Virgil: Objects on the Head of a Pin

Ben L. Titzer
UCLA Compilers Group
4810 Boelter Hall
Los Angeles, CA 90095
1-310-206-3844
titzer@cs.ucla.edu

Abstract

Embedded microcontrollers are becoming increasingly prolific, serving 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 paper, I will present the design and implementation of Virgil, a lightweight object-oriented language designed with careful consideration for resource-limited domains. Virgil explicitly separates *initialization time* from runtime, allowing an application to build complex data structures during compilation and then run directly on the bare hardware without a virtual machine or any language runtime. This separation allows the entire program heap to be available at compile time and enables three new data-sensitive optimizations: *reachable members analysis*, *reference compression*, and *ROM-ization*. Experimental results demonstrate that Virgil is well suited for writing microcontroller programs, with five demonstrative applications fitting in less than 256 bytes of RAM with fewer than 50 bytes of metadata. Further results show that the optimizations presented in this paper reduced code size between 20% and 80% and RAM size by as much as 75%.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features –*Classes and Objects, Dynamic Storage Management, Inheritance*

General Terms Algorithms, Management, Performance, Design, Experimentation, Verification.

Keywords Embedded systems, microcontrollers, static analysis, data-sensitive optimization, heap compression, systems software, standalone programs, multi-stage computation, whole-program compilation, dead code elimination, sensor networks

1. Introduction

1.1 Background

Microcontrollers are tiny, low-power processors that contain a central processing unit, flash memory, RAM, and I/O devices on a single physical chip. Marked by limited computational power and very small memories, they often serve as the central or auxiliary control in systems where the presence of a computer may not be readily apparent, such as a fuel-injection system. Microcontrollers represent one of the most extreme instances of a software-programmable resource-constrained embedded system. For example, the 8-bit AVR architecture offers chips with flash sizes between 8 and 128 kilobytes and just 64 to 4096 bytes of RAM, with clock rates usually between 4 and 20mhz.

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 processes, threads, and semaphores; thus most programs run directly on the microcontroller without any of the protection mechanisms that are common on desktop computing platforms.

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. Within the sensor network domain, a number of parallel and distributed languages and programming paradigms have been proposed [25]. Virgil is not an attempt to solve the distributed problems in sensor networks but rather a compiler/language system for programming resource-constrained embedded systems, of which sensor nodes are one example. In this space the most popular sensor systems are either programmed in C or nesC [16], an extension to C that includes module capabilities and a simple task model.

1.2 Motivation

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 paper considers the question of how object technology can benefit developing software in this domain. I believe that 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 new languages and paradigms.

The Virgil programming language is an attempt to address the challenge of matching objects to microcontrollers at the language and compiler level. The most important design consideration when taking this approach is the space overhead that language features add to the program implementation. For the purposes of this paper, this overhead can be divided into two categories: *runtime*, which consists of libraries, routines

OOPSLA'06, October 22–26, 2006, Portland, Oregon, USA.

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.

1.3 Structure

The rest of this paper is structured as follows. Section 2 sets out the most important design criteria that have guided the development of the Virgil language. Section 3 discusses the language features that have survived the crucible of design criteria and how each can be implemented efficiently with known techniques. Language features that were rejected are also discussed. Section 4 describes the compilation model of a Virgil program, including the initialization time concept. Section 5 explores the implications of the compilation model by describing three new optimizations that it makes possible. Section 6 discusses experience with the language and its implementation, providing experimental results. Section 7 discusses related work. Section 8 gives a conclusion and Section 9 describes future work.

2. Design Principles / Constraints

In this section, we will explore the design principles for the Virgil language and the constraints that the domain imposes. Each principle is oriented toward easing the difficult task of building embedded software through modularity, static checking, and expressive features. To this end, Virgil's 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.

While the design principles outline desirable properties of such a language, the domain imposes 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. In Virgil, one of the primary underlying efficiency considerations 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 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.** Virgil programs will run as the lowest layer of software, so the notion of a language runtime underlying a Virgil program is a bit problematic. Secondly, because of the severe resource limitations of a microcontroller, any language runtime system can represent a significant cost in terms of both code space, data space and execution efficiency. Because the runtime system is often implemented "under the language," it is generally not under the control of the application or system programmer and may vary from implementation to implementation. For the microcontroller domain, a programmer needs 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. While a general-purpose garbage collector eliminates most problems in modern languages, most microcontroller programs are already written to statically pre-allocate all the memory that will be needed during execution. For example, one of the nesC language's primary design criteria was that dynamic memory allocation is unnecessary for the targeted class of systems. Thus even without dynamic allocation,

nearly all microcontroller and sensor network programs are realizable, though some must resort to statically allocated and manually managed object pools of fixed size.

