Real-Time control of Humanoid Robots using OpenJDK

Jesper Smith
IHMC
40 South Alcaniz St
Pensacola, Florida 32502
jsmith@ihmc.us

Douglas Stephen
IHMC
40 South Alcaniz St
Pensacola, Florida 32502
dstephen@ihmc.us

Alex Lesman
MIT
77 Massachusetts Ave
Cambridge, MA 02139
lesman@mit.edu

Jerry Pratt
IHMC
40 South Alcaniz St
Pensacola, Florida 32502
jpratt@ihmc.us

ABSTRACT

At IHMC we create state of the art walking algorithms for humanoid robots in Java. We successfully used our Java algorithms to control the Atlas humanoid, a 150kg robot with 28 actuated degrees of freedom, in the Darpa Robotics Challenge, placing second in an international competition. To execute our control loops within the deadlines imposed by the robot (1ms), we have brought POSIX real-time threads to the OpenJDK using a JNI library. Lockless synchronization primitives are used to communicate between the different threads, while Garbage Collection is avoided altogether. We have released the IHMCRealtime library created for this project under the Apache license [1].

1. INTRODUCTION

At the Florida Institute for Human and Machine Cognition (IHMC) we have been developing state-of-the-art control software for a variety of robots in Java for more than 10 years. While our choice to employ Java is often seen as a bit of an oddity in the robotics community, we feel that the productivity benefits of using Java make it an ideal choice for teams with diverse backgrounds. The most recent application of our software to a real robot was our entry in the international DARPA Robotics Challenge (DRC), where we competed using our control strategies on the Atlas robot designed by Boston Dynamics, see Figure 1.

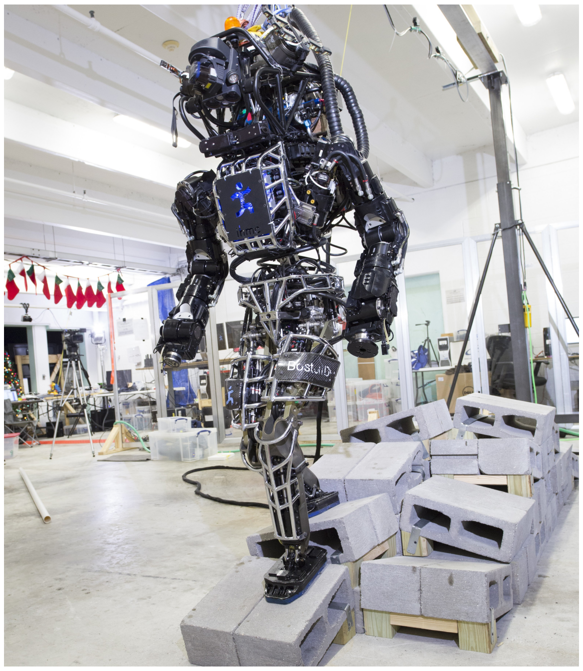

Figure 1: Atlas.

The Atlas robot is a humanoid with size comparable to that of a large adult male, requiring a great deal of computational power to control successfully. Controlling the robot requires processing input from a wide array of sensors as well as computing output from our control software to command the robot to walk, perceive, and manipulate. Successfully controlling a machine like this requires a very different approach from traditional factory or industrial robotics where the controlled devices are fixed to a known point in a known space; simply commanding desired positions in space is not adequate in the types of environments that the Atlas robot operates in. Control engineering for complex legged humanoids that can balance dynamically benefits greatly from architectures that employ real-time execution constraints similar to other complex control domains like avionics and automated medical tooling. Control commands have to be provided to a humanoid every 10ms or less, larger delays result in the robot falling over; in addition, we need to process incoming data at 1ms in order to accurately estimate the current state of the system. Our particular architecture leverages two separate threads of execution with real-time deadlines, running

[1]https://bitbucket.org/ihmcrobotics/ihmcrealtime

JTRES '14, October 13 - 14 2014, Niagara Falls, NY, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2813-5/14/10...$15.00.
http://dx.doi.org/10.1145/2661020.2661027

on a fully-preemptible soft real-time kernel. One thread is used to estimate the state of the robot in the world based on information from its sensors, and the other is used to solve the control-domain equations using input from the state estimator and in turn commanding all of the powered joints on the robot, leading to locomotion and manipulation. This control strategy was shown to be highly effective by leading us to a first place finish in the Virtual Robotics Challenge (VRC) and a second place finish in the VRC Trials running the same algorithms. For the VRC, very soft real-time constraints had to be met and the standard OpenJDK virtual machine provided the performance we needed [8].

