SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2014)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2300

Exploiting array manipulation habits to optimize garbage collection and type flow analysis

Dominique Colnet[1,*,†] and Benoît Sonntag[2]

[1] *Université de Lorraine, LORIA, Villers-Lès-Nancy, F-54600, France*
[2] *Université de Strasbourg, LSIIT, Illkirch, F-67412, France*

SUMMARY

A widespread practice to implement a flexible array is to consider the storage area into two parts: the used area, which is already available for read/write operations, and the supply area, which is used in case of enlargement of the array. The main purpose of the supply area is to avoid as much as possible the reallocation of the whole storage area in case of enlargement. As the supply area is not used by the application, the main idea of the paper is to convey the information to the garbage collector, making it possible to avoid completely the marking of the supply area. We also present a simple method to analyze the types of objects, which are stored in an array as well as the possible presence of NULL values within the array. This allows us to better specialize the work of the garbage collector when marking the used area, and also, by transitivity, to improve overall results for type analysis of all expressions of the source code. After introducing several abstract data types, which represent the main arrays concerned by our technique (i.e., zero or variable indexing, circular arrays and hash maps), we measure its impact during the bootstrap of two compilers whose libraries are equipped with these abstract data types. We then measure, on various software products we have not written, the frequency of certain habits of manipulation of arrays, to assess the validity of our approach. Copyright © 2014 John Wiley & Sons, Ltd.

Received 14 September 2013; Revised 23 October 2014; Accepted 29 October 2014

KEY WORDS: content of arrays; dynamic type; garbage collector; null pointer detection; type analysis

1. INTRODUCTION

A well-known technique for implementing a flexible array is to organize the storage area into two parts: the memory area currently being used and the reserve memory area. The size of the used area gives the current size of the flexible array, and if the program behaves normally, only the used area is accessed for read/write. When the array is to be extended, extra space is taken first in the supply area. If the supply area is exhausted, the whole storage area must then be reallocated in order to provide for a large enough new storage area, which most often involves copying out the used area. To limit reallocations and copies as much as possible, one has to find the right balance to resize the supply area. As the supply area is not visible by the user program, zeroing of that area is not required. In case of decrease in the size of the flexible array, it is enough to change the boundary mark between the two areas, leaving all values in the storage unchanged. Thus, all along the execution, the supply area may grow or shrink, containing whatever values.

The main idea of this paper is to adapt the garbage collector (GC [1]) so that supply areas, which correspond to inaccessible objects of the application, be completely ignored during the marking phase. Our contributions are summarized in the following text:

*Correspondence to: Dominique Colnet, LORIA, Campus Scientifique, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France.
†E-mail: Dominique.Colnet@loria.fr

- Presentation of the abstract data type for flexible arrays indexed from zero. Type analysis for the content of arrays with detection of the NULL value. Tuning of the GC to ignore the supply area and to use type information.
- Adaptation of our technique for other kinds of arrays. Abstract data types for user-defined indexing, circular arrays and hash maps.
- Evaluation of the garbage collection savings during the bootstrap of the SmartEiffel compiler, a large program using the previously defined abstract data types.
- Measurements of the impact of the type analysis carried out for arrays during the bootstrap of the Lisaac compiler. Also a large benchmark.
- Detailed experimental evaluation of the arrays habits using a suite of programs including real-life programs written in C or Java.

The results presented here come from the work performed in the SmartEiffel project [2, 3] and also in the Lisaac project [4, 5]. SmartEiffel consists of a compiler for the Eiffel language as well as a class library including all the kinds of arrays described in this article. The new optimization for marking arrays we present here is to be added to the previously published results in [6]. Lisaac is also a compiler with a large library, for a prototype-based language. The arrays described in this article are also part of the Lisaac's library. Both compilers are completely bootstrapped, written with the language they translate, and both are using the arrays presented here. The compilation strategy of Lisaac is more advanced than the one of SmartEiffel, particularly with regard to type analysis (see [5] for details).

The rest of the paper is organized as follows. In Section 2, we present, using the example of a simple flexible array, how it is possible to draw part of the filling up strategy to optimize both the garbage collection and type analysis. The foundations being laid, we present in Section 3 how it is possible to implement similar strategies for other kinds of more complex arrays. To show the gain obtained, we present in Section 4, the measurements we made during the bootstrap of two compilers, SmartEiffel and Lisaac, as well as the measurements for some other large applications, written with other languages/libraries. We then discuss in Section 5 the importance of the choices made by the languages designers to ensure the initialization of memory areas, and we also present work in connection with this article. Finally, Section 6 concludes this article.

2. SETTING UP WITH SIMPLE FLEXIBLE ARRAYS

2.1. *Abstract data type for flexible arrays*

Before taking into account other forms of arrays to start, we now consider the simple case of a flexible array indexed from zero. The array can dynamically expand or shrink its size during execution. The leftmost element is always accessible with index 0. Figure 1 shows how to manage the array using three variables. The `storage` variable allows access to the elements storage area, and the `capacity` variable stores the allocation size of this area. The `size` variable is the current size of the array as it is seen by the user. This variable also determines the boundary between the used area and the supply area. The following four operations completely define the abstract data type that represents the flexible array.

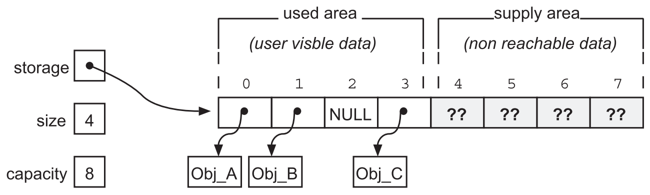

Figure 1. Handling a flexible ARRAY with three variables: `storage`, `size` and `capacity`. The array is filled up from left to right, cell after cell, in order to avoid uninitialized values.

Create(cap) The creation operation to be used in order to prepare a new empty array with a given `capacity` `cap`:

```
assert (cap >= 0);
capacity = cap; size = 0; storage = malloc(cap);
```

Extend(obj) To extend by one the array on its right, writing `obj` in the new slot. In case of reallocation (i.e. when the supply area is empty), the `capacity` is increased twofold:

```
if ( size >=  capacity ) {
   capacity = capacity * 2;
   storage = realloc(storage, capacity);
}
storage[size] = obj;
size = size + 1;
```

Read(ind) This function returns the object stored at index `ind` assuming that the index is correct, that is, a valid index in the used area:

```
assert ((0 <= ind) && (ind < size));
return storage[ind];
```

Write(ind, obj) Change the value at index `ind` using `obj` for the replacement, assuming that the `ind` is a valid index in the used area:

```
assert ((0 <= ind) && (ind < size));
storage[ind] = obj;
```

The four operations that we have given (`Create`, `Extend`, `Read` and `Write`) completely define the abstract data type for flexible arrays. We will see later how the GC can be made to ignore the supply area altogether. Before that, note that these operations force voluntarily the filling up from left to right by requiring the user to extend the array element by element. We made this decision to facilitate the type analysis of the content of arrays and, in particular, to facilitate the detection of NULL.

2.2. *Type analysis for the content of arrays*

Type analysis in object-oriented languages allows a better implementation of dynamic dispatch or even, under certain circumstances, the replacement of some late binding call sites with direct static calls. Prediction of the NULL values makes it possible to detect statically that some calls will fail at runtime. Many research papers have been published regarding type analysis or NULL prediction: [2, 5, 7–15]. Previous research papers address type analysis for the `self` or `this` variable, instance variables, formal parameters, local and global variables. When looking for type analysis inside arrays, there is, as far as we know, nothing published yet.

In the Lisaac compiler, we perform a *type flow analysis* that one must not mix up with *data flow analysis*. Data flow analysis consists in taking into account the order of statements to gather a precise piece of information, sometimes even perfect, that is, allowing to know exactly the reference of the real object or the real value of certain expressions. Often, data flow analysis makes it possible to improve type flow information, nevertheless, such an accurate type of information is not required to implement our type analysis for the content of arrays. What we call type flow analysis is the computation, for a given expression, of its set of possible dynamic types, *not taking into account the order of statements*. For instance, using all the reachable assignments in a variable, we compute all the possible types for that variable. The analysis is flow insensitive, that is, the order of reachable assignments does not matter. We consider on the whole the set of reachable assignments.

