On Real-Time Performance of Ahead-of-Time Compiled Java

Anders Nilsson and Sven Gestegård Robertz
Department of Computer Science
Lund University
Box SE-221 00 Lund Sweden
{andersn|sven}@cs.lth.se

Abstract

One of the main challenges in getting acceptance for safe object-oriented languages in hard real-time systems is to combine automatic memory management with hard real-time constraints, while providing adequate general execution performance.

An approach to real-time Java based on ahead-of-time compilation is presented, and real-time properties and problems are examined. In particular, achieving both low latency and high throughput in an environment where neither the back-end compiler nor the scheduler is aware of automatic memory management is considered. Optimizations in both the compiler and run-time system, aimed at reducing the execution time overhead while still allowing very short latency times, is presented and experimentally verified.

1. Introduction

Java, as a programming language and execution environment, has gained a lot of interest from the real-time developer community during the last decade. As the complexity of real-time software increase, the inherent limitations of the programming languages used today (typically C, C++, or assembly languages) become more and more apparent. Programming errors like memory leaks and dangling pointers are caused by programming language limitations and would not occur if a *safe programming language* with *automatic memory management* was used.

A safe language is characterized by the fact that *all possible results of the execution are expressed by the source code of the program*. Of course, there can still be programming errors, but they lead to an error message (reported exception), or to bad output as expressed in the program. In particular, an error does not lead to uncontrollable execution such as a “blue screen”. If, despite a safe language, uncontrolled execution would occur , that implies an error in the platform; not in the application program. Clearly, a safe programming language is highly desirable for embedded systems. As of today, Java is the only safe, fully portable object-oriented programming language available that has reached widespread industrial acceptance (although not yet for hard real-time systems), due to these previously mentioned qualities, and its platform independence[1].

The benefits of safety are often referred to as the ”sandbox model”, which is a core part of both the Java language and the run-time system in terms of the JVM. Safety is guaranteed by the rule that objects cannot refer to data outside its scope of dynamic data, so activities in one sand-box cannot harm others that play elsewhere. This is particularly important in flexible automation systems where configuration at the user’s site is likely to exhibit new (and thereby untested) combinations of objects for system functions, which then may not deteriorate other (unrelated) parts of the system. Hence, raw memory access and the like should not be permitted within the application code, and the enhancements for real-time programming should be Java compatible and without violating the safety of the language.

2. Using Java for real-time systems

Given a program, written in Java, there are basically two different alternatives for how to execute that program on the target platform. The first alternative is to compile the Java source code to byte code, and then have a—possibly very specialized—Java Virtual Machine (JVM) to execute the byte code representation. This is the interpreted solution (as required to be Java certified from Sun) used today for Internet programming, where the target computer type is not known at compile time. The second alternative is to compile the Java source code, or byte code, to native machine code for the intended target platform.

A survey of available JVMs, more or less aimed at the embedded and real-time market, reveals two major prob-

[1]Or rather, its good platform portability, since it takes a platform dependent Java Runtime Environment (JRE), and JREs are not quite fully equivalent on all supported platforms.

lems with the interpreted solution. Since Just-In-Time (JIT) compilation is very hard to combine with real-time demands, an interpreter will typically suffer a performance penalty of being up to 10 times slower than natively compiled code. To improve performance, some JVMs (e.g. mackinac [8]) use the JIT compiler to compile the application at initialization time. This, however, comes at the cost of a significantly larger memory footprint.

The set of target systems considered in our work include small (350 MHz PPC G3 with 32 MB ram) and very small (AVR μcontroller at 8MHz/32 kB RAM) embedded computers. Therefore, we prefer ahead-of-time compilation to using a JVM. One thing in common for almost all CPUs, is that there exists a C compiler with an appropriate back-end. In the interest of maintaining good portability while compiling Java to native code, we use C as the intermediate language; The Java front-end generates C code which, in turn, is compiled by a standard C compiler.

2.1. Java in an uncooperative environment

Due to external requirements, we need our system to operate in an uncooperative environment; we want to be able to use an off-the-shelf C compiler and RTOS as well as external, legacy or automatically generated, C code. That means that we cannot rely on detailed assumptions on the behavior of the back-end C compiler or the thread scheduler, which makes implementation of a real-time garbage collector (GC) more challenging. For instance, it means that any synchronization required between collector and application, or *mutator*[2], needs to be done explicitly. It also means that the generated C code must be written so that it ensures, in a portable way, that no back-end optimization causes interference with the GC.

In particular, the combination of uncooperative compiler, uncooperative scheduler, and tight real-time requirements (low latency) makes a demanding challenge. Without control over the scheduling, some compiler optimizations cannot be allowed, as threads may be preempted at any time. For instance, if we are using a copying or compacting GC algorithm, pointers must always be read from memory, and not kept in registers, as the collector may move objects at (from the mutator's point of view) any time.

The details of accurate concurrent GC in an uncooperative environment are outside the scope of this paper, but is investigated in more depth in [14].

3. The Lund Java system

Research on real-time memory management and compiler technology at our department has evolved into a

[2]From the GC's point of view, the application is a process that changes, or mutates, the object-reference graph, causing objects to become garbage.