In order to meet the much harder real-time constraints imposed by a physical system, we tested several commercial real-time virtual machines including IBM WebSphere RT for RT Linux [3], Atego Perc [1], aicas JamaicaVM [12] and the older Sun JRTS implementation [2]. While these solutions would all allow for meeting of deadlines they proved to sacrifice too much in the way of performance, with the best case measured execution time failing to meet the deadlines required by our controller. They also tended to lag behind the current language specification, posed licensing or pricing that was restricting for an organization of our size, and some were limited to 32-bit implementations. Therefore, we decided to implement the minimum real-time support required to run our controller in an open source library, called IHMCRealtime.

In section 2, we will introduce the control architecture and computational requirements. In section 3 we compare various commercially available real-time Java solutions. In section 4 we describe our approach to bring the necessary real-time guarantees to the OpenJDK virtual machine. In section 5 we provide results of our real-time implementation.

2. CONTROL ARCHITECTURE

The DARPA Robotics Challenge dictates a certain computation and networking architecture; partially as a way to deliver a level playing field and partially as a way for the organizers to impose certain effects on the competition.

Figure 2 show a high level overview of our command an control architecture. The Atlas robot has an onboard computer(which is a black box) and a Multisense-SL sensor head that communicate over a high bandwidth, low latency fiber-optic network link to two separate field computers. The field computers are stand-ins for a future Atlas design which will have two developer-accessible computers on-board. One of the field computers runs a vision/networking process, while the other computer runs the real-time IHMC full body controller process. In addition, during testing in our lab, a third field computer acts as a logging system; it receives the full state of the controller at a regular rate, which it saves to disk for later review and analysis using in-house tools.

The user controls the robot using a user interface developed by IHMC. The user interface shows the user the video feed from the robot, as well as a 3D world based on the LIDAR data. The user provides high level commands ,for example desired footstep locations, to the IHMC full body controller. The user interface communicates with the field computers trough a network shaper. The network shaper introduces latency and limits the available bandwidth in order to simulate degraded communication in a disaster situation.

The IHMC full body controller consists of two major parts

- State estimator
  Reads joint angles, velocities and Inertial Measurement Unit orientation, rotational velocity and linear acceleration from the robot and uses this data to estimate linear velocity and the position in a defined "world" coordinate system. The state estimator is also responsible for filtering incoming signal data from the robot.
- Controller
  The implementation of the full body control and walking algorithm. Computes the desired joint forces necessary for a stable walking gait of the robot, including during compound tasks like manipulating objects that may introduce reaction forces that must be accounted for (e.g. turning a tight valve, carrying a tool, etc.). Desired tasks are received asynchronously from the user interface and are used to plan the next motions.

Both the controller and the state estimator have their own periodic real-time deadlines. We will refer to the real-time deadlines as loop rates. The internal state of the state estimator and controller is communicated to the visualization and logging system at the estimator loop rate. About 8000 variables are currently monitored.

2.1 Runtime constraints

To allow our robots to handle disturbances and unmodeled obstacles in the environment, we control the robot using compliant force control [11]. With force control, desired joint forces have to be provided at a fast and predictable rate. Large delays lead to instabilities in the numerical algorithms employed in the controller due to over-shooting of the desired configuration. Furthermore, the robot is not statically stable while taking steps. This means we must actively command the joints during the leg swing phases of walking and failure to meet deadlines during leg swing will lead to the robot falling.

Our experience tells us that the maximum acceptable delay between measuring the robot state and providing force commands is about 10ms. Subtracting the delay introduced by the communication with the robot, which is approximately 3ms, we need to provide commands to the robot within approximately 7ms.

2.2 Computational cost

Currently, we are using a relatively simple scheme in the state estimator. Joint positions and velocities are filtered using a very simple first order filter and a Kalman filter is used to find the center of mass position and velocity. These quantities are extremely important for controlling a walking robot and form the core of many modern walking algorithms.