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. The programmer is likely willing to trade some space for better language features, provided the overhead is acceptably small and readily apparent during implementation and tuning.

3. 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 as presented here will further reduce program footprint, lightening the burden on the programmer, leading to higher productivity and more robust systems.

3.1 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 implicitly 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. [3] Discusses efficient implementation techniques for Java interface dispatch in the Jikes RVM; it 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 object instance. A Virgil object requires only a single-word header that holds a pointer to its class's meta-object containing an integer type id and a virtual dispatch table. Class-based inheritance, whether Java, C++, or Virgil, requires the meta-object for a class to be at least as large as the meta-object for its super-class, plus the number of new methods declared in the class. Because Virgil has no universal super-class, a root class inherits nothing and contains only a type id and the virtual methods declared in the class. Additionally, Virgil meta-objects are read-only and can be stored in ROM, saving 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 [28], 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.

3.1.1 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 metadata (the `getClass()` method), hashing, equality, etc. A number of these services require meta-data in object headers, in addition to mark bits needed by the garbage collector. Bacon et al [6] 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 needed for object headers. For meta-objects, as we saw in the previous section, inheritance requires the meta-object for a class to be at least as large as that of its super-class. `Object` in Java 5 contains 11 virtual methods, which bloats every meta-object in the application.

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, as in Java, rejects casts between unrelated classes, 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. My personal experience with large applications in Java gives me the impression 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 [13][26] 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 [37][39].

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 section 5, 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. 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. However, since Virgil is focused on building systems rather than applications, a Virgil program will likely be mostly self-contained or tightly coupled with a single library like a small microcontroller operating system.

Clients also have the option to employ the Adapter [15] pattern by writing wrapper classes to adapt their classes to the API of various libraries. Delegates (Section 3.3) reduce this problem by allowing limited functional programming, but as Virgil applications are scaled up, developers begin to stress the language and a more robust solution is needed. One implementation technique is fat pointers, which encode the type information for an object in the reference itself rather than in the header of the object, allowing orphan objects to still be represented without a header. However, fat pointers can increase the size of references and may negate the space saved by omitting headers for orphans. I believe the best future solution overall is generic types [8], which will allow library designers to write classes that are polymorphic over any type, including primitives, while providing static type safety. The implementation issues for this change are discussed briefly in the future work section.

3.2 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, all class members in Virgil 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 types. Components also serve an important purpose that will be explored more in Section 4: they encapsulate the initialization portion of the program and their fields serve as the roots of the live object graph.

3.3 Delegates

Purely class-based languages have one important drawback that design patterns such as the Adapter, Observer, and Visitor [15] 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. In my opinion, this is a language flaw that can lead to needlessly complicating applications and libraries with interfaces. Parametric types are 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 types. 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 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 transparently 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 constitute the delegate tuple. Referring to a component method as a delegate creates a tuple with `null` as the object. Applying a delegate to its argument is implemented as a simple indirect function call, passing the bound object reference as the hidden `this` parameter if necessary. Since method resolution takes place at creation time rather than invocation time, delegate invocations actually require one fewer memory access than a virtual dispatch. 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.

```
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 1: 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()`.

3.4 Hardware Registers and Interrupts

Although hardware access is beyond the scope of this paper, Virgil has support for directly accessing hardware I/O registers in a controlled way, without having to resort to calls to native methods, indirect accesses through pointers, "fake" classes, 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 fields of a special component named `device` that can be read or written with primitive types only. 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. Hardware interrupts can be handled directly by component methods, allowing a complete hardware device driver to be written entirely in Virgil, without any underlying unsafe code.

3.5 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 the program's type system. This requires a significant runtime support structure. Additionally, dynamic loading can invalidate essentially any compiler optimization, which forces a static compiler to be overly conservative.

iii. **Reflection.** The ability to inspect the members of objects, modify them, search by name, and various other things that reflection allows 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 such as [23] should rather be replaced with static configuration mechanisms.

iv. **Garbage collection.** Garbage collection is unnecessary under the current design constraints, because no dynamic memory allocation is allowed.

v. **Method Overloading.** C++, Java, and C# all allow overloading methods by their parameter types. Although overloading is a purely static form of polymorphism and thus has no inherent runtime cost, it conflicts with Virgil's delegate mechanism. Virgil supports using a method as a delegate by simply referring to its name; the presence of overloading would introduce ambiguity that would require a resolution mechanism that detracts from the simplicity of Virgil delegates.

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.

4. 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.

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

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

Figure 2: 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.

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, a constructor might not terminate, which of course is undecidable. The Virgil compiler makes no attempt to enforce that the initialization phase terminates; this is left 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.

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.

4.1 Initialization Order

The order in which component constructors are executed is deterministic and given by their order in the master program declaration in which the programmer lists the components that are part of the program. 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 at the beginning of 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). One drawback of persistent systems such as Smalltalk

has been that replicating the initialization environment for a particular program can be nontrivial. In Virgil, the program does not depend on the compilation environment's objects, but instead builds its own object heap.

4.2 Initialization Garbage

When the constructors of the components have terminated, the compiler will perform a garbage collection phase that removes objects that were allocated by the initialization code but are unreachable. The fields of components serve as roots into the graph of objects that represents the entire heap of the program. The compiler traces from the component fields through objects and object fields to discover everything transitively reachable from the roots. Temporary objects allocated during initialization that are not reachable are discarded. Only the code and metadata associated with live objects are included in the final program binary.

4.3 Code Generation and Runtime

After the initialization phase and garbage collection, 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.

5. Optimizations

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, every Virgil compiler is required to employ Class Hierarchy Analysis [12] 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 that are described in this section. The first, *reachable members analysis*, removes code, objects, and fields of objects that are unused in the program. *Reference compression* 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.

5.1 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 statically reachable but are not used 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. To encapsulate this problem, we need a more general notion, *semantic reachability,* where objects and their constituent fields are considered live only if they are accessed at runtime.

A compiler may employ semantic reachability to slice the program and remove objects and fields that are dead. 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 [32][35], but they require conservative assumptions due to the consideration of initialization code, because dynamic memory allocation in the program may cause object constructor code to be invoked. 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.

5.1.1 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. Then the code of each method would be inspected for accesses of root fields **R** and instance fields of objects. Unused fields would be considered dead and removed from the root set and from each object instance in the heap.