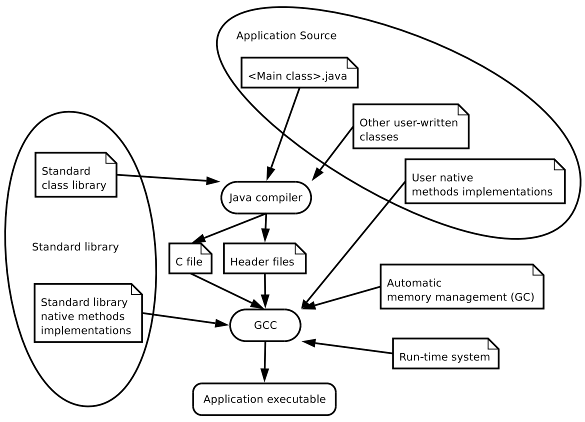

Figure 1. The Java2C compiler translates a Java application to C, which is then compiled and linked with memory management-, and run-time system modules to form an executable.

prototype compiler and run-time environment for hard real-time Java compatible applications [10, 9]. By Java compatibility we mean that a hard real-time application written in Java, and using a well defined subset of the standard Java2 Standard Edition (J2SE) class libraries, should run concurrently correct on any J2SE platform. The actual real-time performance of course depends on the underlying run-time system. The main design goal of our work is to keep the simple memory model of standard Java so that it should be easy to develop and debug an application on a desktop JVM, and then recompile for the real-time target and hopefully only have to sort out remaining timing-related issues on the target.

The Lund Java system consists of three modules: a Java-to-C translator, a subset of the J2SE standard class library slightly refurbished for use in hard real-time systems, and the automatic memory management module with a generic Garbage Collector Interface (GCI). The relationship between these modules is shown in Figure 1.

The execution platform—scheduler, GC, class library, etc.—is very important for the behavior of a Real-Time (RT) Java system. Compiled Java code relies on the task scheduler on the underlying operating system. It will also need to cooperate closely with the RT memory manager in such a way that timing predictability is accomplished, while memory consistency is maintained at all times.

One important design criterion in the Lund Java system is that we want to retain the platform independence of standard Java. It should be possible, with a very reasonable amount of work, to port our run-time system to a new platform equipped with a task scheduler.

3.1. Java2C

The Java2C compiler [11] is a research prototype compiler, developed for the purpose of testing our ideas on how hard real-time Java could be implemented. The compilation process, see also figure 1, is sketched as follows:

- Given the main class of the Java application, the compiler front-end performs name-, and type analysis, looking up and parsing depending classes when needed. All dependant classes are inserted in one large Abstract Syntax Tree (AST).
- The AST is transformed from representing the parsed source-code to a, semantically equivalent, representation suitable for code generation and the Garbage Collector Interface (GCI).
- The compiler back-end produces C code where all heap references are done using the GCI.

3.2. Real-time garbage collection

In order to be feasible for use in real-time systems, the memory manager, including the GC, must be non-intrusive. I.e., the GC should disturb the mutator threads as little as possible. This means that using an incremental collector is necessary, but not enough; care must also be taken in how the increments are scheduled. The GC must also be accurate, as a conservative GC cannot guarantee that all garbage is reclaimed.

The basic idea behind the GC implementations in our work is using a fine-grained incremental collector to allow very low latency, and running it concurrently, coupled with a suitable scheduling strategy, to ensure non-intrusiveness.

In 1998, Henriksson presented the *semi-concurrent* GC scheduling model [4], and showed that by analyzing the application, it is possible to schedule GC in such a way that the execution of high priority threads is not disturbed. This is accomplished by freeing high priority threads from doing any GC work during object allocation; that work is performed by a medium priority GC thread. Low priority threads perform a suitable amount of GC work at each allocation, using traditional incremental GC. Hence the term semi-concurrent, as the GC is concurrent to high priority threads but inlined in low priority ones. The analysis needed for tuning the GC scheduling parameters, to guarantee that the application will never run out of memory when a high priority thread tries to allocate an object, is based on calculating worst case response times, for the high priority threads and the GC, using generalized rate monotonic analysis [16].

In 2003, Robertz and Henriksson presented the notion of *time-triggered GC scheduling* [13], an approach aimed at making it possible to schedule a concurrent GC as any other thread, and thus transferring the responsibility for making the low-level scheduling decisions from the GC to the thread scheduler. By treating GC scheduling on the GC cycle level instead of on the increment level, this strategy also fits well into an adaptive, feedback scheduling system, which makes the approach suitable for flexible real-time systems. Taking that work into account, it appears reasonable to accomplish real-time Java without extending the memory allocation model, in contrast with what is done in for example the two real-time Java specifications [1, 2].

3.3. Garbage collector interface

Different GC algorithms require different interaction with the application. For instance, a compacting or copying collector requires a read-barrier[3], as objects may move, where a non-moving mark-sweep collector only requires a write barrier[4]. These differences makes it error-prone and troublesome to write code generators supporting more than just one type of GC algorithm, and it gets even worse considering hand-written code, which would need a major rewrite for each supported GC type.

In order to separate and hide the GC implementation from the application, we have specified an interface that provides heap access primitives to the application [6]. The GCI makes it possible to change the underlying GC algorithm without requiring any changes to the application code. This includes providing the necessary synchronization for reference and heap operations.

