Worst-Case Analysis of Heap Allocations *

Wolfgang Puffitsch[1], Benedikt Huber[1], and Martin Schoeberl[2]

[1] Institute of Computer Engineering, Vienna University of Technology, Austria
wpuffits@mail.tuwien.ac.at, benedikt@vmars.tuwien.ac.at
[2] Dept. of Informatics and Mathematical Modeling, Technical University of Denmark
masca@imm.dtu.dk

Abstract. In object oriented languages, dynamic memory allocation is a fundamental concept. When using such a language in hard real-time systems, it becomes important to bound both the worst-case execution time and the worst-case memory consumption. In this paper, we present an analysis to determine the worst-case heap allocations of tasks. The analysis builds upon techniques that are well established for worst-case execution time analysis. The difference is that the cost function is not the execution time of instructions in clock cycles, but the allocation in bytes. In contrast to worst-case execution time analysis, worst-case heap allocation analysis is not processor dependent. However, the cost function depends on the object layout of the runtime system. The analysis is evaluated with several real-time benchmarks to establish the usefulness of the analysis, and to compare the memory consumption of different object layouts.

1 Introduction

In hard real-time systems, failing to deliver results in time may lead to catastrophic consequences. Deadlines must be met even in worst-case scenarios. As such situations cannot be reliably provoked through measurements, hard real-time systems must be statically analyzed to ensure that all deadlines will be met. Scheduling analysis determines whether all tasks can meet their deadlines. The input for this analysis are the worst-case execution times (WCETs) of the individual tasks and their respective deadlines. However, it is not only important to consider the tasks' timing. An application that runs out of memory cannot deliver its result on time either. Therefore, it is important to bound not only the worst-case execution time, but also the worst-case heap allocations (WCHAs). With the known WCHAs, the memory management system, be it scoped memory or a real-time garbage collector (GC), can be correctly dimensioned.

Allocations are often "hidden" behind syntactic features or in libraries. Even innocent-looking expressions such as `println("info: "+i)` allocate several objects.

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

The expression "info: "+i is processed as follows: a StringBuilder object is allocated, which contains an array to store the actual characters. Then, "info: " is appended to that object, potentially allocating a larger character array. The integer i is also appended to that object; converting a number to its decimal representation requires another character array. Finally, the StringBuilder is converted to a String, allocating yet another object. In total, two objects and two to four arrays are allocated. Considering the simplicity of the above expression, manual analysis of realistic programs is clearly not an option.

Manual analysis is also complicated by the fact that the requested amount of memory is not always the same as the allocated amount. In object-oriented languages, objects include some meta-information about the type of an object to allow dynamic method calls. Some real-time GCs (RTGCs) split objects, either to avoid [20] or to overcome [5] fragmentation issues. Programmers must have intimate knowledge about the runtime-system to find out how much memory is actually allocated.

Knowledge about heap allocations is useful both when using a RTGC and when using scoped memory. Scoped memory was introduced in the real-time specification for Java RTSJ [3] to eliminate the need for garbage collection. As the size of the scoped memory area has to be provided when the area is created, it is important to know how much memory will be allocated in that scoped memory. The analysis helps to size the scoped memory area such that allocation demands can be met even in worst-case scenarios.

Real-time garbage collection has gained more acceptance since the RTSJ was formulated, and the use of scoped memory often can be avoided. Unlike a GC for general purpose systems, hard RTGCs require some knowledge about the application for correct operation. A RTGC must be paced correctly—otherwise it cannot keep up with the allocations from the application. In such a case the system would fail, either because it runs out of memory or because tasks are delayed beyond their deadline by the GC. The allocation rate alone is not enough to determine an upper bound for the period of the GC thread [14, 15]. However, the allocation rate is a necessary prerequisite for pacing the RTGC.

We propose to use the existing technologies for WCET analysis for the analysis of heap allocations and provide cost formulas for several object layouts. We evaluate the tighness of our approach by comparing the analysis results to measurements, and identify programming idioms that introduce pessimism.

The following section provides an overview of work related to this paper. Background on WCET analysis, on which the WCHA analysis builds upon, is given in Sec. 3. In Sec. 4, we present the analysis to automatically determine the WCHAs of tasks, which is evaluated in Sec. 5. Section 6 concludes the paper and provides an outlook on future work.

