Toward Libraries for Real-time Java*

Trevor Harmon
Electrical Engineering and Computer Science
University of California, Irvine
tharmon@uci.edu

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Raimund Kirner
Institute of Computer Engineering
Vienna University of Technology, Austria
raimund@vmars.tuwien.ac.at

Raymond Klefstad
Electrical Engineering and Computer Science
University of California, Irvine
klefstad@uci.edu

Abstract

Reusable libraries are problematic for real-time software in Java. Using Java's standard class library, for example, demands meticulous coding and testing to avoid response time spikes and garbage collection. We propose two design requirements for reusable libraries in real-time systems: worst-case execution time (WCET) bounds and worst-case memory consumption bounds. Furthermore, WCET cannot be known if blocking method calls are used. We have applied these requirements to the design of three Java-based prototypes: a set of collection classes, a networking stack, and trigonometric functions. Our prototypes show that reusable libraries can meet these requirements and thus be viable for real-time systems.

1 Introduction

General-purpose software libraries, such as the Standard Template Library for C++, the Base Class Library for .NET, and the Class Library for Java, provide helpful functions, such as sorting lists or parsing strings. But since they are designed for good *average-case* performance, rather than to minimize *worst-case* performance, their highly variable execution times are unsuitable for real-time systems.

Adapting general-purpose libraries for real-time systems is even more problematic with the Real-Time Specification for Java (RTSJ) [1]. Java libraries often rely on unpredictable Java idioms and design patterns. In the RTSJ's scoped memory model, using objects in mixed contexts, either shared by heap and non-heap objects or accessed from different memory areas, causes illegal assignment errors and memory leaks. Efforts such as the Javolution [2] library mitigate these problems, but its reliance on exception handling and dynamic memory allocation hinders formal analysis techniques to ensure timeliness.

In a hard real-time system, all tasks must meet known deadlines. Meeting deadlines is guaranteed by the system's design and by schedulability analysis. A library, in order to provide this guarantee, must meet two requirements: 1) it must have predictable worst-case execution time (WCET), which further requires non-blocking functions, and 2) it must have predictable worst-case memory consumption.

Known Execution Time. To be useful for real-time systems, all operations in the library must be statically analyzable for WCET. The maximum bound of any loop must be known in order to calculate the WCET; information about all loop bounds must therefore be incorporated into the real-time library. This information allows the WCET analysis tool we developed, called Clepsydra [3], to verify timeliness of individual tasks.

In addition, real-time systems must avoid all blocking calls to the operating system. The unknown time of blocking calls defeats schedulability analysis. Instead, real-time systems must use periodic tasks, a form that is easily analyzable.

Known Memory Consumption. Time-predictable execution of tasks can usually be guaranteed only when virtual memory management is disabled. Therefore, a guarantee must be made that the application never runs out of memory. In particular, dynamic memory management must be carefully analyzed: the maximum stack size of each thread and interrupt handler must be determined, and the individual sizes must be included in the link process.

Without dynamic creation of objects, Java becomes a restricted subset of itself. Recent work on real-time garbage

*The work on the Canteen library was supported in part by the Wiener Innovationsförderprogramm für betriebliche Forschung & Entwicklung – Call IKT Vienna 2004.

collection (GC) tries to relax this restriction [10, 4, 9]. However, the allocation and deallocation rate must still be known so that the garbage collection thread can be scheduled.

Three Real-time Library Prototypes. To show that a library can meet all of the above requirements, we have built three prototype libraries: 1) a library of collection classes, 2) a non-blocking network stack, and 3) a set of trigonometric functions. These prototypes vary in the techniques they must use to meet real-time requirements: collection classes are memory intensive; network stacks must also eliminate extensive use of blocking functions; complex mathematical functions are computationally intensive with unbounded loops. Collectively, these three varying prototypes demonstrate that predictable real-time Java libraries are possible.

This work is built on JOP [7], a predictable platform consisting of a Java-only processor without any operating system, and uses Clepsydra [3], a WCET analyzer for JOP and other Java-based processors. JOP provides a predictable platform because, unlike most microprocessors, every instruction on JOP has a known, bounded execution time. Clepsydra can therefore immediately and automatically display the WCET for each line of code as it is written, provided that the maximum number of iterations for each loop is known. (For loops with variable iterations, the programmer must insert an annotation with a constant for the maximum number of iterations.)

2 Analyzable Collection Classes

Our first real-time library prototype is a library of collection classes called Canteen, which offers features for convenience, type-checking, and compatibility.[1] Canteen implements the same List, Set, and Map interfaces declared in Java's standard library, allowing them to act as drop-in replacements for the standard collection classes. Canteen also achieves running time complexities that are asymptotically identical to standard non-real-time algorithms, demonstrating that performance need not be sacrificed to ensure predictability.

