Using Real-time Java for Industrial Robot Control

Sven Gestegård Robertz
Department of Computer Science
Lund University
sven@cs.lth.se

Roger Henriksson
Department of Computer Science
Lund University
roger@cs.lth.se

Klas Nilsson
Department of Computer Science
Lund University
klas@cs.lth.se

Anders Blomdell
Department of Automatic Control
Lund University
anders.blomdell@control.lth.se

Ivan Tarasov
Department of Applied Mathematics — Control Processes
St. Petersburg State University,
Sun Microsystems
Ivan.Tarasov@sun.com

ABSTRACT

Safe languages like Java provide a much more programmer-friendly environment than the low-level languages in which real-time and embedded software have traditionally been implemented. However, an obstacle for widespread use of Java in control applications has been the predictability and real-time performance of garbage collection, and the cumbersome memory management associated with RTSJ NoHeapRealtimeThreads. The current version of the Sun Java Real-Time System includes a real-time garbage collector, and therefore, it is interesting to examine its feasibility for robot motion control.

We have implemented a motion control system, and an application, for an ABB IRB 340 industrial robot entirely in real-time Java, using standard computer hardware, off-the-shelf EtherCAT servo drives, and the Sun Java Real-Time System 2.0 on Solaris 10. To our knowledge, this is the first robot control system implemented entirely in Java and executed on a certified virtual machine.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Memory management, run-time environments*; J.7 [**Computer Applications**]: Computers in other systems—*Real time, Industrial control*

General Terms

Experimentation, Performance

JTRES'07, September 26–28, 2007, Vienna, Austria.

Keywords

Real-time, Java, garbage collection, industrial robot, motion control

1. INTRODUCTION

The required flexibility in manufacturing implies that both machine control and equipment interfaces need to be easy to change for new application scenarios. Also, the trend for shorter time to market and increased customization of the manufactured products often result in configurations that are quite dynamic, and therefore less suited for implementation in unsafe languages such as C or C++.

Automation equipment contains a variety of processors and interfaces for the involved embedded systems, so portability of (source and binary) code is an issue. But, in contrast to other areas of embedded systems, power consumption is not a big issue, as the machines consume a lot of energy for the mechanical motions.

Another aspect is that controlled machines often need to be simulated as part of the concept of digital factories, e.g. for evaluation of suitable production line designs. These simulations need to run on host computers, preferably without recompiling (or even rewriting as today) the motion control software.

In total this means that the software requirements get similar to those of Internet applications (portable execution with language restrictions on data access), but with real-time requirements. In other words, we would very much benefit from implementing machine control systems in Java.

Up to now there has been a tradition to implement the time-critical parts in C, and other parts (such as operator interfaces) in Java or other languages. In the following we explore if using Java for low-level control is at all possible, using a high-performance robot control system as a demanding test case, and the Sun Java Real-Time System 2.0, as the test platform.

1.1 Organization of the paper

Section 2 opens the discussion by presenting some key technologies for the implementation of the robot control system. It goes on by presenting the robot control system itself, both from hardware and software point-of-view in Section 3. The experiences gained from the work is discussed in Section 4. In Section 5, related work is presented, and Section 6 concludes the paper.

2. BACKGROUND

There are a couple of technologies that can be said to be enabling technologies for making it possible for us to assemble and implement the robot control system with relatively little effort. One is the introduction of real-time garbage collection (RTGC) into the Sun Java Real-Time System 2.0 which made it possible to utilize the standard Java memory model for critical control threads and the other was EtherCAT, the open field bus which made it easy to assemble off-the-shelf hardware into a complete system.