For each execution of the controller, a full model of the current configuration of the robot is computed based on the

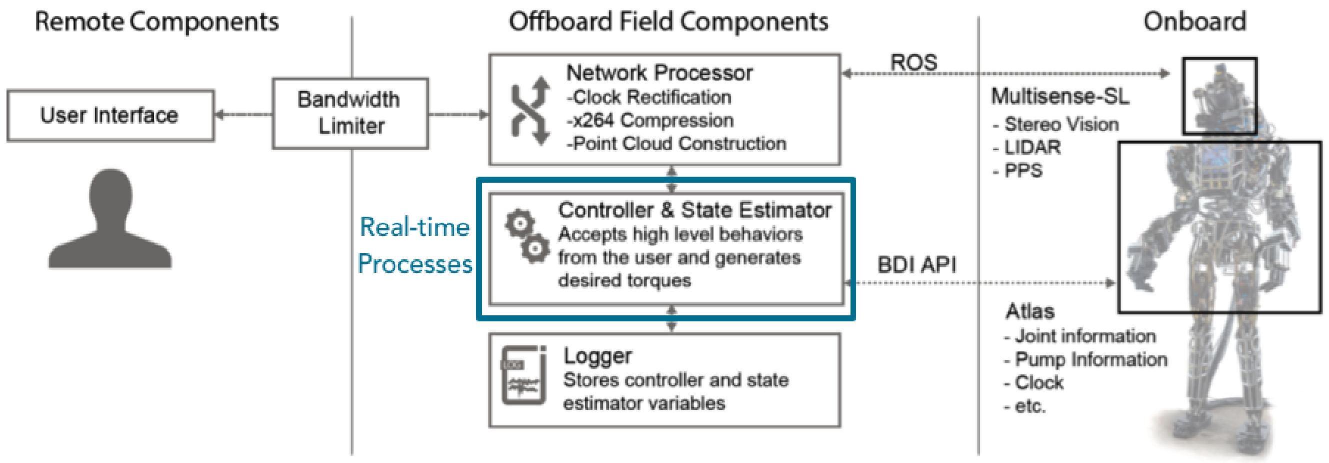

Figure 2: High level architecture of our command and control setup for the Atlas robot.

Table 1: Execution time of the state estimator and controller is simulation taken on a 2.6GHz Intel i7-4960HQ with 16GB of RAM.

	Average	Maximum
State estimator	$0.15ms$	$0.26ms$
Controller	$2.5ms$	$4.6ms$

state received from the state estimator. This model includes the spatial relationship between all joints and rigid bodies on the robot. Our control algorithm is designed for controlling the whole body of the robot in order to achieve specified goals [8]. This includes computing quantities such as desired forces exerted on the environment for balance or motion (e.g. effort on the feet at the ankles, pushing against walls with the hands, etc.) as well as desired poses of various parts of the robot such as the pelvis, chest, head and hands. Using this information, desired rotational accelerations of the joints are calculated using Quadratic Programming techniques, a form of mathematical optimization. The computed full model of the robot is then used to calculate desired joint forces based on the desired accelerations. A large portion of the calculations involve 4-by-4 transformation matrices. The eventual goal of the controller is to run on a fast mobile processor embedded on the robot. We timed our control algorithm in simulation on a mobile Intel i7-4960HQ (2.6GHz base frequency) with 16GB of RAM, results are shown in Table 1.

2.3 Multi-Threading

Previous controllers written at IHMC were designed to use a single threaded model, greatly simplifying the software architecture. Atlas provides control data at a steady 1ms rate, which needs to be handled by the state estimator. The higher rate data the state estimator receives, the more accurate the estimation becomes. Therefore, we need to run the state estimator in lockstep with the rate at which data is received, leading to a 1ms period/deadline. Total execution time for the controller and state estimator from the simulation precludes running this in the same 1ms control tick. This forces us to use a multi-threaded approach for controlling our robot. The main threads running our software are described in Table 2. We selected a 6ms loop rate for the controller, which gives us ample overhead for new features while still providing new control commands to the robot below the 7ms upper bound. A schematic of the threading architecture is shown in Figure 3. Data is communicated from the Atlas robot toward the state estimator, which processes and publishes the data for the controller to use. Both the state estimator and controller communicate their internal state to the logger.

Table 2: Main threads running the control software on the robot.