Canteen provides the following four implementations of the three most common types of collection interfaces (List, Set, and Map):

PredictableArrayList A simple sequence that allows precise control over element ordering. It allows multiple entries of the same element, including null. It is implemented as a simple linear array.

PredictableLinkedList Identical to the PredictableArrayList but implemented as a linked list.

PredictableTreeSet A sorted collection that guards against duplicate elements. As the name implies, it models the mathematical abstraction known as a set. It is implemented as a red-black tree.

PredictableTreeMap A sorted dictionary-type collection that maps keys to values. Each key can map to at most one value. It is also implemented as a red-black tree.

2.1 Known Execution Time

Since predictability of performance is the key factor in a real-time library, Canteen's collection classes are analyzed with Clepsydra to calculate the known WCET. For loops, we use source code annotations to declare the loop's maximum number of iterations.

The bound of almost every loop in a collection class depends on the maximum size of a particular instance of that collection class. For example, if two list objects have maximum sizes of 10 and 100, then the loop bound of a search operation for the latter list will be 10 times more than the former. In other words, the loop bound annotation of a collection class method cannot be specified using a simple constant. To address this issue, loop bound annotations in Canteen are expressed using specially named constants. For example:

```
// User code
@CollectionBound(max=1024)
private PredictableList<Int> list;
...

// Canteen code
class PredictableList ... {
  ...
    @LoopBound(max=COLLECTION_BOUND)
    for (int i = 0; i < array.length; i++)
      ...
```

To preserve the intended logarithmic time for all operations in Canteen's Set and Map classes, we developed a *list-tree hybrid* data structure. It consists of a normal binary tree whose entries contain the usual left and right pointers to child nodes. In addition, each entry also contains previous and next pointers to form the doubly-linked list of a memory pool; this allows entries to be removed and returned to the memory pool in constant time. The red-black tree algorithms in Canteen were then modified so that these list pointers are properly updated across all rebalancing operations. Figure 1 shows an example of how the list-tree hybrid is altered by the removal of an element.

2.2 Known Memory Consumption

To satisfy the requirement of known memory consumption, Canteen allocates no memory during the mission phase. This approach prevents unpredictable delays caused by the GC and allows total memory consumption to be known immediately after initialization. Instead of dynamic memory management, Canteen relies on static memory

[1]Distributed under an open-source license as part of the Volta project: http://volta.sourceforge.net/

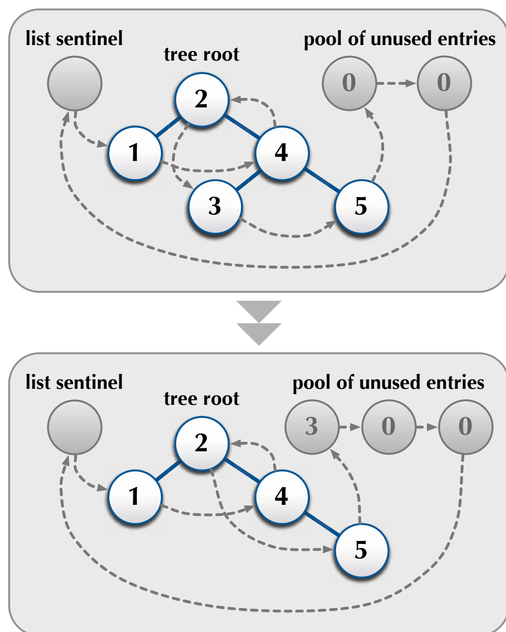

Figure 1. This hybrid list-tree data structure allows binary trees to use memory pools. Element #3 is removed from a tree of current size 5 and maximum size 7. The element is returned to the pool, and the list pointers are updated accordingly.

pools to retrieve pre-allocated elements to be added to a collection. When an element is removed from the collection, Canteen returns it to the collection's pool for later use. The maximum size of a pool must be programmer-specified during instantiation, and the effect of exceeding this maximum is undefined.

Implementing memory pools in the List classes is straightforward: the memory pool is stored at the unused end of the preallocated array. For PredictableTreeSet and PredictableTreeMap, however, supporting memory pools is much more involved. These classes are backed by a red-black tree structure, and the memory pool must be maintained across the elaborate rebalancing operations that occur after adding or removing elements in the tree.

3 A Non-blocking Network Stack

Our second prototype of a real-time Java library is an implementation of a TCP/IP stack called *ejip* (Embedded Java Internet Protocol).[2] The lowest level of the stack contains device drivers for Ethernet (a CS8900 chip driver adapted from the Linux sources), SLIP, and a PPP implementation. At the next level, IP, UDP[3], and a restricted subset of TCP are implemented. The restriction is that we handle only one packet on the fly. TCP is a streaming protocol where blocking operations are natural; as a proof of concept, we implemented a TCP handler on port 23—the Telnet port—on top of TCP/IP. On top of that stack, a TFTP server, a simple web server, and a minimal telnet server are available.