2.1 Real-time garbage collection

In order to satisfy the demands of hard real-time systems, a technique must be found to schedule the GC work of a concurrent GC such that the application is guaranteed to meet all of its hard deadlines. One such scheduling technique was presented by Henriksson in [4]. That work focuses on embedded systems which are assumed to have a number of high-priority (typically periodic) threads that must meet hard deadlines. It can be observed that in most embedded systems, a relatively small number of such threads exist. Apart from these, low-priority (periodic or background) threads are often executing with more relaxed deadline requirements. This leads to the fundamental idea of Henriksson's work, which is as follows: do not perform any GC work when the high-priority threads are executing. Instead, assign the work motivated by high-priority allocations to a separate GC thread which is run when no high-priority thread is executing. When invoked, it performs an amount of GC work proportional to the amount of memory allocated by the high-priority threads. Since the garbage collector may temporarily get behind with its work in this way, there must always be an amount of memory reserved for the high-priority threads. Slightly modified generalized rate monotonic analysis can be used both for calculating the amount of memory which need to be reserved and to verify that the garbage collector thread will always keep up with the high-priority threads. Garbage collection work motivated by low-priority threads are performed incrementally at allocation time.

The effect of this scheme is that it makes it possible to guarantee hard real-time performance for threads that actually require it in a system scheduled by a fixed-priority scheduler. It should be noted that by scheduling GC this way, the critical high priority threads pay no penalty for allocating memory, in contrast with traditional incremental GC scheduling strategies, where each thread has to carry it's own weight as far as the CPU overhead of memory management is concerned. That is the key property, as in a real-time system, fairness in scheduling is not desired — the most critical threads should run with as small latency and jitter as possible.

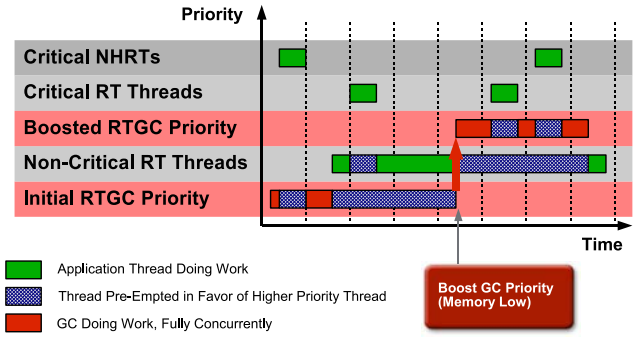

Figure 1: Sun RTGC scheduling in a memory-low situation. The RTGC priority is boosted in order to guarantee the availability of memory to critical RealtimeThreads.

The described scheduling principle is not targeted at, or limited to, any particular GC algorithm. Any incremental GC could be used, provided that the increments can be made sufficiently small (typically around 10 microseconds) to keep the latency suffered by high priority threads low.

Another important aspect of a real-time run-time system is that it must provide isolation between high and low priority threads, in the sense that no actions of a low priority thread may cause a higher priority thread to miss a deadline. Since memory is a global resource, this means that care has to be taken that low priority threads are not allowed to allocate memory if that would cause an out-of-memory situation in a higher priority thread. That can be achieved by calculating the allocation requirements of critical threads, and reserving a sufficient amount for such allocations [5].

2.2 The Sun Java Real-Time System 2.0

The Sun Java RTS 2.0 is an implementation of the Real-Time Specification for Java [2], and includes a real-time GC based on the basic principles described described in the previous section [3]. The RTGC is fully concurrent and preemptable by NoHeapRealtimeThreads and RealtimeThreads with priorities higher than the RTGC. It is also suitable for multiprocessor machines since it can run on one processor while application threads can run in parallel on the other processors.

Normally, all RealtimeThreads run with a priority higher than the RTGC, but the RTGC is allowed to boost its priority in low-memory situations to a higher, programmer-configurable, level as shown in Figure 1. RealtimeThreads can thus be divided into two categories, critical threads with a priority higher than the boosted RTGC priority and non-critical. Critical RealtimeThreads are guaranteed to never be preempted by the RTGC as long as the system is not overloaded.

The critical RealtimeThreads are protected against excessive allocations in low priority threads by reserving a certain amount of memory for exclusive use by critical threads. This is done by setting the command line parameter RTGC-CriticalReservedBytes.

