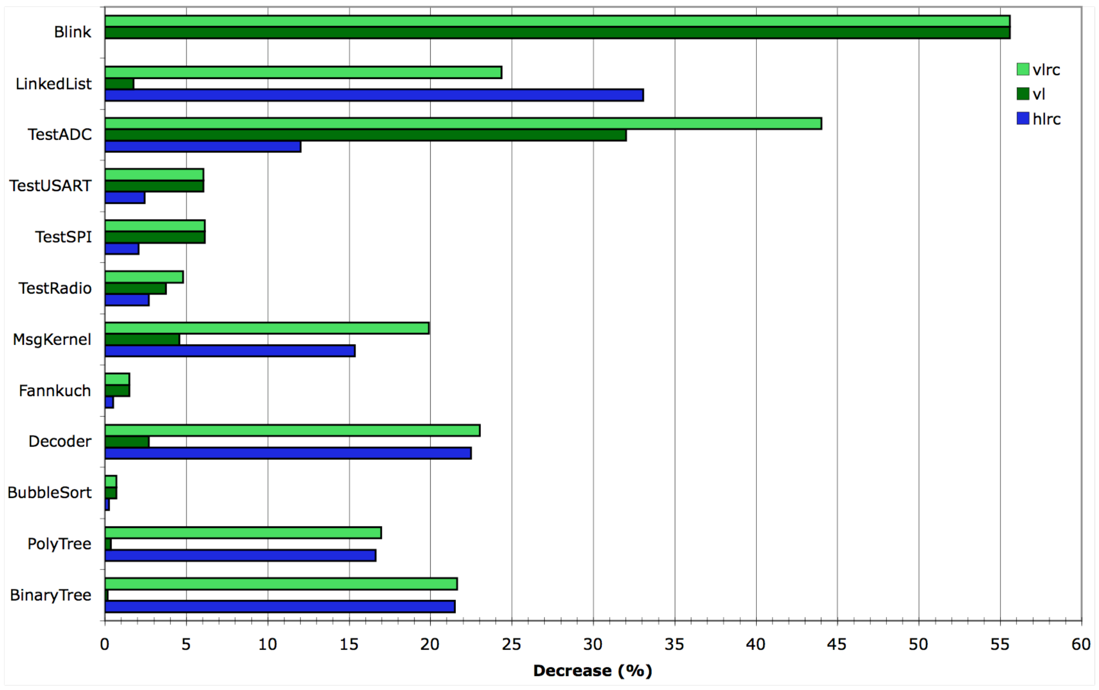

Figure 4.4: Heap Size Decrease
This figure compares relative heap decrease for three different object models, with each normalized to the standard horizontal object layout. Higher is better.

Figure 4.4 shows a comparison of the relative data sizes for our benchmark programs for the three main configurations, normalized against the base configuration of horizontal layout with no reference compression. First, we notice that vertical layout (`vl`) often saves some memory over the base configuration. This is because it does not require type identifiers in the meta objects because the object numbers have been assigned so that they encode the type information. Also, the horizontal layout sometimes produces zero-length objects; `avr-gcc` allocates a single byte of memory to such objects. The second observation is that the compressed vertical layout typically saves a similar amount of memory to the compressed horizontal layout, although some of this is due to the empty object anomaly and the lack of type identifiers in meta-objects. As expected, compressed vertical layout (`vlrc`) is uniformly better than vertical layout (`vl`) alone.

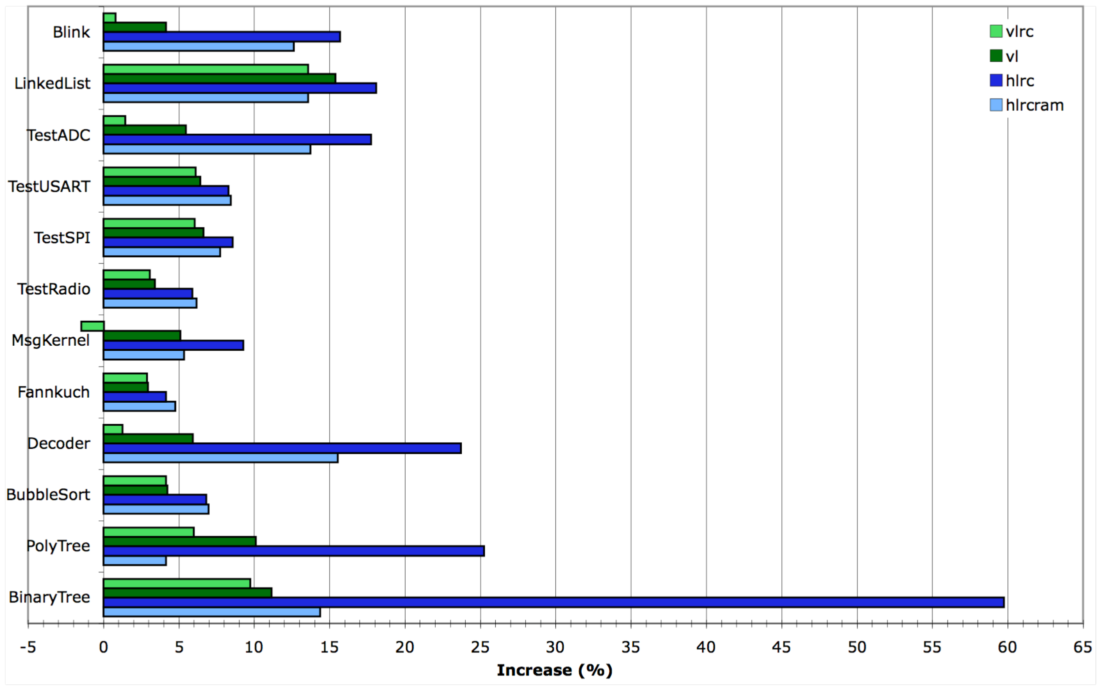

Figure 4.5: Code Size Increase
This figure compares relative code size increase for three different object models, with each normalized to the standard horizontal object layout. Lower is better.

Figure 4.5 shows the relative increase in code size for the same benchmarks with an added configuration, `hlrcram`. As in Figure 4.4, all configurations are normalized against the base configuration of horizontal layout without reference compression. Here, we can see that all configurations increase the code size of all programs (with the sole exception of `vlrc` on `MsgKernel`), with both `vl` and `vlrc` performing better than `hlrc` in each case. The increase for `vlrc` is less than 10% for most programs and less than 15% for all programs. Here, adding compression to the vertical layout actually reduces code size. This is because all field arrays become smaller, down to a single byte (because the Virgil compiler does not pack field arrays at the bit level), therefore the code to access them becomes smaller.

Horizontal reference compression increases the code size in two ways. First, it introduces compression tables that are stored in the read-only code space. Second, it requires extra instructions for each object operation due to the extra indirection. When the compression tables are stored in ROM, the Virgil compiler must emit short inline AVR assembly sequences because `avr-gcc` does not support directly accessing the ROM at the source level. These assembly instructions are essentially unoptimizable by `avr-gcc`. To better isolate this effect, this figure includes code size results for a new configuration, `hlrcram` (or horizontal layout with reference compression tables in RAM). This configuration of course does not save RAM overall, but allows us to explore the effect of the special ROM assembly sequences on the code size in comparison to accessing the RAM. Comparing the `hlrc` configuration against the `hlrcram` shows that most of the code size increase is due to these special inlined ROM access sequences. The difference could be reduced if either `avr-gcc` understood and optimized accesses to ROM, or if the AVR architecture offered better addressing modes to access the ROM with fewer instructions. It is important to note that the largest proportional code size increases are for the smallest programs, as can be seen in Figure 3.9 in the previous chapter (the base code size here is equivalent to the `RMA` configuration in that figure).

