Optimizing JVM Object Operations to Improve WCET Predictability

Angelo Corsaro[1], Corrado Santoro[2]

[1]Washington University
Department of Computer Science and Engineering
1 Brookings Drive, BOX 1045, St. Louis, 63130
Missouri, USA
EMail: corsaro@cse.wustl.edu

[2]University of Catania
Dept. of Computer Science and
Telecommunication Engineering
Viale A. Doria, 6 - 95125 - Catania, Italy
EMail: csanto@diit.unict.it

Abstract

This paper describes the optimizations introduced in Juice*, a J2ME virtual machine for embedded systems. These optimizations are designed to make possible the determination of the WCET of the JVM bytecodes related to object and array management. The solution proposed, which is based on subdividing the heap in a set of* chunks *of fixed size, allows to execute those bytecodes either in a constant time or in a linear time with an upper bound that can be determined.*

1 Introduction

In real-time systems, determination of the worst-case execution time (WCET) plays a fundamental role in task feasibility analysis and scheduling. Frameworks for WCET analysis [1] are based on determining the expected execution time of each instruction of the given task. In a real-time Java environment, this implies to obtain the WCET of each Java bytecode. Such an analysis could be hard for those bytecodes that need to manipulate Java heap or access the structure of involved objects, class/interface hierarchy, etc. In such a context, this paper describes the optimizations introduced in *Juice* [3], a J2ME virtual machine designed by the authors to be run upon NUXI [5], a light executive for Intel-based embedded systems[1]. Juice uses a heap management technique and an object layout that facilitate object allocation, object's attributes access and garbage collection. The employed technique allows to perform these operations in a predictable time. The paper focuses on object allocation/deallocation and attribute reading/writing, showing how these operation can advantage of the proposed heap management technique.

[1]NUXI can be downloaded at `http://nuxi.iit.unict.it`

2 Heap Management

Operations related to heap management are those executed when an object has to be allocated or collected. In general, the time required to perform the creation of a new object depends on the size of the object that, in turn, depends on the amount of attributes declared in the object's class and in its ancestors. The operations required for object allocation can be summarized as: (*i*) determine the number and the type of the attributes, in order to compute the size of the memory area to allocate in the heap, and (*ii*) find a *contiguous* area of free memory, in the heap, where to allocate the created object.

The former operation could imply to navigate class hierarchy in order to find all the attributes the object possesses; indeed, number of attributes can be computed at class loading time, thus storing object size in a field of the structure representing the class in memory. The latter operation instead implies to scan the heap until a piece of free memory, whose size is greater than or equal to the requested amount, is found. This operation requires, in general, a time dependent on the size of the heap and of the object [4, 6]. This means that the WCET of such an operation cannot be exactly computed, but only upper bounded with a limit that depends on heap size.

To overcome the problems above, we propose a technique that, by borrowing some principles from Unix-style file system handling, provides an efficient algorithm to allocate any object in a time that depends only on the size of the allocated object, a parameter that can be exactly estimated with a static bytecode analysis. Our solution subdivides the entire heap in

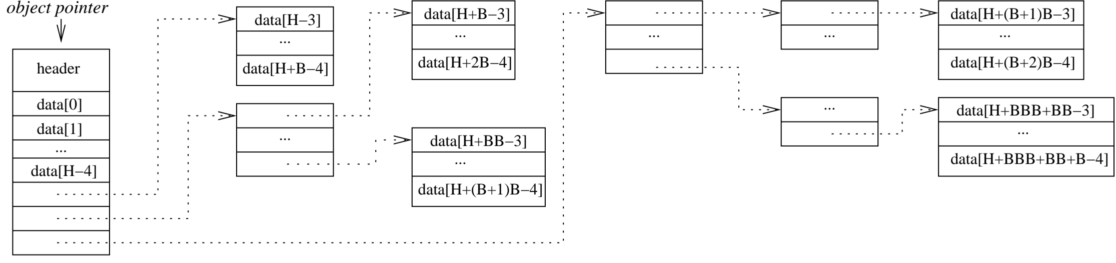

Figure 1. Object Data Allocation Policy

a sequence of *chunks* of a fixed size CS. All chunks are organized in a linked list, started by a pointer FC representing the *free list* of chunks (the first four bytes of each chunk represent the pointer to the next free chunk). Allocating a chunk means to pick it from FC, moving the latter to the next free chunk; while releasing a chunk implies to place it at the head of the free list, thus updating FC accordingly.

Using such a memory layout, allocating an object, given its size S, implies to pick a number of free chunks equal to $NC = \lceil \frac{S}{CS} \rceil$, operation that does not require to walk the heap and whose duration can be predictable. Similarly, releasing an object implies to return allocated chunks to the free list.

With such an allocation policy, the main issue is that the allocated chunks could not be contiguous and object's data could be spread over different chunks; a technique to suitably link those chunks together is thus needed, and it also must take into account that object's data access has to be fast and predictable.