```
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: 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.

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 [7], 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. However, 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 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.

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.

5.1.2 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 live and puts the referenced object into a "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 considers only method implementations corresponding to object types in the current live set. The algorithm iterates until there are no new method implementations or objects to analyze.

Figure 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,

```
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 4: The RMA algorithm's 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.

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()` method 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.

5.1.3 Algorithm Complexity

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 expected 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.

5.1.4 Pull Members Down

The algorithm as presented can be used to compute the necessary information for the *pull members down* optimization that saves space in instances of super-classes where fields are declared but accessed only in instances of its subclasses. This transformation originally appeared in automated refactoring tools, but admits a small opportunity for space savings in this context by allowing the field to be deleted in the super-class and retained in the subclasses. Tyma in [36] 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.

5.2 Reference Compression

The most severe resource constraint of a microcontroller is the RAM space available to store the objects of the heap and the runtime stack. While reachable members analysis saves RAM space by removing unreachable objects and removing fields of objects that are statically unused in the program, *reference compression* is an optimization that exploits the type-safe property of Virgil to encode object references in a more compact way, reducing the size of reference fields in objects.

On microcontroller architectures with between 256 bytes and 64 kilobytes of RAM, pointers into the memory are usually represented with a 16-bit integer that contains a byte address. In a weakly typed language like C, a pointer is not restrained to point to values of any particular type and can conceivably hold any byte address. In fact, pointer arithmetic relies on the fact that pointers are represented as integers and allows arithmetic 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 Virgil, as in any strongly typed language, references into the heap are typed and may only refer to heap entities of the corresponding type. For example, strong types enforce that an object reference of declared type `A` must only refer to objects of type `A` and its subtypes. References cannot be converted to integers or vice versa; the implementation of references is entirely opaque to the program. Recall that after initialization time, a Virgil program has already allocated all the objects of type `A` and its subtypes that will ever exist in the heap and no further objects can be allocated at runtime. The compiler can exploit the 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.

Consider a program that allocates some number of objects of type `A`. Everywhere a reference of declared type `A` occurs, because of type safety, the reference may only refer to one of the allocated objects (or possibly `null`). Conceptually, if `K` is the number of objects of type `A` that exist in the entire program heap, the field representation requires only `log(K+1)` bits to distinguish between the possible objects that can be referenced by the field. This idea forms the basis of the reference compression algorithm.

5.2.1 Heap Layout

Before choosing the representation of references, the compiler must arrange objects in memory by assigning them addresses. Consider a Virgil program **P** and a heap **H** that has been obtained by running the initialization phase of the program. Assume that each object in the heap **H** will reside in memory at some fixed address. The *heap layout problem* is therefore to assign each object $\mathbf{O_i}$ in **H** an address $\mathbf{A_i}$ in memory that does not overlap with other objects. For the simplest version of the reference compression algorithm, we will assume that the solution to the heap layout problem may place an object anywhere in memory, but later we will explore more advanced layout techniques.

5.2.2 Reference Representation

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 and rely on the type-safety of the language to limit the referencible set to those live objects whose dynamic type is a subtype of the declared type.