Concerning instance variables, we do not discriminate between the different instances from the same class (Figure 2); the type information is identical for all instances of a given class. For a formal method argument, we use the set of reachable effective arguments, that is, all reachable calls. Classically, to obtain the set of dynamic types of a given method call, we consider all the methods

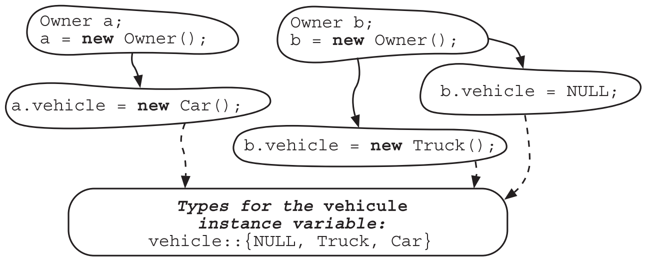
Owner a;
a = **new** Owner();
Owner b;
b = **new** Owner();
a.vehicle = **new** Car();
b.vehicle = NULL;
b.vehicle = **new** Truck();
Types for the vehicule instance variable:
vehicle::{NULL, Truck, Car}

Figure 2. Type flow analysis for instance variable `vehicle` from class OWNER. Order of statements is not relevant, and the result, which is {NULL, TRUCK, CAR}, is for all instances of the class OWNER.

that could be dynamically linked with the receiver type set, that is, a kind of dynamic dispatch simulation. To complete the coverage of all kinds of expressions, we propose in the following a simple technique to propagate type flow analysis for array read/write operations.

For the type analysis, all cells of the array are considered as a whole, regardless of index variation. We are using a unique type set for all elements of the array. As an example, if a TRUCK is written at index 1, we consider that all read accesses, whatever the index used, may yield a TRUCK. If another write sequence of a CAR exists somewhere in the reachable code, all cells are considered as potential holders of objects of type TRUCK or CAR. More generally, assuming that S_{array} is the type set of the array, as soon as one *index* is possibly assigned with an expression having S_{expr} as type set, S_{array} is changed as $S_{array} \cup S_{expr}$. This is a direct application of the *array smashing* technique of [16] using dynamic types instead of scalar values.

In the rest of the paper, the NULL value, which represents the lack of reference is simply considered as a possible type, and its occurrence inside the type set indicates that the corresponding expression can have the NULL value. For instance, in Figure 2, the set of possible types of the `vehicle` instance variable includes NULL. In the case of flexible array, the choice to start by creating an empty array, which extends cell by cell, facilitates the detection of the NULL value. Indeed, starting from an empty array, if no write operation (i.e. `Extend` or `Write`) introduces NULL, we statically know that this array never contains NULL at runtime. With this knowledge, it is possible to customize the marking procedure in order to focus on the sole used area, completely ignoring the supply area. Furthermore, inside the used area, it is possible to specialize the marking code according to type analysis results. Obviously, this is also true for all method calls applied to objects read from the array: `Read(ind).methodCall()`.

For the compiler implementation, storing the type flow information for an array can be performed in two ways. The first keeps all the possible types for an array by making successive unions with the set of types that an expression inserted in the array can take. This is a straightforward application of the previous definition. However, if the set of types of the expression inserted in the array is incomplete or not known, which can happen at the beginning of the compilation, then there is another way to proceed. This second solution involves a list of expressions. Every expression ever written to the

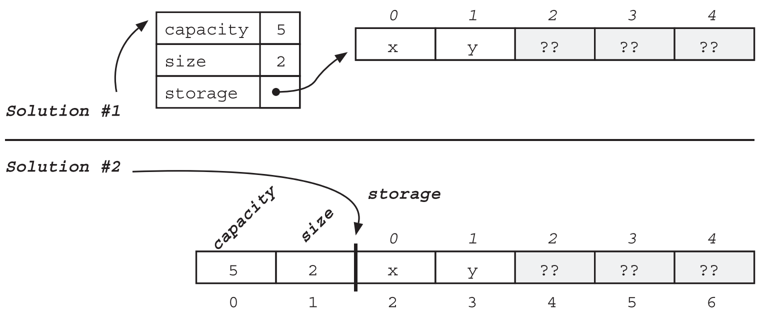

Figure 3. Two solutions to wrap all three array manipulation variables, `capacity`, `size` and `storage`, in order to handle the array with a single reference.

array is memorized. This is just a matter of keeping the list of calls to the `Extend` and `Write` for a particular array. The type set of an array can then be computed on demand by propagating the call to the expressions stored earlier, when all of those expressions obtain their set of possible types.

Figure 3 presents two solutions to reference as a whole all the components of the flexible array. Solution 1 is well adapted to object-oriented programming as it encapsulates variables `capacity`, `size` and `storage` in an object. This is also very convenient in case of reallocation of the `storage` area because the reference to the object does not need to be changed. Solution 2 avoids the extra indirection to access the `storage` area. The `size` and `capacity` are to be reached by moving the pointer by one or two words backward. Solution 2 is actually the usual C `malloc` solution to memorize the size of the memory chunk. That size is necessary in case of a `free` call.

2.3. Avoiding supply area traversal during garbage collection

The most straightforward optimization is to ensure that the GC scans only the useful area of the array (i.e. the indexes `0` to `size - 1`), completely ignoring the supply area. In the SmartEiffel library, zero indexed flexible arrays are represented with the FAST_ARRAY class, using solution 1 of Figure 3. The compiler allows to partially redefine the marking procedure for the `storage` area, making it possible to traverse only the useful part, that is, the user-visible part of the array. Thus, the FAST_ARRAY class is equipped with a template method definition to drive the `storage` area scanning:

GarbageCollectorMark()

```
  int idx = size - 1;
  while ( idx > 0 ) {
      MarkOneItemOf(storage, idx); // Generated by the compiler (Section 2.4).
  idx --;
 }
```

What is performed in the SmartEiffel library to ignore the supply area is easily applicable in Java whatever the category of the GC (i.e. mark and sweep, copying collector or reference counting). Indeed, Java arrays are managed by the virtual machine and are always initialized to zero. Arrays of references are already provided with a header that indicates the size of the array and other information such as those concerning, for example, the type of the elements. All write operations in arrays of references go through the execution of the `aastore` bytecode. It is particularly easy to change the Java virtual machine to avoid, during a GC cycle, the unnecessary traversals of supply areas. One extra index in each array descriptor is enough for the GC to be informed of the boundary of the supply area. Thus, the `aastore` bytecode can be modified as follows:

```
  aastore(array, value, index) {
      // Unmodified original aastore code here (i.e. array bounds check followed by
      // memory write as well as dynamic type check of the value).
      // As an exception is raised when bounds check fails, the index is always valid in the
      // following extra code:
      if ( index > gc_boundary ) {
          gc_boundary = index; // Supply area shrinked.
      }
  }
```

The index `gc_boundary` must be initialized to zero when creating the array. In Java, for security reasons, all arrays of references are initialized with `null`. It is therefore possible to move the boundary of the supply area (i.e. `gc_boundary`) without special precautions. Note that no change is needed for read operations that could occur beyond the index `gc_boundary`. All values beyond the `gc_boundary` index return `null` and do not affect the GC. Of course, one must also modify the code of the GC so that it takes into account the new frontier (i.e. the GC uses `array.gc_boundary` instead of `array.length - 1`).

2.4. Using type information to optimize garbage collection of arrays

The type flow information of the array elements is integrated to the GC as it is already the case for the objects' attributes. SmartEiffel generates a specialized and precise marking function for every object type and without interpretation during execution (for details about SmartEiffel's GC, see [6]). The GC code is generated differently for every application; it is adapted to every manipulated object and is integrated in the final executable. The marking code has a static (hard-coded) knowledge of the object's structure. As an example, an instance variable, which is of type integer, character, or floating point number, is simply ignored (i.e. no marking code is generated for them), because such a variable is not a possible path toward another object. Only attributes that are references are inspected. Yet another improvement, thanks to type flow analysis, in case an attribute is monomorphic (i.e. when the type set of that attribute has only one type), the function call to the marking code is direct. Polymorphic attributes still need the usual dispatching code to the corresponding marking code.