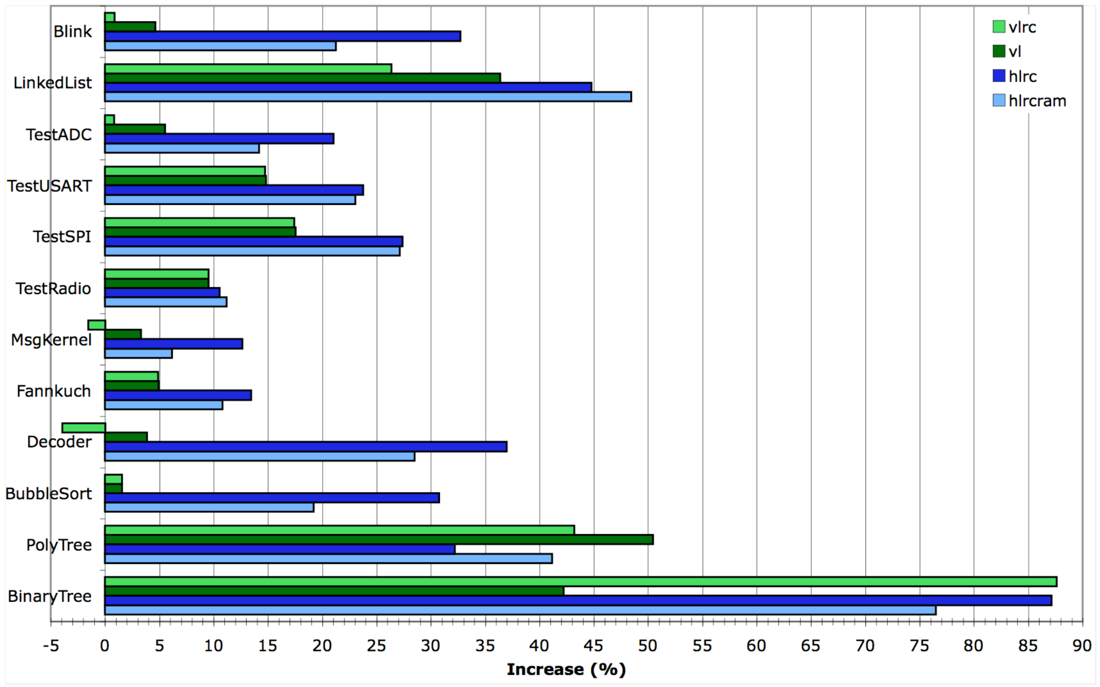

Figure 4.6: Execution Time Increase
This figure compares relative execution time increase for three different object models, with each normalized to the standard horizontal object layout. Lower is better.

Figure 4.6 gives the relative increase in execution time obtained by executing each benchmark in the Avrora [103] instruction-level simulator. The vertical layout technique performs better than horizontal compression in all but one case, and the execution time overhead for the compressed vertical layout is less than 20% in 9 of the 12 benchmarks, less than 10% in 7, and actually performs better by than the baseline by a small amount in two cases. This is because these two programs perform a significant number of dynamic casts, which are cheaper in the vertical layout. This figure also includes results for the `hlrcram` configuration from Figure 4.5. We wanted to isolate how much of the execution time overhead is due to the cost of a ROM access versus a RAM access. In most cases, the execution time of `hlrcram` is noticeably better than that of `hlrc`, which means that a significant fraction of the overhead is due to this ROM access cost. Also notice that that the largest proportional execution time increases tend to be for the smaller, pointer-intensive programs like `BinaryTree`, `PolyTree`, `LinkedList`, and `Decoder`.

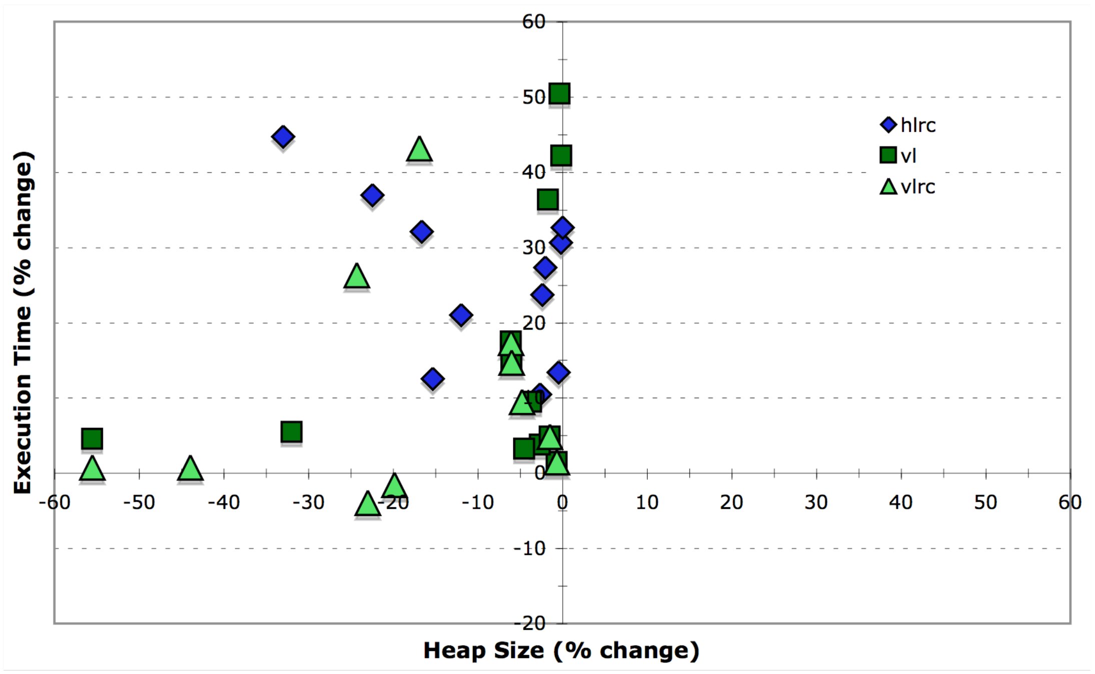

Figure 4.7: Heap Size vs. Execution Time
This figure compares relative heap decrease and relative execution time increase for three different object models, with each normalized to the standard horizontal object layout.

Figure 4.7 combines the data from figures 4.4 and 4.6, showing the tradeoff between increase in execution time and the savings in heap size for the three main configurations. First, we can see that the vertical layout without reference compression (`vl`) usually increases execution time without saving any heap space, while adding reference compression to vertical layout (`vlrc`) increases heap savings and usually has better execution time than vertical layout alone. Also, `hlrc` compression tends to have a larger increase in execution time with some savings in heap size, but not as much as `vlrc`. Overall, there is significant variation across the benchmarks, suggesting that the two factors are not intrinsically correlated. Instead, it is more likely that the factors are correlated to benchmark characteristics, therefore the compiler should take these characteristics into account and avoid reference compression when it will save little heap space.

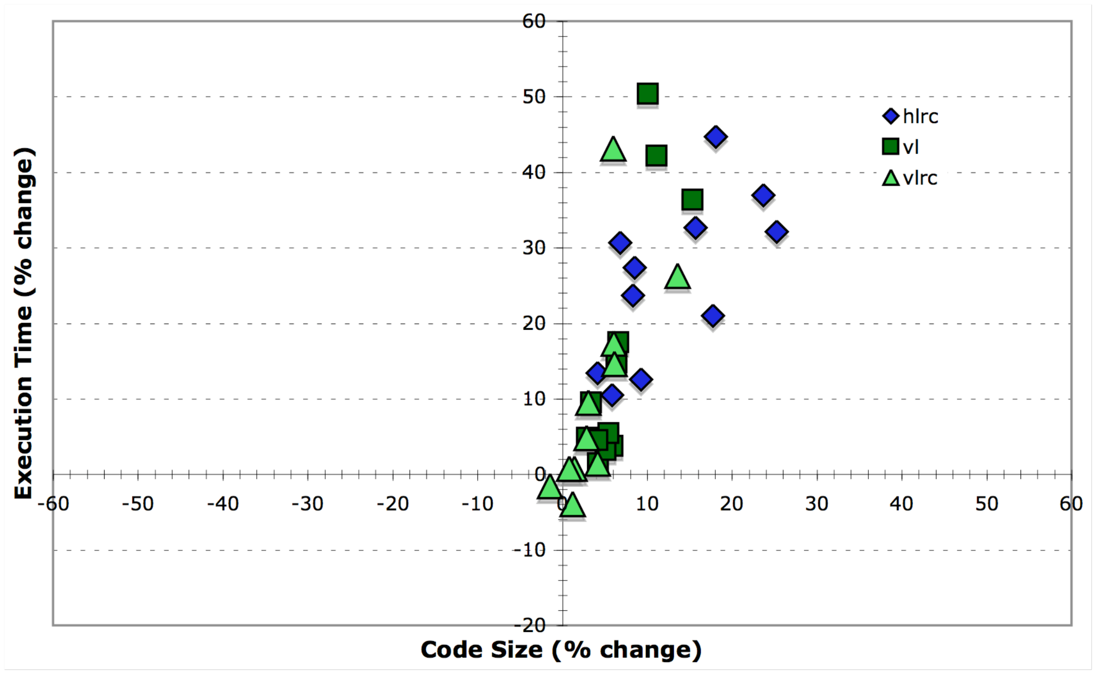

Figure 4.8: Code Size vs. Execution Time
This figure compares relative code size increase and relative execution time increase for three different object models, with each normalized to the standard horizontal object layout.