5.2.3 Compression Tables

One straightforward way to implement type-based reference compression is to use a compression table. In this approach, each compressed reference is an object handle: an integer index into a table that contains the actual addresses of each object. Conceptually, we can compress each type of reference by creating a compression table for its associated referencible set. 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 storing a 16-bit address.

The compression table is read-only and can therefore be stored in the ROM, which is considerably larger than RAM; this represents a classic tradeoff by consuming some ROM space for the table and saving some RAM space with compressed references. The compression table also introduces a slight runtime overhead for field reads due to the extra indirection. Field updates must also write a compressed reference into a heap field and not an object address; to avoid the need to compute the compressed reference from an object address, object handles can be used throughout the program and classical compiler optimizations employed to cache the actual object addresses whenever possible to reduce repeated accesses of compression tables, especially within loops, etc.

Note that when applying the type-based referencible set approximation on programs with subtyping, the subtyping induces a subset relation on referencible sets; in particular, each referencible set for a type is the union of the referencible sets of each of its subtypes. We can simplify the problem by unifying the referencible set for each type with its parent type. Thus the lack of a universal super-class in Virgil avoids unifying the referencible sets for all classes; therefore there is one referencible set for each class hierarchy.

5.2.4 Indexed Addresses

Another possible scheme for compressing references is to choose a heap layout with particular properties that can be exploited to represent references more compactly. 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 can be used as the compressed reference. With this approach, we could again use the type-based referencible set approximation and place all objects of the same type next to each other in memory. 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.

5.3 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 can 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 where reads occur. If it is constant by subclass [4], the compiler can move it to the meta-object, and otherwise, the compiler could introduce a constant table stored in ROM that is indexed by the object handle obtained with reference compression.

5.4 Metadata Optimizations

In addition to optimizing the layout of objects within the heap and compressing their reference fields, 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 entries in the meta-object that correspond to calls that have been devirtualized can be removed as well. Secondly, the compiler can apply reference compression to 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.

6. Experience

I have implemented a prototype compiler that compiles the complete Virgil language. The front-end parses, typechecks, and runs the initialization phase to obtain the complete program heap. The middle of the compiler implements reachable members analysis and optimizes the program with the results. The compiler generates a program binary by way of a native C compiler. The C source code emitted by the prototype compiler implements the Virgil program, including the live objects in the heap, their metadata, and all the reachable code as C structures and functions. This C program includes all code necessary to run on the bare device, and does not require the use of any C libraries, including `libc`, the C language runtime. 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 prototype compiler has been used to develop a number of small real programs that have been successfully compiled and run on the Mica2 sensor network nodes.

6.1 Decoder Example

After some experience writing code in Virgil, the initialization time concept has proved to be quite useful. An application can run a substantial initialization routine at compile time in order to allocate and configure its data structures. 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.