An advantage of the Java RTS 2.0 garbage collector is that it makes it possible to use RTSJ RealtimeThread instances for hard real-time tasks, as they can run with a higher priority than the GC. This removes one of the previous obstacles in writing RTSJ code; it is now possible to use dynamic memory in real-time threads just like in standard Java, and to share objects between real-time and non-real-time threads.

Another benefit of using Real-Time Java is the availability of the standard Java libraries and the ability to use some language features (notably, generics, since Sun RTJS 2.0 is based upon Java 1.5).

2.3 EtherCAT

EtherCAT [1] is an open field bus technology, based on real-time ethernet, originally developed by Beckhoff, providing very efficient transmission of process data. The EtherCAT protocol has an officially assigned Ether-type in the ethernet frame, making it possible to transmit process data directly in the ethernet frame while being compliant with the ethernet standard. The fundamental item of communication on the EtherCAT level is the telegram, and many telegrams can be packed together in one single ethernet frame, each addressing a particular area in the logical process image.

The simplest and most efficient way of communicating — which was used in the presented application — is sending EtherCAT telegrams directly in the ethernet frame. For larger systems, there is also an option to put EtherCAT telegrams inside UDP datagrams, to allow more flexible network topologies.

EtherCAT slaves introduce very small delays (in the order of nanoseconds) as the ethernet frames are not received, processed and then transmitted. Instead, frames are processed on-the-fly using a fieldbus memory management unit (FMMU), allowing each slave to read and write its process data while the frame is being copied directly from input to output. Addressing is also quite flexible and the data sequence is independent of the physical order of slaves on the bus.

In addition to transmission of process data, mailbox-based communication is possible, for instance using CANopen over EtherCAT (CoE) service data objects (SDOs). It is possible to send such mailbox telegrams in the same ethernet frame as process data telegrams.

For our application, one of the main advantages of EtherCAT is that it doesn't require any special hardware on the master computer; a standard ethernet card is all that is required. Also, if the ethernet driver of the operating system provides raw ethernet access with sufficient real-time performance, no low-level driver implementation is required.

3. THE APPLICATION

The application developed for the real-time Java powered robot demo at JavaOne 2007 was a portrait-drawing robot. A digital camera was used to take pictures of attendees. The pictures were vectorized and turned into robot paths which were sent to the robot controller. Figure 2 shows a photo of the robot from JavaOne 2007.

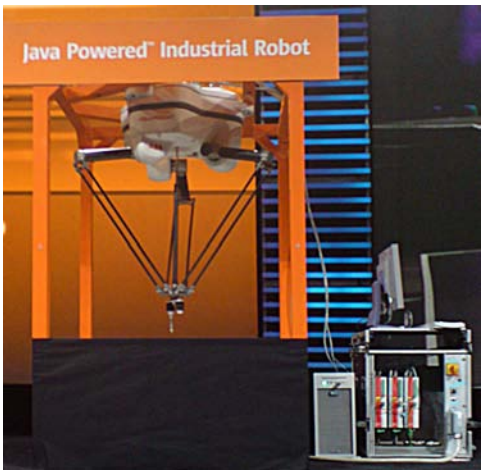

Figure 2: The robot, controller computer, and drive rack at JavaOne 2007.

.

The computer running the controller was a Sun Ultra 40 workstation, a 4 way AMD machine, running Sun Java RTS 2.0 on Solaris 10. The etherCAT communication was done using one of the built-in ethernet interfaces.

The motion controller consisted of a trajectory generator running with 12 *ms* period time, and a position controller running with 1 *ms* period time. Velocity and torque control was done in the drive units. The application software was written entirely in Java, with the exception of a few lines of JNI code for sending raw ethernet packets. In addition to the real-time control parts, there was a graphical user interface providing an operator's console and logging and plotting of process values. These parts were run in ordinary Java threads, at lower priority than both real-time threads and the RTGC.