This specialization used for attributes (described in [6]) is now also applied to arrays when iterated in their initialized area. In the particular case of an array that cannot contain `NULL` references, the `NULL` test is omitted while scanning the array. It is also possible to avoid a dynamic call if all the elements of an array have only one possible dynamic type, and if it is the same for all the elements. For the general case, type analysis results are used to produce the dispatch marking code, as we do for ordinary late binding calls. The internal `MarkOneItemOf` call in the loop is processed by the compiler to profit from type analysis results as follows:

MarkOneItemOf(array, index)

```
item = array[index];                  // Array access into the local item variable.

if ( item == NULL ) {                 // Code generated if and only if the null
   return;                            // type is in the type set of the array.
}

switch ( item->dynamic_type ) {       // Switch generated only when the type set
                                      // has more than one non null element.

   case TRUCK:                        // Case branch generated only when the
      MarkTRUCK(item);                // array type set has TRUCK as element.
      break;

   case CAR:                          // Case branch generated only when the
      MarkCAR(item);                  // array type set has CAR as element.
      break;

   ...
}
```

2.5. Extra built-in operations for flexible arrays

As might be expected, the `FAST_ARRAY` class is equipped with many other methods. Most of them are simple combinations of previously defined methods of Section 2.1, as such, they require no special support by the compiler, type analysis results being carried out by transitivity. Given our type analysis technique, methods that reduce the size of the visible area do not need particular compiler support. As an example, the method to remove the rightmost element can be simply defined as follow:

RemoveLast()

```
assert (size > 0);
size--;
```

Note that it is not necessary to reset with NULL the cell that has been assigned to the supply area, as such, that cell will be simply ignored when marking. Thus, in theory, only the methods defined in Section 2.1 are subject to compiler support for type analysis. In practice, as it is not always practical to build an array, little by little, cell by cell, the library features some other built-ins. The `Calloc` built-in allows to allocate and initialize an array with just one call. While the real implementation relies on the efficient C `calloc`, the effect, in terms of type analysis is equivalent to the following code:

Calloc(siz)

```
assert (siz >= 0);
Create(siz);
while ( size < siz ) {
   Extend(NULL);
}
```

As a consequence, such an allocated new flexible array is a potential holder of the NULL value. We needed this built-in to optimize the implementation for all hashing collections (hash table, dictionary, hashed set, ...). Implementing a hash table requires creating an array initialized with NULL and having its size match its capacity. Notice that for performance reasons, `calloc` actually uses the `memset` function from the C language, making it possible for the code to use the specific processor instructions that initialize entire areas in memory. Also for performance reasons, the FAST_ARRAY class is also equipped with another built-in to extend arrays with a large number of cells. That built-in uses the C `realloc` function to minimize and optimize data moving in memory.

3. HANDLING OTHER KINDS OF ARRAYS

Zero indexed array (e.g. SmartEiffel's FAST_ARRAY) is not the only one class to be handled with a progressive filling up strategy and possibly equipped with high-level garbage collection tuning.

3.1. *The* ARRAY *class*

An object of the ARRAY class is an array that can have any integer value for the leftmost index. Positive value, zero or a negative value as well. Except for index translation, which needs an extra attribute, gradual left-to-right filling is appropriate. The implementation is straightforward and very similar to the one of FAST_ARRAY. Obviously, we obtain the same benefits in terms of garbage collection and type analysis. Let us now see how to handle circular arrays.

3.2. *The* RING_ARRAY *class*

Adding elements in front of a FAST_ARRAY or in front of an ARRAY cannot be made efficiently. As all elements have to be shifted to the right, this makes the operation very slow. To address this problem, the SmartEiffel library provides the RING_ARRAY class for managing an array circularly. In order to keep the flow analysis that can detect the absence of NULLs, it is necessary to adapt the filling up technique. Figure 4 shows how it is implemented and the new variables required. In the example, the same array with indexes from 11 to 14 is shown twice. The first shows a case where the used area is in one part, and the supply area is split in two parts. The second shows the inverse case, but many other situations are possible.

As for previous arrays, the `storage` variable points to the beginning of the allocated area, and the `capacity` variable memorizes its size. The `lower` and `upper` variables memorize the lower and the upper bound from the user's point of view, that is, the publicly visible part of the RING_ARRAY. The `storage_lower` variable is used internally and matches the physical index for the `lower` indexed element in `storage`. As such, the `storage_lower` variable always stays between 0 and `capacity - 1`.

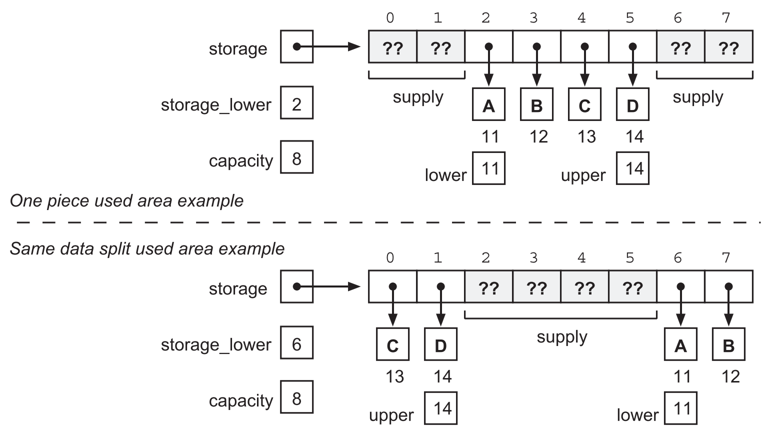
storage
storage_lower
capacity
supply
lower
upper
One piece used area example
Same data split used area example

Figure 4. Implementation of a RING_ARRAY with five variables. The user's visible range is `[lower, upper]`. The `storage_lower` internal variable is the corresponding index of element at `lower` index inside `storage`. Because the used area unfolds circularly starting at any possible `storage_lower`, there are many possibilities.

The following operations define the abstract data type for the RING_ARRAY, keeping type flow analysis properties.

RingCreate(cap, low) Creation of an empty RING_ARRAY. The lower index is initialized with `low` and the initial `capacity` with `cap`:

```
assert (cap >= 0);
capacity = cap; lower = low;
storage = malloc(cap);
storage_lower = 0; upper = low - 1;
```

As for the creation of noncircular arrays, the initial used area is empty, and the set of possible types is initialized with the empty set.

RingExtend(obj) To extend by one the array on its right. When the supply area is empty, as previously, the `capacity` is made twice as big:

```
if ( upper - lower + 1 >= capacity ) {
   capacity = capacity * 2;
   storage = realloc(storage, capacity);
}
sidx = upper + 1 - lower + storage_lower;
if (sidx >= capacity) {
   sidx = sidx - capacity;
}
storage[sidx] = obj; upper = upper + 1;
```

As for other arrays, as soon as a call of `RingExtend` is in the reachable code, all slots are considered as possible holders of `obj`'s types.

RingPrepend(obj) To extend by one the array on its left. The type flow information is maintained the same way we did for `RingExtend`. Only the index computation in the `storage` area is different:

```
if ( upper - lower + 1 >= capacity ) {
   capacity = capacity * 2;
   storage = realloc(storage, capacity);
}
```

```
storage_lower = storage_lower - 1;
if (storage_lower < 0) {
   storage_lower = storage_lower + capacity;
}
storage[storage_lower] = obj; lower = lower - 1;
```

RingRead(ind) To access the element, which is at index `ind`, assuming that `ind` is a valid index in the visible area:

```
assert (lower <= ind) && (ind <= upper);
sidx = ind - lower + storage_lower;
if (sidx >= capacity) {
   sidx = sidx - capacity;
}
return storage[sidx];
```

As this only reads the array and does not modify it, it does not change the type flow information previously gathered.

RingWrite(ind, obj)

```
assert (lower <= ind) && (ind <= upper);
sidx = ind - lower + storage_lower;
if (sidx >= capacity) {
   sidx = sidx - capacity;
}
storage[sidx] = obj;
```

As usual, because this adds a new value in the array, the possible types of `obj` must be all added into the set of possible types contained in the array.

And finally, to complete the description of the abstract data type for RING_ARRAY, we have to make the GC aware of the visible area with the following template definition:

GarbageCollectorRingMark()

```
int cnt = upper - lower + 1;
int idx = storage_lower - 1;
while ( cnt >= 0 ) {
   idx ++;
   if (idx >= capacity) {
      idx = idx - capacity;
   }
   MarkOneItemOf(storage, idx);
   cnt --;
}
```

Again, the `MarkOneItemOf` call is generated by the compiler and customized according to type analysis results. In the best case, there is only one possible type and the exact marking code for only one kind of element is inlined.