The GCI is used both in the Infinitesimal Virtual Machine (IVM) [5] and in our Java2C compiler, and with both non-moving, compacting and copying collectors. The varying requirements, both on the set of memory access primitives and run-time aspects causes the interface to contain quite many operations, which makes it less than ideal for manually written code, but as the intended use for GCI is generated code (especially from our Java to C translator) or low-level routines in a virtual machine, this is no major concern. As always, there is a trade-off between keeping the interface small and limiting the power of expression as little as possible. Manually writing code that accesses the heap through the GCI is, however, also quite doable.

The interface is implemented as a set of C preprocessor macros, and consists of primitives for initialization, object layout declaration, reference variable declaration, object allocation, reference access, field access, and function declaration and call, which adds up to 50 primitives.

[3]In GC terminology, a read barrier is the operation transforming a reference to an object into a pointer to the memory location the object. The term barrier stems from that it prevents accesses to obsolete copies of objects.

[4]A write barrier is used to inform an incremental GC about reference assignments, to ensure that no live object is missed due to changes to the reference graph being made while it is being traversed by the GC.

4. Performance issues

The key requirements on a run-time system for real-time applications are predictability and low latency, and the real-time properties of our approach have been previously verified [4, 9, 10, 12]. However, for a system to be practically feasible, the inlined overhead must not be unacceptably large. This section discusses how to achieve good general execution performance while maintaining the hard real-time properties.

As our Java system needs to operate in an uncooperative environment, it must ensure that correct behavior and real-time performance is not jeopardized by compiler optimizations, concurrency issues or interference from external code. In isolation, each of these aspects do not pose a problem; the difficulty comes from the combination, which gives conflicting requirements. In our experience, the main bottleneck is the synchronization between mutator and collector. Under an uncooperative scheduler, preemption can occur at any instant. Therefore, all reference operations must be protected to ensure mutual exclusion between collector and mutator and, if we want low latency, the critical sections must be small. However, this means a lot of synchronization, which may add up to a significant execution time overhead. It should be noted that this problem is due to the uncooperative scheduler, and not the ahead-of-time compilation; a JVM using native threads would face the same problems if a non-intrusive concurrent GC was desired.

The execution time overhead can be reduced in two basic ways: reducing the number of operations that require synchronization or using cheap synchronization primitives.

4.1. Reducing the need for synchronization

The level of required synchronization is affected both by the choice of GC algorithm (e.g., if a read barrier is required or not) and by different implementation decisions in the compiler and run-time system. This section gives examples of how those issues can be addressed in the compiler and in the run-time system, respectively.

Root alias analysis A *GC root* is a reference from outside the garbage collected heap to an object on the heap, typically a global variable or a local variable on a stack. The root references are used as starting points when the GC traverses the reference graph to identify live objects, and a GC cycle in a tracing collector typically starts with a stack scanning phase, where the root references are identified [7].

As our implementation is constrained by an uncooperative environment, we cannot scan the C stacks directly, as they contain no type information which means that we cannot discriminate between pointers and data. Therefore, we use an auxiliary *root stack* for each thread to keep track of the set of live local reference variables [4, 14].

```
GC_REF(Type1, tmp1);          // Type1 tmp1;
GC_REF(Type2, tmp2);          // Type2 tmp2;
GC_PUSH_ROOT(tmp1);
GC_PUSH_ROOT(tmp2);
GC_GET_REF(tmp1, a, b);       // tmp1 = a.b;
GC_GET_REF(tmp2, tmp1, c);    // tmp2 = tmp1.c;
GC_GET_REF(foo, tmp2, d);     // foo = tmp2.d;
GC_POP_ROOT(tmp1);
GC_POP_ROOT(tmp2);
```

Figure 2. Example of how GCI requires temporary reference variables

In a typical object oriented program, a large part of local variables will be of reference types, and thus there will be many root references. The use of GCI also makes it necessary to introduce many temporary variables as complex constructs, such as `foo = a.b.c.d`, has to be split up into simple attribute accesses as shown in Figure 2. This means that a lot of roots has to be pushed on and popped from the root stack, causing a significant execution time overhead, primarily from the required synchronization.

It can, however, be observed that in order to ensure correct GC behavior, it is enough that each live object is reachable from one root[5]. This means that the amount of necessary root operations, and thereby the overhead, can be reduced; if it can be statically determined that a variable will only reference objects that are also referenced by another variable with longer lifetime, the "inner" variable does not have to be registered as a root. We call this *root alias* analysis, and the compile-time analysis is trivial, as we do whole-program compilation. With this optimization, the push and pop operations in Figure 2 would be removed, which means that there will be no additional overhead of having the temporary variables explicitly in the code. In a typical Java program, the amount of "root duplication" is, in our experience, very high, as the associativity between objects tend to be high — between $50\,\%$ and $70\,\%$ of roots (including temporaries) were redundant in our experiments. A large portion of the required roots are temporary references required to keep a newly allocated object live before its constructor has completed. This is needed to keep latency low; as the constructor can be of arbitrary length it cannot be treated as atomic.