2 Related Work

An early attempt at automated computation of upper bounds for different performance measures was presented by Wegbreit [22]. The analyses are formulated for Lisp and computation takes place on a symbolic level. Recursion (which is also used to implement loops) leads to recurrence equations that are solved to derive results in a closed form. When aggregating worst-case results, the analysis must be conservative, and assume that all program fragments exhibit their worst-case behavior at the same time. This leads to results that are similar to the timing schema [18]. In contrast, our analysis uses a more powerful approach based on integer linear programming, which allows expressing cases where the execution of program fragments is not independent.

Unnikrishnan et al. presented analyses for both allocations and live memory [21], which are to some degree similar to the analyses by Wegbreit. However, recursion is handled implicitly rather than through explicit equations. The analyses are formulated for a first-order functional language and assume that the input programs are purely functional. It is not clear whether their findings can be directly applied to imperative languages such as Java.

Albert et al. [1] developed an analysis framework where a sub-set of Java bytecodes is transformed to a rule-based procedural representation. Loops are transformed into recursions, and recurrence equations are generated to characterize the program. The solution to these equations provides parametric bounds for the memory consumption.

The analysis presented by Braberman et al. [4] computes the memory usage of “regions” within a program. The memory consumptions of these regions are combined to derive symbolic bounds on the minimum memory consumption and the total amount of allocated memory.

Mann et al. [11] developed a data-flow analysis to determine worst-case allocation rates. They use an instruction “window”, and determine how much data is potentially allocated within such a window. Clustered allocations, as they are common at the start of a task’s period, can lead to considerable pessimism when using an instruction window. Considering the whole execution of a task, as our analysis does, levels out such allocation spikes.

3 WCET Analysis

WCET analysis, similar to many other program analyses, is performed on a program abstraction, the control flow graph (CFG). In a CFG the basic blocks are represented by vertices and the directed edges represent possible control flows. The cost of an edge is set to the maximum time needed to execute the basic block it originates from. WCET analysis needs to find the most expensive execution path between the program’s entry and exit node. In order to bound the execution time of a task or method, an upper bound for the number of times loops are executed and for recursion depths has to be known. Additionally, one aims to exclude infeasible paths, which are never taken but are part of the CFG abstraction.

A common technique to find the WCET is implicit path enumeration (IPET) [13, 10]. The problem of finding the most expensive execution path is transformed to a network flow problem. The variables of the problem correspond to the

```
for (int i = 2; i <= 10; i++) // outer loop
{
  for (int j = i; a[j] < a[j - 1]; j--)
  // @WCA loop <= 9
  // @WCA loop <= 45 outer
  {
    swap(a,j,j-1);
  }
}
```

Listing 1.1. Loop bound annotation example

execution frequency of CFG edges. Unique start and end vertices are created with a single outgoing and a single incoming edge with execution frequency one. For all other vertices, representing basic blocks in the CFG, the flow into the vertex is equal to the flow out of the vertex. Furthermore, linear constraints to bound the maximum number of loop iterations and to exclude infeasible paths are added. Each edge is assigned a constant execution cost. The problem of finding the WCET now amounts to finding the flow with maximal cost. The solution to the resulting integer linear programming (ILP) problem can be found by a standard ILP solver, such as lp_solve.[3]

WCET analysis usually calculates a safe bound of the real WCET. Due to simplifications of the processor model some features are modeled with conservative execution times. Furthermore, infeasible paths in the CFG can stay undetected by the tool. Elaborated annotation languages have been developed to reduce the pessimism due to infeasible paths [9].

3.1 Loop and Recursion Bounds

In order to bound the WCET, it is necessary to bound the maximum number of loop iterations and the maximum depths of recursions. Simple loop bounds can be automatically extracted from the program source. For this purpose, a data-flow analysis (DFA) framework providing a loop bound analysis is integrated in the WCET analysis tool [17].

However, if a bound cannot be determined automatically, programmers must provide annotations. An example for such an annotation is shown in Listing 1.1. The annotation @WCA loop<=9 tells the analysis that the loop body is executed at most 9 times whenever the loop is entered. The annotation @WCA loop<=45 outer further restricts the number of times the loop body is executed by stating that it is executed at most 45 times whenever the *outer* loop is entered. The analysis can therefore compute tight bounds for triangular and other non-rectangular loops.