Figure 4.8 combines data from figures 4.5 and 4.6 to show the correlation between increase in code size and increase in execution time for the three main configurations. First, we can see that the two factors appear closely correlated because the points cluster near a line from the origin into the upper right quadrant. This is most likely due to the simplicity of the AVR instruction set architecture and lack of an instruction cache; adding more instructions has a predictable effect on the execution time. Second, we can see that `vlrc` performs significantly better than the other configurations, with most of its points clustered near the origin. Third, we can see that `hlrc` performs the worst, with the largest increases in code size and execution time.

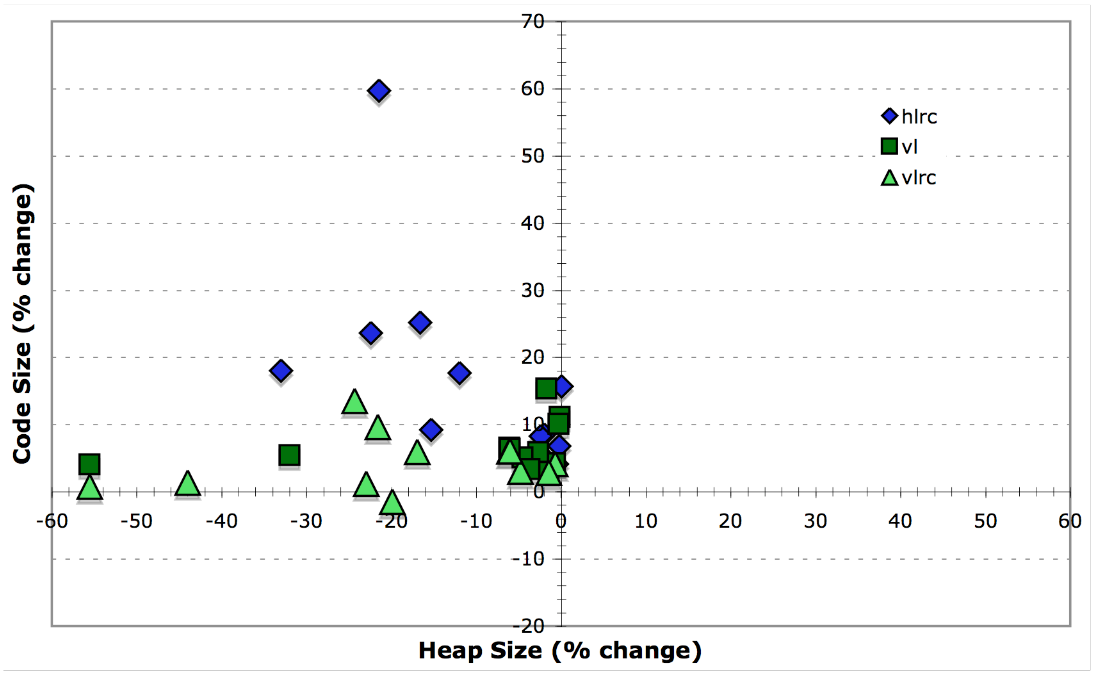

Figure 4.9: Heap Size vs. Code Size
This figure compares relative heap size decrease and relative code size increase for three different object models, with each normalized to the standard horizontal object layout.

Figure 4.9 combines the data from figures 4.4 and 4.5, comparing relative increase in code size versus decrease in heap size. Here we can see for a given heap size reduction (horizontal axis), `vlrc` tends to produce smaller code than `hlrc` because of the lack of ROM compression tables and simpler field access sequences.

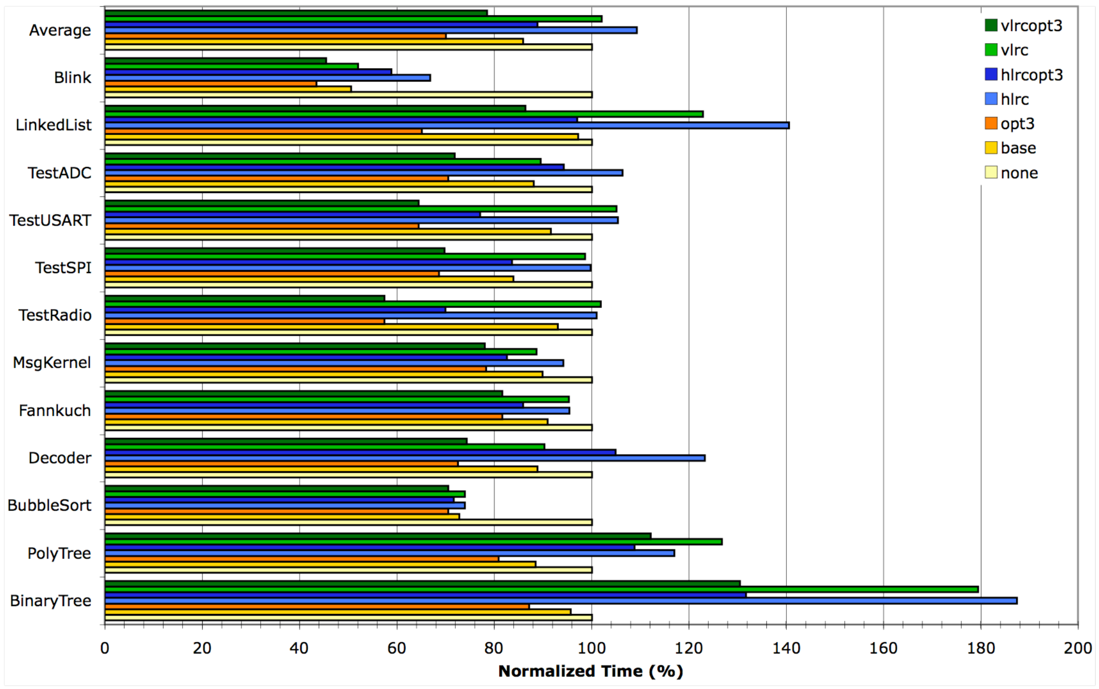

Figure 4.10: Execution Time with Compression and Inlining

This figure compares the execution time of the reference compression strategies when more aggressive inlining optimizations are also applied. The configurations here correspond closely to the configurations from results in Figure 3.8.

Figure 4.10 compares execution time of the different reference compression strategies when more aggressive code optimizations (inlining and constant propagation) are also applied, providing a more complete picture of the overall performance. All results are normalized to the execution time of the `none` (where RMA is not applied) configuration, which corresponds to the `none` configuration from Chapter 3. The `base` configuration corresponds to RMA; `opt3` corresponds to inlining at a slightly more aggressive level than `inline2` in Chapter 3; `hlrc` and `vlrc` are as before; `hlrcopt3` corresponds to horizontal layout with reference compression and optimization level 3, and `vlrcopt3` corresponds to vertical layout with reference compression and optimization level 3. Here we can see that applying aggressive optimization significantly reduces execution time for all configurations, though it increases code size as seen in Figure 4.11. In fact, applying aggressive optimization allows the compressed configurations `hlrcopt3` and `vlrcopt3` to run faster than the `none` configuration (no RMA) in most cases, and faster than the `base` configuration (RMA only) on average. One way to view these results is that aggressive code optimizations provide a performance "budget" that allows reference compression to be applied without an overall loss of performance relative to the baseline. Because aggressive optimization increases code size, we can view the overall tradeoff as trading code size for data size, without a loss of performance.