2.1 Object Allocation Policy in Juice

In Juice, a Java object is composed of a **header**, which contains information such as the pointer to the corresponding class, the object's monitor, etc., and a **data**, which contains the array of object's fields. If the object is an array, **data** contains the array elements. We make the following assumptions: (*i*) chunks are double-word (32-bit) aligned, thus CS is a multiple of 4. We call $B = \frac{CS}{4}$ the number of d-words of a chunk[2]; (*ii*) the chunk size is greater than the size of **header**, i.e. $size(header) < CS$; and (*iii*) **header** is structured in such a way as to be double-word aligned, we call $H = \frac{CS - size(header)}{4}$ the number of d-words left in a chunk after the object header. When an object is small, i.e. $size(header) + size(data) \leq CS$, a single memory chunk is enough; otherwise, the first part of the chunk is filled with **header** while **data** is placed in the remaining chunk part and in other chunks linked using a hierarchical structure of forwarding pointers as depicted in Figure 1. In particular, d-words from 0 to $H-4$ after object header store the corresponding elements of object's data, while d-words from $H-3$ to $H-1$ are used as single-, double- and triple-indirection links to other chunks, each one containing B data elements. Therefore, as detailed in Figure 1, d-word $H-3$ is a pointer to a chunk containing elements from $H-3$ to $H+B-4$, d-word $H-2$ points to a chunk containing *pointers* to chunks containing *elements*, etc.

[2] This choice is due to the fact that most of the JVM types are 4-bytes long (integer, floats, object and array pointers, etc.).

3 WCET for Object Operations

3.1 Object Allocation

In traditional heap management techniques, the time required to allocated a new object depends on the sizes of both the heap and the object to allocate. In the proposed approach we need to pick a number of chunks, from the free list, equal to:

$$1 + \left\lceil \frac{n - H + 3}{B} \right\rceil + \lfloor log_B(n - H + 3) \rfloor + \left\lfloor \frac{n - H + 3}{BB} \right\rfloor \quad (1)$$

where n is the number of object fields or array elements. This number depends only on object size and, if B is a power of 2, it can be easily calculated using bit-shift and *if* instructions.

Using the relation above, the WCET of the `new` bytecode can be exactly computed. The only exception is the use of the `Class.forName()` construct to load and instantiate a new object; in this case, the type of the object—and thus its size—is unknown until runtime and the WCET cannot be exactly computed: only an upper bound can be determined by assuming a reasonable maximum number of attributes that an object could have (in Juice, we assumed that an object cannot have more than 255 attributes).

3.2 Array Allocation

Java treats arrays as objects: an array of elements of type "T" is treated as an object of class "`[T`" ("array of `T`"). For this reason, the structure of an array, in Juice, is the same of an object, given that it has no attributes and the **data** part is used to represent array elements. Allocating an array, given that its size is known, implies to perform the same operations done for object allocation, and thus the calculation of the WCET is subject to the same formula 1. However, if the array size is known only at runtime (and this could happen very often), a different approach is needed. Indeed, this is a common problem of WCET computation in presence of dynamic arrays, and should be solved with other well-known techniques, such as by determining an upper bound, using annotation, etc. [1].

3.3 Reading/Writing Attributes

Reading and Writing object attributes is performed, in Java, by means of the bytecodes `getfield`/`getstatic` and `putfield`/`putstatic`.

Since Juice is a virtual machine for embedded system, Java classes are intended to be "ROM'ized". To this aim, Juice adopts an ahead-of-time pre-link and resolution process that, together with transforming classes into a ROMable representation, replaces each `get-/putfield` attribute with the "quick" version. Attribute index is thus referred to the array stored in the object. Given that attributes can be spread over different chunks, the access could require to navigate the chain of pointers. The code of such an operation is reported in Figure 2 for the `getfield` bytecode: as it can be seen, it is fast and its WCET can be exactly determined.

3.4 Juice Heap Layout and Garbage Collection

One of the main known issues that impede the use of Java in (hard) real-time environments is the presence of the garbage collector. The instants in which the GC is activated and the duration of its execution cannot be predicted, and thus any WCET/schedulability analysis is, in general, impossible. Such problems are overcome by the *Real-Time Specification for Java (RTSJ)* [2] with the introduction of *scoped memory.*

In our approach, the use of memory chunks greatly simplifies garbage collection, independently of the particular algorithm that is then used (reference-counting, three-color-marking, etc.). In fact, chunks are fixed-sized and free chunks are organized in a linked list, therefore *no compacting process is needed.* Collecting

```
dword getfield_quick (HOBJECT p, int index)
{
  dword * p1, * p2, * p3;
  if (index < (H - 3)) return p->data[index];
  index -= (H - 3);
  if (index < B) {
    // follow index at H - 3
    p1 = (dword *)p->data[H - 3];
    return p1[index];
  }
  index -= B;
  if (index < B*B) {
    // follow index at H - 2
    p2 = (dword *)p->data[H - 2];
    p1 = (dword *)p2[index / B];
    return p1[index % B];
  }
  // follow index at H - 1
  index -= B*B;
  int i0 = index / (B*B);
  int i1 = (index % (B*B)) / B;
  int i2 = index % B;
  p3 = (dword *)p->data[H - 1];
  p2 = (dword *)p3[i0];
  p1 = (dword *)p2[i1];
  return p1[i2];
}
```

Figure 2. Juice's `getfield` code fragment

an object no longer used implies to return the associated chunks to the free list, one-by-one, operation that can be performed also *incrementally*, n chunks per time. Using such a characteristic, the garbage collector of Juice[3] is designed to perform a *known number* of operations each time it is invoked. More specifically, the cost to pay when an object occupying n chunks has to be allocated is to ask the garbage collection to free, at most, n unreferenced chunks. With such an approach, execution of the GC is always tied to object allocation (i.e. when new free memory could be needed) and its duration can be predicted.