As an example of how the root alias analysis works, we take the code fragment in Figure 3. There, `f` and `b` will (or may) reference objects that are allocated in the context of `main`, so these variables must be registered as roots, as they are the only references to the new objects. On the other

[5]This does not hold for copying collectors that use forwarding pointers in the objects, as the roots are used for updating pointers as well as for finding live objects; for this optimization to work, the read barrier must be implemented using an indirect table outside the object.

```
void main() {
     Foo f; Bar b;
     ...
     f = new Foo();
     b = new Bar();
     ...
     proc(f,b);
}
void proc(Foo foo, Bar bar) {
     Test t1, t2; Bar b1;
     ...
     t1 = foo.test1;
     t2 = foo.test2;
     b1 = bar.x();
     ...
}
class Foo {
   Test test1, test2;
   ...
}
class Bar {
   Bar b;
   ...
   public Bar x() { return b; }
}
```

Figure 3. Root alias example

hand, in `proc`, we know that the parameters have been registered as roots in the calling scope. Local analysis in `proc`, can statically determine that `t1` and `t2` only reference objects that are reachable from (the attributes of) the parameters, and therefore it is not necessary to register these variables as roots. In contrast, we cannot tell if `b1` is an alias for something already rooted, or not. However, by analyzing the method `Bar.x()` it is seen that `x` only returns an object reachable from an attribute. Therefore, `b1` does not need to be registered as a root.

If we are doing whole-program compilation, all calls to functions returning references can be analyzed and will finally boil down to either an attribute access (which doesn't require rooting) or an allocation (which does). In a separate compilation context, it is not generally possible to perform the whole-program root alias analysis, but the local analysis may still be used to get rid of unnecessary roots caused by temporary variables.

The implementation of the root alias analysis is quite simple, and the majority of the code is shown in figure 4. In the case of class overloading, the analysis of whether a method call may return a new root must analyze all overloaded implementations of the method which may be executed, which may yield a conservative result. For the sake of readability, that code has been left out from the figure.

Function calls For function calls, the level of locking required depends on how reference arguments are passed — as references or as actual pointers (i.e., if the read barrier is executed in the caller or in the callee). In our implementation, reference structures are stack allocated and thus will not be moved by the GC. Therefore, if references are called by reference (i.e., a pointer to the reference structure is passed) no new roots are pushed in the callee and no heap locking is required. As the caller will always out-live the callee, if parameters to functions are known to be rooted in the calling context they don't have to be rooted again in the called context. Similarly, we know that the return value of a function will be used in the calling function (or not at all). Therefore, the variable that will receive the return value must already be rooted so if we pass a reference to this variable to the called function, it can be assigned before the return which removes the need to protect the return value. If function arguments and return values are handled in this way, no locking is required for function calls.

```
boolean VariableDeclaration.isNewRoot() {
  boolean result = false;  Stmt stmt = null;
  ASTNode scope = getSurroundingScope();
  foreach stmt in scope {
    result |= stmt.isNewRoot(this);  }
  return result;
}
boolean ExprStmt.isNewRoot(VariableDeclaration varDecl) {
  if (getExpr() instanceof AssignSimpleExpr) {
    AssignSimpleExpr expr = (AssignSimpleExpr) getExpr();
    return expr.getDest().isUse(varDecl) &&
           expr.getSource().isNewRoot();  }
  return false;
}
boolean MethodAccess.isNewRoot(){return decl().isNewRoot();}
boolean VarAccess.isNewRoot(){return decl().isNewRoot();}
boolean MethodDecl.isNewRoot(){ return returnsNewRoot();}
boolean InstanceExpr.isNewRoot(){return true; }

boolean Block.returnsNewRoot() {
  boolean result = false;
  for (int i=0; i<getNumStmt(); i++) {
    result |= getStmt(i).returnsNewRoot(); }
  return result;
}
boolean ReturnStmt.returnsNewRoot() {
  boolean result = false;
  if (hasResult()) { result = getResult().isNewRoot(); }
  return result;
}
boolean MethodDecl.returnsNewRoot() {
  // Native methods do not have bodies, so let's be conservative
  boolean result = true;
  if (hasBlock()) { result = getBlock().returnsNewRoot(); }
  return result;
}
```

Figure 4. Root alias analysis in the front-end

Root stacks in multi-threaded programs Another example of overhead caused by an uncooperative environment is the root stacks. In multi-threaded programs, each thread has its own root stack, and therefore, all root operations (i.e. push and pop) requires a pointer to the root stack of the current thread. In a system where the thread scheduler is Java-aware, the root stack pointer is part of the execution context of each thread and is saved and restored automatically.

In systems which cannot rely on scheduler cooperation, this has to be handled in the application code. As the root operations are part of the application code, and the current thread is not known at compile time, this must be looked up at run time. However, looking up the root stack at each root operation is quite inefficient so this should be done once for each function call and cached. Similarly, if no root operations are done in a function (like in e.g. a typical math function of the standard library), such lookup is unnecessary. Therefore, lookup of the thread root stack is done lazily at the first root operation of each function and the result is cached. This can be implemented quite efficiently.