3.1 ABB IRB 340 robot

The robot used was an ABB IRB 340 FlexPicker, a very fast and light pick-and-place robot. We used only the ABB robot itself, and not the ABB controller. The robot is a parallel kinematic robot, which differs from the traditional *serial* robot (where each axis is in line relative to the preceding one) in that the axes operate in parallel. That means that such a robot can be made light and flexible but also that, due to the simpler dynamics, it opens possibilities to control them using computers and software other than that supplied by the robot manufacturer.

3.2 Servo drives

A servo drive is an electronic amplifier used for driving an electrical motor — in this application the AC motors in the robot — using feedback control. The motors have resolvers providing angle measurements and the drive controls the motor current in order to achieve a desired torque and velocity. The power electronics part of the drive converts the three phase mains feed to the phase currents required to drive a particular type of motor.

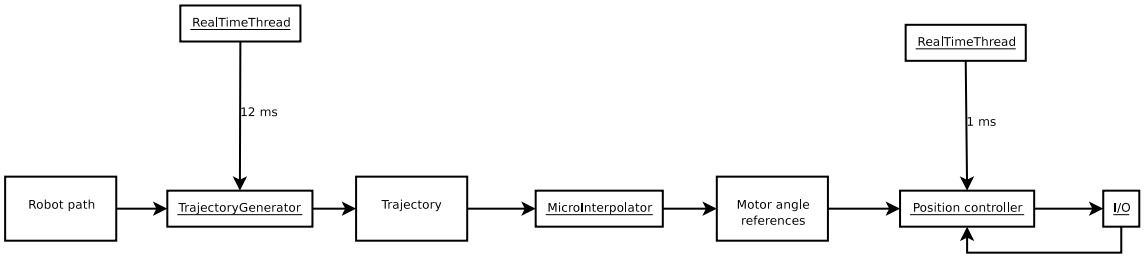

Figure 3: Overview of the control data flow. The trajectory generator reads the robot path and generates position and velocity references with 12 ms time step. Those are interpolated into 1 ms time steps by the micro interpolator, and buffered. The position controller runs at 1 kHz, sampling the current motor positions, computing and outputting velocity references to the motors.

The drives used in the presented application were Bechhoff AX2006, with a rated current output of 6A, and a peak output of 12A. The drive outputs were connected through the standard ABB motor cable, but the resolver cables were directly connected to the motors in the robot, bypassing the ABB measurement electronics. The communication between the Java controller and the drives was done over an EtherCAT bus.

3.3 Robot motion control

The robot motion control in the presented application has two main parts, trajectory generation and position control. The former is run at 12 ms and takes the robot path, which is simply a list of coordinates, and computes a trajectory, a fine-grained list of coordinates and velocities for each 12 ms period. The trajectory generation takes the physical constraints of the robot and work cell into account, e.g., maximum speed and acceleration, and the allowed working area of the robot. From the trajectory, the position and velocity references are interpolated down to 1 ms intervals, and sent to the position controller, which runs the feedback loop, computing motor velocities based on the references and measured motor positions.

The controller used for the position control is a proportional controller with feed-forward. The output signal is the sum of two parts, one proportional to the error, and one depending on the desired velocity in the trajectory. I.e.,

$$v = K_{fb}\,(x_{ref} - x) + K_{ff}\,v_{ff}.$$

The feed-forward part is useful as it allows the controller to follow the desired path more closely; with the feedback control alone, there has to be a control error ($x \neq x_{ref}$) for a non-zero output to be generated. With feed-forward, a control signal can be generated without such an error.

3.4 Java implementation

The control was performed by two RealtimeThreads, running at 12 ms and 1 ms, respectively. Both the position controllers and the EtherCAT I/O (reading positions and sending velocity references) were done in the context of the 1 ms thread. as sketched in Figure 3. It is worth noting that the control did not use NoHeapRealtimeThreads.

The micro-interpolator, or joint interpolator, is implemented as a buffer, where the interpolation is done on the "slow" side, to minimize latency and jitter in the fast position control loop.