Ejip meets real-time requirements: all of its operations are WCET analyzable, and its memory consumption is known before runtime. A key challenge to achieve known execution time was to eliminate blocking method calls. In the following sections, we describe how ejip is designed and implemented to meet real-time requirements.

3.1 Known Execution Time

In a general-purpose system, read from and write to an I/O channel are designed as blocking I/O operations. By contrast, however, ejip is designed as a set of periodic activities: the periodic task model is used for all layers of the stack. We do not use any asynchronous notification mechanisms, such as interrupts, in the stack. Waiting for an operation to finish has to be performed at the level of the application.

At the lowest layer—the network layer—the Ethernet driver polls the Ethernet chip and the packet buffer within a periodic loop. It polls for packets that are in the Ethernet chip buffer and also for packets that are ready to send. A received packet is put into the packet pool.

The next layer, the IP layer, periodically polls the packet pool. When a pending packet is received from the network layer, it processes one packet at each iteration and hands them over to the next layer (ICMP, UDP, or TCP).

Normal blocking to wait on a TCP packet to receive an application-defined message does not always work as expected. One can never be sure that a complete packet is received with one blocking read operation. It is possible to get less than the application-defined message, in which case a second (or more) blocking read operation is necessary to retrieve the whole message. This behavior is ignored in some applications, which rely on the chance that a single message (that is short enough) will not be split by the transport layer. However, this is not guaranteed. For correct handling of packet splitting, reassembly of application messages would be necessary at the application level. That reassembly is similar to our API. We provide the application with the raw TCP packets as they are received at the network interface. The application code must split and reassemble them into the application messages.

Handling more complex application protocols can be performed by offloading the work to another thread. The handler simply does the preprocessing and hands the packet over to the other thread.

The WCET is known, as we showed with an earlier ver-

[2] Ejip is available as part of the JOP distribution at `http://www.opencores.org/`.

[3] UDP is used for an application-designed protocol of a soft real-time application in the railway domain described in Section 3.3.

sion [8]. The protocol stack processes only one packet per period, using protocol-specific handlers. The WCET of the packet processing thread is the WCET of the slowest handler. For example, UDP-based applications register a UDP handler which includes the listening port number; when a UDP packet arrives, the corresponding handler is invoked. If no corresponding handler is found, the packet is dropped. The TCP handler works in a similar way.

Furthermore, all loops are bounded, and no recursion is used. We annotate all loops with the maximum loop bound. In previous WCET analysis of an earlier version of ejip, the WCET is excessively conservative for small packets, since the loops for packet processing are bounded by the maximum Ethernet packet size [8]. A data-flow analysis or special versions of some functions with a smaller packet size can help to reduce the pessimism. Nevertheless, we achieve the goal of safe bounds on execution time.

3.2 Known Memory Consumption

Ejip uses a packet buffer pool in which all buffers are allocated at initialization time. The packet buffer management plays a central role in the TCP/IP stack, providing free buffers, buffers that are received, and packets that are ready to send. The number of buffers is controlled by a constructor parameter. Each layer merely grabs the packets that it expects, works with them, and puts them back into the pool.

All memory allocation is bounded. A buffer can contain a single Ethernet packet, and the number of packet buffers is fixed. Without an OS and due to the safety of Java, we do not need any memory protection or OS protection levels. Therefore, we implemented a zero-copy TCP/IP stack. The data is read at the network interface layer (Ethernet or serial line) into a packet buffer and moved up the whole stack without any copy. When the application consumes the data, it releases the packet buffer. For a simple UDP-based command/response protocol, the buffer can also be reused and passed back down the network stack.

This conservative memory management is common in real-time systems. It eliminates analysis of mission-phase memory allocation and fits the upcoming standard for safety-critical Java (JSR 302) prohibiting a GC [5].

3.3 An Example Application

In a soft real-time application for the railway industry, we use the UDP function heavily. An end module in each locomotive is equipped with a GPRS modem. That module receives commands from the master station and sends its location data to the master station.

The flow control between the master station and the module is handled by the application. Each command or message is acknowledged by the receiver through a reply packet. The sender is responsible for retransmission when the message or the acknowledgment packet is lost. This protocol relieves the receiver from handling retransmission.

Retransmission is performed after a fixed timeout and a fixed number of tries. A failure after a fixed number of retries is reported to the uppermost layer, the user.

All these timeouts and retries are implemented in periodic tasks. Therefore, the code is time-predictable. The Internet and GPRS do not provide real-time capabilities; however, we handle this non real-time property at the communication level with the application code. We *hide* it to some extent, but also *expose* it when all retries fail. In that case, the user must find a way around the problem.