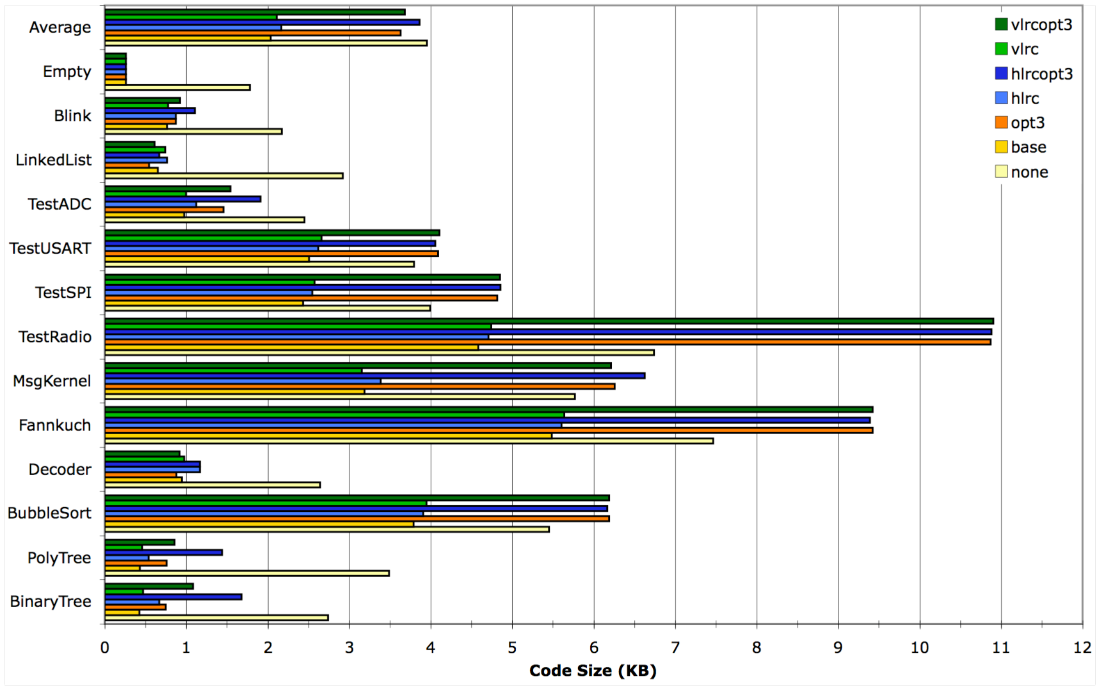

Figure 4.11: Code Size with Compression and Inlining
This figure compares the code size of the reference compression strategies when more aggressive inlining optimizations are also applied. The configurations here correspond closely to the configurations from results in Figure 3.8.

Figure 4.11 compares absolute code size for combined reference compression and inlining, using the same configurations from Figure 4.10. Here, we can see that in terms of absolute size, the effect of reference compression alone is small, while the effect of aggressive inlining is large. Of course, applying RMA (the `base` configuration), reduces code size substantially, which in the case of the smaller applications means that the highly optimized configurations are actually smaller than the `none` configuration. We can see that on average, RMA provides enough dead code removal to negate the increase in code size due to inlining. Similar to the performance "budget" provided by inlining, we can view RMA as providing a "budget" of code space that allows the inliner to improve performance. Inlining heuristics deserve further study here. Because these results show that for larger applications, the effect of inlining far outweighs the code size increase from reference compression, it is more important to develop and tune heuristics so that only the most frequently executed calls are inlined, rather than focus on reducing code size increase from reference compression.

5. CONCLUSION

This dissertation tackles the problems of developing systems software for very small devices with language and compiler technology. I have established my thesis statement that *advanced language and compiler technology can bring the benefits of object-oriented programming to even the most constrained of embedded systems.* The two systems I have built and described here offer compelling evidence.

The ExoVM explores a new approach in developing virtual machines for embedded systems, employing *pre-initialization*, *closure*, and *persistence* to an existing, state-of-the-art virtual machine to reduce memory footprint on a per-program basis. The feature analysis technique has exposed a new dimension of modularity in language implementations: the cost of a language feature in terms of the reachable virtual machine entities that it requires. The ExoVM is one important contribution towards a grand challenge in virtual machine construction: a language runtime and compilation model that seamlessly adapts across static and dynamic views of execution and scales from extremely small systems up to very large systems. The experimental results show that pre-initialization coupled with feature analysis can vastly reduce the footprint of the JVM's internal data structures and the VM code size by removing unnecessary entities on a per-program basis.

The ExoVM also has wider applicability because it can provide the basis for relating language features to their efficiency considerations more directly. We illustrated how the use of constraints in feature analysis has shed light on the interconnectedness of the virtual machine and the class library implementation. We believe that this is just a first step to exposing the efficiency implications of feature use to application developers to whom footprint matters, such as embedded system programmers.

Virgil is a major step forward in language technology for microcontrollers. Careful attention to detail and adherence to design constraints brings most of the expressiveness of object-oriented languages to this most severely resource-constrained class of devices, without sacrificing type safety and without requiring any language runtime system, imposing only minor metadata overheads. Virgil is the first language to recognize that explicitly separating initialization time from run-time at the language level leads to a convenient programming model for embedded systems by allowing objects to be freely allocated at compile time and then stored for use at run time. The commitment to language safety eliminates a large class of pernicious software bugs through strong static type safety and some dynamic checks, like Java. In fact, Virgil's type safety is key to efficient implementation, enabling a new class of optimizations that exploit the static availability of the program heap.

The Virgil compiler exploits the design choices and language model to introduce an unprecedented level of data optimizations. It represents a significant shift in how we should view the compiler's role in optimizing for heap space, relieving much of the burden of optimization from application programmers. The Virgil compiler introduces heap-sensitive optimizations that serve to significantly reduce the size of programs by removing unused members and object headers, representing reference fields in a more compact manner, and making whole program object layout decisions—all without changing the programming language semantics or model.

Reference compression exploits the type-safe nature of object references to achieve significant heap compression without requiring expensive program analysis; references can therefore be stored far more efficiently than the standard implementation practices of pointer-based languages like C, which cannot compress pointers by type. Vertical object layout demonstrates the possibilities of allowing the compiler complete control over the data representation of all objects in the entire heap, leading to compression techniques that radically alter the memory layout while reducing its size and preserving most of the runtime performance—without any manual effort and without changing the programming model or semantics. This surprising result leaves us to ponder the suggestion that objects, under the control of a highly optimizing compiler, may in fact be better than pointers for embedded systems, since the strong types of references and objects gives the compiler much richer information for making good layout decisions.

5.1. Limitations

Any software system of significant complexity that exists in a large design space inevitably has tradeoffs, limitations, and pitfalls. This section discusses some of the limitations of Virgil and the ExoVM. First, as a new language and compiler system, Virgil is very young and has not been fully stressed by the demands of large software projects. It does not yet have a large body of code and has not yet faced the common feature-explosion stage of language evolution. On the other hand, the ExoVM system suffers from many of the same weaknesses as any technique for changing large existing software systems. In particular, it does not address all of the corner cases and does not cover the entire JDK or VM internals.

5.1.1. Virgil Limitations

Virgil is remarkably useful and expressive for simple programs that do not require dynamic memory allocation, but larger applications require allocation and hence, automatic memory management. While the use of statically allocated and manually managed pools of objects is a partial solution for intermediate size programs, as systems become larger, static allocation becomes infeasible. To scale to these larger systems, the core Virgil runtime model must offer some form of dynamic allocation. There are numerous techniques that could be fruitful, including explicit or implicit regions, stack allocation, and various garbage collection techniques. For extremely resource-constrained devices such as microcontrollers, efficient automatic memory management remains an open problem.