Thread description	Priority	Execution rate
State estimator	Real-time	$1ms$
Controller	Real-time	$6ms$
Logging	Non-real-time	$1ms$
Networking	Non-real-time	on-demand

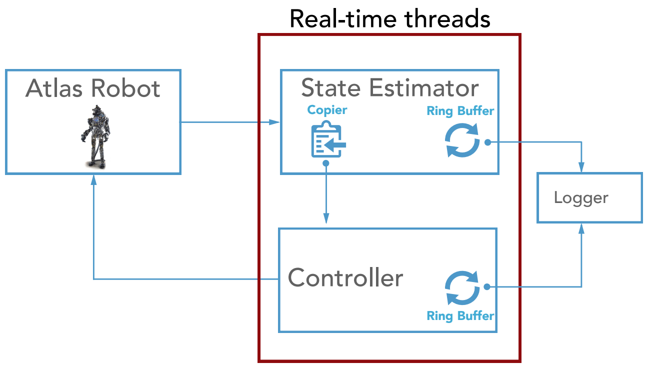

Figure 3: Block diagram of the threading flow.

2.4 Scheduling

Data from the Atlas robot comes in at a nominally $1ms$ rate, with a maximum of 20% timing jitter. This jitter comes from the proprietary software on the Atlas robot and is outside of our control. The state estimator executes immediately upon receiving new data. The wall time t_{est} on data arrival is communicated to the controller together with the robot

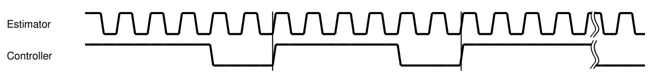

Figure 4: Timing diagram of the state estimator and controller threads.

state.

At the beginning of each controller execution, it reads the state published by the last state estimator execution. After writing the desired forces to the robot, the controller thread will sleep until $t_{wall} = t_{est} + dt_{est} + dt_{con}$. Here dt_{est} is the state estimator loop rate, and dt_{con} is the controller loop rate. This guarantees that the controller and state estimator run nominally in a lock-step fashion, even with hiccups in the data from the robot or clock drift between the onboard Atlas computer and our control computer. A timing diagram is shown in Figure 4.

3. VIRTUAL MACHINE COMPARISON

During the VRC portion of the Challenge, we were forced to use a specific execution environment normalized across all of the competing teams and managed by external agents. This limited the amount of work we could do in order to enforce our desired deadlines. Primarily, the execution environment would not allow for custom kernel installation, and the simulated physics environment inside of which the virtual challenges took place only enforced very soft real-time constraints with a large window on updates to simulated rigid bodies. Combined with the fact that the physics loop existed asynchronously from our control and state estimator loops meant that non-deterministic updates to the state of the simulated robot lead to difficulty in enforcing control deadlines with the strictness that we prefer. We did not begin investigating real-time execution environments until after the VRC had ended. We began by investigating the off-the-shelf commercial offerings for real-time execution of Java code.

3.1 Benchmarks

Our previous real-time execution environment for our older controllers was the Sun JRTS implementation of JSR 282 [5]. However, as the Sun JRTS ceased to be actively developed and supported, we investigated other options allowing for modern language features introduced after the Sun implementation stalled. We evaluated several virtual machine options including Atego PERC, and the JamaicaVM and ahead-of-time compiler options offered by aicas. IBM's websphere was ruled out for non-technical reasons. We assumed that the computational determinism of these products met our requirements and focussed our comparison on performance; our software is computationally demanding and some of our deadlines were fixed by the robots architecture, meaning that sacrificing too much performance for increased determinism was not an option.

In addition, any option that required an ahead-of-time compilation step needed to be evaluated in terms of how it impacted our ability to deploy changes to the robot rapidly. We tested all options with a suite of micro-benchmarks for evaluating several real-time execution options across several numerical benchmarks including Whetstone [4], Dhrystone [14], and SciMark 2 [9].

Figure 5 shows a condensed version of the benchmark results; only the best performers out of all of our evaluations on the most computationally taxing - and similar to our use-case - test (SciMark 2 "Large") are presented. It is clear that the commercially available real-time virtual machines lag the performance of the OpenJDK by a significant amount. Furthermore, ahead-of-time compilation of our code base took well over an hour for all virtual machines that provided that option, while still incurring a large performance penalty. In addition to not meeting performance needs, many of these implementations did not implement the 1.7 language specification or are 32-bit only, making them incompatible with the 64bit driver provided for Atlas.

4. BRINGING REAL-TIME GUARANTEES TO THE OPENJDK VIRTUAL MACHINE

Extrapolating from the benchmarks shown in Figure 5, we assume that the execution times of the controller as shown in Table 1 will increase at least three times using available real-time virtual machines. Running the VRC controller inside the real-time virtual machines showed that the performance penalty was even larger than the synthetic benchmarks. Consequently, we concluded that the commercially available real-time virtual machines did not have the performance necessary to control our robot using the algorithms developed for the Virtual Robotics Challenge. Therefore, we brought the bare minimum real-time capabilities to run our control algorithm on the real robot to the OpenJDK.

