Vertical Object Layout and Compression for Fixed Heaps

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

Jens Palsberg
UCLA Compilers Group
4531K Boelter Hall
Los Angeles, CA 90095
palsberg@ucla.edu

ABSTRACT

Research into embedded sensor networks has placed increased focus on the problem of developing reliable and flexible software for microcontroller-class devices. Languages such as nesC [8] and Virgil [14] have brought higher-level programming idioms to this lowest layer of software, thereby adding expressiveness. Both languages are marked by the absence of dynamic memory allocation, which removes the need for a runtime system to manage memory. To provide data structures, nesC offers modules, and Virgil offers the application an opportunity to allocate and initialize objects during compilation. This paper explores techniques for compressing fixed object heaps with the goal of reducing the RAM footprint of a program. We explore table-based compression and introduce a novel form of object layout called *vertical object layout.* We provide experimental results that measure the impact on RAM size, code size, and execution time for a set of Virgil programs. Our results show that compressed vertical layout has better execution time and code size than table-based compression while achieving more than 20% heap reduction on 6 of 12 benchmark programs.

Categories and Subject Descriptors

E.2 [**Data Storage Representations**]: Object Representation. D.3.3 [**Programming Languages**] Language Constructs and Features.

General Terms

Management, Measurement, Performance, Languages.

Keywords

Microcontrollers, program data compression, pointer compression, reference compression, object layout, heap optimization, vertical object layout.

1. INTRODUCTION

Microcontrollers are tiny, low-power embedded processors deployed to control and monitor consumer products from microwaves, to fuel injection systems, to sensor networks. A typical microcontroller integrates a central processing unit, RAM, reprogrammable flash memory, and IO devices on a single chip. They have very limited computational power, and their main memory is often measured in hundreds of bytes to a few kilobytes. For example, a popular microcontroller unit from Atmel, the ATMega128, has 4KB of RAM and 128KB of flash to store code and read-only data; it serves as the central processing unit of the Mica2 sensor network node.

The primary resource limitation in many microcontroller applications is the RAM space available for storing the program heap and stack. This often precludes a dynamic memory management system, either a manual one such as C's `malloc()` and `free()`, or an automatic system such as a garbage collector. For that reason, many microcontroller applications are written to pre-allocate all of their needed data structures and do not use any form of dynamic memory allocation. In the C programming language, this is done by statically declaring arrays and structures as global variables. In nesC [8], modules contain state as fields and are instantiated by a module wiring system at compile time. In Virgil [14], applications allocate and initialize their heap within the compiler, which contains an interpreter for the complete language.

Reducing RAM consumption allows larger applications to be built and deployed on the same microcontroller model, while reducing the resource requirements of a single application allows a smaller, cheaper microcontroller to accomplish the same task. There are several general techniques to reduce RAM consumption. One is to simply remove unused data structures through compiler or manual analysis. Another is to reduce the average footprint of a program by employing virtual memory techniques that move infrequently used data to larger, slower storage such as disk. A third is to compress infrequently used data and dynamically decompress it as it is accessed. A fourth technique, employed here, is to statically compress program quantities so that dynamic decompression is unnecessary. We discuss previous techniques in more detail in the related work section.

This paper evaluates two offline heap compression techniques for programs written in the Virgil language. Both techniques exploit the availability of the entire program heap at compile time and are employed on top of our compiler that already performs sophisticated dead code and data elimination. The first compression technique, table-based reference compression, was described in our previous work [14] and is evaluated in more detail here. The second technique is a novel object layout model that we call *vertical object layout*. Vertical object layout represents objects in a more compact way by viewing the heap as a collection of field arrays that are indexed by object number, rather than the traditional approach of a collection of objects that are accessed via pointers. Object numbers can then be compressed without additional indirections. We present a simple object numbering system for identifiers that ensures that each field array is stored without wasting space, even in the

CASES'07, September 30–October 3, 2007, Salzburg, Austria.

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

The outline of this paper is as follows. Section 2 gives background for this paper by describing the key features of Virgil that are relevant to this work. Section 3 describes the basic idea of reference compression, summarizing our previous work in the area. Section 4 describes our new object model and its application to reference compression. Section 5 gives our experimental results. Section 6 describes related work, and Section 7 gives our conclusion and vision of future work.

2. VIRGIL BACKGROUND

In our previous work [14], we have developed the Virgil programming language and compiler system that targets microcontroller-class devices. Virgil is a lightweight object-oriented language that is closest to Java, C++ and C#. Like Java, Virgil provides single inheritance between classes, with all methods virtual by default, except those declared `private`. Objects are always passed by reference, never by value, as they can be in C++. However, like C++ and unlike Java, Virgil has no universal super-class akin to `java.lang.Object` from which all classes ultimately inherit. But Virgil differs from C++ in two important ways; it is strongly typed, which forces explicit downcasts of object references to be checked dynamically, and it does not provide pointer types such as `void*`. This inheritance model allows for object-oriented design in applications using familiar patterns and provides for a straightforward and efficient object model. In addition to its fields, each Virgil object has a single word header that stores a pointer to a meta-object containing a type identifier and a virtual dispatch table. This type information is unnecessary for Virgil classes that are unrelated to any other classes (so-called *orphan* classes) because complete type information is known statically. Such objects are represented without object headers and all virtual dispatches are resolved at compile time.