Virgil makes some tradeoffs between efficiency and extensibility. For example, the lack of a universal super-class combined with the lack of interfaces allows the object model to be implemented simply and efficiently, requiring no metadata for orphan objects. However, it makes it somewhat harder to abstract interfaces between software modules. Virgil's parametric type system addresses this problem for collections and many other situations, but there are situations where specifying an interface between parts of a system is easier with Java-style interfaces or ML-style modules. Here, it may be useful to have a mechanism to place restrictions, either Java-style bounds [70], or

Haskell-style type classes [58]. The Virgil compiler is somewhat suboptimal in its treatment of parametric types; it sometimes duplicates code more than necessary, since in many situations (e.g. copying or searching an array) duplicating the code to the representation size of the type parameter is sufficient. A more efficient solution that preserves the language's current simple semantics may be important for highly parametric code.

Another important limitation is that the Virgil compilation model currently precludes the use of dynamically loaded or updatable code. While this is reasonable for devices where the program binary is replaced wholesale, if it all, dynamic extensibility is needed in other domains. In the future, Virgil may be able to benefit from a module system where initialization and optimization is applied to modules at a time and programs are allowed to dynamically load new modules.

The compression strategies discussed in Chapter 4 have some limitations as well. Currently, the Virgil compiler applies one compression scheme to the entire program, so that all references are compressed with the same strategy and all objects use either horizontal or vertical layout. Results by Cooprider and Regehr [31] suggest that most of the execution time penalty for compression is due to a small number of data items that are accessed frequently but represent small space savings overall. Therefore selectively deciding object layouts or reference compression strategies for different parts of the same program could reduce some of the disadvantages of reference compression. For example, the compiler may apply the best-performing object model for references that are accessed frequently at runtime, while choosing the most space-efficient model for less frequently accessed references. This would allow saving some RAM while avoiding most of the execution time penalty.

The vertical object layout model has some clear weaknesses as well. Most importantly, it requires that no new objects be created at runtime, because object allocation would require growing the field tables individually. Maintaining the contiguous nature of object identifiers while growing tables might be tricky in the presence of subtyping. Further, a garbage collector would need to reclaim entries in the field tables and thus add some bookkeeping overhead; it is not clear whether the costs of such maintenance would outweigh the benefits. One might instead consider a hybrid strategy that uses the vertical layout for those classes that are allocated only at initialization time and not at runtime. Another technique might to be to hybridize both horizontal and vertical layouts for the same class—for example, only part of an object might be stored horizontally, and the rest of the object is stored vertically, with the index stored in the horizontal layout for access. Even in the presence of dynamic allocation of objects, vertical layout may still be useful for meta-objects if the class hierarchy is statically known, which would allow object headers to be compressed.

Our compiler detects read-only component fields and object fields and inlines the values of those that are constant over all objects, but currently it does not move other read-only object fields to ROM. This would be complex in the horizontal layout model because an object might be split into a read-only portion stored in ROM and a read-write portion stored in RAM. An uncompressed horizontal object reference must point to the address of one half of the object, and that half must have a pointer to the other half.

However, when compression is applied to the horizontal layout, the compiler can use one object index but instead have two compression tables, one that holds the address of the RAM portion of the object, and one that holds the ROM address of the object. Even more promising is the idea of using vertical object layout to radically simplify moving individual fields to ROM. Because an entire field is stored contiguously and object indexes are used instead, moving a field array to ROM is trivial; the compiler can generate code to access the appropriate memory space at each field usage site. However, none of these strategies is currently implemented in the Virgil compiler.

5.1.2. ExoVM Limitations

The ExoVM also has several important limitations. First, the persisting techniques that we used are an artifact of the implementation technology of the J9 virtual machine. Much more would be possible if the derivation of constraints and the persistence mechanism could be automated. In our work, we did not completely decompose the entire virtual machine, but only the core parts that were necessary in order to run the benchmark programs. A full-fledged system would need to support the entire language, runtime system, and libraries, which would require considerably more manual effort than our project could muster.

There are more opportunities for static optimization that we could not explore due to either time constraints of the limitations of the underlying programming model. For example, in the ideal static closed-world scenario, the imaging process should be able to copy both the application’s code, the internal data structures of the VM, and also the live code of the VM into the image, producing a completely customized VM compiled together with the application into a standalone program. This would allow the VM and its JIT compiler to be reused as a static compilation system, perhaps allowing it to employ sophisticated compiler optimizations like partial evaluation or static specialization to itself and the application code together. The ExoVM cannot currently achieve this because of the limitations of the linking model of C and C++.

The ExoVM was designed for a very static world, but a dynamic scenario may also benefit from a more flexible VM infrastructure. For example, it might be possible to employ a dynamic feature analysis so that parts of the program and VM infrastructure are loaded as needed by the program. The VM might reduce the granularity of dynamic loading to single methods rather than single classes, only loading methods as they are used. Similarly, the VM might defer the construction of internal data structures until they are demanded by the first use of a particular programming language feature. This may significantly improve performance for small dynamic programs and help combat large class libraries. The ExoVM system is currently not able to support any such techniques because the analysis mechanisms are not built into the loading mechanisms themselves.