3.3. Hash tables or hash maps using an array

More complex data structure using hashes, as for example hash sets, are using a primary array that might contain NULL references or a reference to a storage cell (Figure 5). Access to this array uses the *hash code* of the targeted object. If the array contains a NULL at the index corresponding to the *hash code*, it means that the object is not present in the set. If the array contains a valid reference to

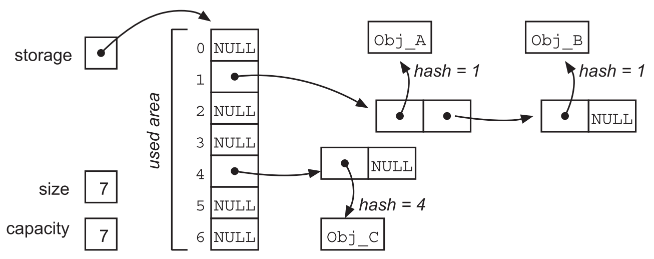

Figure 5. Using an array as a *hash map* to implement a set. All the `capacity` of the `storage` area is used, consequently, the supply area is empty. NULL values inside the array are meaningful.

a storage cell instead, the object can then be searched in the linked list composed of cells to account for any collisions. In order to create the primary array, the built-in `calloc` is used. As the primary array does not contain a reserved area for future expansion, the GC normally works by exploring the whole array. And as it is absolutely common for the primary array to contain NULLs, the use of `calloc` is clearly the best choice as it implies the presence of NULL in the set of possible types.

4. BENCHMARKS AND MEASUREMENTS

4.1. *Garbage collection for arrays in SmartEiffel*

We equipped the source code of the SmartEiffel's GC with extra code in order to examine the impact of arrays on memory footprint. To have a significant execution, we have used the whole source code of the SmartEiffel compiler itself, that is, 180,000 lines of Eiffel source code during its own bootstrap. The self recompilation of the compiler is a very good benchmark because it uses many arrays and requires about 330 Mb of memory during the process. Moreover, the GC is triggered 32 times while compiling the compiler[‡].

For the following measurements, only arrays of references are considered, because other arrays, for example, arrays of integers or arrays of characters, are not directly concerned by our type flow analysis technique. Moreover, the SmartEiffel GC does not even scan the content of arrays of scalars.

We modified the marking procedure for the content of arrays so as to count the number of marked arrays during one recompilation. The measurement shows that the GC processes 6,399,198 arrays. The total size of the corresponding used area scanned is 12,548,963 cells. As the total `capacity` of processed arrays is 21,714,957 cells and because the supply area is not scanned, the GC avoids scanning 9,165,994 cells, that is, a gain of 42%.

In order to gather more information, we examined the distribution of arrays during the very last run of the GC, just before the final exit of the program. During the last run of the GC, among the 838,681 marked arrays, 31,859 are actually empty arrays. These empty arrays take direct advantage of our optimization because they are marked in a constant time. By contrast, there are 453,215 arrays, which are completely filled up and, as a consequence, need to be entirely scanned. Figure 6 displays the distribution of arrays according to the relative size of the supply area. The filled up arrays are represented in the leftmost bar, and the empty arrays are in the rightmost bar. In Figure 6, the largest bar is for arrays having less than 10% of supply area, which seems to indicate that SmartEiffel does not waste too much memory using arrays with large supply areas.

In addition, during the last run of the GC, we also measured that the total `capacity` of storage was 3,242,679 cells including 1,440,568 cells in the supply areas. Using the distribution of arrays from Figure 6, Figure 7 gives the corresponding capacities for storage. The values *Min*, *Max* and *Avg* give, respectively, for each family (the 10% family, the 20% family and the 30% family), the smallest capacity, the largest capacity and the average capacity. The average size of arrays is small and filled up arrays represent 35% of the total memory used for arrays. Still using the same distribution of

[‡]The heap size is automatically tuned with the results of the memory collection.

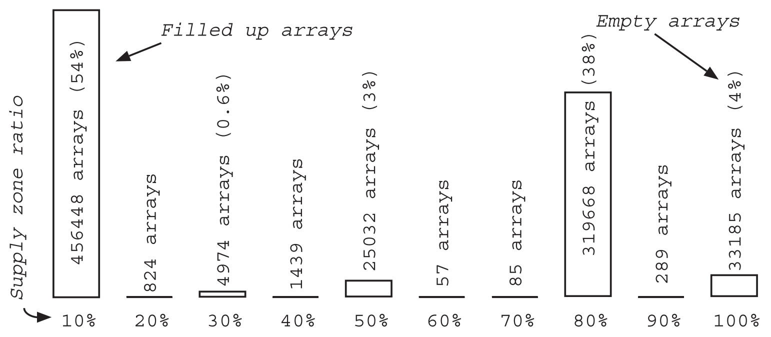

Figure 6. Bootstrap of SmartEiffel. Distribution of arrays according to the relative size of the supply area. Only arrays of references are considered for Figures 6, 7 and 8 (same run).

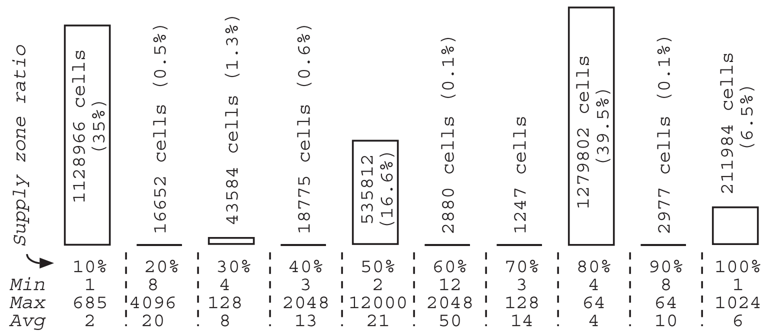

Figure 7. Bootstrap of SmartEiffel. Following Figure 6 with the same distribution, the relative size of capacities in number of cells.

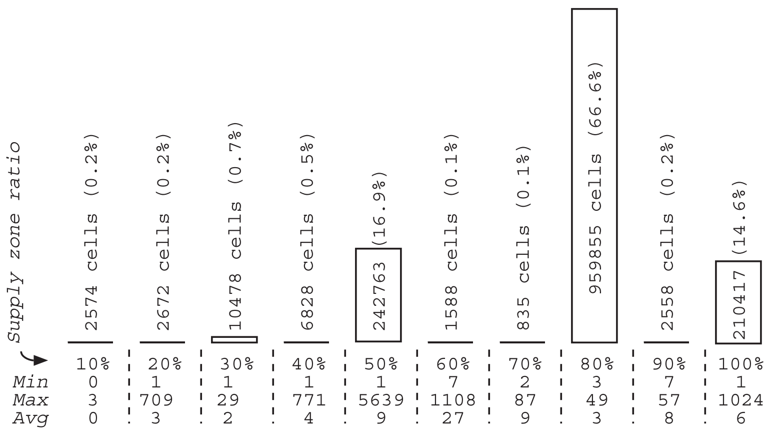

Figure 8. Bootstrap of SmartEiffel. Following Figure 6 with the same distribution, the relative size of supply areas in number of cells.

arrays, Figure 8 shows the corresponding sizes for the supply areas. The 80% family represents the largest part of the memory used for the supply areas, which probably correspond to arrays used as temporary buffers, growing and shrinking all along the execution.

If one compares the memory used for arrays of references (3,242,679 references) with the total memory footprint during bootstrap (330 Mb, i.e. 82×10^6 references), the memory used for arrays of references represents only 4% of that memory. Considering only the `supply` areas, it represents even less, actually 1.7% of the total memory footprint. Thus, during bootstrap, the total memory used for arrays of references is so small compared with the memory used for other objects that the gain of runtime because of optimized array marking is probably low. We modified the GC of SmartEiffel in order to force scanning of the supply areas, but as expected, we did not notice any discernible modification of runtime, either of memory used or in the number of GC calls.

4.2. *Type flow analysis in the Lisaac compiler*

While trying to perform a true global system analysis, the lack of type analysis concerning the content of arrays has consequences in the rest of the analysis. For instance, as soon as a variable is assigned with a reference read from an array, the lack of type information on the content of the array is echoed on the information that was previously gathered for this variable. If this variable is then copied into another variable or passed as an argument, the unclear information about the array content propagates rapidly. Array read or array write operations are quite frequent, and it is thus important to collect, then to transmit, type flow information concerning array contents.