The real-time demands of the sampler thread, driving the position control, come both from the control itself (jitter in the sampling causes degraded control performance and may even cause instability), and from the fact that the servo drives have a safety feature requiring periodic sampling; if the jitter gets too high, the motors are stopped.

Figure 4 shows an overview of the major classes of the robot application. RobotApplication is the main class, and connects the three parts of the system; an AbstractRobot instance providing an interface to the robots sensors and actuators, the DrawingRefGen responsible for reading paths and generating trajectories, and the RobotSampler driving the feedback control loop.

There are two I/O abstractions; IOTrigger and IOGroup. The former provides an interface for triggering I/O operations like sending EtherCAT telegrams. The latter contains the structured interface to the data. For instance, in this application the IOGroup consists of four VelocityIO ob-

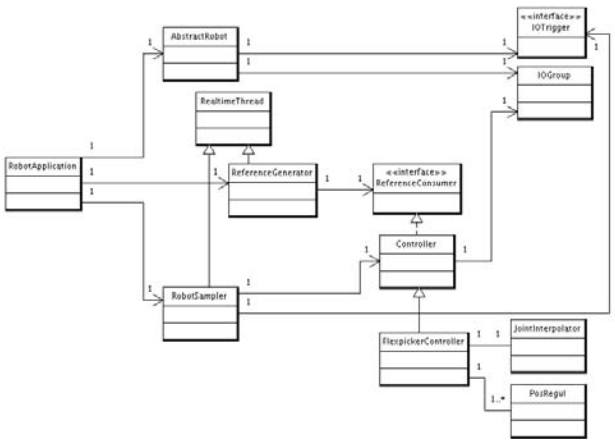

Figure 4: Class diagram of the central parts of the robot application.

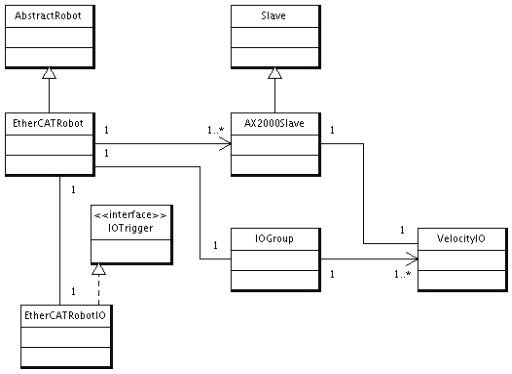

Figure 5: Class diagram showing the specialized EtherCAT robot.

jects, which have operations for reading motor positions and outputting motor velocities. If torque control were desired instead of velocity control, an IOGroup with the corresponding TorqueIO objects would be implemented.

Figure 5 shows how the concrete class EtherCATRobot relates to the EtherCAT slave class and the I/O interfaces. This class contains the knowledge about the EtherCAT specific details of communication, implemented in the EtherCATRobotIO class.

3.5 EtherCAT protocol implementation

The EtherCAT master code has two main pieces of functionality: configuration of the slaves and transmission of process data. The former is carried out by a non-real-time supervisor thread, monitoring the state of the bus (number of slaves, etc), and reacting to any changes. That includes assigning station addresses, configuring process data objects, setting up FMMUs, and so on. Transmission of process data is done through I/O objects (IOTriggers), which cause EtherCAT logical read/write telegrams to be sent on the bus. The periodic thread driving I/O and the feedback control typically has the following structure:

```
while (operational) {
    ioTrigger.doInput();
    controller.doControl();
    ioTrigger.doOutput();
    waitForNextPeriod();
}
```

The EtherCAT master does, in itself, not depend on any real-time Java classes. The transmission and reception of process values must — if hard real-time is required — be done in a real-time thread, but for applications with less strict timing requirements the EtherCAT master can be run on any Java virtual machine.

We will now briefly describe the EtherCAT master implementation. Figure 6 shows a diagram of the core classes of the EtherCAT master code. The central class is Master, which is responsible for monitoring the state of the bus, and reacting to changes. When a new slave is discovered the corresponding SlaveFactory will instantiate a new Slave object (based on manufacturer ID and product ID codes read from the slave). For each slave type, there is a specialization of the Slave class describing that particular device.