One of Virgil's key innovative features is the notion of *initialization time*, whereby an application can allocate and initialize its entire object heap during compilation. This allows an application to build all of its data structures during compilation; memory allocation during execution on the device is disallowed. This feature is motivated by the observation that many microcontroller programs already written in C and nesC statically allocate all of their memory via global variables and reuse this memory at execution time. Static allocation removes the need for a memory management library such as `malloc()` and `free()`, or in the case of a safe language, the need for a garbage collector or other automatic memory management system. The lack of dynamic memory allocation also removes the need for programs to be made robust against the possibility of exhausting memory, since memory is statically allocated and apportioned at compile time.

Virgil elevates the common practice of static memory allocation on microcontrollers to a first-class language feature. In addition to implementing the standard phases of compilation such as parsing, typechecking, optimization and code generation, the Virgil compiler includes an interpreter for the complete Virgil language. After typechecking is complete, the compiler executes the application's initialization code with the built-in interpreter. The interpreter allows the application code to perform any Turing-complete computation using the complete Virgil language, including, for example, allocating objects, initializing them, calling methods on them, etc. This phase allows unrestricted allocation; a general-purpose garbage collector ensures that unreachable objects allocated by the application are discarded. When the application's initialization routines have terminated, the compiler will generate an executable that includes not only the code of the program, but also the data of the entire reachable heap. This initial heap is immediately available to the program at runtime on the device. During execution the program can perform unrestricted computation and can read and write heap fields, but further memory allocation is disallowed and will result in an exception.

The initialization time model provides new opportunities for compiler optimizations, since the compiler has not only the entire code of the program, but the entire initialized heap as well. This allows the compiler to employ data-sensitive optimizations that can exploit the size and shape of the heap, as well as the initial values of fields in objects. One such optimization is a combined dead code and dead data elimination optimization called *reachable members analysis*. RMA can be thought of reachability over program code and data object simultaneously; it serves to remove dead code and objects from the program that cannot be reached over any program execution. For the purposes of this work, it can be considered complementary. In fact, all of our experimental results are obtained against a baseline of performing the RMA optimization first.

Initialization time also makes the optimizations described in this paper possible. We exploit the availability of the entire program heap at compile time to represent references specially and to layout objects in novel ways to achieve better results on small architectures.

3. REFERENCE COMPRESSION

Microcontrollers have severe limitations on the RAM space available to store heap objects and the program stack, which makes a space efficient representation of objects of primary importance. While the Virgil language was designed to have straightforward and low costs for the implementation of objects, as well as sophisticated dead code and dead data elimination passes, we would like to employ advanced object layout strategies and compression techniques to further improve on the basic implementation. Compression has two advantages: first, it allows larger applications to be built and deployed on the same hardware; and second, it allows a given application to be deployed on a smaller, cheaper microcontroller model with less RAM.

Our previous work on Virgil [14] explored table-based compression of object references for Virgil, but did not evaluate the impact on performance. This paper provides those experimental results and compares them with our new technique, compressed vertical object layout. We briefly describe the previous table-based compression technique in this section for completeness.

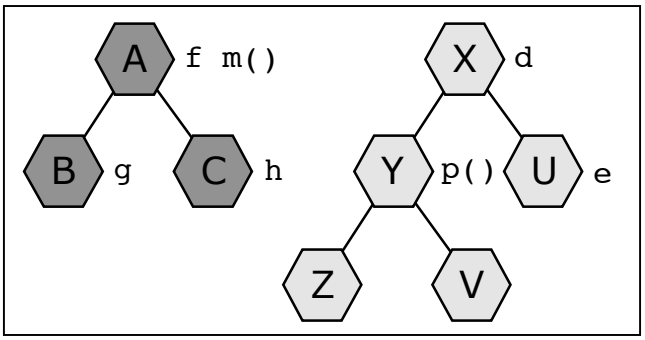

Figure 1: Example Virgil class hierarchy. Hexagons represent classes; their field and method declarations are to their right.

Reference compression relies on the strong type safety of Virgil. On microcontroller architectures with between 256 bytes and 64 kilobytes of RAM, pointers into the memory are typically represented with a 16-bit integer byte address. In a weakly typed language like C, a pointer is not constrained to point to values of any particular type and can conceivably hold any value. In fact, pointer arithmetic relies on the fact that pointers are represented as integers and allows operations such as increment, addition, subtraction, and conversion between types. Worse, C allows pointers to be converted to integers, manipulated, and converted back to pointers. In Virgil, the type of a reference restricts the set of possible objects that it may reference to only those objects that are of the corresponding type or one of its subtypes. Recall that after initialization time in the compiler, a Virgil application has allocated all of its objects that will exist over any execution. The compiler can use this fact to represent references specially.