The Lisaac compiler carries out a global type flow analysis for all kinds of variables: formal parameters, local, global and instance variables. We also implemented our type flow analysis with NULL detection inside arrays thus making type flow analysis really global. The gathered information for arrays may impact all kinds of variables and transitively, all kinds of expressions, from methods to methods without any border. We have used the source code of the Lisaac compiler itself, 53,000 lines of Lisaac source code, as a benchmark for the following measurements.

We modified the compiler in order to inspect the types of variables after type flow analysis completion. For the measurement, we kept only variables whose type is an array of references. Arrays of scalar values (arrays of integers, arrays of characters or arrays of floats) are not considered because there is no possible polymorphism inside such arrays. Furthermore, only arrays of references may hold NULL values. As explained previously, NULL is considered as a possible element in the set of types, which describe cell contents. Figure 9 shows the distribution of type sets according to the number of items in each set. The level of polymorphism is variable from 1 to 56 for the compiler source code. As shown in the lower part of the figure, many arrays are statically detected without possible NULL inside cells. For the source code of the compiler, this stands for 56.2% of arrays. Also in Figure 9, the leftmost bar in the upper part indicates that 79 variables are holding arrays with only one kind of elements (i.e. there are 79 variables holding *monomorphic* arrays). The corresponding bar in the lower part indicates that 96% of those arrays never hold NULL values. As an example, for such an array, which contains only TRUCK, the marking function generated by the compiler is

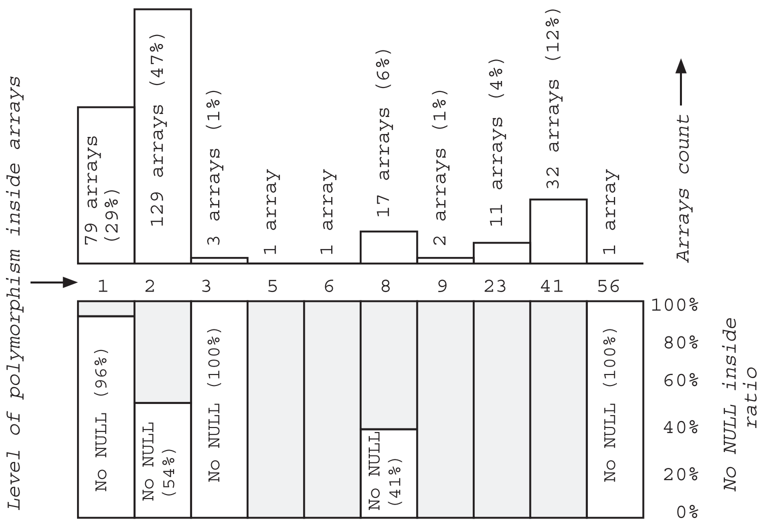

Figure 9. Bootstrap of Lisaac. Compile time information regarding types of arrays. The upper part gives the distribution of arrays according to the number of elements in type sets. The lower part gives the corresponding ratio of arrays without NULL.

MarkOneItemOf(array, index)

```
  MarkTRUCK(array[index]);      // Direct static call of the marking function (no NULL check).
```

The upper right-hand bar of Figure 9 shows that only one variable obtains an array type with 56 kinds of objects mixed together in its cells. A close look at the source code taught us that this variable was actually an instance variable. Thus, at runtime, each object holding that instance variable will have its own megamorphic array, and as indicated by the lower right-hand bar, all those arrays will never include the NULL value.

To examine the propagation of the type flow information from arrays to local variables, we modified the compiler in order to add artificially the NULL value inside all sets of types for all arrays. Among the 10,429 local variables, before the modification of the compiler, only 3,202 have the NULL value inside their type set, that is to say 30.7% of the local variables. Artificial addition of the NULL value inside arrays infects 7,043 local variables, making 98.2% of local variables possible NULL holders. This experimentation shows that the result of type flow information for arrays directly impacts on the type flow information for local variables. Figure 10 gives the detail of the NULL propagation, local variables being sorted according to their level of polymorphism, which varies from 2 to 57.

We then performed the same measurement for global variables and instance variables. Before the artificial addition of the NULL value inside all arrays, 70.5% of global variables were already NULL holders. After the modification, 98.5% of global variables became NULL holders. For instance variables, the NULL propagation changes from 93.2% of NULL holders to 99.4%. Figure 11 details the NULL propagation for global variables and instance variables.

To continue the study of array type flow propagation, we then artificially added into the type sets of arrays, not only the NULL value, but also all compatible subtypes of the elements. For instance, with an array of VEHICLEs, all encountered subtypes of VEHICLE are artificially added into the type set: CAR is added, TRUCK is added, BIKE, and so on, not forgetting to add the NULL value. For local variables, before the modification, there were 17 different type set sizes in the whole source code of the compiler. After the modification, only 12 different sizes remain. Detailed distribution presented in Figure 12 shows that the number of sets decreases and that these sets are bigger, thus indicating that the accuracy of type flow analysis for local variable is drastically impacted by arrays.

4.3. *Memory behavior analysis of some other C programs*

In order to inspect other software products than SmartEiffel and Lisaac, we analyzed the memory behavior of several other applications (Figures 13 and 14). To do this, we have developed a new module for Valgrind [17] in order to observe the allocation of memory blocks that may correspond

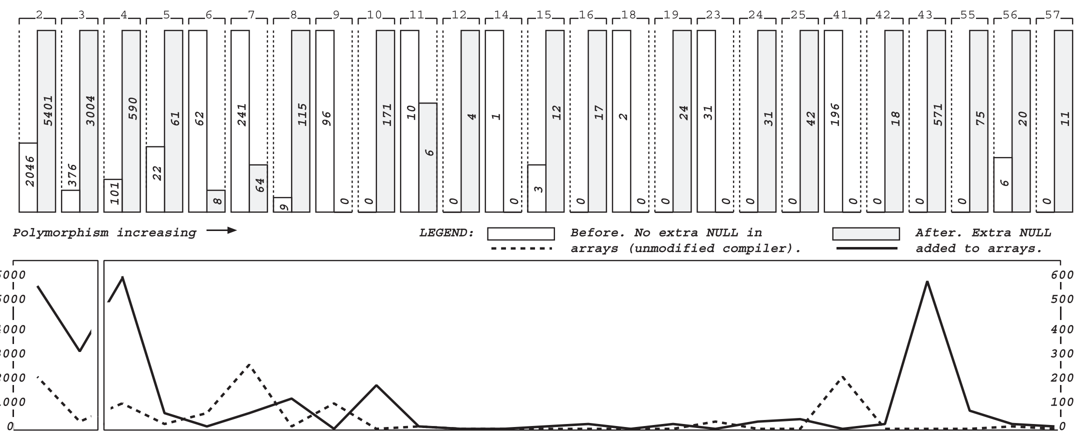

Figure 10. Bootstrap of Lisaac. *Before* and *after* compiler modification. Adding extra NULL to each type of array to view the propagation of the wrong information into local variables type sets. The number of local variables is indicated inside each bar (upper part).

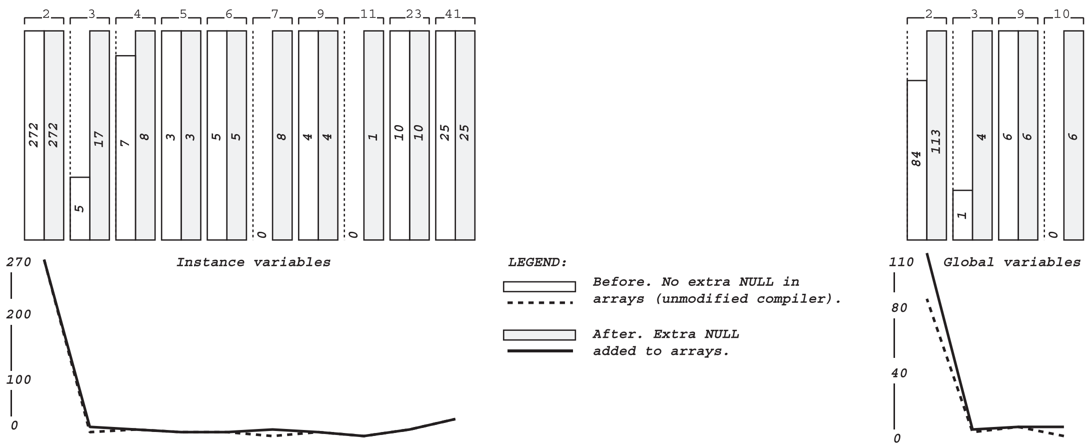

Figure 11. Bootstrap of Lisaac. *Before* and *after* compiler modification. Adding extra NULL to each type of array to view the propagation of the wrong information into instance variables (left) and global variables (right).