4.2. Reducing the cost of synchronization

With fine-grained memory operations and heap-intensive applications, such as Java programs, the heap is almost always locked, so whenever preemption occurs, the probability that the heap is locked is high. Assume that a thread (`T1`) is executing and is in the middle of a memory operation. Then, a context switch occurs; the thread that is scheduled to run (`T2`) will probably try to lock the heap very soon after the context switch and be blocked. Then `T1`, which is holding the heap lock, is scheduled to run again until it releases the heap lock, allowing `T2` to continue its execution. This means that there will be three context switches instead of one, increasing the execution time overhead due to such *context switch chatter*.

Low latency due to locking is a requirement, so just increasing the size of the critical sections is not a viable solution. Therefore, we need a solution that allows very fine-grained preemption without the overhead of frequent unlocking and re-locking. We also need to make sure that context switches are not performed when the heap is locked.

This section will sketch three possible solutions based on turning off interrupts, preemption points, and a proposed technique, lazy locking, respectively.

Turning off interrupts The straight forward solution is to simply implement `gc_lock()` by turning off (clock) interrupts and `gc_unlock()` by turning them on again. On most architectures, interrupt requests that arrive when interrupts are masked are latched, so that when the interrupts are turned back on, any missed interrupt will be generated and the corresponding interrupt routine is executed. On such an architecture, this will give the desired semantics that if a time-slice ends, and preemption should take place, when the heap is locked, the context switch is delayed until the heap lock is released. Turning off interrupts may, however, not be allowed by the OS, or have negative effects on other parts of the system, e.g., interrupt-based drivers for peripherals, etc.

Preemption points By using a scheduler which only allow preemption at certain, pre-determined points, we can avoid frequent locking/unlocking. In fact, if the memory accesses are taken into account when placing preemption points so that preemption is only allowed when the heap is in a consistent state, no additional housekeeping or synchronization is needed in order to ensure correct GC operation.

However, preemption points are problematic for two reasons. The first is that most standard real-time operating systems don't support them. The second one is that calling external native code (that doesn't have preemption points) may cause priority inversion. An illustrating example is a background thread calling an external routine with a long execution time. As external code doesn't have preemption points, high priority threads may be delayed indefinitely. One solution is switching to "native" preemption when calling external code and then switching back to preemption-points when executing known code. However, calling external code would then have a performance penalty due to the additional housekeeping required and scheduler implementation would be more complex.

```
        gc_lock();
        ...
-->     gc_unlock();
-->     gc_lock();
-->     ...
-->     gc_unlock();
-->     gc_lock();
        ...
        gc_unlock();
```

Figure 5. Locking example: Small atomic operations cause frequent locking.

Lazy locking If turning off interrupts or using preemption points is not possible or desirable, an alternative strategy for reducing the locking overhead is based on the observation that, while the frequent locking and unlocking is required in order to achieve low latency, in the common case, the heap is unlocked, and then shortly re-locked by the same thread. Thus, most of the locking operations are really unnecessary and could be removed without changing the behavior of the program (other than reduced overhead). The problem is just determining which lock and unlock operations that need to be performed. This could be done statically, but the analysis would be difficult and highly dependent on the low-level scheduling, control flow based on input data, etc. Therefore, a dynamic, on-line approach is preferable.

For example, take a code sequence like in Figure 5. If we are executing in the marked region, and no clock interrupt has arrived (i.e., the thread will not yet be preempted), it is unnecessary to perform the unlocking and re-locking operations. Thus, if we could dynamically decide whether to perform the unlock/lock operations (in a way that is much cheaper than actually performing the locking), the overhead could be reduced. Then, when a clock interrupt occurs, the heap should really be unlocked at the next unlock instruction and the context switch performed.

One way of implementing this is by having two versions of the operations: the actual lock/unlock operations (which are executed when the locking is required) and "NOP" versions that are used when unlocking and re-locking isn't necessary. Then, the run-time system ensures that the correct version is run at each time to both guarantee the correct semantics and achieve the best performance. In principle, an implementation of this scheme looks like in Figure 6. This method gives similar behavior as preemption points with regard to heap accesses, but without requiring additional

```
void (*gc_lock)(void);
void (*gc_unlock)(void);

void gc_lock_real(void)
{   lock(heap_mutex);
    gc_lock   = f_nop;
    gc_unlock = f_nop;
}
void gc_unlock_real(void)
{   unlock(heap_mutex);
    yield();
}
void f_nop(void) { return; }
void reschedule(void)
{   if(is_locked(heap_mutex)) {
       gc_lock   = gc_lock_real;
       gc_unlock = gc_unlock_real;
      } else {
         /* perform actual context switch */
    }
}
```

Figure 6. Lazy locking implementation sketch

housekeeping in order to allow external native code to be run with real-time guarantees.

If modifying the scheduler is not possible, or practically feasible, much of the benefit of lazy locking can still be obtained if the OS has a call-back hook for a method to be called at context switches. In fact, this is the method used in our Linux/RTAI prototype, and it gives the same reduction of the number of locking operations, but does not address context switch chatter. That may, however, be a reasonable trade-off for not having to modify the scheduler.

There are, of course, many other small details that must be taken care of when implementing such a scheme; e.g., the system must ensure that the heap is always unlocked before a blocking call is made or before a thread dies; otherwise there is a risk of deadlock.