The most straightforward way to implement reference compression is to use a compression table where each compressed reference is an object handle: an integer index into a table that contains the actual addresses of each object. Because Virgil has disjoint inheritance hierarchies, the compiler can compress each reference by creating a compression table for its associated root class, with one entry in the table for each object whose type is a subclass of that root. The number of bits needed to represent the integer index is therefore the logarithm of the table size. For example, if the table has 15 live objects plus `null`, we could use a 4-bit integer index, a savings of 75% over storing a 16-bit address. Because there is no garbage collector which may move objects at runtime, object addresses are fixed, which allows the compiler to store the table in ROM or flash, which is considerably larger than RAM, though usually slightly slower to access. Figure 1 introduces an example class hierarchy and Figure 2a gives the corresponding table-based compression implementation.

The table adds a level of indirection to all object operations. Reads and writes on object fields require first looking up the object's RAM address from the compression table before performing the operation as before. In some situations, the compiler may be able to avoid the cost of the indirection by using standard optimizations to cache the actual address of frequently accessed objects, e.g. within loops. Accesses to reference fields in the heap may also be slower if the fields are bit-packed in memory and require masks and shifts. (However, if fields are packed only at the byte level, accesses can be faster if the field requires only one byte of storage instead of two.) Thus table-based compression represents a classic space/time tradeoff: it consumes some ROM space for the tables and may reduce performance, but saves RAM.

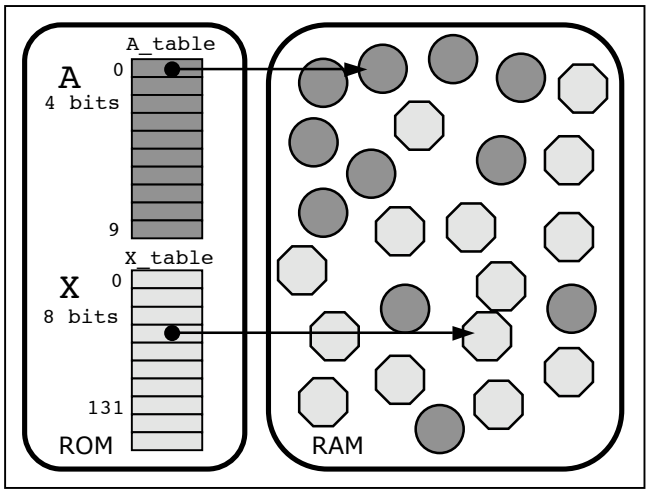

Figure 2a: Table-based reference compression for the example. Each root class (`A`, `X`) has its own compression table in ROM that stores the RAM addresses of the objects.

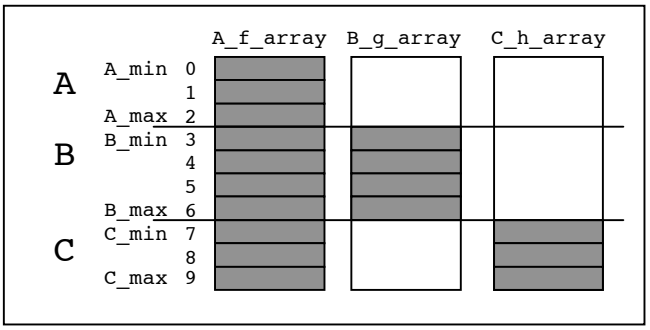

Figure 2b: Vertical object layout for the example (only classes `A`, `B`, `C` shown). Each field is represented by an array and only the occupied portion is stored. Preorder numbering gives objects of type `A` ids 0-2, `B` 3-6, and `C` 7-9.

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

4. VERTICAL OBJECT MODEL

In traditional high-performance object-oriented systems, each object is represented in memory as a contiguous region of words that contain the values for each of the object's fields. An object reference is represented as a single-word pointer to this contiguous memory region, and the different fields of a single object are located at fixed offsets from this base address. Advanced features such as mix-ins, multiple inheritance, etc may be implemented by indirection to further contiguous memory blocks. This layout strategy has the best performance in a scenario

where objects are created, moved, or reclaimed dynamically. An object allocation operation amounts to little more than an acquisition of a small contiguous region of memory, often simply bumping a top-of-heap pointer by a fixed amount. Field accesses in this model are implemented straightforwardly as a read or write of a memory address that is a small fixed offset from the object pointer; nearly all architectures allow this operation to be implemented with a single instruction. We will refer to this implementation strategy as the standard or *horizontal* layout, for reasons that will become obvious in this section.

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

Imagine the heap of the program after initialization has completed. The program has allocated some number of objects of various types, with each object having values for all of its declared and inherited fields. If we consider this set of objects to be a two-dimensional matrix, we can consider the storage for the fields' values to be entries in the matrix. Each object corresponds to a row in the matrix, and each declared field in the program corresponds to a column. If we represent objects in the standard layout, a reference corresponds to a pointer to a row of the matrix, where the elements of a single row are adjacent in memory. In a sense, the matrix is laid out *horizontally*. But one can also explore the implications of arranging this matrix in memory *vertically*, where an entire column has its elements adjacent in memory.