[3] http://lpsolve.sourceforge.net/5.5/

3.2 Data-flow Analysis

The data flow analyses are run prior to the WCET calculation, and provide information to deal with dynamic dispatch (receiver type analysis) and cycles in the CFG (loop bound analysis). Both analyses are based on the techniques described in [12], and operate directly on Java bytecode.

Receiver Types. A receiver type analysis computes which types an object may actually have. This is useful to reduce the pessimism that is introduced to the WCET/WCHA analysis by virtual method calls. The term *receiver* refers to the object which *receives* a message through the method call.

Our receiver type analysis take call strings into account and is similar to k-CFA ("k^{th}-order control-flow analysis") [19]; a detailed description of the analysis can be found in [17]. We acknowledge that techniques like the ones described in [2] are more efficient than our approach. However, these techniques trade precision for analysis time; the amount of pessimism introduced by this loss in precision remains to be evaluated.

Loop Bounds. The loop bound analysis is based on an interval analysis that computes an upper bound for the values integer variables may hold. It is augmented with information whether a variable is only incremented or decremented. From the value range of a loop variable and its possible increments/decrements, it can be deduced how often a loop may be executed. As this analysis is not the focus of this paper, please refer to [17] for details of the analysis.

A by-product of the loop bound analysis is that it also computes ranges for array sizes. As the analysis computes ranges for all integer variables, it also computes ranges for the values that are passed to newarray, anewarray, and multianewarray. The only necessary change in the analysis was to keep those ranges available for further processing.

3.3 Execution Time Calculation

In addition to the high-level program analysis described above, the construction of a low-level timing model is necessary before calculating the WCET. The low-level analysis provides a bound on the execution time of basic blocks, and depends on the particular target platform. As the state of the instruction cache, the pipeline and other hardware components might influences the timing, non-local analyses of these components are necessary to obtain tight bounds.

Given the results of the low-level and high-level program analysis, a system of linear constraints is generated for each CFG. The objective function for the ILP problem is obtained by summing up the cost of all edges. The cost of an edge is the product of the edge's frequency (a variable) and the cost of executing the basic block the edge originates from. The latter is a constant obtained by the low-level timing analysis. Finally, the model is passed to an ILP solver, which calculates the edge's execution frequency on the worst-case path, as well as the WCET itself.

4 Heap Allocation Analysis

The WCHA analysis we propose is based on the WCET analysis described in Sec. 3. Instead of using the execution time as cost function for the analysis, we use the amount of memory a bytecode allocates. This implies that the infrastructure for calculating the WCET can be reused for calculating the WCHA. All path information obtained from value and loop bound analysis and the information extracted from annotations, is available to the allocation analysis as well.

Obviously, most bytecodes do not allocate any memory. The amount of memory a `new` bytecode allocates is determined by the type it allocates. The size of instances of that type are known at compile time, and can be obtained by adding the sizes of all fields for a given class and its superclasses. When allocating arrays, the allocation size is determined at runtime.

4.1 Array Size Bounds

To bound the maximum size of allocated arrays, the DFA has been extended to provide array sizes at the allocation site. During the loop bound analysis, ranges for all integer values have to be computed. As this also includes values on the stack, the analysis has been extended to record bounds for array allocations. When encountering a `newarray` or `anewarray` instruction, a mapping between the allocation site and the range of the appropriate stack location is added to an allocation bound table. For `multianewarray` instructions, the analysis must record multiple stack locations, to bound every level of the multidimensional array. When computing the cost of an array allocation, the appropriate mapping is retrieved from the allocation bound table.

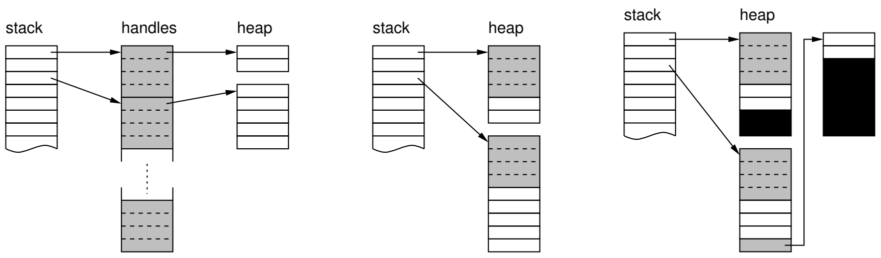
If the analysis does not succeed in computing a bound, annotations have to be provided by the programmer. An example for an array size annotation would be `arr = new int[n]; // @WCA size<=100`, where the array is annotated to occupy no more than 100 elements.