Writing a communication application in the periodic style is unusual and requires some adaptation in programming style. Our experience in writing the modem communication and the PPP link negotiation phase as a periodic task showed that this programming style is not trivial. The complexity is in the state of the link layer. Several dependent state machines are triggered by events such as timeouts, modem responses, and link negotiation protocol responses.

4 Trigonometric Functions

The third type of real-time library functions we implemented are trigonometric functions. We chose the trigonometric *cosine* function as a representative of this class of mathematical library functions; others can be calculated similarly, for example, by shifting the phase of the input of the cosine by $-\pi/2$ to give the *sine* function. In previous research we have compared two different ANSI C implementation styles that are time-predictable and memory-predictable: 1) calculation based on Taylor series, and 2) precomputed values based on lookup tables [6].

4.1 Known Execution Time

The calculation based on Taylor series with polynomials of degree $2n$ is computed as follows:

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \cdot \frac{x^{2n}}{(2n)!} \approx \sum_{n=0}^{7} (-1)^n \cdot \frac{x^{2n}}{(2n)!} \qquad (1)$$

For *double* variables a value of $n_{max} = 7$ is sufficient. To avoid any loop, in practice the iterative calculation of above formula is calculated directly. We have implemented instances with $n+1$ of at maximum 2, 4, 6, and 8, where 8 is used by standard *double* implementations. (We give the maximum value of $n+1$ because n starts at zero.)

We implemented the lookup-table approach with interpolation (cos_lutinterp) and without (cos_lutprim). Both implementations do not contain any loop constructs.

	et_{min} [cycles]	et_{max} [cycles]
cos8	50,971	813,393
cos6	48,623	650,992
cos4	47,132	530,042
cos2	45,433	449,033
cos_lutprim	46,060	688,514
cos_lutinterp	371,010	1,052,518

Figure 2. Execution Time of Cosine Variants.

As both implementation styles are without loops, analyzing the WCET is rather simple. Clepsydra does not yet support analysis of methods using float values. Therefore, we instead measured the execution times for these functions on JOP, as shown in Figure 2.[4] Note that the high variation in execution time is due to the optimizations for argument values of zero in our current floating-point emulation used in the Java execution environment.

Our previous work using lookup-table implementations based on double values in ANSI C showed them to be an interesting alternative to iterative calculations based on Taylor series with respect to WCET [6]. However, as shown in Figure 2, there was no such benefit for our float implementations in Java, due to the inefficient array initialization in Java 1.6.

On the other hand, our experiments have shown that the limited iterative approach for computing a Taylor series is predictable and efficient in Java.

4.2 Known Memory Consumption

Both approaches, the limited iterations and the precomputed lookup tables, have known memory usage. Implementations based on the Taylor series have a small code size and do not allocate data on the heap. As both approaches lack recursive function calls and loops, memory consumption is easily analyzable. The memory consumption is summarized in Figure 3. Precomputed lookup-table functions do use additional memory, even on the heap, but the table size is fixed and known before runtime.

5 Conclusion

Designing reusable shared libraries for real-time Java systems requires careful design to ensure that WCET and memory consumption can be known. Blocking functions must be eliminated. Our work thus far in three types of libraries with very different challenges demonstrates that predictable libraries for real-time Java are possible. The combination of JOP, Clepsydra, and the prototype libraries provide a predictable foundation and an interactive tool with which to build hard real-time applications in a high-level language.

[4]The actual WCET may be slightly different, since the measurements do not ensure that we have sufficiently covered hardware effects like the method cache.

	Code [byte]	Const [byte]	Heap [byte]	Sum (N=100)
cos8	119	40	0	159
cos6	107	36	0	143
cos4	97	32	0	129
cos2	89	24	0	111
cos_lutprim	$N \cdot 6 + 104$	$N \cdot 4 + 24$	$N \cdot 4$	1,528
cos_lutinterp	$N \cdot 6 + 138$	$N \cdot 4 + 20$	$N \cdot 4$	1,588

Figure 3. Memory Demand of Cosine Variants. Code **shows code size,** Const **shows memory attributed to the constant pool, and** Heap **shows memory allocated on the heap.**

References

[1] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[2] J.-M. Dautelle. Validating java for safety-critical applications. In *AIAA Space 2005 Conference*, 2005.

[3] T. Harmon and R. Klefstad. Interactive back-annotation of worst-case execution time analysis for Java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007.

[4] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[5] Java Expert Group. Java specification request JSR 302: Safety critical java technology. Available at http://jcp.org/en/jsr/detail?id=302.

[6] R. Kirner, M. Grössing, and P. Puschner. Comparing WCET and resource demands of trigonometric functions implemented as iterative calculations vs. table-lookup. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2006.

[7] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[8] M. Schoeberl and R. Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[9] M. Schoeberl and J. Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[10] F. Siebert. Real-time garbage collection in multi-threaded systems on a single processor. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, Arizona, 1999.