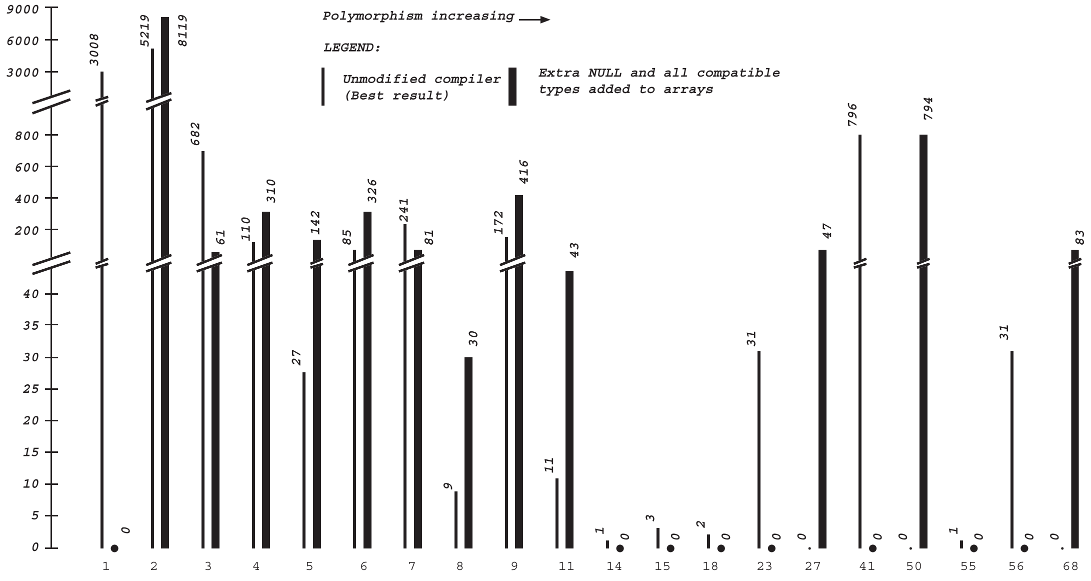

Figure 12. Bootstrap of Lisaac. Propagation into local variables of array type sets saturation: NULL added as well as all compatible types.

to arrays of pointers. All reads and writes in these blocks are dynamically analyzed until the end of execution. Each time a read or write operation involves another data size than the memory pointer size[§], the corresponding block is excluded from the analysis because we want to study the sole memory devoted to arrays of pointer, not the memory used for other objects.

The objective is to determine if a memory block is filled gradually or not. We only detected the gradual left-to-right filling up, from the beginning of the block. The other gradual filling strategies, such as the right-to-left filling up, have not been investigated. Just after the allocation of each new block (i.e. `malloc` or `calloc`), the supply area occupies the whole memory of that block. Each new block is initially classified into the gradual-filling category. A block is kept in the gradual-filling category only if the first write operation is performed into the leftmost slot of the supply area and so

[§]The tests were performed on a 64-bit architecture. We also used the function `my_get_chunk` from Valgrind, which allows to check that a value corresponds to a pointer to a block in the heap.

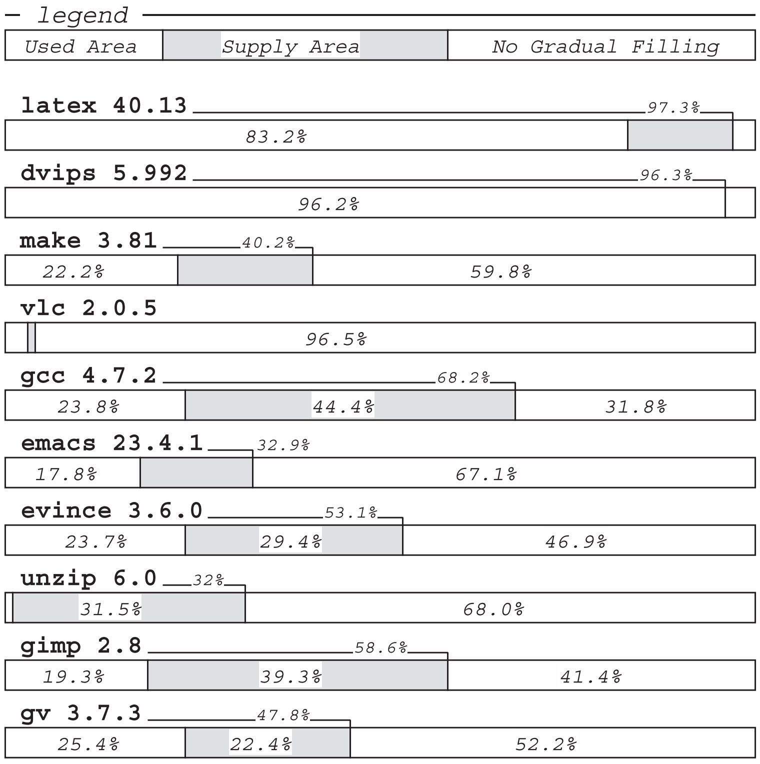

Figure 13. Detection of memory blocks, which are filled gradually. Relative memory used for all arrays of pointers. Measured just before the program exits.

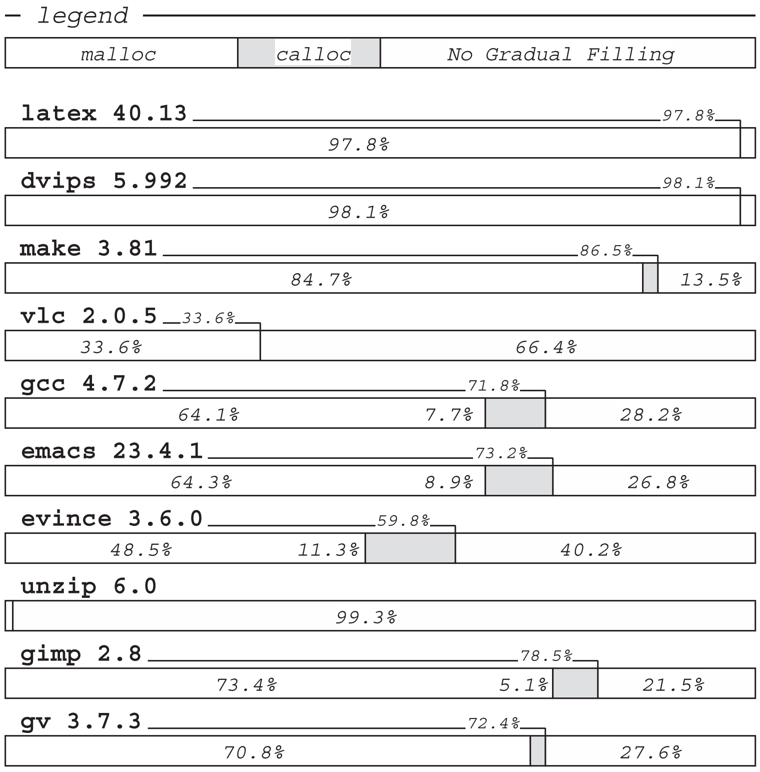

Figure 14. Detection of memory blocks, which are filled gradually. Relative number of arrays of pointers. Useless `calloc` detection. Measured just before the program exits.

on. Once a write operation is out of the used area and is not equivalent to the `Extend` operation of Section 2.1, the corresponding block is definitively removed from the gradual-filling category.

Figure 13 shows the results in terms of memory space. For each program we considered, the size of the *Used Area* bar represents the total memory space occupied by used areas of blocks, which are filled gradually. The *Supply Area* bar represents the total memory space occupied by supply areas of blocks, which are filled gradually. Then, the *No Gradual Filling* bar corresponds to the memory used by arrays, which are not filled progressively. Figure 13 shows that gradual filling is a quite common phenomenon and that many software could benefit from the GC optimization presented before. The best result is obtained by `gcc` with 44.4% of supply area (i.e. ignored by the GC), which is a similar result to what we found for SmartEiffel (42%). Similarly, `evince` (29.4%), `unzip` (31.5%) and `gimp` (39.3%) are also good candidates to benefit from this optimization. Also in Figure 13, we see that `vlc` might not benefit from our optimization because 96.5% of arrays memory is not filled gradually. Finally, for `dvips`, even if arrays, which are filled gradually, represent 96.3%, our optimization would be useless because there is no supply area. Note that measurements were made just before the programs exit. Knowing that a GC cycle may occur at any time in the life of a program, the situation can be completely different (larger supply areas or arrays which are not yet allocated).