The SlaveHandler manages information about the process data layout (by reading the list of process data objects (PDOs) from the slaves) and sets up the slaves for the desired mode of operation. It also contains an IOHandler class, which contains the methods for packing and unpacking process data in EtherCAT telegrams.

The different Telegram classes handles packing and unpacking the corresponding EtherCAT telegram into an ethernet frame. Telegrams can be logical (reading/writing/reading and writing a FMMU mapped address), or physical (with single slave addressing based on bus position (AP) or assigned address (NP) or broadcast to all slaves).

4. EXPERIENCES

From a programmer's point of view, being able to access the heap in the real-time parts of the code means that the design of the system can be made in a more straight-forward way, than if some of the more restricted memory management models had been used. Even though the controller does very little dynamic memory allocations during steady-state operation, having the possibility to occasionally do so allows a simpler code structure.

More imporantly, it makes it possible to use the same thread object both for the non-real-time setup phase, where a lot of object allocations take place, and the steady-state real-time phase. That proved very valuable for the EtherCAT master, as setting up the bus is very dynamic, due to the plug-and-play nature of the bus. For instance, when new slaves are discovered on the bus (typically at startup) a lot of information is sent between the master and the slave, including station address, mailbox configurations, process data mappings, etc., and having the possibility to build dynamic data structures in a straight-forward way makes the implementation easier.

By contrast, after the system has been set up and entered the steady state cyclic operation, the system is very static, and little or no dynamic memory allocation is done. Thus, the steady-state part could be implemented without using heap access, but the changes between setup and steady state get simpler, and the volume of code smaller, by having RealtimeThreads that can perform real-time tasks in the kilohertz range.

The non-real-time parts of the system, on the other hand, are quite allocation intensive. When running the graphical user interface and plotting process values (reference, actual position, velocity) during operation, with 1 *ms* resolution, the system performed a GC cycle every 5 seconds.

The RTGC scheduling of Sun RTS 2.0, allowing RealtimeThreads to run at a higher priority than the garbage collector, introduces the risk of starving the GC thread which leads to stop-the-world GC. However, tuning the RTGCCriticalReservedBytes parameter was enough to eliminate deadline misses due to memory management.

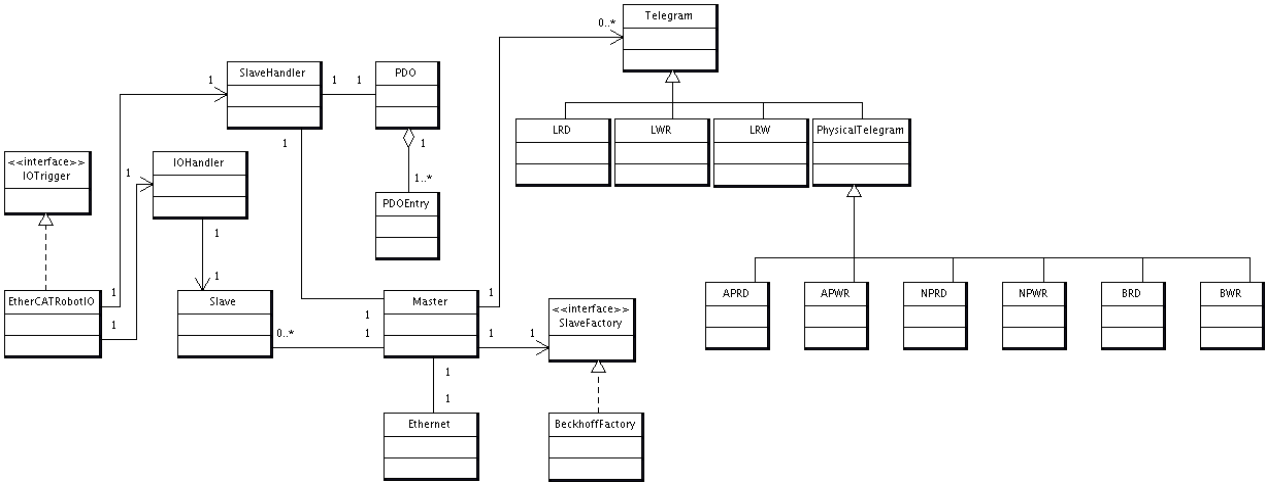

Figure 6: Class diagram of the central parts of the EtherCAT master.