Consider again the example in Figure 1 and the corresponding vertical layout in Figure 2b. The classes `A`, `B` and `C` have declared fields `f`, `g`, and `h`, respectively. Suppose now that we collect all the objects in the initialized heap of these types and number them so that all the objects of exact type `A` are first, `B` second, and `C` third. Then if we put these objects into a table such that the columns are the fields `f`, `g`, and `h`, we can see that each column has a contiguous range of indices for which the field is valid corresponding to the indices of the class in which the field was declared. If we represent an object reference as an index from 0 to 9 (with -1 representing a `null` reference), and represent the field `f` as an array `A_f_array`, we can read and write the field by simply indexing into `A_f_array` by the object number.

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

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

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

Figure 3: algorithm to compute object indices by pre-order traversal of inheritance tree. For each class, `ClassInfo` stores a list of the child classes and an interval representing the valid indices for objects of this class and subclasses. For each object, `ObjectInfo` stores the object id (index) assigned to the object.

It is simple to generalize from the example. For any inheritance tree, we simply assign object identifiers using a pre-order tree traversal. Figure 3 gives the algorithm. The output of the algorithm is an interval of valid indices for each class and an object id for every object. By employing preorder traversal of the inheritance tree, the final assignment guarantees that each class has a contiguous range of indices corresponding to all objects of that type or one of its subtypes. Therefore the array that represents that field in the vertical object layout can be compact without wasting space. This algorithm chooses to restart the object id at zero for each root class in the hierarchy, which means that an object id is unique within its inheritance hierarchy, but not necessarily globally unique.

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

	Horizontal	**Horizontal Reference Compressed**	**Vertical**
`e instanceof A`	`e != null`	`e != -1`	`e != -1`
`e instanceof B`	`e != null && e->meta->id == B_metaid`	`e != -1 && A_table[e]->meta->id == B_metaid`	`e >= B_min && e <= B_max`
`e instanceof C`	`e != null && e->meta->id == C_metaid`	`e != -1 && A_table[e]->meta->id == C_metaid`	`e >= C_min`
`e.f`	`e->f`	`A_table[e]->f`	`A_f_array[e]`
`e.g`	`e->g`	`A_table[e]->g`	`B_g_array[e - B_min]`
`e.m()`	`e->meta->m(e)`	`A_table[e]->meta->m(e)`	`A_m[A_metaid[e]](e)`

Figure 4 shows object operations from the Figure 1 example and each model's corresponding implementation (in pseudo-C). Bold expressions are constants inlined into the code by the compiler. Underlined expressions represent tables stored in ROM.

in the traditional approach). The first access is to get the type information of the object by indexing into the type information array using the object index. The retrieved value is a meta-object id that is then used to index into the appropriate virtual method array, which stores a direct pointer to the code of the appropriate method.

This numbering technique also has another advantage in that the contiguousness of the object identifiers makes dynamic type tests extremely cheap, because the object identifier actually encodes all the type information needed for the cast. The algorithm assigns object identifiers so that every class has an interval of valid indices that correspond to all objects of that type. Thus, given a reference `R` that is represented by an object index and a cast to a class `C`, we can simply check that the index `R` is within the interval for the class `C`. This requires only two comparisons against two constants; no indirections and no memory loads are required. The range check automatically handles the case of a null reference, because `null` is represented with `-1`, which is outside of the range for any type.

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

Vertical layout may also save more memory by eliminating the need to pad fields in order to align their addresses on word boundaries. In the horizontal object layout, compilers sometimes need to add padding between fields in the same object in order to align individual fields on word boundaries. This becomes unnecessary in vertical object layout; as long as each field array is aligned at the appropriate boundary for its type, each element in the array will be aligned by the simple virtue of being of uniform size. However, memory alignment is not generally an issue on 8-bit microcontrollers.

5. EXPERIMENTAL RESULTS

In this section we evaluate the impact that reference compression and the vertical object model have on three program factors: code size, heap size, and execution time. Each of our benchmark programs is written entirely in Virgil and does not rely on external device drivers or libraries, but instead the device drivers necessary to run on the hardware are themselves implemented in Virgil and are included in these results. These applications target the Mica2 sensor network node that contains an ATMega128 AVR microcontroller (4KB of RAM, 128KB of flash). Our Virgil compiler emits C code that is compiled to AVR machine code by `avr-gcc` version 4.0.3. Code size and data size numbers correspond to the size of the `.text` and `.data` sections in the ELF executables emitted by `avr-gcc`, and thus correspond to the exact usage by the program when loaded onto the device. Precise performance numbers are obtained by using the program instrumentation capabilities [15] of the Avrora cycle-accurate AVR emulator.

	Data	Code	Time
BinaryTree	703	432	3613978
PolyTree	602	436	3460648
BubbleSort	874	3878	8419862
Decoder	374	980	4119442
Fannkuch	406	5612	951068
MsgKernel	352	3262	2314365
TestRadio	188	4706	14862672
TestSPI	98	2484	26912
TestUSART	83	2564	1153916
TestADC	50	992	53849859
LinkedList	115	664	1973094
Blink	18	778	8038

Figure 5: Raw data for the standard horizontal model. Code and data sizes are in bytes, and execution time is given in clock cycles (active cycles over 20 seconds for non-terminating programs like `Blink`). All other results are normalized to these.