When an array is filled gradually, it is not necessary or useful to initialize memory when creating a new array. Thus, in C, it is preferable to use the `malloc` function rather than the `calloc` function, which is more expensive (i.e. in addition to what is performed by the `malloc` function, the `calloc` function clears the allocated memory). We performed a rerun of tests of Figure 13 under the same conditions in order to inspect how arrays are allocated (`malloc` or `calloc`). Figure 14 shows the results in number of allocated blocks. The *malloc* bar represents the number of blocks, which have been filled gradually and allocated with `malloc`. The *calloc* bar represents the number of blocks, which have been filled gradually and allocated with `calloc`. Then, the *No Gradual Filling* bar is for all other arrays of pointers. The results in Figure 14 show that the vast majority of arrays that are filled gradually are, which is consistent, allocated with `malloc`. Note, however, that for `evince`, 11.3% of arrays are filled gradually and are built with `calloc`. Similarly, `emacs` (8.9%) and `gcc` (7.7%) have a significant number of arrays that might be allocated more efficiently using `malloc` instead of `calloc`. One could well imagine that the memory tracking tools (e.g.

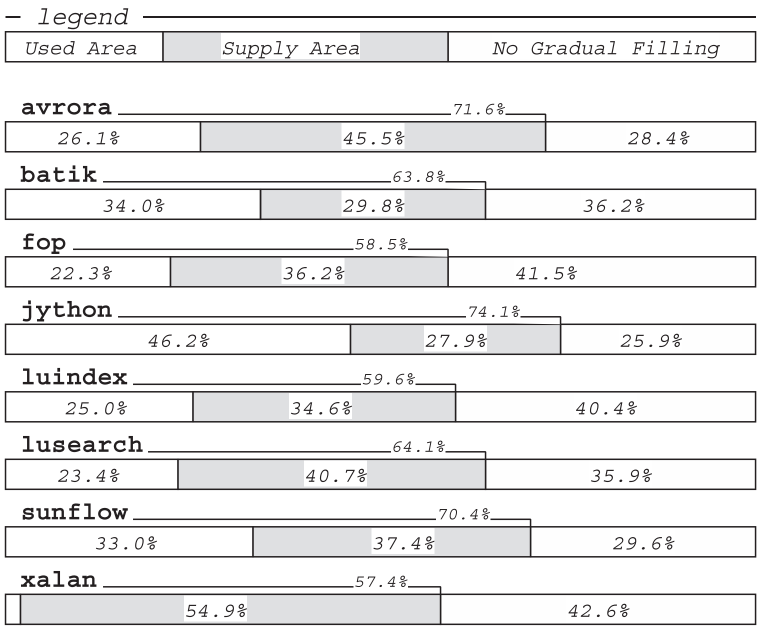

Figure 15. Java arrays benchmark. Modified Jikes `RVM 3.1.3` and DaCapo `9.12-bach`. Detection of memory blocks, which are filled gradually. Relative memory used for all arrays of references.

Valgrind), which are already looking for uninitialized memory usages, could be enhanced with the detection of progressive filling up, in order to suggest the programmer to advantageously replace some calls to `calloc` by calls to `malloc`.

4.4. Memory behavior analysis of some other Java programs

To complete our tests, we measured the behavior of Java programs as regards the handling of arrays of references. As before, the aim is to know if arrays are filled gradually or not. We modified the code of the Java Virtual Machine (Jikes RVM 3.1.3) in order to trace all write accesses inside arrays of references. Thus, we have equipped the `aastore` instruction of the virtual machine, which is the necessary step of all write operations inside arrays of references. As in previous tests, we only detected the gradual left-to-right filling up of arrays throughout the program execution. We used the DaCapo benchmarks version 9.12-bach [18]. Figure 15 shows the results in terms of memory space. The *Used Area* (respectively *Supply Area*) bar represents the total memory space occupied by used areas (respectively supply areas) of blocks, which are filled gradually. The *No Gradual Filling* bar corresponds to the memory used by arrays, which are not filled progressively. The results presented in Figure 15 clearly shows that a very large proportion of arrays are filled gradually and that the supply area is very high (from 27.9% for `jython` to 54.9% `xalan`). Also note that, as before, the measurement was made on completion (i.e. just before the final exit).

5. DISCUSSION AND RELATED WORK

5.1. Related work for type analysis inside arrays

Most of the work performed about arrays concerns array bound checks [19–21]. The automatic analysis of properties of array content was considered only recently. Works in [22, 23] study decidable logics to express properties of array content. If we restrict ourselves to automatic analysis, an important track initiated by [24] concerns verification of programs with arrays using predicate abstraction [25, 26], possibly improved with counter-example guided refinement [27] and Graig interpolants [28]. All these approaches make use of the property to be proved, while [29] aims at discovering properties.

Indeed, the work presented in [29] concerns properties for one-dimensional arrays in nontrivial programs, which manipulate arrays only by sequential traversal when the array content is restricted to scalar (i.e. primitive) values. Discovered properties in [29] are array maximum value detection or array copy detection or even correct insertion of some value inside a sorted array. For object-oriented languages, it is important to address polymorphism inside arrays as well as NULL prediction inside arrays. Indeed, it is quite common to have multiple kinds of objects mixed together with possible NULL values inside the same array.

Concerning automatic program analysis methods based on abstract interpretation, a common approach presented in [16], for safety critical software, consists in summarizing an array with one auxiliary variable, managed to satisfy the disjunction of properties of all cells of the array. We adapted the *array smashing* method of [16] for type flow analysis, and thanks to our gradual filling strategy, we solved its initialization problem pointed out in [29].

While traditional implementations of arrays use contiguous storage, [30–34] are splitting the array in memory by using a *discontinuous storage*. To reduce fragmentation, Siebert [30] organizes array memory in trees. To avoid tree traversal, [31, 33] use a single level of indirection to fixed-size *arraylets*. [32] contemporaneously invented arraylets to aggressively compress arrays during collection and decompress on demand for memory-constrained embedded systems. To remove most indirections, Z-rays of [34] inlines the first *N* array bytes into the array spine. For arrays of small sizes, the most numerous category, Z-ray avoids indirections and reaches the performance we have for the contiguous layout. Somehow, with Z-rays, the right side of arrays (i.e. parts with indirection) looks like our supply area. However, Z-rays storage is inherent in two properties of the target language (e.g. Java): zero-initialization of all arrays to take advantage of the zero-compression and the abstraction of the physical representation of an array (i.e. no pointer manipulation for array traversal).

5.2. *About the initialization of arrays and variables*

An uninitialized variable is a well-known origin of bugs: unpredictable and/or randomly initialized integers, invalid pointers to wrong memory locations, and so on. Furthermore, an uninitialized variable adds uncertainty, which makes type flow analysis impossible or at least useless. As a consequence, to achieve type flow analysis, the language must either force the programmer to initialize all variables or provide default values for uninitialized variables. The C# and Java languages provide default values for all instance/class variables and force the initialization of all local variables. For the vast majority of new programming languages, uninitialized variables are avoided, either with languages default rules or with compile time data flow checks. For less recent languages (e.g. Fortran, Basic or C), it is also possible to force the initialization thanks to external extra tools thus allowing type flow analysis.

Notice that for a language like Java, without default value for local variables, the NULL situation implies that an explicit assignment with NULL exists or, by transitivity, that another assigned expression could hold that value.

In a language like Eiffel, the default is NULL for all kinds of reference variables, that is, local variables or instance variables, which are not of a scalar type. As a drawback, this uniform language decision forces the use of *data flow analysis* to predict that a certain variable cannot hold NULL in its type set. Here, it is necessary to take into account the order of statements, just after the local variable declaration or inside all creation procedures for an instance variable, in order to guarantee that the default value is always overwritten before the variable is read. Insofar as this language decision can be considered a facility from the programmer's point of view, it makes the work of implementing type analysis harder. We believe that explicit initialization of all variables should be the rule as is the case in our initialization strategy of arrays. At least, this makes it easy to perform type flow analysis.