One illustration of the flexibility that this mechanism provides is in the `Decoder` application. The `Decoder` application builds a binary 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 binary tree, or the code to implement the binary tree, by hand; efficient algorithms exist to build a decision tree in time linear in the number of bit patterns. The decoder application runs this algorithm during its initialization phase by creating a `DecoderBuilder` object in the constructor of the main application and inserting bit patterns corresponding to various commands. After the patterns are added, the `build()` method is called, which determines the structure of the tree, allocates the tree nodes, and connects them together. A reference to the completed decoder tree is stored for use at runtime, and 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. The compiler performs reachable members analysis to remove all other code and data structures that are unreachable from the entry point to the program; in particular it removes the complex initialization code for the `DecoderBuilder`. Then reference compression is applied to the data structures, reducing the size of the node objects.

6.2 Optimization Results

This section presents experiments that demonstrate the efficacy of the optimizations described in this paper. The prototype compiler fully implements reachable members analysis and transforms the program according to the results. The results reported for reference compression represent accurate space usage numbers computed by the compiler, but the actual transformation of program code is not yet implemented and therefore performance and code size measurements are not available for this optimization. The ROM-ization optimizations are not currently implemented in the prototype compiler.

The five benchmark programs used in this section are: `Blink`, a small program that uses the timer device driver to toggle an LED once per second; `CntToLeds`, a simple program that displays a counter on the LEDs one per second; `List`, a small program that uses doubly linked lists; `Decoder`, the decoder application with a set of 17 different bit patterns to

disambiguate; and `MPK`, a message passing kernel that uses objects to represent messages and handlers. Both the `Decoder` and the `MPK` applications use objects in a non-trivial way, including inheritance and virtual dispatch. The results in Figure 4 correspond solely to the code size reduction due to removable of dead code. Object-oriented optimizations such as devirtualization and procedure-level optimizations such as inlining are not applied in this experiment, and would further improve the results reported here.

6.2.1 Code Size

Reachable members analysis is the primary mechanism used by the Virgil prototype compiler to detect and remove unused methods in a Virgil program. Figure 5 compares the code size reduction when RMA is applied to the benchmark programs. Each bar gives the code size results of the program when compiled to machine code for the AVR processor by `avr-gcc 3.3`. The first group of four bars for each application represents the size of the program without applying any Virgil-specific optimizations and using four different `avr-gcc` optimization levels. The second group of four bars for each application gives the size of the binary when the Virgil compiler applies RMA with the same `avr-gcc` optimization levels. To summarize the results, RMA reduces the code footprint of the benchmark programs by 38%, 29%, 45%, 79%, and 35%, respectively, for `avr-gcc` with no optimizations, and by 22%, 17%, 39%, 81%, and 28% when `avr-gcc` is run with `-Os`. The large reduction for the Decoder program is because the complex initialization routines that build the decoder tree are removed automatically by RMA.

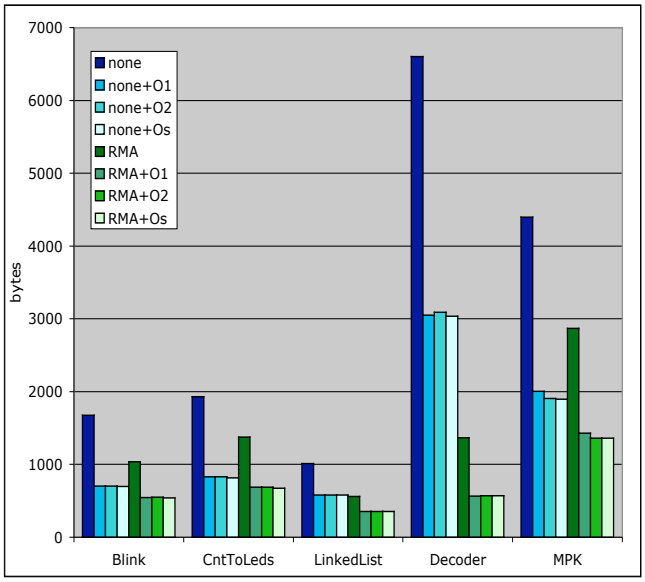


Figure 5: Comparative code size reduction for 8 combinations of Virgil and gcc optimizations.

6.2.2 RAM Size

Reachable members analysis is also used to reduce the size of data structures and metadata in the program by removing semantically unreachable objects, object fields, and component fields. Figure 6 gives the results for each of the benchmark programs under three different optimization scenarios; **base**: the base Virgil compiler configuration, with no optimizations; **RMA**: the compiler performs RMA and removes unused objects, meta-objects, and fields; and **RMA+RC**, where the Virgil compiler performs RMA and then compresses references with the compression table technique described in section 5. It is important to note that C compilers do not generally perform any data representation optimizations and thus the RAM consumption for all five benchmark programs is constant across `gcc` optimization levels.

Figure 6 illustrates the effectiveness of RMA and reference compression for all five benchmark programs. RMA reduces both RAM and ROM space required for the application, and reference compression trades some ROM space for RAM using compression tables. In the three largest applications, the two optimizations combined reduce RAM consumption by 75%, 40%, and 41%, respectively. All applications exhibit very small footprints for the meta-objects stored in ROM; RMA reduces the size of these meta-objects by 42%, 72%, and 40% for the three largest applications respectively. For reference compression, the tradeoff ratio, or the number of RAM bytes saved for each ROM byte spent in a compression table, is 1.05, 1.29, and 1.53.