4.1 Real-time priority

Our controllers are deployed on Linux systems with the RT_PREEMPT [10] patch applied to the kernel. We created a RealtimeThread class that invokes native C++ code to setup a POSIX real-time thread. The native thread is attached to the JVM and calls a user defined Java callback function. Periodic execution is facilitated by providing access to the POSIX call clock_nanosleep through the JNI layer.

4.2 Garbage collection

A major feature of real-time virtual machines and one of the main difficulties in implementing a real-time Java solution is a deterministic garbage collector. Implementing one ourselves would take an enormous effort. Therefore, we make sure we do not invoke the garbage collector during realtime control.

To get acceptable real-time performance in our previous project, we implemented our support libraries with a minimum amount of object allocation. We continued this practice in the VRC and the control algorithm itself allocates a relatively small amount of objects during execution. The DRC competition put a 30 minute time limit on any of the tasks, limiting the maximum execution time of our controller. Small, profiler guided, optimizations reduced the size of newly allocated objects to less than 10GB in 30 minutes.

To accommodate these allocations without invoking the garbage collector, we allocate a large, 12GB new generation heap (-XX:NewSize=12g -XX:MaxNewSize=12g). Because of its

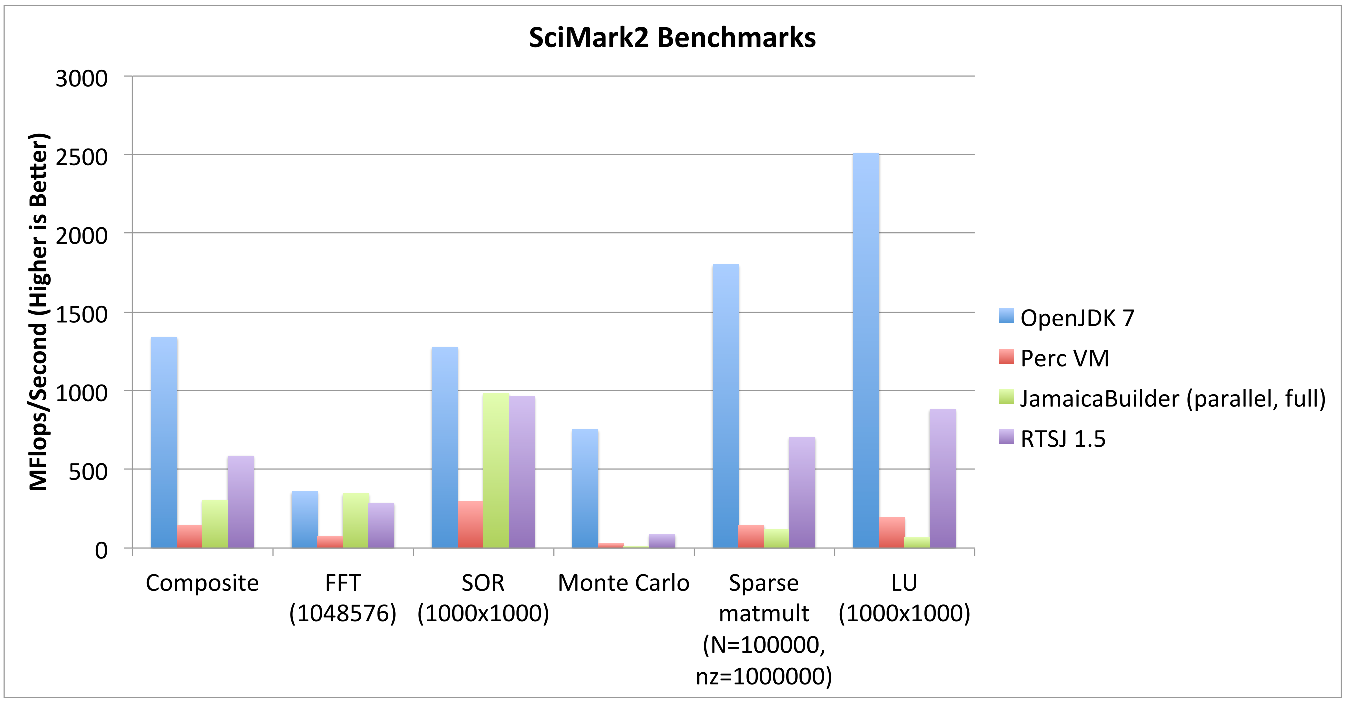

Figure 5: OpenJDK Java 7 compared to offerings from Perc, the full ahead-of-time Jamaica compile with parallelism, and the Sun RTSJ 1.5. It is clear that the performance of the available real-time virtual machines lags the performance of the OpenJDK by a significant amount.