From analyzing GC logs, we saw that at every garbage collection, the GC reclaimed most of the allocated memory since last cycle, so that the space used by the live objects in steady state was roughly constant. The amount of memory reclaimed during each of the garbage collections was greater than the steady state memory usage by an order of a magnitude, indicating that a lot of short-lived objects was allocated in the non-real-time parts. The GC logs also show that the GC doesn't block any of the threads, which means that the garbage collector was using the multi-CPU/multicore features of the system efficiently.

Ethernet timing measurements

For practical use in a control application, the real-time performance of all parts of the system is important. In a control application, apart from the VM itself, the I/O drivers in the operating system are important.The system presented in this paper relies on raw Ethernet frames that are accessed using a Java JNI wrapper around the Solaris DLPI interface. We will now briefly present some experimental evaluation of that wrapper.

Figure 7 shows measured network delays, from the time an ethernet frame is sent in the Java code until it is received:

```
long t1 = System.nanoTime();
ethernet.send(sendPacket);
long t2 = System.nanoTime();
ethernet.receive(receivePacket, 10);
long t3 = System.nanoTime();
long sendDelay = t2 - t1;
long receiveDelay = t3 - t2;
long totalDelay = t3 - t1;
```

Nearly all packages return within 50 μs, but out of 10^7 packages, 300 had a delay larger than 400 μs. To study the effects of cache pressure and critical paths shared between ethernet driver and other kernel code, a disturbance task was run:

```
repeat 100 \
find /usr /export -type f -exec wc {} \; >/dev/null
```

When running without additional load, the CPU usage was less than 2%, with the find process, one of the processors was fully occupied. About 60 μs of the delay is due to cache misses, as can be seen from the less steep slope when there is concurrent disk activity. Other (probably critical sections in the kernel) interactions add an extra delay of up to 500 μs. I.e. using the Solaris ethernet driver may cause jitter of up to around 500 μs (when using only a single ethernet interface; using another interface adds another 300 μs).

Figure 8 shows the same program running on a dedicated processor set. Now, the effects of running a competing find task are barely visible, but it can be noted that the fraction of packets that experience a delay of more than 400μ is slightly higher in the idle processor set example ($2 \cdot 10^{-4}$ as opposed to $8 \cdot 10^{e-5}$).

For the presented application, the real-time performance was

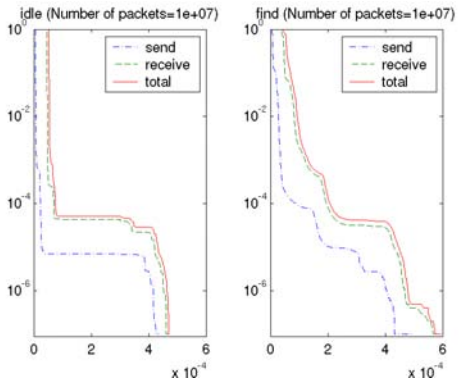

Figure 7: Network delays. The x-axis shows seconds of delay, and the y-axis shows the fraction of requests not serviced within that time. The left plot shows measurements with an otherwise idle system, the right plot shows the effects of concurrent disk accesses and CPU activity.

sufficient. Otherwise, it would be possible to write a new ethernet driver to be used for the EtherCAT communication — especially so as it is a very simple protocol, which does not require a full TCP/IP stack or even advanced package buffering, etc.

5. RELATED WORK

The main advantage of having a real-time GC is that hard real-time code can be written without using NoHeapRealtimeThreads, and thus avoiding the cumbersome memory management associated with them. Another way of dealing with that problem is the recently presented eventrons [6].