4.2 Object Layouts

When analyzing the WCET it is also necessary to include low-level details of the processor in the analysis. Some of the low-level architecture details, such as caches and branch predictors, are irrelevant for the WCHA analyis. However, the analysis must take into account the object layout of the underlying JVM.

Figure 1 shows object layout variants that are used in JVMs with RTGCs. White cells contain user data, while gray cells contain meta-information required by the runtime system. Black cells denote memory which cannot be used due to the object layout. The black cells are the result of internal fragmentation.

In a RTGC, it is necessary to either defragment memory (by relocating objects) or to avoid unbounded fragmentation. In a handle-based layout (Fig. 1(a)), a handle in a separate handle area points to the actual object. The handle area does not need compaction, because the fixed length of the handles eliminates fragmentation. Object relocation is simple, because only the indirection pointer



(a) Handles (b) Headers (c) Fixed Blocks

Fig. 1. Comparison of object layouts

in the handle has to be updated. While the cost for an object is similar to a header-based layout, it has to be taken into account that also the number of allocated objects has to be bounded in order to fit the handle area.

A layout that incorporates header data (Fig. 1(b)) into the object eliminates the indirection. However, when relocating objects, forwarding pointers have to be followed. On average, such a layout speeds up object accesses, but in the worst case, access times are similar to a handle-based layout due to the indirection through the forwarding pointer during object relocation. Such a layout also complicates defragmentation, because all references in the objects and on the thread stacks have to be updated to point to the new object location.

When using fixed block sizes for all allocations, external fragmentation can be eliminated, albeit at the expense of internal fragmentation. Such a layout is shown in Fig. 1(c). Objects that are too large to fit into a single block are organized as linked list. Arrays are organized as a tree to achieve logarithmic costs for accesses. Individual accesses may be more expensive than with the other object layouts. However, when considering the whole system, this is alleviated by the fact that no defragmentation is necessary.

4.3 Cost Functions

Different object layouts lead to different cost functions for the heap allocation analysis. Object fields can be allocated packed or at word boundaries. Object meta-data (e.g., object type, array size, object locks,...) can be organized differently to optimize for size or for speed. Furthermore, large objects can be split into constant sized blocks to avoid heap compaction or to make object relocation interruptible.

In the following, $\mathcal{F}(o)$ are the fields of an object o, and $\mathcal{F}_k(o)$ is the k-th field of object o (with indices starting at 1). With $s(f)$ we denote the size of a field f in bytes, and with $a(f)$ the required alignment for the field f. For the total memory usage of an object o we write $mu(o)$.

To handle alignment requirements, we define $P(n, m)$ such that it pads the address n to a multiple of m.

$$P(n, m) = \left\lceil \frac{n}{m} \right\rceil m$$

$S(n, f)$ returns the memory usage of an object after adding a field f to that object at relative address n. For example, $S(3, f)$ would evaluate to 8 for a 4-byte field f that requires alignment to 4-byte boundaries.

$$S(n, f) = P(n, a(f)) + s(f)$$

Handle-based Layout. In a handle-based layout, header data is stored at the handle site, while the payload is located in the remaining heap space. Together, the handle and the payload must fit the total available memory. Furthermore, it is necessary that the handle area is large enough for the handles, and the rest of the heap is large enough for the object data.

The memory usage of an object can be computed with the following formulas:

$$\begin{aligned}
mu_h(o) &= s(\mathit{handle}) \\
mu_f^0(o) &= 0 \\
mu_f^k(o) &= S(mu_f^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\
mu_f(o) &= P(mu_f^{|\mathcal{F}(o)|}(o), A_{field}) \\
mu(o) &= m_h(o) + mu_f(o)
\end{aligned}$$

We assume that handles are always aligned, and that any required padding or unused fields are part of $s(\mathit{handle})$. Furthermore, the formulas assume that object fields start at an address that never requires padding. The maximum alignment for object fields is denoted by A_{field}. By padding the end of the user data to such an alignment boundary, we ensure that the next object starts at this boundary and its first field indeed does not require padding.