relatively predictable nature, we chose to employ the serial garbage collector (-XX:+UseSerialGC). While System.gc() does not guarantee a collection cycle, studying the OpenJDK source code shows that System.gc() forces a collection cycle to happen when using the Serial GC. Therefore, we call System.gc() to clean up objects allocated in the setup process before starting the periodic threads for the first time.

4.3 JIT Compilation

Another source of non-determinism in the OpenJDK virtual machine comes from the JIT compiler. We chose to employ a small compile threshold (-XX:CompileThreshold=1000) to force early optimization of hot spots. The controller output is not applied to the robot for approximately 10 seconds after startup to allow the JIT compiler to optimize the main code path.

4.4 Synchronization

The OpenJDK does not protect against priority inversion, especially when using native threads. When using built-in locks to synchronize data, the logging and visualization thread or the networking thread could be preempted and possibly block the execution of either the controller thread or state estimator thread. Therefore, we choose to use lockless synchronization to share data between threads.

The existing classes for lockless synchronization inside OpenJDK, the main one being ConcurrentLinkedQueue [7], have as major disadvantage that a small amount of objects are allocated for each object added to the queue. We created two classes that allow high performance lockless synchronization between threads, based on a single publisher and single subscriber.

- ConcurrentRingBuffer
  Based on the LMAX-Disruptor[13] we created a concurrent ring buffer that is lock free and does not allocate objects when adding elements. An volatile long is used to synchronize the position of the producer and the consumer inside the ring buffer. Writing to an volatile long inserts a memory write barrier, guaranteeing data visibility between threads. Multiple elements can be added to the buffer with a single write to the volatile long.

- ConcurrentCopier
  For the communication between the state estimator and controller we are only interested in the latest data and all data that came before can be discarded. Therefore we implemented the ConcurrentCopier, a simplified buffer were only a single element is available for reading at a time. Inserting a new element before the previous element has been read results in an overwrite of the previous element. Internally, a pre-allocated three element buffer is allocated. The producer thread can request an element to write to. When the producer thread is finished writing to the object, it can commit it. The consumer thread can request access to the latest written element. When the consumer requests the next element, it relinquishes control of the previous element. The producer thread writes to the element that is currently not being read by the consumer thread. A single volatile integer is used to hold the state of the buffer.

For non-frequent data such as control commands we use the ConcurrentLinkedQueue, as the amount of objects allocated is small in that case.

4.5 Processor Affinity

To reduce jitter in the loop rates due to migration of the threads between different processor cores, we created an easy to use Java wrapper around the POSIX call sched_setaffinity, allowing the developer to pin the current thread to a single CPU core. Further decreasing non-determinism is done by isolating CPU cores from the kernel scheduler and disabling hyper threading.

5. RESULTS

Our real-time additions to the OpenJDK are written to directly support the walking software. In this section we will show that our implementation meets the deadlines imposed by the robot system. We first show synthetic benchmarks for our synchronization primitives and then provide timing results from running the control software on the robot. All benchmarks were executed on a Intel Xeon E5-2667 v2 (3.30GHz) with 64GB of RAM running Ubuntu 12.10 with Linux kernel 3.12.1 patched with RT_PREEMPT.

5.1 Synthetic benchmarks

We tested the throughput of the ConcurrentRingBuffer based on the number of messages passed per second. The producer thread changes and commits a single mutable long element as fast as possible, possibly spin-locking when no free elements are available in the buffer. The consumer thread polls as fast as possible. Correctness is checked by setting the long element to a simple pseudo-random value. The time the consumer takes to read all values is measured using System.nanoTime(). Results are shown in Figure 6. The throughput increases sharply after start-up, converging to approximately 77 million messages/second. We attribute the increase to the optimization of the code path by the JIT compiler. This means that the ConcurrentRingBuffer will have minimum overhead when used for copying the data from the state estimator and controller to the logger at a rate of 1000 messages/second.