4 Known Issues

The heap management policy presented in this paper, even if it guarantees good allocation performances and predictability in WCET determination, suffers of two main problems: *limited number of fields/array elements* and *memory fragmentation.*

4.1 Limited `data` elements

As shown in Figure 1, the maximum number of elements the **data** part can refer is $H+BBB+BB+B-4$. In Juice, where we chose $B = 32$ (and thus $H = 24$), this limit is equal to 33844. It is enough for object's

[3] In the current implementation, the GC is based on a simple reference-counting

attribute, but it could be a problem for array allocation. A possible solution could be to increase B, but, as we will see in the following, this choice provokes an increment of memory internal fragmentation. The solution adopted in Juice is to flag large arrays with a bit in the **header**, and add another level of indirection in the **data** array. This implies an upper bound equal to $H + BBBB + BBB + BB + B - 5$, i.e. 1082419 elements when $B = 32$. This new upper bound can now be considered enough for embedded applications. However, the introduction of such a variation implies an additional cost in accessing array elements, which is useless when the limit of 33844 elements is not overcome by the application: thus, in Juice, a command-line flag is used to activate the "large array" option.

4.2 Memory Fragmentation

The proposed approach provokes both external and internal fragmentation. The former is due to the fact that an object could be spread over non-contiguous chunks: this does not fit the working scheme of a CPU cache and thus can lead to performance reduction. However, we remind that, in general, the use of caching could be a problem for (hard) real-time systems, since caches may introduce large jitters in CPU opcode executions thus affecting WCET calculation.

Internal fragmentation, derived from the unused space left in chunks, is instead more important, since it implies a reduction of the amount of available memory. For this reason, the value of B should be chosen in such a way as to find a good compromise between the allowed *maximum number* of object's attributes and array elements, which is $M = H + BBB + BB + B - 4$, and the degree of internal fragmentation. Figure 3 reports the trend of M and the amount of wasted memory, due to internal fragmentation, with respect to a value of B ranging from 16 to 128 d-words. The amount of wasted memory is measured considering 10, 50, 100 and 500 allocated objects with no attributes, thus producing the maximum fragmentation. As Figure 3 reports, the wasted memory with 500 objects, choosing $B = 32$ as in Juice, is approximatively 42 KBytes, a not-so-high cost to be payed for the use of fixed-sized memory chunks.

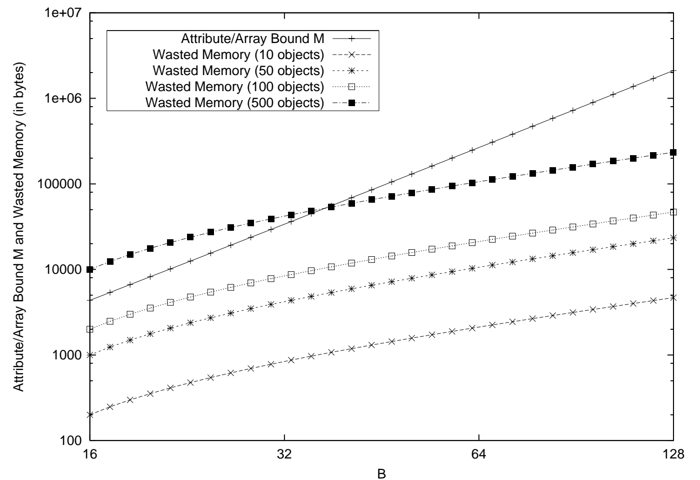

Figure 3. Wasted Memory due to Internal Fragmentation

5 Conclusions

This paper described the heap management and object allocation techniques employed in the Juice virtual machine. As it has been shown, the proposed approach was studied in order to have operations for object allocation and access not only optimized but, above all, with a predictable execution time, making possible the determination of WCET.

References

[1] I. Bate, G. Bernat, and P. Puschner. Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis. In *Proc. 5th IEEE ISORC 2002*, pages 83–90, Apr. 2002.

[2] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[3] A. Corsaro and C. Santoro. A C++ Native Interface for Interpreted JVMs. In *1st Intl. JTRES Workshop (JTRES'03).* LNCS 2889, Springer, 2003.

[4] S. M. Donahue, M. P. Hampton, M. Deters, J. Nye, R. Cytron, and K. Kavi. Storage allocation for real-time, embedded systems. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software: Proceedings of the First International Workshop*, pages 131–147. Springer Verlag, 2001.

[5] C. Santoro. *An Operating System in a Nutshell.* Internal Report, Dept. of Computer Engineering and Telecommunication, UniCT, Italy, 2002.

[6] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK), 1995.