The equations for arrays are the same as for objects, except that arrays use an *array_handle* instead of a *handle*. The handle types can differ, because the *array_handle* must accomodate the size of the array and type information may be treated differently.

When being interested in the overall memory consumption, $mu(o)$ is the appropriate cost function. When considering the handle area and the remaining heap space separately, $mu_h(o)$ and $mu_f(o)$ provide the respective cost functions.

Layout with Header Data. Header data is located in the same place as the payload. The memory usage can be computed as follows:

$$\begin{aligned}
mu^0(o) &= s(\mathit{header}) \\
mu^k(o) &= S(mu^{k-1}(o), \mathcal{F}_k(o)) && k > 0 \\
mu(o) &= P(mu^{|\mathcal{F}(o)|}(o), A_{header})
\end{aligned}$$

Again, we assume that objects start appropriately aligned and add padding at the end of the object as required. The equations for arrays are the same, except that an *array_header* is used instead of *header*.

Fixed-Block Layout. In the fixed-block object layout, the header data and the start of the object are in the same block. If the header data and the payload exceed the size of a single block, the object is split across several blocks. The link to the next block may be located at the start or the end of a block, which leads to slightly different equations for the memory usage computation. With B we denote the size of a block; we assume that fields never require padding at the beginning of a block.

Next Pointer at End of Block If the pointer to the next block is located at the end of a block, it must be checked whether the field and the *next* pointer fit the current block when adding a field to an oject. The function S_{next} returns a value greater than B if these two fields do not fit the current block.

$$S_{next}(n, f) = S(S(n \bmod B, f), next)$$

The function S_{block} uses S_{next} to determine the actual memory usage when adding field f at position n.

$$S_{block}(n, f) = \begin{cases} S(n, f) & \text{if } S_{next}(n, f) \le B \\ P(n, B) + S(0, f) & \text{if } S_{next}(n, f) > B \end{cases}$$

Next Pointer at Beginning of Block If the *next* pointer is located at the beginning of a block, S_{next} and S_{block} have a slightly different definition:

$$\begin{aligned} S_{next}(n, f) &= S(n \bmod B, f) \\ S_{block}(n, f) &= \begin{cases} S(n, f) & \text{if } S_{next}(n, f) \le B \\ P(n, B) + S(s(next), f) & \text{if } S_{next}(n, f) > B \end{cases} \end{aligned}$$

In this flavor of the fixed-block object layout, the *next* pointer must taken into account for all blocks. This is especially true for the first block; the *next* pointer for this block is considered as part of the object header.

Memory Usage of Objects The memory usage for objects can be computed with the following formulas:

$$\begin{aligned} mu^0(o) &= s(header) \\ mu^k(o) &= S_{block}(mu^{k-1}(o), \mathcal{F}_k(o)) & k > 0 \\ mu(o) &= P(mu^{|\mathcal{F}(o)|}(o), B) \end{aligned}$$

The formulas are the same for both flavors of the fixed-block object layout; the placement of the *next* pointer is already taken into account by S_{block}.

Memory Usage of Arrays Arrays have a special header, that includes the size of the array and depth of the tree representation. As all fields are the same size and have the same padding requirements, we do not need to use a recursive definition for the memory consumption. $L(a)$ captures how many array fields fit into a single block. $N(a)$ is the number of blocks the array elements occupy. M is the number of *next* pointers within an inner node of the tree representation.

$$L(a) = \left\lfloor \frac{B}{s(\mathcal{F}_1(a))} \right\rfloor \qquad N(a) = \left\lceil \frac{|\mathcal{F}(a)|}{L(a)} \right\rceil \qquad M = \left\lfloor \frac{B}{s(next)} \right\rfloor$$

$$\begin{aligned} depth(a) &= \lceil \log_M(N(a)) \rceil \\ mu^0(a) &= s(array\_header) \\ mu(a) &= \begin{cases} P(mu^0(a), B) & \text{if } |\mathcal{F}(a)| = 0 \\ P(mu^0(a), B) + \sum_{i=0}^{depth(a)} B \left\lceil \frac{N(a)}{M^i} \right\rceil & \text{if } |f(a)| > 0 \end{cases} \end{aligned}$$

5 Evaluation

To evaluate the heap allocation analysis, three benchmarks and two applications are analyzed and the memory consumption is compared to measurements of the five applications on the target JVM.Measurements cannot reliably capture the the worst-case behavior; comparing the analysis results with measurements only hints at bounds that might be overly pessimistic.