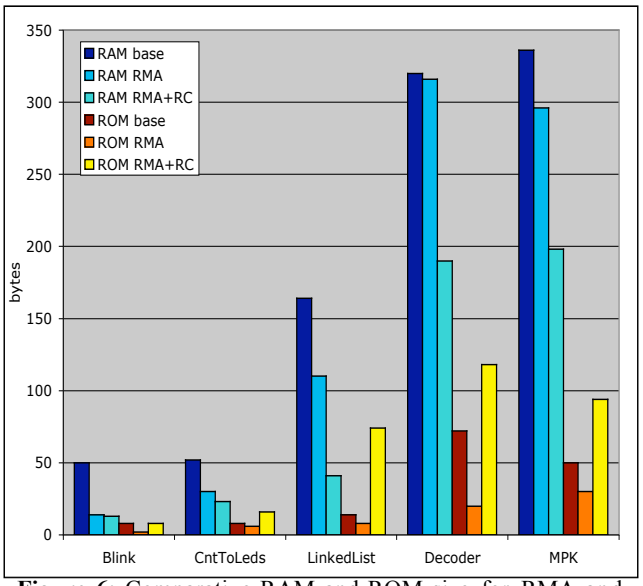
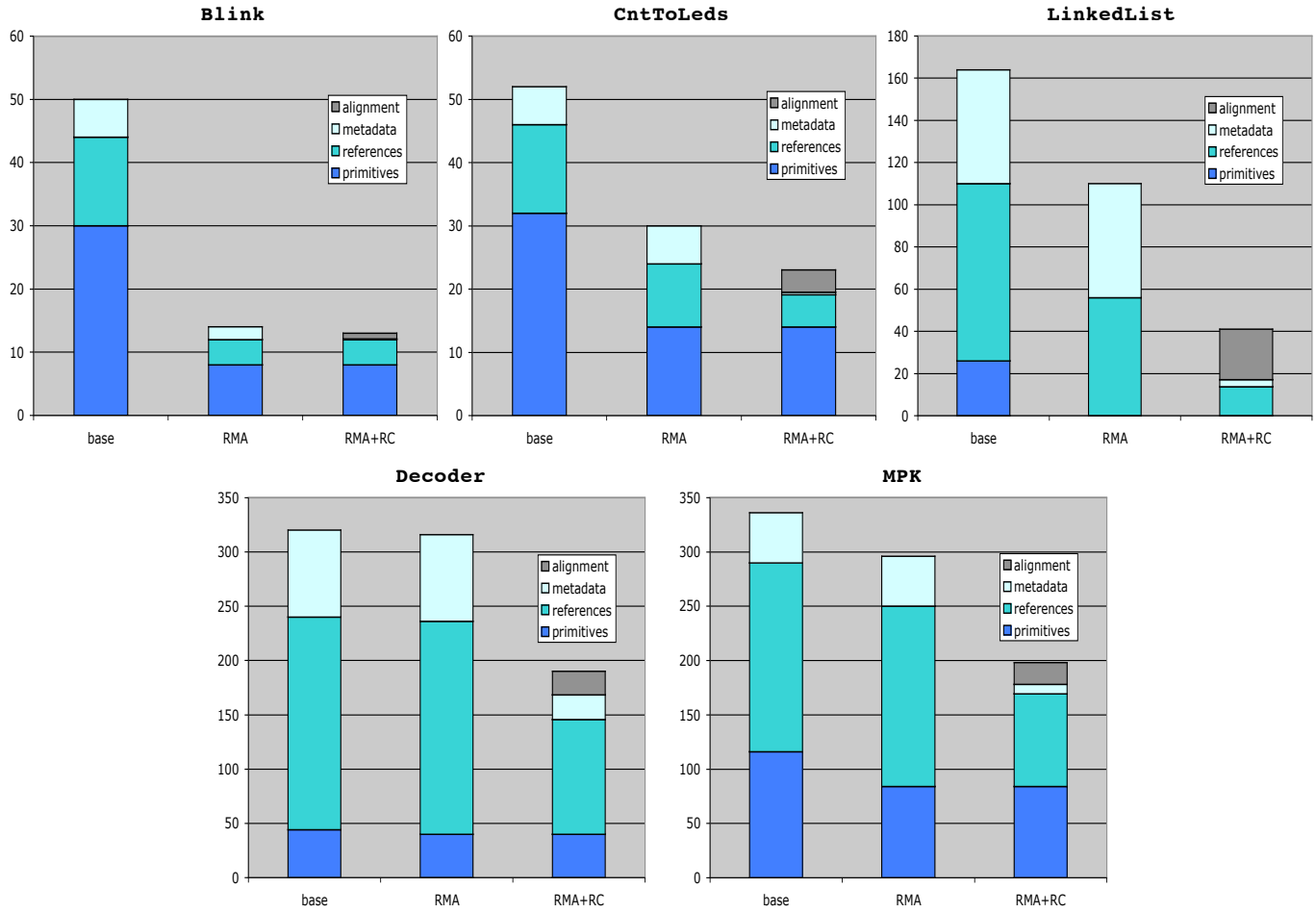


Figure 6: Comparative RAM and ROM size for RMA and RMA plus reference compression optimizations.

6.2.3 Detailed RAM Size Comparison

The charts on the next page examine the RAM reduction in more detail for each application. The RAM usage of each Virgil program as reported in these charts are: **primitives**, which are primitive fields like integers, booleans, etc; **references**, which are object references and delegates; **metadata**, which corresponds to object headers and array length fields; and **alignment**, which are the extra wasted bits of objects needed to align them on byte boundaries after compressing references to sub-byte quantities. Meta-objects are stored in ROM and appear only in the RAM/ROM reduction chart.



RMA may remove all categories of data, including primitives, reference fields, and metadata, because it can remove unused objects and fields. Reference compression is only applied to object headers and reference fields; it does not compress primitive values, except for booleans, which can be represented as a single bit.

The `LinkedList` results are somewhat surprising. Although the example program manipulates and traverses lists, it does not actually touch the data items. RMA aggressively reduces the doubly linked list, removing not only the backward pointers, but also removing the data items (primitives) from the links in the list because it detects they are unused. With reference compression, the link objects can be represented with just a few bits (as orphan classes they require no header and the forward link field can be compressed to a small number of bits). The alignment anomaly is due to the fact that each of these sub-byte sized objects occupies a whole byte in memory. Interestingly, this raises the question of how to deal with extremely small objects in future implementations.

Delegates are used extensively in the `Decoder` and `MPK` applications, but are not currently compressed by the prototype compiler; this represents an opportunity for further optimization in future work.

7. Related Work

This section organizes related work into three categories; object systems (which are broadly considered to include embedded virtual machines and object-oriented operating systems), object compression technologies, and embedded languages.

7.1 Embedded Object Systems

There has been extensive research on embedded virtual machines, particularly for the Java programming language. Most target embedded systems with at least 64KB of RAM. For example the KVM [2] virtual machine from Sun Microsystems targets devices with at least 192KB of memory. This is two orders of magnitude more memory than a microcontroller.

VM* [24] 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.

Ultimately, virtual machines are an attempt to address the challenges of embedded systems primarily for the application level because the virtual machine itself is generally either implemented in C or C++, and more recently, Java. Using Java to implement virtual machines and operating system kernels, e.g. JX [17] 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.

Virgil, however, is intended to address the challenges inherent in 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.

The Singularity Project [21] at Microsoft Research is an ambitious attempt to build a complete desktop operating system primarily in C#, eliminating as much unsafe code as possible. In developing a full-fledged operating system kernel with a safe language, there are several challenging problems that Virgil does not address. For example, Virgil programs, though low-level, don't generally have to manipulate MMU mappings (there is no MMU), program stacks (one stack exists for the main computation and interrupts), thread contexts (there is only a main thread and interrupts), and don't have to scan raw memory for memory-mapped IO devices or perform DMA (the IO space is segregated and devices registers have statically known locations). Larger systems also require dynamic memory allocation and some mix of manual memory management or garbage collection—runtime services that must be implemented in the language using some unsafe code with special magic tunnels under, around, or through the type system. However, the restrictions on the domain allow Virgil programs to be implemented completely in safe code and compiled directly to the bare hardware with no runtime system. Interesting future work remains to scale Virgil to this larger class of software systems.

Eventrons [29] are a programming construct introduced in the context of the IBM J9 virtual machine which 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.