6. CONCLUSION

We have shown how it is possible to optimize garbage collection of arrays by ignoring the supply area, usually present in flexible/resizable arrays. We have considered several kinds of resizable arrays: leftmost zero indexed, user-defined indexing, circular arrays and hash maps. Each kind of array is described as an abstract data type also including a customized marking procedure for the GC.

We have presented a simple technique to analyze the type of elements that are stored in arrays. That analysis does not take into account either the order of elements in the array or the index variation. The whole array, composed of many cells, is considered as a single polymorphic variable. We have shown that the gradual filling up of the array, cell by cell, facilitates detection of the NULL value inside the array.

We measured the impact of our array manipulation strategy during the bootstrap of the SmartEiffel compiler. On that large piece of code, our technique avoids scanning 42% of the memory used for arrays of references (Section 4.1).

Thanks to our technique of type flow analysis inside all kinds of arrays, our Lisaac compiler is able to carry out a true global analysis. Roughly, we measured that 98% of the variables do benefit from array type flow information. We also measured that 56% of arrays never hold NULL values. Furthermore, the level of polymorphism in arrays is significantly reduced (Section 4.2).

We also measured on some other large well-known applications, written in C or C++, that the ratio of arrays using gradual filling is often similar to the one encountered in SmartEiffel or Lisaac (Section 4.3).

ACKNOWLEDGEMENT

The authors wish to thank Jean-Pierre Camal and Vasilica Le Floch for proofreading.

REFERENCES

1. Jones R, Lins R. *Garbage Collection*. John Wiley and Sons Ltd: Chichester, West Sussex, England, 1996. ISBN 0-471-94148-4w.

2. Zendra O, Colnet D, Collin S. Efficient dynamic dispatch without virtual function tables. The SmallEiffel compiler. *In "12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)",* ACM SIGPLAN Notices, Volume 32, Issue 10, Atlanta, Georgia, 1997; 125–141.
3. Zendra O, Colnet D. Adding external iterators to an existing Eiffel class library. *TOOLS Pacific'99*, Melbourne, Australia, 1999; 188–199.
4. Sonntag B, Colnet D. Lisaac: the power of simplicity at work for operating system. *TOOLS Pacific'2002*, Sydney, Australia, 2002; 45–52.
5. Sonntag B, Colnet D. Efficient compilation strategy for object-oriented languages under the closed-world assumption. *Software Practice & Experience* 2012; **44**(5):565–593.
6. Colnet D, Coucaud P, Zendra O. Compiler support to customize the mark and sweep algorithm. *ISMM'98*, Vancouver, BC, Canada, October 1998; 154–165.
7. Palsberg J, Schwartzbach MI. Object-oriented type inference. *OOPSLA'91*, Phoenix Arizona, USA, October 1991; 146–161.
8. Corney D, Gough J. Type test elimination using typeflow analysis. *Proceedings of Programming Languages and System Architectures*, Vol. 792, LNCS, Zurich, Switzerland, 1994; 137–150.
9. Dean J, Grove D, Chambers C. Optimization of object-oriented programs using static class hierarchy analysis. *ECOOP'95*, Springer Verlag, Aarus, Denmark, August 1995; 77–101.
10. Agesen O, Palsberg J, Schwartzbach MI. Type inference of SELF: analysis of objects with dynamic and multiple inheritance. *Software Practice & Experience* 1995; **25**(9):975–995.
11. Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. *OOPSLA'96*, ACM Press, San Jose, California, Oct 1996; 324–341.
12. Dean J, DeFouw G, Grove D, Litvinov V, Chambers G. Vortex: an optimizing compiler for object-oriented languages. *OOPSLA'96*, ACM SIGPLAN Notices, San Jose, California, USA, Oct. 1996; 83–100.
13. Collin S, Colnet D, Zendra O. Type inference for late binding. The SmallEiffel compiler. *JMLC'97*, Vol. 1204, LNCS, Linz Austria, March 1997; 67–81.
14. Fitzgerald R, Knoblock TB, Ruf E, Steesgaard B, Tarditi D. Marmot: an optimizing compiler for Java. *Software Practice & Experience* 2000; **30**(3):199–232.
15. Fähndrich M, Leino R. Declaring and checking non-null types in an object-oriented language. *OOPSLA'2003*, Anaheim, California, Oct. 2003; 302–312.
16. Blanchet B, Cousot P, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. A static analyser for large safety-critical software. *PLDI 2003*, Vol. 38, ACM SIGPLAN Notices, New York, NY, USA, May 2003; 197–207.
17. Bond M, Nethercote N, Kent S, Guyer S, McKinley S. Tracking bad apples: reporting the origin of null and undefined values errors. *OOPSLA'2007*, Montreal, Quebec, Canada, Oct. 2007; 405–422.
18. Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, Diwan A, Feinberg D, Frampton D, Guyer SZ, Hirzel M, Hosking A, Jump M, Lee H, Moss JEB, Phansalkar A, Stefanovic D, VanDrunen T, von Dincklage D, Wiedermann B. The DaCapo benchmarks: Java benchmarking development and analysis. *OOPSLA'2006*, Vol. 41, ACM Press, Portland Oregon, USA, 2006; 169–190.
19. Cousot P, Cousot R. Static determination of dynamic properties of programs. *2nd International Symposium on Programming*, Dunod, Paris, April 1976; 106–130.
20. Gupta R. A fresh look at optimizing array bound checking. *PLDI 1990*, New York, NY, USA, 1990; 272–282.
21. Chin WN, Goh EK. A reexamination of "optimization of array subscripts range checks". *TOPLAS 1995*, Vol. 17, No. 2, New york, NY, USA, March 1995; 217–227.
22. Bradley AR, Manna Z, Sipma HB. What's decidable about arrays? *VMCAI 06*, LNCS 3855, Springer Verlag, Berlin Heidelberg, 2006; 427–442.
23. Iosif R, Habermehl P, Vojnar T. What else is decidable about integer arrays? In *FOSSACS 2008*, Amadio R (ed.). LNCS, Springer Verlag: Budapest, Hungary, April 2008; 474–489.
24. Flanagan C, Qadeer S. Predicate abstraction for software verification. *POPL 2002*, Portland, 2002; 191–202.
25. Lahiri SK, Bryant RE, Cook B. A symbolic approach to predicate abstraction. *CAV 2003*, LNCS 2725, Boulder, Colorado, USA, July 2003; 141–153.
26. Lahiri SK, Bryant RE. Indexed predicate discovery for unbounded system verification. *CAV 2004*, LNCS 3114, Boston, MA, USA, July 2004; 135–147.
27. Beyer D, Henzinger TA, Majumdar R, Rybalchenko A. Path invariants. *PLDI 2007*, San Diego (CA), 2007; 300–309.
28. Jhala R, McMillan KL. Array abstraction from proofs. *CAV 2007*, LNCS 4590, Springer Verlag, Berlin, Germany, July 2007; 193–206.
29. Halbwachs N, Péron M. Discovering properties about arrays in simple programs. *PLDI 2008*, Vol. 43, ACM SIGPLAN Notices, Tucson, Arizona, June 2008; 339–352.
30. Siebert F. Eliminating external fragmentation in a non-moving garbage collector for Java. *CASES 2000*, San Jose, 2000; 9–17.
31. Bacon D, Cheng P, Rajan VT. A real-time garbage collector with low overhead and consistent utilization. *POPL 2003*, Louisiana, 2003; 285–298.
32. Chen G, Kandemir M, Vijaykrishnan N, Irwin MJ, Mathiske B, Wolczko M. Heap compression for memory-constrained Java environments. *OOPSLA'2003*, Vol. 38, ACM Press, Anaheim, California, Nov. 2003; 282–301.
33. Pizlo F, Ziarek L, Maj P, Hosking A, Blanton E, Vitek J. Schism: fragmentation-tolerant real-time garbage collection. *PLDI 2010*, Vol. 45, ACM SIGPLAN Notices, Toronto Canada, June 2010; 146–159.
34. Sartor JB, Blackburn S, Frampton D, Hirzel M, McKinley KS. Z-Rays: divide arrays and conquer speed and flexibility. *PLDI 2010*, Vol. 45, ACM SIGPLAN Notices, Toronto, Canada, June 2010; 471–482.