In order to compare our work with existing memory allocation analyses, we use the JOlden benchmark suite [6], which was also used in [4, 1]. We use the subset of the benchmarks that does not require recursion and hence can be analysed by our tool. The benchmarks were modified such that they do not get their parameters via the command line arguments. We cannot compute a worst-case bound for unknown input values and hence initialize the appropriate variables internally. Where the DFA could not find loop bounds, we provided manual annotations.

The first application we evaluate is based on the demo application presented in [5].[4] It emulates a multi-threaded financial transaction system, which must react to market changes within a bounded amount of time. For the evaluation, we chose the methods MarketManager.onMessage() and OrderManager.checkForTrade(), both of which perform core functionality of the respective thread.

The application was adapted in three ways: First, the execution model of our execution platform is closer to the thread model of safety-critical Java [7], than the thread model of the RTSJ. The thread management therefore had to be reorganized. Second, our platform does not support the libraries for receiving and transmitting messages. This part had to be rewritten such that messages are read from and sent to standard in- and output. Third, we used string buffers

[4] We thank Eric Bruno and Greg Bollella for open-sourcing this demo application. It is available at `http://www.ericbruno.com`.

Table 1. Analysis and measurement results

		Allocated Objects		Allocated Words	
Benchmark	Method	Analysed	Measured	Analysed	Measured
MST	MST.main()	242	221	501	459
Em3d	Em3d.main()	814	805	11627	7298
BH	Tree.createTestData()	464	464	1954	1954
Trading	MarketManager.onMessage()	8	8	4104	2004
Trading	OrderManager.checkForTrade()	35	29	7876	741
CD_x	Main.run()	197936	22907	590150	73841

without automatic resizing where suitable. The reasoning behind this is discussed in Sec. 5.3.

The second application used for evaluation was extracted from the CD_x benchmark for RTSJ [8]. The source code of the original benchmark is freely available.[5] We analyze the real-time thread responsible for collision detection of airplanes.The benchmark in its original form is unsuitable for WCET analysis, as it makes heavy use of hash tables, which have poor worst-case performance. For heap allocation analysis on the other hand, the benchmark is both challenging and, with a few modifications, within the capabilities of our tool.

We first adopted the benchmark to meet the requirements of our target platform. The recursive voxel intersection procedure, which needs large amounts of stack space, was replaced by an efficient iterative version. The number of planes and other constants were reduced to meet the memory restrictions of our embedded system. For the analysis it was necessary to annotate some loops that use iterator objects, as these are beyond the capabilities of our data flow analysis.

5.1 Results

The analysis results for the six benchmark methods is shown in Tab. 1. In order to keep the pessimism within reasonable bounds, we used a modified version of the JDK suitable for real-time applications. It requires to pass a maximum capacity for lists and maps in the constructor, and prohibits on-demand reallocations, which cannot be handled by the analysis (see Sec. 5.3 for a discussion of the respective issues). Furthermore, the results for the trading engine application were obtained under the assumption that input messages are at most 1024 characters long. This is enforced by the input routines, but not a constraint that is visible at the application level. For unbounded input messages, the memory consumption of the trading engine would not be boundable.

Table 1 compares the results of the analysis with the results of a measurement; the numbers in the last two columns of this table refer to the raw amount of memory allocated by the application, excluding object meta-data. Results

[5] `http://adam.lille.inria.fr/soleil/rcd/`

that take into account the overhead of the object layout are discussed in the following section.

The figures in Tab. 1 show that the analysis yields relatively tight results for some benchmarks, while introducing a considerable pessimism for others. One reason for this pessimism is the fact that the measurement is not guaranteed to actually trigger the worst case. Some of the pessimism is however introduced by the analysis itself.