We use 12 Virgil benchmark programs that are available as part of the driver kit we developed for the AVR microcontroller. `Blink` is a simple test of the timer driver, toggling the green LED twice per second; `LinkedList` is a simple program that creates and manipulates linked lists; `TestADC` repeatedly samples the analog to digital converter device; `TestUSART` transmits and receives data from the serial port; `TestSPI` stresses the serial peripheral interface driver; `TestRadio` initializes the CC1000 radio and sends some pre-computed packets; `MsgKernel` is an SOS excerpt that sends messages between modules; `Fannkuch` is adapted from the Programming Language Shootout Benchmarks and permutes arrays; `Decoder` is a bit pattern recognizer that uses a data structure similar to a BDD; `Bubblesort` sorts arrays; `PolyTree` is a binary tree implementation that uses parametric types; `and BinaryTree` is the same tree implementation but uses boxed values.

We tested five configurations of the Virgil compiler (version `vpc-b03-013`) including the standard horizontal object layout; the four new configurations are normalized against the results of the standard layout to show relative increase and decrease in code size, data size, and execution time. The three main configurations are: `hlrc`, which is the standard horizontal layout with table-based compression; `vl`, which is the vertical object layout without compression; and `vlrc`, which is the vertical layout with compression applied to object indices. The last configuration, `hlrcram`, is only shown for code size and execution time comparison; it corresponds to horizontal layout with reference compression, but instead of storing the compression tables in ROM, they are stored in RAM. This of course does not save RAM overall, but allows us to compare the cost of accessing ROM versus accessing RAM.

Figure 6a shows a comparison of the relative data sizes for our benchmark programs for the three main configurations, normalized against the base configuration of horizontal layout with no reference compression. First, we notice that vertical layout (`vl`) often saves some memory over the base configuration. This is because it does not require type identifiers in the meta objects; object numbers have been assigned so that they encode the type information. Also, the horizontal layout sometimes produces zero-length objects; `avr-gcc` allocates a single byte of memory to such objects. The second observation is that the compressed vertical layout typically performs as well as the compressed horizontal layout, although some of this is due to the empty object anomaly and the lack of type identifiers in meta objects. As expected, compressed vertical layout (`vlrc`) is uniformly better than vertical layout (`vl`) alone.

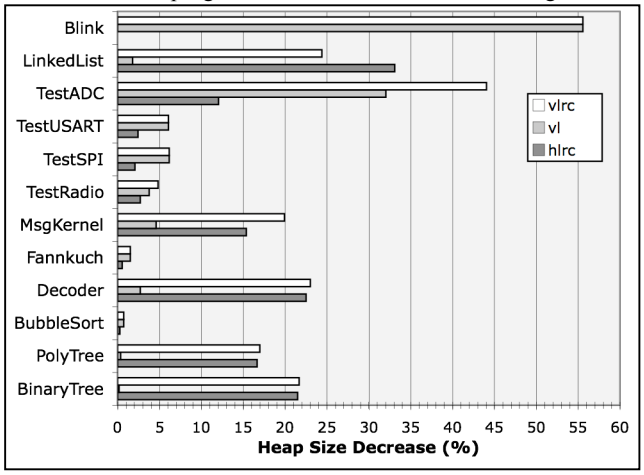

Figure 6a shows the heap size decrease for three object models normalized against the heap size for the standard horizontal layout. (larger is better)

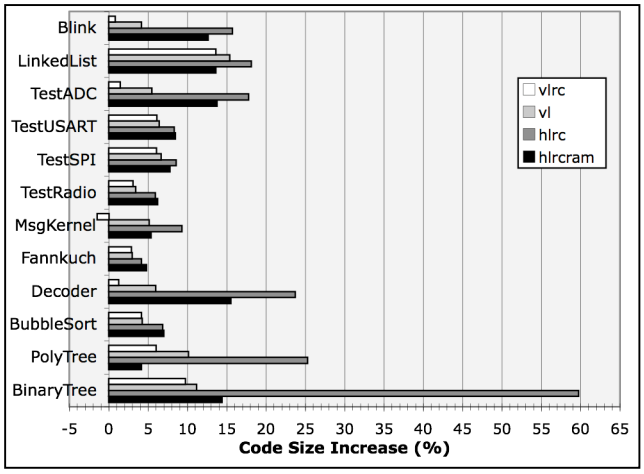

Figure 6b shows the code size increase for four object models normalized against the code size of the standard horizontal layout. (smaller is better)