4.3. Compiler optimization effects

Another problem with locking is that the lock/unlock operations are function calls or inline assembler, and that tend to break basic blocks and interfere with compiler optimizations. This is, partly, intentional, as many optimizations are not safe in the general case. E.g., we must make sure that pointers (gotten through the read barrier) to objects are always read from memory as objects may have moved since the last access, etc., when we enter the next critical section, and such race conditions will lead to memory corruption.

However, this is really only needed when a context switch actually has taken place; as long as the same thread is executing, any optimization is legal, as long as the heap and all references in memory are consistent at the next context switch. Thus, performance could be improved significantly if it was possible to implement lazy locking in a way that the fast case did not break basic blocks. We believe that this could be done with self-modifying code, injecting the lock-/unlock operations into the code where they are needed and modifying the lock/unlock instructions so that they ensure heap consistency. This, of course, requires detailed information about the inner workings of the optimizing back-end and target architecture and cannot be done in a simple or portable way.

5. Experimental results

We have performed a number of experiments in order to verify the applicability of our proposed real-time Java system. We will first present measurements on latencies for a hard real-time Java application using automatic memory management. Then we will show a comparison of general execution performance between our Java system and some other Java compilers and run-time systems, including a native C solution as a reference. Finally, we will study how the presented techniques and optimizations affect performance.

The experiment setup used in Section 5.1 was a synthetic benchmark resembling a typical real-time application; periodic threads allocating a number of different objects each sample, corresponding to a set of threads controlling a physical process. It was executed on a 333 MHz Pentium II running Linux and RTAI. The experiment setup used in Sections 5.3 and 5.4 was a low level servo controller for an ABB IRB-2000 industrial robot. Given a desired motor angle for each of the six joints, suitable torque values and the corresponding AC motor currents are calculated. Both servos executed on a 350 MHz PowerPC G3 with 32 MB RAM running Linux/RTAI. The execution performance comparison in Section 5.2 consists of two benchmarks from the *emb_bench* suite [15] and was executed on a P4 2,8 GHz workstation with 1 GB RAM running Debian GNU/Linux.

5.1. Latency

This experiment examines if the use of GC in a real-time Java system causes any jitter in the highest priority thread. The heap is sufficiently small so as to guarantee that at least one full GC cycle will be run during the experiment.

Figure 7 shows measured latency and response times for the highest priority thread in a heavily loaded system (CPU utilization >90%) using mark-compact and mark-sweep GC algorithms. Latency is in the interval $3-12\,\mu s$ for the mark-compact GC and $2-14\,\mu s$ for the mark-sweep GC while response times are measured to be in the intervals $32-43\,\mu s$ and $27-41\,\mu s$ for the respective GC. The execution time of a sample was about $30\,\mu s$. The experiment shows that the main source of response time jitter is the release latency and when a task has been started it is not disturbed further.

5.2. Performance

For the performance comparison with some non real-time Java runtime environments, two applications from

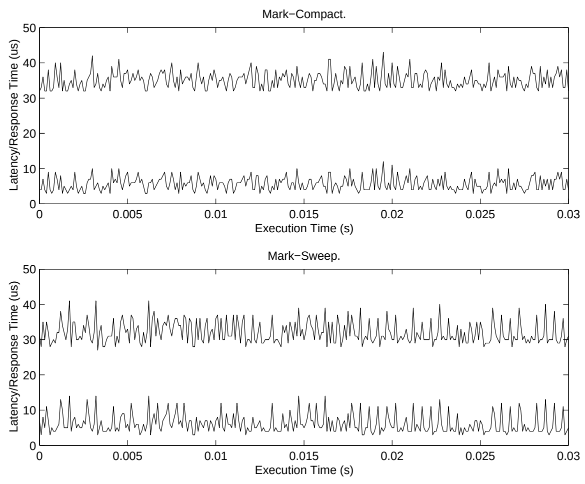

Figure 7. Release latency (bottom) and response time (top) jitter for the highest priority thread at 10 kHz, for two different GCs.

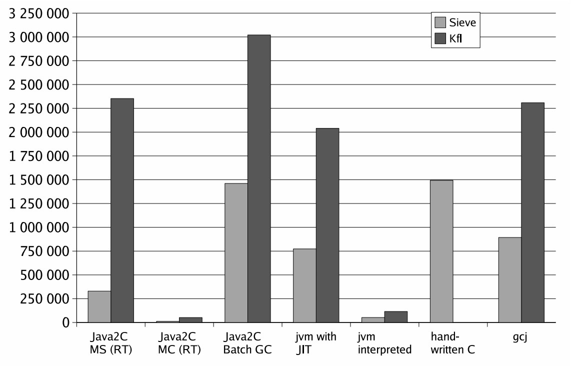

Figure 8. Performance comparison chart with two real-time GCs, one non real-time GC, and four other non RT execution examples. Performance measured as number of iterations performed in one second.

Martin Schoeberls [15] embedded benchmarks suite were used; *sieve* does little more than access elements in arrays, while *kfl* is a real control application for a small embedded system here run in a simulated environment. The Java2C tests were compiled and run using mutex synchronization and with root alias analysis.