5.2 Robot measurements

To establish the baseline of the real-time we measured the jitter on the incoming data from Atlas. This data is transferred from the onboard proprietary computer to our control software at a rate of $1ms$. A frequency graph of time difference between two data packets, dt, is shown in Figure 7, taken over a 60 second interval. The smallest dt was $0.872ms$ while the largest dt measured was $1.18ms$. The average dt was $0.994ms$. A non-significant amount of jitter is introduced by the onboard system on the robot and the network communication.

5.2.1 State estimator and Controller computation time

To show that our state estimator and controller are capable of meeting the deadlines imposed (1ms for the state estimator, 6ms for the controller) we measured the average

After a minute warmup period, we measure the execution time of the controller and state estimator. We also measure the total computation time, defined as the time between the

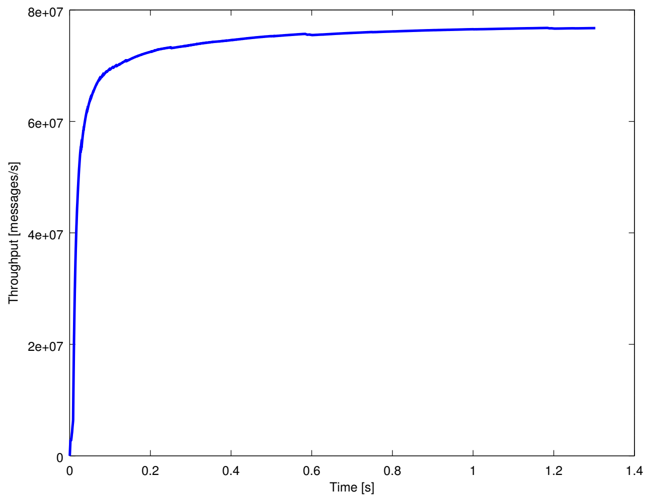

Figure 6: Throughput of the ConcurrentRingBuffer measured versus time.

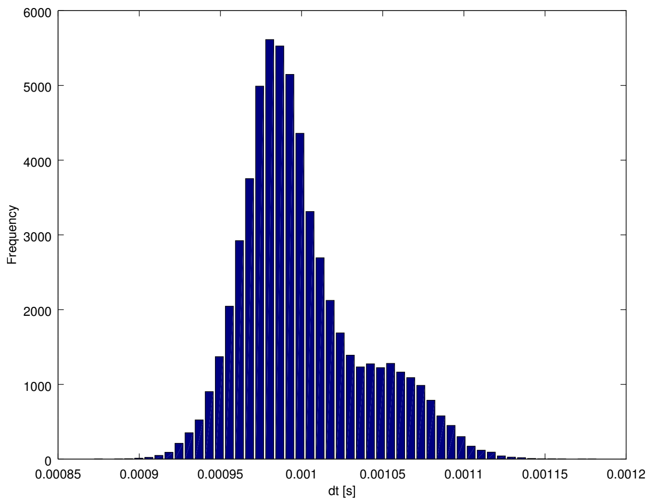

Figure 7: Histogram of the time differential between incoming packets from the Atlas robot.

Table 3: Non-synchronization events in the controller.

	Controller lags state estimator events	Controller leads state estimator events	Number of non-synchronized events
Run 1	0	2	2
Run 2	0	0	0
Run 3	1	1	2

Table 4: Execution time of the state estimator, controller and total computation time between receiving robot state and sending control commands.

State estimator DT	$0.12ms$	$0.13ms$	$0.20ms$
Controller DT	$1.3ms$	$1.4ms$	$2.0ms$
Total delay	$2.24ms$	$2.51ms$	$3.34ms$

moment the state of the robot is received and a control command calculated from that state is sent to the robot. Results for a 10 minute run, allowing for a one minute warmup, are shown in table 4.

5.2.2 Missed deadlines

During the Darpa Robotics Challenge Trials, the maximum runtime of the robot is 30 minutes. During this time, the controller and state estimator should ideally not miss any deadlines in this period.

Table 3 shows the amount of missed deadlines in a 30 minute run, recorded after a minute initialization time. A controller lags state estimator event is defined as an event where the controller executes with state estimator data that is less than 6 state estimator ticks old. Consequently, A controller leads state estimator event is defined as an event where the controller executes with state estimator data that is more than 6 state estimator ticks old. In both cases, the state estimator and controller are not in synchronization.