The three benchmarks from the JOlden benchmark suite are relatively simple and their execution is independent from input data. The analysis can therefore find reasonably tight bounds. The figures for the object count are tighter than the figures for the memory consumption because array sizes are overestimated at a few occasions. The pessimism for the object count is similar to the pessimism reported in [4] for these benchmarks.

The analysis results for the MarketManager.onMessage() method are off by a factor of around two. The method parses the input string for a name and a price and updates the market price of a traded item accordingly. Although the measurement was performed with a message that was designed to trigger as much memory allocation as possible, the analysis fails to find tight bounds on the string variables and assumes that all strings are 1024 characters long.

OrderManager.checkForTrade() shows considerably more pessimism. This is mainly caused by conversions from numbers to strings. Within these conversions, the analysis is not able to bound the length of the result string and assumes that such strings are 1024 characters in size. It is notable that the number of objects is overestimated by only about 20%, but the number of allocated words by about an order of magnitude.

The heap allocations reported for the collision detector thread of the CD_x benchmark are relatively high. The main reason for the overestimation is that it is difficult to find tight bounds for all collection sizes and loops in the benchmark, and that the tool does not yet support context dependent manual annotations. On the other hand, the benchmark showed that the analysis scales up for larger programs, and helped us to identify many problematic language constructs which complicate the analysis.

5.2 JVM Comparison

To compare the effects of different object layouts, we variated the cost function for the WCHA analysis as described in Sec. 4.2. The results are shown in Tab. 2. We assume 4 words for header data, and a blocks size of 8 words. We also include the number of allocated objects in Tab. 2, as this number is crucial to correctly dimension the handle area for a handle-based object layout.

As it is the case on our evaluation platform, the Java Optimized Processor (JOP) [16], fields are always stored at word boundaries and do not require any further padding. Due to the simple model for alignment requirements, a handle-based layout and a header-based layout consume the same amount of memory. The heap has to be large enough to fit the number of words given in the “Han-

Table 2. Analysis results for different object layouts

			Allocated Words		
Benchmark	Method	Objects	Raw	Handles/Headers	Blocks
MST	MST.main()	242	501	1469	2040
Em3d	Em3d.main()	814	11627	14883	22776
BH	Tree.createTestData()	464	1954	3810	5768
Trading	MarketManager.onMessage()	8	4104	4136	4768
Trading	OrderManager.checkForTrade()	35	7876	8016	9528
CD_x	Main.run()	197936	590150	1381894	1915336

dles/Headers" column. Similarly, the total memory consumption of a fixed-block layout is provided in the "Blocks" column.

For the handle-based layout, not only the total amount of memory must fit the heap, but also the handle area has to be dimensioned correctly. The number in the "Object Count" column times the handle size has to fit the handle area, and the rest of the heap has to be large enough to fit the number of words given in the "Raw" column.

When relating the total amount of allocated memory to the number of allocated objects, the trading engine benchmarks differ considerably from the other benchmarks. While the former allocate a few relatively large objects (mostly arrays), the latter allocate a many small objects. The overhead for the header data is therefore considerably higher for these benchmarks than for the trading engine benchmarks.

Using a fixed-block layout increases the memory consumption by 15 to around 50%, when comparing it to a simple header layout. A GC that uses such a layout must be considerably more efficient in other areas to make up for this increased memory consumption.

5.3 Programming Style

During the evaluation of the analysis, we encountered several times that relatively simple operations resulted in seemingly excessive memory allocations. A closer look at these operations revealed that this was due to the automatic resizing of data structures. Except for a few special cases, the analysis assumed that such a resizing would always occur. For example, when appending characters to a StringBuffer, the analysis assumed that the array to hold the actual characters would be resized for each invocation of append(). Converting a float to a String was reported to allocate several megabytes of memory, instead of a just few dozen words. Similar effects were observed for other common data structures of the Java library, such as ArrayLists.

Such situations can be circumvented in two ways. On the one hand, some data structures exhibit more analysis-friendly behavior than others. For example, adding an element to a LinkedList requires only the allocation of a single list element. The drawback of this solution is that such data structures do not always have the desired performance characteristics. On the other hand, it is sometimes possible to size the data structure upon allocation such that no resizing is necessary. However, this solution requires programmers to correctly predict the sizes of data structures. We believe that further research is necessary to find a suitable tradeoff between analyzable memory allocation and ease of use for the respective data structures.