6. APPENDIX A – Virgil Grammar

```
Module ::= ( ProgramDecl )? ( TypeDecl )* <EOF>

ProgramDecl ::= "program" <IDENTIFIER> "{" ( ProgramMember )* "}"

ProgramMember ::= EntryPoint
                | ComponentList

EntryPoint ::= "entrypoint" EntryPointName "=" <IDENTIFIER> "."
<IDENTIFIER> ";"

EntryPointName ::= <IDENTIFIER>

ComponentList ::= "components" "{" ComponentRef ( "," ComponentRef )*
"}"

ComponentRef ::= <IDENTIFIER>

TypeDecl ::= ( ClassDecl | ComponentDecl ) "{" ( Member )* "}"

ClassDecl ::= "class" <IDENTIFIER> ( TypeParamDecl )? ( "extends"
TypeRef )?

ComponentDecl ::= "component" <IDENTIFIER>

Member ::= MethodDecl
         | FieldDecl
         | ConstructorDecl

MethodDecl ::= MethodModifiers "method" <IDENTIFIER> ( TypeParamDecl )?
FormalParams ( ":" TypeRef )? MethodBody

MethodModifiers ::=  ( "private" )?

FieldModifiers ::= ( "private" )?

FieldDecl ::= FieldModifiers "field" oneFieldDecl ( "," oneFieldDecl )*
";"

ConstructorDecl ::= "constructor" FormalParams SuperClause MethodBody
```

```
SuperClause ::= ( ":" "super" Arguments )?

oneFieldDecl ::= <IDENTIFIER> ":" TypeRef ( "=" Initializer )?

Initializer ::= ( ArrayInitializer | Expr )

ArrayInitializer ::= "{" ( InitializerList )? "}"

InitializerList ::= Initializer ( "," Initializer )*

FormalParams ::= "(" ( ParamDecl ( "," ParamDecl )* )? ")"

ParamDecl ::= <IDENTIFIER> ":" TypeRef

MethodBody ::= ( ";" | Block )

Block ::= "{" ( BlockStmt )* "}"

BlockStmt ::= LocalVarDecl ";"
            | Stmt

LocalVarDecl ::= "local" oneLocalVarDecl ( "," oneLocalVarDecl )*

oneLocalVarDecl ::= <IDENTIFIER> ( ":" TypeRef )? ( "=" Initializer )?

Expr ::= ConditionalExpr ( ( “=” Expr ) | ( <TK_CASSIGN> Expr ) )?

ConditionalExpr ::= ConditionalOrExpr ( "?" Expr ":" ConditionalExpr )?

ConditionalOrExpr ::= ConditionalAndExpr ( "or" ConditionalAndExpr )*

ConditionalAndExpr ::= InclusiveOrExpr ( "and" InclusiveOrExpr )*

InclusiveOrExpr ::= ExclusiveOrExpr ( “|” ExclusiveOrExpr )*

ExclusiveOrExpr ::= AndExpr ( “^” AndExpr )*

AndExpr ::= EqualityExpr ( “&” EqualityExpr )*
```

```
EqualityExpr ::= TypeQueryExpr ( ( “==” | “!=” ) TypeQueryExpr )*

TypeQueryExpr ::= RelationalExpr ("<:" TypeInExpr )?

RelationalExpr ::= ConcatExpr ( RelationalOp ConcatExpr )*

RelationalOp ::= ( "<" | ">" | ">=" | "<=" )

ConcatExpr ::= ShiftExpr ( “#” ShiftExpr )*

ShiftExpr ::= AdditiveExpr ( (“<<” | “>>”) AdditiveExpr )*

AdditiveExpr ::= MultiplicativeExpr ( (“+” | “-“) MultiplicativeExpr )*

MultiplicativeExpr ::= TypeCastExpr ( (“*” | ”/” | ”%”) TypeCastExpr )*

TypeCastExpr ::= PostIncDecExpr ( "::" TypeInExpr )*

PostIncDecExpr ::= UnaryExpr ( “++” | “--” )?

UnaryExpr ::= ( NegativeLiteral | UnaryOp UnaryExpr | PreIncDecExpr |
Term )

NegativeLiteral ::= “-” <DECIMAL_LITERAL>

PreIncDecExpr ::= ( “++” | “--” ) UnaryExpr

UnaryOp ::= ( “~” | “!” | “+” | “-” )

Term ::= ( ( TermPrefix ( Suffix )* ) | ( NewExpr ( NewSuffix )* ) )

Suffix ::= ( NewSuffix | IndexSuffix )

NewSuffix ::= ( MemberSuffix | AppSuffix )

TermPrefix ::= ( ( VarUse ) | ( Literal ) | ( "(" Expr ")" ) )

VarUse ::= <IDENTIFIER>

Literal ::= <ZERO_LITERAL>
          | <BIN_LITERAL>
          | <OCTAL_LITERAL>
          | <DECIMAL_LITERAL>
          | <HEX_LITERAL>
          | <STRING_LITERAL>
          | <CHARACTER_LITERAL>
          | "null"
          | "this"
          | "true"
          | "false"

NewExpr ::= ( "new" TypeRef ) ( NewArraySuffix | NewObjectSuffix )

NewArraySuffix ::= ArrayDims

NewObjectSuffix ::= Arguments

ArrayDims ::= ( "[" Expr "]" )+

MemberSuffix ::= "." <IDENTIFIER>

AppSuffix ::= Arguments

IndexSuffix ::= "[" Expr "]"

Arguments ::= "(" ( ListExpr )? ")"

ListExpr ::= Expr ( "," Expr )*

Stmt ::= Block
       | EmptyStmt
       | BreakStmt
       | ContinueStmt
       | ReturnStmt
       | WhileStmt
       | ForStmt
       | IfStmt
       | DoWhileStmt
       | SwitchStmt
       | ExprStmt
```

```
IfStmt ::= "if" "(" Expr ")" Stmt ( "else" Stmt )?

SwitchStmt ::= "switch" "(" Expr ")" "{" ( SwitchCase )* "}"

SwitchCase ::= ValueCase
            |  DefaultCase

ValueCase ::= "case" "(" ListExpr ")" Stmt

DefaultCase ::= "default" Stmt

ForStmt ::= "for" "(" ( ListExpr )? ";" ( Expr )? ";" ( ListExpr )? ")"
Stmt

ExprStmtList ::= ExprStmt ( "," ExprStmt )*

ExprStmt ::= Expr ";"

EmptyStmt ::= ";"

BreakStmt ::= "break" ";"

ContinueStmt ::= "continue" ";"

ReturnStmt ::= "return" ( Expr )? ";"

WhileStmt ::= "while" "(" Expr ")" Stmt

DoWhileStmt ::= "do" Stmt "while" "(" Expr ")" ";"

TypeRef ::= ( NestedType | ParameterizedType | SimpleType | FuncType )
( "[" "]" )*

NestedType ::= "(" TypeRef ")"

FuncType ::= ( "function" "(" ( TypeList )? ")" ( ":" TypeRef )? )

TypeParamDecl ::= "<" TypeParam ( "," TypeParam )* ">"

TypeParam ::= <IDENTIFIER>
```

```
ParameterizedType ::= <IDENTIFIER> "<" TypeList ">"

TypeList ::= TypeRef ( "," TypeRef )*

TypeInExpr ::= ( SimpleType | NestedType )

SimpleType ::= ( SingularType | RawType )

SingularType ::= <IDENTIFIER>

RawType ::= <DECIMAL_LITERAL>
```

7. REFERENCES

[1] ECMA Standard 334. C# Language Specification. Available at: `http://www.ecma-international.org/`

[2] Connected Limited Device Configuration (CLDC). `http://java.sun.com/j2me`

[3] The Scala Programming Language. `http://www.scala-lang.org`

[4] Java Technology: The Early Years. `http://java.sun.com/features/1998/05/birthday.html`

[5] Objective CAML. `http://caml.inria.fr/ocaml`

[6] Java in the Small. `http://www.lifl.fr/RD2p/JITS/`

[7] Forth Interest Group Homepage. `http://www.forth.org/`

[8] D. Abrahams and A. Gurtovoy. C++ Template Metaprogramming. Addison-Wesley, 2004.

[9] A. Adl-Tabatabai, J. Bharadwaj, M. Cierniak, M. Eng, J. Fang, B. Lewis, B. Murphy, and J. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. *In CGO'04, International Symposium on Code Generation and Optimization*. San Jose, CA. March 2004.

[10] O. Agesen and D. Ungar. Sifting out the Gold: Delivering Compact Applications from an Exploratory Object-oriented Programming Environment. In *OOPSLA '94,*

the 9th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications. Portland, OR. October 1994.

[11] A. Aiken. Introduction to Set Constraint-Based Program Analysis. In *Science of Computer Programming* 35(2-3): 79-111. November 1999.

[12] A. Aiken, E. Wimmers, and J. Palsberg. Optimal Representations of Polymorphic Types with Subtyping. UCB Tech Report CSD 96-909. July 1996.

[13] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *OOPSLA '03, the 18th Annual Conference on Object-Oriented Systems, Languages, and Applications*. Anaheim, CA. October 2003.

[14] E. Allen, R. Cartwright, and B. Stoler. Efficient Implementation of Run-time Generic Types for Java. *In Proceedings of the IFIP TC2/WG2.1Working Conference on Generic Programming*, pp. 207-236. July 2002.

[15] B. Alpern, D. Attanasio, J. Barton, A. Cocchi, S. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeno in Java. In *OOPSLA '99, the 14th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications*. Denver, CO. November 1999.

[16] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *OOPSLA '01, the 16th Annual Conference on Object-Oriented Systems, Languages, and Applications*. Tampa, FL. October 2001.

[17] C. Ananian and M. Rinard. Data Size Optimizations for Java Programs. In *LCTES '03, Workshop on Languages, Compilers, and Tools for Embedded Systems*. San Diego, CA. June 2003.

[18] M. Atkinson, M. Dmitriev, C. Hamilton, T. Printezis: Scalable and Recoverable Implementation of Object Evolution for the PJama1 Platform. *Lecture Notes in Computer Science Volume 2135*: 292-314, 2000.

[19] D. Bacon. Kava: a Java Dialect with a Uniform Object Model for Lightweight Classes. *Concurrency and Computation: Practice and Experience 15(3-5): 185-206.* 2003.

[20] D. Bacon, S. Fink, and D. Grove. Space- and Time-efficient Implementation of the Java Object Model. In *ECOOP '02, the 16th European Conference on Object-Oriented Programming*, University of Malaga, Spain. June 2002.

[21] D. Bacon and P. Sweeney. Fast Static Analysis of C++ Virtual Calls. In *OOPSLA '96, the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*. San Jose, CA. October 1996.