At the core, Eventrons and their verification are similar to the Virgil concepts of initialization time and reachable members analysis. While Eventrons are a programming construct within the larger Java language, initialization time and runtime are Virgil language concepts and are an integral part of the programming and compilation model. Whereas Eventrons use the results from analysis for verification and to pin reachable objects, the Virgil compiler uses the analysis results to remove dead code and objects completely. Virgil also provides a richer programming environment because Virgil programs can modify reference fields in an unrestricted manner.

7.2 Object Compression

Chen et al [10] study dynamic heap compression techniques in an embedded Java VM setting, based on the KVM 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 authors describe a number of enhanced garbage collectors that dynamically compress and decompress objects in response to memory demands. Their system employs a simple and fast compression algorithm that operates on the raw memory values of objects, without considering the types. Virgil, however, is designed to run without a garbage collector or any runtime system, and in this context cannot use dynamic compression and decompression. The Virgil compiler uses the types of references in its reference compression scheme to reduce the size of objects, rather than considering raw memory values, which might require dynamic decompression.

Ananian and Rinard [4] 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. Of these, field reduction applies only to primitive fields and could be considered complementary to techniques described here; unread and constant field elimination is less general than RMA, which extends the technique to the complete live heap; static specialization can be detected with the results from RMA, externalization does not apply because it requires a dynamic hash table, and class pointer compression is less general than reference compression, which can compress object references of all types.

7.3 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 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 reference optimization for C++ could not be made sound. Despite promises of its adherents, C++ has not succeeded for microcontroller class systems.

NesC [16] is an extension to C that adds module capabilities and a simple task model. NesC hides some of the messiness that can normally appear in C code such as copious macros and mostly eliminates the need for header files, but inherits C's weak type system. NesC provides modules and interfaces that can be configured by "wiring" them together in a configuration language. 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 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 like Virgil's are not possible. NesC is also coupled closely with TinyOS; in fact, the nesC compiler assumes the availability of certain TinyOS-specific header files.

8. Conclusion

This paper tackles the problems of developing microcontroller software at the language and compiler level. By careful attention to detail and adherence to design constraints that are reasonable for this domain, the Virgil language brings most of the expressiveness of object-oriented languages to this most severely resource-constrained of domains, without sacrificing type safety. In fact, the opposite is true: Virgil's type safety enables a new class of optimizations and is key to achieving efficient implementation. Each of Virgil's features has been crafted carefully to provide better expressiveness while still requiring no language runtime and imposing only minor metadata overheads on the program. The type-safe nature of Virgil objects eliminates a large class of pernicious software bugs through strong static type safety and some dynamic checks, like Java.

This paper is the first 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.

This paper also introduces three data-sensitive optimizations that serve to further reduce the size of programs by removing unused members, optimizing dispatches, representing reference fields in a compact manner, and moving read-only portions of objects to the ROM without changing the programming model. In particular, reference optimization exploits the type-safe nature of object references to achieve heap compactions of up to 75%; object 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. This surprising result leaves us to ponder the suggestion that objects may in fact be better than pointers for embedded systems, since the strong types of references and restrict their referencible sets, thereby allowing compression techniques like those in this paper.

9. Future Work