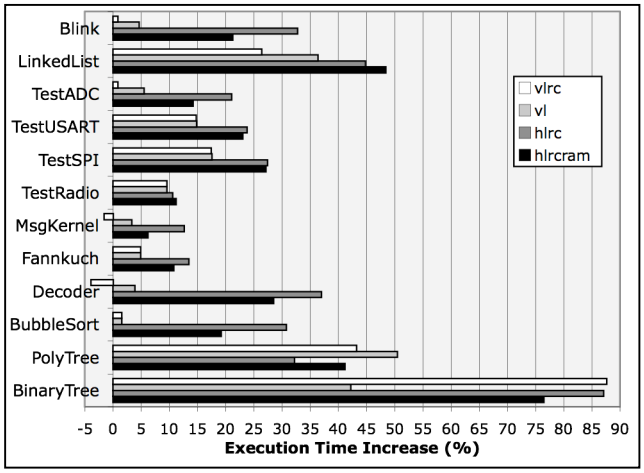

Figure 6c shows the execution time increase for four object models normalized against the execution time of the standard horizontal layout. (smaller is better)

Figure 6b shows the relative increase in code size for the same benchmarks with an added configuration, with all configurations normalized against the base configuration of horizontal layout without reference compression. Here, we can see that all configurations increase the code size of all programs (with the sole exception of `vlrc` on `MsgKernel`), with both `vl` and `vlrc` performing better than `hlrc` in each case. The increase for `vlrc` is less than 10% for most programs and less than 15% for all programs. Here, adding compression to the vertical layout actually reduces code size. This is because all field arrays become smaller, down to a single byte (because the Virgil compiler does not pack field arrays at the bit level), therefore the code to access them becomes smaller.

Horizontal reference compression increases the code size in two ways. First, it introduces compression tables that are stored in the read-only code space. Second, it requires extra instructions for each object operation due to the extra indirection. When the compression tables are stored in ROM, (the `hlrc` configuration), the Virgil compiler must emit short inline AVR assembly sequences because C does not expose the ROM address space at the source level. These assembly instructions are essentially unoptimizable by `avr-gcc`. To better isolate this effect, this figure includes code size results for a new configuration, `hlrcram` (or horizontal layout with reference compression tables in RAM). This configuration of course does not save RAM overall, but allows us to explore the effect of the special ROM assembly sequences on the code size in comparison to accessing the RAM. Comparing the `hlrc` configuration against the `hlrcram` shows that most of the code size increase is due to these special inlined ROM access sequences. The difference could be reduced if either `avr-gcc` understood and optimized accesses to ROM, or if the AVR architecture offered better addressing modes to access the ROM with fewer instructions. It is important to note that the largest proportional code size increases are for the smallest programs, as can be seen from the raw data in Figure 5.

Figure 6c gives the relative increase in execution time obtained by executing each benchmark in the Avrora [16] instruction-level simulator. The vertical layout technique performs better than horizontal compression in all but one case, and the execution time overhead for the compressed vertical layout is less than 20% in 9 of the 12 benchmarks, less than 10% in 7, and actually performs better by than the baseline by a small amount in two cases. These two programs perform a large number of dynamic type tests, which are cheaper in the vertical layout. This figure also includes results for the `hlrcram` configuration from Figure 6b in order to isolate how much of the execution time overhead is due to the cost of a ROM access versus a RAM access. In most cases, the execution time of `hlrcram` is noticeably better than that of `hlrc`, which means that a

significant fraction of the overhead is due to this ROM access cost. Also notice that that the largest proportional execution time

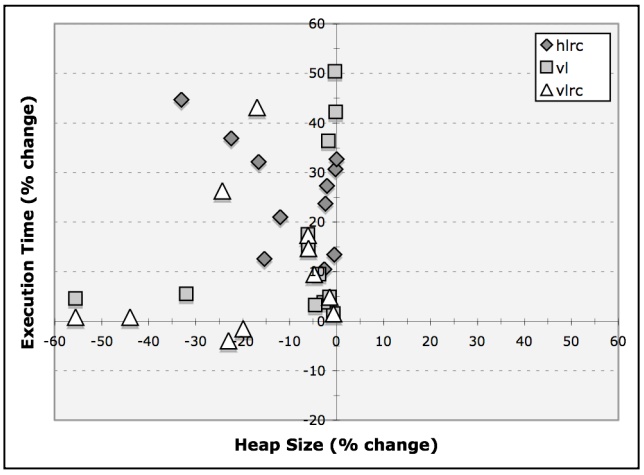

Figure 7a compares heap size change versus execution time change for the three object models, normalized against the standard horizontal object model.

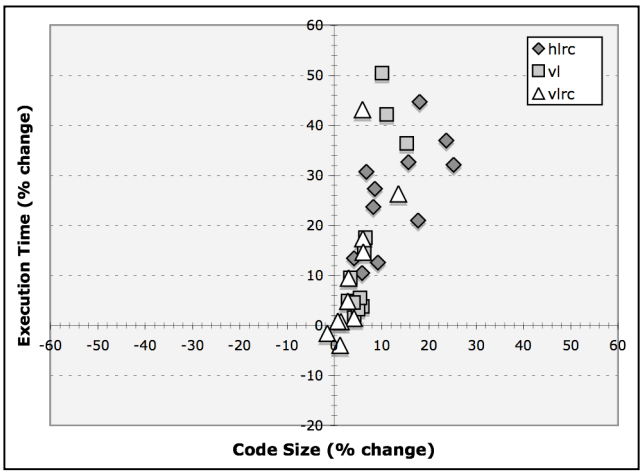

Figure 7b compares code size change versus execution time change for three object models, normalized against the standard horizontal object model.