[22] G. Bracha, N. Cohen, C. Kemper, M. Odersky, D. Stoutamire, K. Thorup, and P. Wadler. Adding Generics to the Java Programming Language. Java Community Process JSR-000014. September 2004.

[23] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *PLDI '00, the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vancouver, Canada. June 2000.

[24] M. Budiu, S. Goldstein, M. Sakr, and K. Walker. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *EuroPar 2000, the EuroPar 2000 European Conference on Parallel Computing*. Munich, Germany. August 2000.

[25] D.-W. Chang and R.-C. Chang. Ejvm: an Economic Java Run-time Environment for Embedded Devices. *Software Practice & Experience*, 31(2):129-146, 2001.

[26] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. Henzinger, and J. Palsberg. Stack Size Analysis for Interrupt-driven Programs. *Information and Computation* 194:144-174. 2004.

[27] Z. Chen. Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Longman Publishing Co., Inc. 2000.

[28] G. Chen, M. Kandemir, N. Vijaykrishnan, M. Irwin, B. Mathiske, and M. Wolczko. Heap Compression for Memory-constrained Java Environments. In *OOPSLA '03, the 18th Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* Anaheim, CA. October 2003.

[29] N. Cohen. Type Extension Type Tests Can Be Performed in Constant Time. *ACM Transactions on Programming Languages and Systems*, 13(4), 626-629. 1991.

[30] A. Courbot, G. Grimaud, and J. Vandewalle. Romization: Early Deployment and Customization of Java Systems for Constrained Devices. In *CASSIS '05, the Second International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices.* Nice, France. March 2005.

[31] N. Cooprider and J. Regehr. Offline compression for on-chip RAM. In *PLDI'07, ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA. June 2007.

[32] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *ECOOP '95, the 9th European Conference on Object-Oriented Programming.* Aarhus, Denmark. August 1995.

[33] I. Diatchki, M. Jones, and R. Leslie. High-level Views on Low-level Representations. In *ICFP '05, International Conference on Functional Programming*. Tallinn, Estonia. September 2005.

[34] I. Diatchki and M. Jones. Strongly Typed Memory Areas; Programming Systems-level Data Structures in a Functional Language. In Haskell '06 Workshop. Portland, Oregon. September 2006.

[35] A. Diwan, K. McKinley, and J. E. Moss. Using Types to Analyze and Optimize Object-Oriented Programs. *In ACM Transactions on Programming Languages and Systems,* 23(1), 30-72. 2001.

[36] L. Dragan and S. Watt. Parametric Polymorphism Optimization for Deeply Nested Types in Computer Algebra. In *Proceedings of the 2005 Maple Conference*, pp. 243-259. Waterloo, Canada. July 2005.

[37] B. Delsart, V. Joloboff, and E. Paire. JCOD: Lightweight Modular Compilation Technology for Embedded Java. In *EMSOFT '02, the Second International Conference on Embedded Software*. London, UK. October 2002.

[38] D. Dreyer, R. Harper, and M. Chakravarty. Modular Type Classes. In *POPL '07, the ACM Symposium on Principles of Programming Languages*. Chicago, IL. January 2007.

[39] N. Eckel and J. Gil. Empirical Study of Object-layout Strategies and Optimization Techniques. In *the 14th European Conference on Object-Oriented Programming (ECOOP '00)*. Sophia Antipolis and Cannes, France. June 2000.

[40] B. Emir, A. Kennedy, C. Russo, and D. Yu. Variance and Generalized Constraints for C# Generics. *In ECOOP '06, the 20th European Conference on Object-Oriented Programming*. Nantes, France. July 2006.

[41] A. Gal and M. Franz. Incremental Dynamic Code Generation with Trace Trees. University of California, Irvine Tech Report No. 06-16. November 2006.

[42] A. Gal, C. Probst, and M. Franz. HotPathVM: An Effective JIT Compiler for Resource-Constrained Devices. In *VEE '06, the Second International Conference on Virtual Execution Environments*. Ottawa, Canada. June 2006.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[44] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. *In PLDI '03, the ACM Conference on Programming Language Design and Implementation*. San Diego, CA. June 2003.

[45] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX Operating System. *In USENIX 2002, the USENIX Annual Technical Conference*. Monterey, CA. June 2002.

[46] J. Gosling, B. Joy, and G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[47] C. Grothoff. Expressive Type Systems for Object-Oriented Languages. PhD Dissertation, University of California Los Angeles. September 2006.

[48] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In MOBISYS '05, *the International Conference on Mobile Systems, Applications, and Services*. Seattle, WA. June 2005.

[49] P. Hansen. The Solo Operating System: A Concurrent Pascal Program. In *Software—Practice and Experience* 6(2): 141-149. April 1976.

[50] T. Harbaum. NanoVM: Java for the AVR. Available at: `http://www.harbaum.org/till/nanovm/`

[51] R. Harper and G. Morrisett. Compiling Polymorphism Using Intensional Type Analysis. In *POPL '95, the ACM Conference on Principles of Programming Languages*. San Francisco, CA. January 1995.

[52] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. IEEE Micro, 22(6):12-24, November/December 2002.

[53] G. Hunt, et al. An Overview of the Singularity Project. Microsoft Technical Report MSR-TR-2005-135. October 2005.

[54] A. Igarashi and M. Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. In *ACM Transactions on Programming Languages and Systems*, 28(5):795-847, September 2006.

[55] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, T. Nakatani. A Study of Devirtualization Techniques for a Java Just-in-time Compiler. In *OOPSLA '00, the 15th Annual Conference on Object-Oriented Systems, Languages, and Applications*. Minneapolis, MN. October 2000.

[56] P. Jain, and D. Schmidt. Service Configurator: A Pattern for Dynamic Configuration of Services. *In Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems*. Anaheim, CA. June 1997.

[57] J. Jarvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++. In PLDI '06, *the ACM Conference on Programming Language Design and Implementation*. Ottawa, Canada. July 2006.

[58] S. Jones, M. Jones, and E. Meijer. Type Classes: An Exploration of the Design Space. In *Haskell '97, the ACM SIGPLAN Haskell Workshop*. Amsterdam, The Netherlands. May 1997.

[59] M. Kawahito, H. Komatsu, and T. Nakatani. Effective Null Pointer Check Elimination Utilizing Hardware Trap. In *ASPLOS '00, the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*. Cambridge, MA. November 2000.

[60] A. Kennedy and D. Syme. Combining Generics, Pre-compilation and Sharing Between Software-Based Processes. In *SPACE '04, Semantics, Program Analysis, and Computing Environments*. Venice, Italy. January 2004.

[61] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI '01, the ACM Conference on Programming Language Design and Implementation*. Snowbird, UT. June 2001.

[62] O. Kiselyov and C. Shan. Position: Lightweight Static Resources: Sexy types for embedded and systems programming. In *TFP '07, the 8th Symposium on Trends in Functional Programming*. New York, NY. April 2007.

[63] J. Koshy and R. Pandey. VM*: A Scalable Runtime Environment for Sensor Networks. In *SENSYS '05, the 3rd annual conference on Embedded Network Sensor Systems*. San Diego, CA. November 2005.

[64] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *MSP '02, ACM Workshop on Memory System Performance*. Berlin, Germany. June 2002.

[65] C. Lattner and V. Adve. Transparent pointer compression for linked data structures. In *MSP'05, ACM Workshop on Memory System Performance.* Chicago, IL. June 2005.

[66] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS '02, the 11th Conference on Architectural Support for Programming Languages and Operating Systems*. San Jose, CA. October 2002.

[67] G. Manjunath and V. Krishnan. A Small Hybrid JIT for Embedded Systems. *ACM SIGPLAN Notices*, 35(4):44-50, 2000.

[68] J. Mogul, J. Bartlett, R. Mayo, and A. Srivastava. Performance implications of multiple pointer sizes. In *USENIX'95, Technical Conference on UNIX and Advanced Computing Systems*, pp.187-200, 1995.

[69] A. Myers. Bidirectional Object Layout for Separate Compilation. In *OOPSLA'95, the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. Austin, TX. October 1995.

[70] M. Naftalin and P. Wadler. Java Generics and Collections. O'Reilly, October 2006.