As shown in figure 8, performance suffers heavily from using a mark-compact GC due to the read barrier; as reads are typically more common than writes, the expensive synchronization mechanism has a bigger impact on overall performance. The mark-sweep GC performs much better, especially in the more realistic *kfl* benchmark program. Note that the differences between using a batch GC and the two real-time GC algorithms is the cost for synchronizing with an incremental algorithm. For a fair comparison, only the batch-copy example should be compared to the other three Java execution environments in terms of throughput performance as they do not have real-time GC. It does, however, illustrate that the cost of synchronization may be devastating to performance if care is not taken. The configuration used here corresponds to number 2 in Figure 10, so with more efficient synchronization, the penalty of mark-compact would be significantly less.

Incrementality always comes at the cost of increased runtime overhead, and for batch applications it yields no benefit; as the application never sleeps, any GC work will delay the application. A typical real-time control system, on the other hand, consists of a set of periodic tasks. Thus, an incremental GC can be scheduled so that it will not disturb the application, reducing the impact of the GC overhead significantly. In addition, the long GC pauses make a batch GC unsuitable for real-time applications.

5.3. Lazy locking

This experiment investigates the impact of lazy locking on the number of lock operations that are actually performed. Figure 9 shows the frequencies of locks in the vanilla version and real and lazy locks in the lazy version. This shows that only a small fraction of the locks actually need to be performed and thus that the locking overhead can be significantly reduced. For instance, in the receiver thread, which performs most of the computations, only 0.03% of the lock instructions in the code actually cause a mutex operation.

5.4. GC algorithm, synchronization mechanism and root alias optimization

Figure 10 shows how the choice of locking primitive and the root alias optimization affects total throughput, i.e. the maximum possible sample rate. The big difference between the mark-sweep and the mark-compact collector is caused by the extra synchronization required for the read barrier in the mark-compact case. With synchronization turned off[6], there is no big difference between a moving and a non-moving collector. In this example, the overhead of the read barrier is compensated by the cheaper allocation[7].

In this experiment, lazy locking was implemented with the call-back method, instead of modifying the scheduler,

[6]Of course, running without synchronization is not safe and may cause race conditions and memory corruption, so this is done for reference only and is not a practically usable configuration.

[7]In the mark-compact collector, allocation is done by simply incrementing a pointer, whereas in the mark-sweep case, free-list search and block splitting is done.

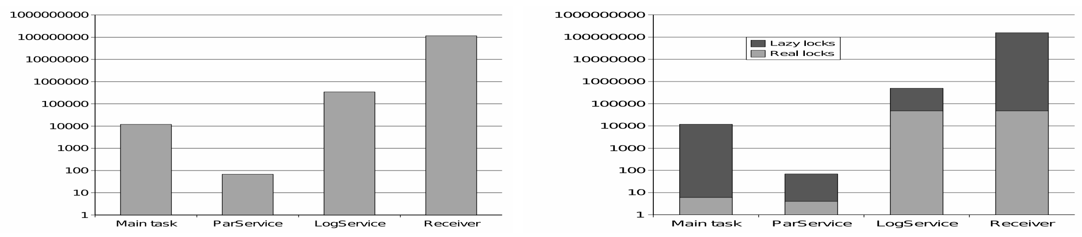

Figure 9. Comparing the vanilla version (left) to the one with lazy locks (right), showing the frequencies of real and lazy locks for each of the application threads. Please note that the scale is logarithmic. In this experiment, the mark-compact collector was used. The application was run for a fixed amount of time, so the numbers should not add up.

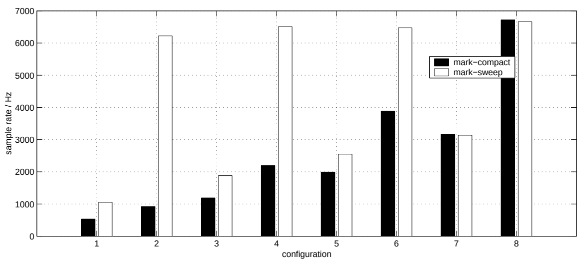

Figure 10. The effect on throughput of different locking primitives and root optimization. The configurations are 1) Mutex locking, 2) Mutex locking with root alias optimization, 3) Lazy mutex locking, 4) Lazy mutex locking with root alias optimization, 5) Interrupt masking (cli/sti), 6) Interrupt masking with root alias optimization, 7) No locking and 8) No locking with root alias optimization

so it only shows the savings from doing fewer locks. It remains to be investigated how large the effects of context switch chatter are. The call-back method also adds to the overhead of each context switch, so an implementation inside the scheduler would yield a much bigger improvement.

6. Related work

The concept of natively compile Java code and/or making Java viable for use in systems with hard timing constraints is not new and there is plenty of both academic and industrial work published. However, in the range of small sized embedded systems we are targeting there are not very many projects trying to implement Java for hard real-time systems *with* automatic memory management.

The Real-Time Specification for Java (RTSJ) [1, 20] is generally acknowledged as the specification to follow when implementing real-time Java. Implementations include:

Mackinac: Based on Sun's HotSpot technology, but compiling Java classes at initialization time instead of during runtime, applications executing on the mackinac is predicted to achieve similar performance as equivalent C++ applications. Just like HotSpot, it will take quite some CPU power and memory to run mackinac.