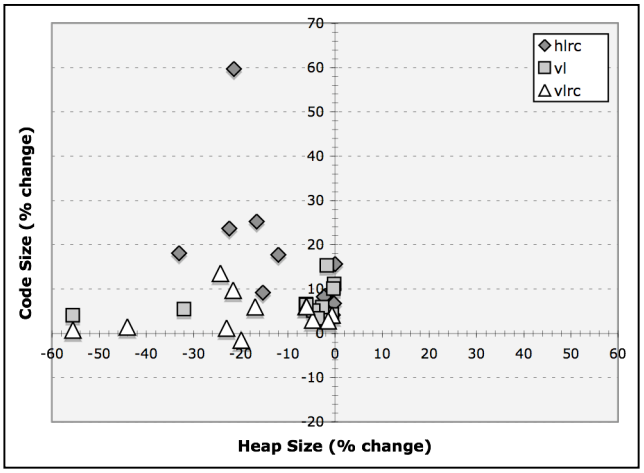

Figure 7c compares heap size change versus code size change for three object models, normalized against the standard horizontal object model.

increases tend to be for the smaller, pointer-intensive programs like `BinaryTree`, `PolyTree`, `LinkedList`, and `Decoder`.

Figure 7a combines the data from figures 6a and 6c, showing the tradeoff between increase in execution time and the savings in heap size for the three main configurations. First, we can see that the vertical layout without reference compression (`vl`) usually increases execution time without saving much heap space, while adding reference compression to vertical layout (`vlrc`) increases heap savings and usually has better execution time than vertical layout alone. Also, `hlrc` compression tends to have a larger increase in execution time with some savings in heap size, but not as much as `vlrc`. Overall, there is significant variation across the benchmarks, suggesting that the two factors are not intrinsically correlated. Instead, it is more likely that the factors are correlated to benchmark characteristics, therefore the compiler should take these characteristics into account and avoid reference compression when it will save little heap space.

Figure 7b combines data from figures 6b and 6c to show the correlation between increase in code size and increase in execution time for the three main configurations. First, we can see that the two factors appear closely correlated because the points cluster near a line from the origin into the upper right quadrant. This is most likely due to the simplicity of the AVR instruction set architecture and lack of an instruction cache; adding more instructions has a predictable effect on the execution time. Second, we can see that `vlrc` performs significantly better than the other configurations, with most of its points clustered near the origin. Third, we can see that `hlrc` performs the worst, with the largest increases in code size and execution time.

Figures 7c combines the data from figures 6a and 6b, comparing relative increase in code size versus decrease in heap size. Here we can see for a given heap size reduction (horizontal axis), `vlrc` tends to produce smaller code than `hlrc`.

6. RELATED WORK

The Virgil notion of initialization time enables the compiler to have the entire heap available before generating code. In the traditional case where the compiler does not know all run-time objects, researchers have developed static analysis techniques that estimate a range of values for each object field. If such a range of values is small, then the compiler can optimize the heap usage by compressing fields using various strategies. For example, Ananian and Rinard [2] use static analysis of Java programs to eliminate fields with constant values and reduce the sizes of fields that can assume a small number of values. Cooprider and Regehr [7] use static analysis of C programs to pack scalars, pointers, structures, and arrays using a compression-table scheme. Lattner and Adve [9] use static analysis to convert and compress 64-bit pointers to 32 bits. Unlike these previous approaches, our compression techniques do not require computationally intensive program analysis but instead exploit the type safety of Virgil. Compilation time for all our benchmarks is less than two seconds, of which the compression time is not measurable, compared to [7] which reports analysis times measured in minutes.

While traditional static compilers do not have the complete heap, the run-time system can track all objects that have been created and use the information to dynamically compress pointers. Some research systems exist that employ dynamic techniques, sometimes assisted by hardware. For example, Zhang and Gupta [19] use special hardware instructions to help compress pointers and integers on the fly; they use profiling information to guide

what data should be compressed and when compression should be done. Chen et al. [6] use a garbage collector that compresses objects when a compacting garbage collector is not sufficient for creating space for the current allocation request; this may require dynamic decompression of objects upon their next use. Wright, Seidl, and Wolczko [18] present a memory architecture with hardware support for mapping object identifiers to physical addresses, thereby enabling new techniques for parallel and concurrent garbage collection; such an architecture could support compression of pointers as well. Wilson [17] supports large address spaces with modest word sizes by using pointer swizzling at page fault time to translate large pointers into fewer bits.

Optimization of heap usage can sometimes help performance as well. For example, Mogul et al. [10] observed in 1995 that pointer sizes could affect performance significantly on a 64-bit computer because larger pointers occupy more space, putting greater stress on the memory system, affecting cache hit ratios and paging frequency. Adl-Tabatabai et al. [1] represent 64-bit pointers as 32-bit unsigned offsets from a known base resulting in a significant performance improvement.