Virgil is a remarkably simple but expressive language, despite the lack of dynamic memory allocation, but as both software and hardware systems grow in complexity and capabilities, dynamic allocation becomes more necessary. It can be approximated in Virgil with statically allocated and manually managed pools of objects, allowing recycling within a type, but as larger systems are developed, static allocation becomes less workable. I would like to explore augmenting the core runtime model with various forms of dynamic allocation, including explicit or implicit regions, stack allocation, and various garbage collection techniques.

Virgil's lack of a universal super-class combined with the lack of interfaces represents a tradeoff between efficiency and expressiveness. It allows the object model to be implemented simply and efficiently, requiring no metadata for orphan objects. However, it limits code reuse by making it more difficult to reuse collection classes. I believe the best solution for this problem is a parametric type system such as Java generics [8]. A solid parametric type system for Virgil should encompass primitive types without resorting to boxing as in Java 5, which requires dynamic memory allocation, or code duplication as in C++, which may result in code explosion. Virgil will ultimately benefit from ongoing research into the tradeoffs between implementation efficiency and expressiveness in parametric type systems for Java, C#, and other languages.

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. 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.

I believe the optimizations presented in this paper can be explained to embedded programmers without advanced knowledge of compiler techniques, inspiring more confidence in the efficiency of objects for small devices. However, there is certainly room to improve on these techniques with more sophisticated analyses. For example, a points-to analysis [30] with live objects would likely give more precise results than reachable members analysis and might allow objects of the same type to be laid out differently depending on which parts of the program manipulate them. There are a number of other reference compression and object layout techniques [4][10][14] that could be applied with various tradeoffs between size and performance; further study is required to determine what works the best in practice for this domain. Such tradeoffs are likely to be fruitful research topics in the future if the initialization time model becomes more widely employed.

10. Acknowledgments

Thanks to Jens Palsberg, Todd Millstein, and David Bacon for comments on a preliminary version of this paper. Thanks to Simon Han for contributing the MPK application, which is a proof-of-concept port of the SOS [19] message passing mechanism to the Virgil language. Thanks to Doug Lea for a deep and fruitful discussion about Virgil's inheritance model. Ben Titzer 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.

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

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

[5] 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.

[6] D. Bacon, S. Fink, and D. Grove. Space- and Time-efficient Implementation of the Java Object Model. In *the 16th*

European Conference on Object-Oriented Programming (ECOOP '02), University of Malaga, Spain, June 2002.

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

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

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

[16] 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 Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '03)*. San Diego, CA. June 2003.

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

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

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

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

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

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

[23] 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*. June 1997.

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

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

[26] J. Palsberg. Type-based Analysis and Applications. *In Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools (PASTE '01)*. Snowbird, UT. June 2001.

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

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

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

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

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

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

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

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

[35] 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.

[36] P. Tyma. Optimizing Transforms for Java and .NET Closed Systems. Phd Thesis, Syracuse University. 2004.

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

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

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