The presented application has an allocation behaviour that makes it possible to implement using eventrons; much allocation is done during the non-real-time setup phase but in steady-state operation, very little allocation is performed in the hard real-time parts, but there is still communication between real-time and non-real-time code.

While programming with eventrons requires less effort than when using NHRTs, it still is more restrictive than using RealtimeThreads and a RTGC. Therefore, for tasks with sampling rates of up to a few kHz, using RealtimeThreads gives sufficient real-time performance, and leaves more flexibility to the programmer.

6. CONCLUSION

We have implemented a control system for an industrial robot entirely in Java, and discussed our experiences from the work.

In the presented application, RealtimeThreads were used for all the time-critical tasks, including the $1\,kHz$ position control loop. Being able to access the heap from the real-time threads allows a much simpler design, and makes all the benefits of dynamic memory and automatic memory management available to the real-time programmer.

The use of a safe object-oriented language such as Java, and EtherCAT based communication (which requires no special hardware on the control computer — just an ethernet interface — and no driver development), made it possible to develop the application quite rapidly. The EtherCAT master and the low-level control code was written from scratch by two persons in less than three months.

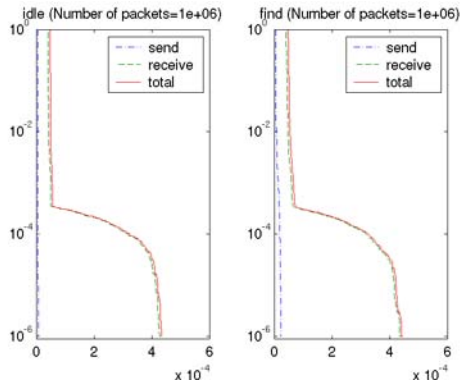

Figure 8: Network delays when the test task is mapped to a processor set.

Java SE compatibility makes development of real-time software much easier, as the bulk of the code can be developed and tested on a normal desktop system and a standard JVM. In the presented project, most of the control code (kinematics, trajectory generation, etc.) and the EtherCAT master code was developed and tested in this way. Of course, care has to be taken when writing code that will eventually run in the context of a real-time thread, but that only requires sound real-time programming, not a real-time run-time system. Furthermore, this makes the code very portable, and we're planning an open source release of the code.

In robotics, safety and predictability are critical, for instance when machine motions are controlled next to a human operator. However, standard off-the-shelf types of computing systems can still be used since it is permitted to stop the machine in case of a fault or error. That is, systems may need to be safe but not safety critical (as, for instance, a drive-by-wire system in car), and networking does not need to be redundant.

Based on our experiences we believe that real-time Java is a feasible alternative for industrial control applications. To our knowledge, this is the first robot controller implemented entirely in Java and excecuted on a certified virtual machine.

Acknowledgements

Greg Bollella and Keith Hargrove, Sun Microsystems, did much of the high-level application implementation. The algorithms for the robot kinematics and trajectory generation were implemented by Tomas Olsson and Mathias Haage, Lund University. We also want to thank ABB and Beckhoff for valuable support. This work was partially financed by Sun Microsystems. The EtherCAT master development was partially financed by the EU FP6 project SMERobot.

7. REFERENCES

[1] Ethercat technology group web site. `http://www.ethercat.org/`.

[2] G. Bollella et al. *The Real-Time Specification for Java*. Addison-Wesley, 2001.

[3] B. Delsart, T. Printezis, G. Bollella, and D. Hofert. A real-time garbage collector for a real-time java virtual machine. Technical session 2901, JavaOne conference, 2007.

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

[5] S. G. Robertz. Applying priorities to memory allocation. In *Proceedings of the 2002 International Symposium on Memory Management (ISMM'02)*, Berlin, Germany, June 2002.

[6] D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: A safe programming construct for high-frequency hard real-time applications. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI'06)*, Ottawa, Ontario, Canada, June 2006.