6 Conclusion

Bounds for the worst-case heap allocations of real-time tasks are needed to correctly size scoped memories or to calculate the maximum period of the GC task. We have adapted technologies from the WCET analysis field to analyze the heap allocations of tasks. Instructions that allocate memory get a cost equivalent to the the size of the allocated data structure. All other instructions have zero cost. Analyzing the program with those costs gives the maximum memory allocation for a task instead of its maximum execution time.

We have shown that our analysis can find reasonably tight bounds for moderately complex programs. However, more realistic Java programs that are not explicitly designed for real-time systems are hard to analyze and result in considerable pessimism for the WCHA bounds. As future work we plan to investigate the right Java based programming style for real-time applications. Furthermore, we will investigate better analyzable replacements for library elements of the JDK.

References

1. Albert, E., Genaim, S., Gómez-Zamalloa Gil, M.: Live heap space analysis for languages with garbage collection. In: ISMM '09: Proceedings of the 2009 international symposium on Memory management. pp. 129–138. ACM, New York, NY, USA (2009)
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 324–341. ACM, New York, NY, USA (1996)
3. Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M.: The Real-Time Specification for Java. Java Series, Addison-Wesley (Jun 2000)
4. Braberman, V., Fernández, F., Garbervetsky, D., Yovine, S.: Parametric prediction of heap memory requirements. In: ISMM '08: Proceedings of the 7th international symposium on Memory management. pp. 141–150. ACM, New York, NY, USA (2008)
5. Bruno, E.J., Bollella, G.: Real-Time Java Programming: With Java RTS. Prentice Hall PTR, Upper Saddle River, NJ, USA (2009)
6. Cahoon, B., McKinley, K.S.: Data flow analysis for software prefetching linked data structures in java. In: PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques. pp. 280–291. IEEE Computer Society, Washington, DC, USA (2001)

7. Henties, T., Hunt, J.J., Locke, D., Nilsen, K., Schoeberl, M., Vitek, J.: Java for safety-critical applications. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009). York, United Kingdom (Mar 2009)
8. Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., Vitek, J.: Cdx: a family of real-time java benchmarks. In: JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems. pp. 41–50. ACM, New York, NY, USA (2009)
9. Kirner, R., Knoop, J., Prantl, A., Schordan, M., Wenzel, I.: Wcet analysis: The annotation language challenge. In: Rochange, C. (ed.) 7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2007)
10. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, & tools for real-time systems. pp. 88–98. ACM Press, New York, NY, USA (1995)
11. Mann, T., Deters, M., LeGrand, R., Cytron, R.K.: Static determination of allocation rates to support real-time garbage collection. SIGPLAN Not. 40(7), 193–202 (2005)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1999)
13. Puschner, P., Schedl, A.: Computing maximum task execution times – a graph-based approach. Journal of Real-Time Systems 13(1), 67–91 (Jul 1997)
14. Robertz, S.G., Henriksson, R.: Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In: LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems. pp. 93–102. ACM Press, New York, NY, USA (2003)
15. Schoeberl, M.: Real-time garbage collection for Java. In: Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006). pp. 424–432. IEEE, Gyeongju, Korea (April 2006)
16. Schoeberl, M.: A Java processor architecture for embedded real-time systems. Journal of Systems Architecture 54/1–2, 265–286 (2008)
17. Schoeberl, M., Puffitsch, W., Pedersen, R.U., Huber, B.: Worst-case execution time analysis for a Java processor. Software: Practice and Experience 40/6, 507–542 (2010)
18. Shaw, A.C.: Reasoning about time in higher-level language software. IEEE Trans. Softw. Eng. 15(7), 875–889 (1989)
19. Shivers, O.: The semantics of scheme control-flow analysis. In: PEPM '91: Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 190–198. ACM, New York, NY, USA (1991)
20. Siebert, F.: Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages. No. ISBN: 3-8311-3893-1, aicas Books (2002)
21. Unnikrishnan, L., Stoller, S.D., Liu, Y.A.: Automatic accurate stack space and heap space analysis for high-level languages. Tech. Rep. 538, Indiana University (Apr 2000)
22. Wegbreit, B.: Mechanical program analysis. Commun. ACM 18(9), 528–539 (1975)