JamaicaVM: Aicas GmBH and IPD Universität Karlsruhe have implemented a combined JVM and Java bytecode-to-native compiler called Jamaica [18, 17, 19]. The Jamaica VM is always responsible for garbage collection and the task scheduling, while some classes may be natively compiled and call the VM for services such as memory allocation. The GC principle used is a non-moving type with fixed memory block size for eliminating external fragmentation. The amount of GC work to do at each object allocation is scheduled dynamically with respect to the current amount of free memory, and task latency (also for high priority tasks) will vary accordingly.

The varying task latency and the fact that the fixed size memory block scheme makes linking with non-GC-aware code modules complicated, make the Jamaica system inappropriate for small embedded systems and for flexible hard real-time systems.

JRate: JRate [3] is implemented as an extension of GCJ to support the RTSJ. Since it is an ahead-of-time compiled solution, performance should be acceptable also on modest platforms.

However capable of achieving good real-time performance, all implementations of the RTSJ do have one large drawback in common. Instead of focusing on how to solve the real-time garbage collection problem, they resort to introducing additional memory types which can be used by high priority threads. This will, in effect, return memory management to the error-prone programmer, who will have to figure out which objects may reference which other objects without violating the various memory access rules.

7. Conclusions

In order to make Java a viable programming language for embedded real-time systems development, performance is the crucial factor in both senses of the term. Short and constant latencies must be guaranteed, while execution performance must not degrade too much compared to implementations in other programming languages. Another important factor is to preserve the flexibility of Java allowing real-time Java applications to execute on many different platforms with different operating systems, including non real-time operating systems for development and debugging.

However, flexibility in natively compiled Java also means that we must consider an uncooperative environment. Neither the task scheduler, nor the back-end C compiler can be assumed aware of incremental GC. Hence, there is a tradeoff to be made between flexibility, latency, and throughput. This paper has identified some potential bottlenecks caused by the uncooperative environment, and presented techniques, both in our Java compiler and in the runtime system, for reducing the execution time overhead and enhance throughput while maintaining low task latency.

Experimental evidence show that we can achieve very low latency and jitter as well as reasonable throughput. Given our contributions and results, we do see compiled real-time Java, or a similar language such as C#, as industrially viable in a near future.

Acknowledgements

The work presented in this paper has been carried out within the *FLEXCON* and *Network Programming* projects, at the LUCAS[8] applied software research center with financing from the Swedish Agency for Innovation Systems (Vinnova) and the Swedish Foundation for Strategic Research (SSF). We would like to thank Anders Blomdell at the Department of Automatic Control, for sharing his deep knowledge in compilers and real-time kernels, and many of our colleagues at the Department of Computer Science.

[8] `http://www.lucas.lth.se`

References

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

[2] J. Consortium. Real-Time Core Extensions. P.O. Box 1565, Cupertino, CA 95015-1565, September 2 2000.

[3] A. Corsaro and D. C. Schmidt. he design and performace of the jrate real-time java implementation. In *Proceedings of the 4thInternational Symposium on Distributed Objects and Applications, DOA 2002*, October 2002.

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

[5] A. Ive. Towards an embedded real-time java virtual machine. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2003.

[6] A. Ive, A. Blomdell, T. Ekman, R. Henriksson, A. Nilsson, K. Nilsson, and S. Gestegård-Robertz. Garbage collector interface. In *Proceedings of NWPER 2002*, August 2002.

[7] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[8] The Real-Time Java Platform, a technical white paper. `http://research.sun.com/projects/mackinac/mackinac_whitepaper.pdf`, June 2004.

[9] A. Nilsson. Compiling Java for Real-Time Systems. Licentiate thesis, Department of Computer Science, Lund Institute of Technology, May 2004.

[10] A. Nilsson, T. Ekman, and K. Nilsson. Real java for real time – gain and pain. In *Proceedings of CASES-2002*, pages 304–311. ACM, ACM Press, October 2002.

[11] A. Nilsson, A. Ive, T. Ekman, and G. Hedin. Implementing java compilers using rerags. *Nordic Journal of Computing*, 11(3):213–234, 2004.

[12] S. G. Robertz. *Flexible automatic memory management for real-time and embedded systems*. Licenciate thesis, Lund Institute of Technology, Lund University, April 2003.

[13] S. G. Robertz and R. Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems - 2003 (LCTES'03)*, pages 93–102. ACM SIGPLAN, ACM Press, June 2003.

[14] S. G. Robertz and R. Henriksson. Accurate concurrent GC in an uncooperative environment — performance vs predictability? In preparation, 2005.

[15] M. Schoeberl. JOP - Java Optimized Processor. World Wide Web, 2004. `http://www.jopdesign.com`.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.

[17] F. Siebert. Hard real-time garbage collection in the jamaica virtual machine. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*, Hong Kong, December 1999. IEEE.

[18] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES 2000)*, San José, November 2000.

[19] F. Siebert and A. Walter. Deterministic execution of java's primitive bytecode operations. In *Java Virtual Machine Research & Technology Symposium '01*, Monterey, CA, April 2001. Usenix.

[20] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, September 2004. ISBN 0-470-84437-X.