For object-oriented languages such as Java, each object has a header that contains such data as type information, a hash code, and a lock. Bacon, Fink, and Grove [4] presented compression techniques that allow most Java objects to have a single-word object header.

Languages such as Java and Virgil allow single inheritance of classes. For languages such as C++, List Flavors, and Theta that allow multiple inheritance among classes, researchers have developed object layouts that enable fast field access with few indirections. For example, Pugh and Weddell [12] and later Myers [11] use both positive and negative offsets of fields. It remains to be seen whether vertical object layout can be useful for languages with multiple inheritance of classes and more complex object layout models. For example, it may be possible to apply ideas from PQ-Encoding in [20].

Like most languages in common use, Virgil uses primitive types of data such as integers. Bacon [3] presented Kava, a language without primitive types in which all data is programmed in an object-oriented manner. An interesting future direction might be to explore whether our techniques can be useful for a Kava-like language with a Virgil notion of initialization time.

7. CONCLUSION AND FUTURE WORK

In this paper, we evaluated static heap compression strategies that are made possible by the compilation model of Virgil—specifically, the availability of the entire program heap at compile time. Our experimental results show that programs compressed using vertical object layout have better execution time and code size than the compression-table approach while achieving nearly the same RAM savings. For six of the 12 benchmark programs, vertical layout with reference compression reduces heap size by more than 20%, while no program suffers more than 15% code size increase.

The lack of dynamic memory allocation is also common in hard real-time systems and high-integrity systems. For example, SPARK [5], an industry-standard subset of Ada, disallows dynamic memory allocation in order to simplify software verification. Recently, Taha Ellner, and Xi [13] described a functional meta-language for generating heap-bounded programs using a staged computation model and sophisticated types. The techniques described here for compression could have applicability to software written for both of these systems due to the fixed size of the heap.

Currently, the vertical object layout model requires that no new objects be created at runtime. Object allocation may require growing the field tables individually, and maintaining the contiguous nature of object identifiers might be tricky, especially in the presence of subtyping. Also, as objects become unreachable, entries in the field tables become unused and would need to be recycled. It is not clear whether the costs of such maintenance would outweigh the benefits. One might instead consider a hybrid strategy that "verticalizes" those types that are allocated only at initialization time and not at runtime. Another technique might to be to hybridize both horizontal and vertical layouts for the same type in interesting ways—perhaps only part of an object is stored horizontally, and the rest of the object is stored vertically, with the index stored in the horizontal layout for access. When the class hierarchy is fixed and known statically, it is possible to layout the meta-objects (i.e. dispatch tables) vertically, even though new objects can be created at runtime. This allows the object header to be compressed to a small meta-object identifier; a virtual dispatch is then implemented as an index operation into the appropriate virtual method array.

Our compiler detects read-only component fields and object fields and inlines the values of those that are constant over all objects, but currently it does not move other read-only object fields to ROM. This would be complex in the horizontal layout model because an object might be split into a read-only portion stored in ROM and a read-write portion stored in RAM. An uncompressed horizontal object reference must point to the address of one half of the object, and that half must have a pointer to the other half. However, when compression is applied to the horizontal layout, the compiler can use one object index but instead have two compression tables, one that holds the address of the RAM portion of the object, and one that holds the ROM address of the object. Even more promising is the idea of using vertical object layout to radically simplify moving individual fields to ROM. Because an entire field is stored contiguously and object indexes are used instead, moving a field array to ROM is trivial; the compiler can generate code to access the appropriate memory space at each field usage site. However, none of these strategies is currently implemented in the Virgil compiler.

In this work, our compiler employs a single object model for all inheritance hierarchies in the program, but one could consider a compiler that employs different object models for different hierarchies depending on the relative execution frequency of object operations and space consumption. For example, the compiler might elect to compress the most infrequently used objects using the most space efficient strategy, while employing the best performing (but possibly larger) strategy for the most frequently accessed objects. Such a compiler might employ heuristics on access frequencies or use feedback from profiling runs. We would like to extend our compiler and explore these more sophisticated strategies.

7.1 Acknowledgments

Thanks to undergraduates Ryan Hall and Akop Palyan for developing many of the AVR hardware drivers in Virgil. The authors were 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.

8. REFERENCES

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

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

[3] 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.

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

[5] R. Chapman, J. Barnes, and B. Dobbing. On the Principled Design of Object-Oriented Programming Languages for High-Integrity Systems. In the 2nd NASA/FAA Object-Oriented Technology Workshop. 2003.

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

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

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

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

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

[11] A. Myers. Bidirectional Object Layout for Separate Compilation. In *OOPSLA'95, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Austin, TX. October 1995.

[12] W. Pugh and G. Weddell: Two-directional record layout for multiple inheritance. In PLDI'90, *ACM SIGPLAN Conference on Programming Language Design and Implementation*. White Plains, NY. June 1990.

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

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

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

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

[17] P. Wilson. Operating system support for small objects. In Object Orientation in Operating Systems, pp.80-86, 1991.

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

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

[20] Y. Zibin and J. Gil. Efficient Subtyping tests with PQ-Encoding. In *OOPSLA '01, ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Tampa, FL. October 2001.