[71] M. Naik and J. Palsberg. Compiling with Code-Size Constraints. In *ACM Transactions on Embedded Computing Systems*. 3(1): 163-181. 2004.

[72] G. Necula, S. McPeak, S. Rahul, and W. Weiner. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02, the International Conference on Compiler Construction*. Grenoble, France. April 2002.

[73] R. Newton, Arvind, and M. Welsh. Building up to Macroprogramming: An Intermediate Language for Sensor Networks. In *IPSN '05, the Fourth International Conference on Information Processing in Sensor Networks*. Los Angeles, CA. April 2005.

[74] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, M. Zenger. An

Overview of the Scala Programming Language. EPFL Technical Report LAMP-REPORT-2006-001. 2006.

[75] M. Odersky, and P. Wadler. Pizza into Java: Translating Theory into Practice. In *POPL '97, ACM Conference on Principles of Programming Languages*. Paris, France. January 1997.

[76] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *OOPSLA '99, the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* Denver, CO. October 1999.

[77] J. Palsberg. Type-based Analysis and Applications. In *PASTE '01, the ACM SIGPLAN Workshop on Program Analysis for Software Tools*. Snowbird, UT. June 2001.

[78] C. Probst, A. Gal, and M. Franz. Code Generating Routers: A Network-Centric Approach to Mobile Code. In *CCW '03, the 18th Annual IEEE Workshop on Computer Communications*. Dana Point, CA. October 2003.

[79] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for Applications, a Way Ahead of Time (WAT) Compiler. *In USENIX '97, the Third USENIX Conference on Object-Oriented Technologies*. Anaheim, CA. June 1997.

[80] W. Pugh and G. Weddell: Two-directional record layout for multiple inheritance. In PLDI'90, ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.85-91, 1990.

[81] D. Rayside and K. Kontogiannis, Extracting Java library subsets for deployment on embedded systems, Science of Computer Programming 45(2-3): 245-270, 2002.

[82] D. Rayside, E. Mamas, and E. Hons. Compact java binaries for embedded systems. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, p. 9, IBM Press, 1999.

[83] K. Redwine and N. Ramsey. Widening Integer Arithmetic. In *CC '04, the 13th Annual Conference on Compiler Construction*. Barcelona, Spain. April 2004.

[84] J. Regehr, A. Reid, and Kirk Webb. Eliminating Stack Overflow by Abstract Interpretation. In *EMSOFT '03, the 3rd International Conference on Embedded Software*. Philadelphia, PA. October 2003.

[85] M. Sakkinen. The darker side of C++ revisited. *Structured Programming*, 13:155-177, 1992.

[86] M. A. Schubert, L.K. Papalaskaris, and J. Taugher. Determining Type, Part, Colour, and Time Relationships. Computer, 16:53–60, October 1983.

[87] J. Siek and A. Lumsdaine. Essential Language Support for Generic Programming. In *PLDI'05, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005. Chicago, IL. June 2005.

[88] D. Spoonhower, J. Auerbach, D. Bacon, P. Cheng, and D. Grove. Eventrons: A Safe Programming Construct for High-Frequency Hard Real-Time Applications. In *PLDI '06, the ACM SIGPLAN Conference on Programming Language Design and Implementation.* Ottawa, Canada. June 2006.

[89] M. Sridharan, S. Fink, and R. Bodik. Thin Slicing. In *PLDI'07, the ACM SIGPLAN Conference on Programming Language Design and Implementation*. San Diego, CA. June 2007

[90] B. Steensgard. Points-to Analysis in Almost Linear Time. Microsoft Technical Report, MSR-TR-95-08. 1995.

[91] B. Stroustrup. Evolving a Language in the Real World: C++ 1991-2006. In *HOPL-III, the ACM Conference on History of Programming Languages*. San Diego, CA. June 2007.

[92] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *OOPSLA '00 the 15th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Minneapolis, MN. October 2000.

[93] P. Sweeney and M. Burke. Quantifying and Evaluating the Space Overhead for Alternative C++ Memory Layouts. *Software—Practice and Experience*. 33(7): 595-636. June 2003.

[94] P. Sweeney and J. Gil. Space and Time-Efficient Memory Layout for Multiple Inheritance. In *OOPSLA '99, the 14th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Denver, CO. October 1999.

[95] P. Sweeney and F. Tip. A Study of Dead Data Members in C++ Applications. In *PLDI '98, the ACM SIGPLAN Conference on Programming Language Design and Implementation*. Montreal, Canada. June 1998.

[96] W. Taha, S. Ellner, and H. Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. In *EMSOFT '03, the 3rd Annual International Conference on Embedded Software*. Philadelphia, PA. October 2003.

[97] A. Taivalsaari, B. Bush, and D. Simon. The Spotless System: Implementing a Java System for the Palm Connected Organizer. 1999.

[98] F. Tip, C. Laffra, P. Sweeney, A. Eisma, and D. Streeter. Practical extraction techniques for Java. In *ACM Transactions on Programming Languages and Systems* 24(6): 625-666. November 2002.

[99] F. Tip and J. Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In *OOPSLA '00, the 15th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Minneapolis, MN. October 2000.

[100] F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical Extraction Techniques for Java. *ACM Transactions on Programming Languages and Systems*, 24(6): 625-666, 2002.

[101] B. Titzer. Virgil: Objects on the Head of a Pin. In *OOPSLA'06, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Portland, OR. October 2006.

[102] B. Titzer, J. Auerbach, D. Bacon, and J. Palsberg. The ExoVM System for Automatic VM and Application Reduction. In *PLDI'07, ACM SIGPLAN*

Conference on Programming Language Design and Implementation. San Diego, CA. June 2007.

[103] B. Titzer and J. Palsberg. Nonintrusive Precision Instrumentation of Microcontroller Software. In *LCTES '05, ACM SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. Chicago, IL. June 2005.

[104] B. Titzer and J. Palsberg. Vertical Object Layout and Compression for Fixed Heaps. In *CASES '07, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. Salzburg, Austria. November 2007.

[105] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *IPSN '05, The Fourth International Conference on Information Processing in Sensor Networks*. Los Angeles, CA. April 2005.

[106] P. Tyma. Optimizing Transforms for Java and .NET Closed Systems. PhD Dissertation, Syracuse University. 2004.

[107] A. Varma and S. Bhattacharyya. Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems. In *DATE '04, the Conference on Design Automation and Test in Europe*. Paris, France. February 2004.

[108] J. Vitek, B. Bokowski. Confined Types. In *OOPSLA '99, the 14th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Denver, CO. Oct. 1999.

[109] J. Vitek, R. N. Horspool, and A. Krall. Efficient Type Inclusion Tests. In *OOPSLA '97, the 12th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*. Atlanta, GA. October 1997.

[110] G. Wagner, A. Gal, and M. Franz. SlimVM: Optimistic Partial Program Loading for Connected Embedded Java Virtual Machines. University of California, Irvine Tech Report No. 06-18. November 2006.

[111] P. Wilson. Operating system support for small objects. In *the 1991 International Workshop on Object Orientation in Operating Systems*. Palo Alto, CA. October 1991.

[112] I. Wirjawan, J. Koshy, R. Pandey, and Y. Ramin. Balancing Computation and Code Distribution Costs: The Case for Hybrid Execution in Sensor Networks. In *SECON '06, the IEEE Conference on Sensors, Mesh, and Ad Hoc Communications and Networks*. Reston, VA. September 2006.

[113] N. Wirth. The Programming Language Oberon. In *Software—Practice and Experience 18(7): 671-690*. July, 1988.

[114] G. Wright, M. Seidl, and M. Wolczko. An object-aware memory architecture. Science of Computer Programming 62(2): 145-163 (2006).

[115] Y. Zhang and R. Gupta. Compressing heap data for improved memory performance. *Software—Practice and Experience*, 36(10):1081-1111, 2006.

[116] T. Zhao, J. Palsberg, and J. Vitek. Lightweight Confinement for Featherweight Java. In *OOPSLA '03, the 18th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Anaheim, CA. October 2003.