The number of non-synchronized events is extremely low. The robot can recover from a single missed deadline, given that it is not followed by another missed deadline shortly after. At no point in time was the controller more than 2ms out of synchronization with respect to the state estimator. A single garbage collection event was seen in several runs around the 25 minute marks, but this was extremely short (less than 0.3ms) and did not affect thread synchronization.

6. DISCUSSION

We successfully implemented the bare minimum real-time guarantees necessary to use our control software in combination with the physical Atlas robot. Using lockless synchronization primitives and avoid allocating objects we can use OpenJDK to control the Atlas robot within the strict deadlines imposed by the physical system. Control deadlines are reliably met during the 30 minute runs. During approximately 300 hours of runtime with this control architecture we have not seen instability or falls that can be attributed to the real-time performance of our solution. However, we cannot provide guarantees that internal JVM housekeeping tasks like JIT compilation and garbage collection will not preempt the controller. For now, we have not had any humans within the reach of the robot and Atlas is robust enough to handle unexpected control commands, avoiding any danger to human life and financial burden.

We have chosen Java as it is an easy to use programming language compared to traditional embedded languages such as C/C++ while providing comparable performance. This allows us to quickly bring visiting researchers and interns with a mechanical or control engineering background up to speed and have them implementing their control algorithms within a few weeks. The additional safety provided by Java compared to C/C++ provides an additional safe-guard against non-obvious bugs. Using the OpenJDK instead of a proprietary solution not only gives us a performance benefits. A free and open runtime is extremely useful for collaboration with external research groups, as the cost and especially effort of acquiring a copy of the runtime is very low. Another main advantage of the OpenJDK instead of a commercial real-time VM is that we have the guarantee to always support the latest version of the Java specification. To foster collaboration and convince more robotic scientists to look at Java as a feasible solution for real-time control we released our real-time routines and helper classes as an Open Source project under the Apache 2.0 license, called IHMCRealtime. The source code can be found on BitBucket [2].

7. FUTURE WORK

Up until now, the control computer we have used has been a high performance Intel Xeon E5-2667 v2 (3.30 GHz) server with 64GB of RAM connected using a 10GBit/s fiber optic tether to the robot. In the near future, this computer will be replaced by an onboard embedded mobile processor (Intel i7-4700EQ) with a maximum of 16GB of RAM. Initial tests have shown that the performance is acceptable, but more data needs to be collected. During the DRC finals the robot will have to run continuously for over an hour, while the memory of the onboard computer is limited. Increasing the heap size is thus not an option and the current object allocation is non-trivial to reduce. We are looking at instrumenting the heap usage to determine when garbage collection is necessary. We then plan to freeze the robot in a stable state using the onboard position controllers and halting the controller for several seconds while a garbage collection cycle is executed, as proposed in project Golden Gate [6]. Finally, should there be interest from the community, we are looking at implementing a special lock that avoids priority inversion within the IHMCRealtime package. We propose to do this by exposing POSIX mutexes through a JNI wrapper. While not useful for a project with throughput requirements like our control software, it might simplify programming real-time controllers with lower requirements.

[2]https://bitbucket.org/ihmcrobotics/ihmcrealtime

8. REFERENCES

[1] Atego perc ultra smp. http://www.atego.com/products/atego-perc-ultra-smp/, 2004. [Online; accessed 2014-06-23].

[2] Sun java real-time system 2.2. http://docs.oracle.com/javase/realtime/rts_productdoc_2.2u1.html, 2010. [Online; accessed 2014-06-23].

[3] D. F. Bacon, P. Cheng, and V. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *On the Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 466–478. Springer, 2003.

[4] H. J. Curnow and B. A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, 1976.

[5] P. Dibble. Jsr 282: Rtsj version 1.1.

[6] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Project golden gate: towards real-time java in space missions. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 15–22. IEEE, 2004.

[7] B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice.* Addison-Wesley, 2006.

[8] T. Koolen, J. Smith, G. Thomas, S. Bertrand, J. Carff, N. Mertins, D. Stephen, P. Abeles, J. Englsberger, S. Mccrory, J. V. Egmond, M. Griffioen, M. Floyd, S. Kobus, N. Manor, S. Alsheikh, D. Duran, L. Bunch, E. Morphis, L. Colasanto, K.-l. H. Hoang, B. Layton, P. D. Neuhaus, M. J. Johnson, and J. E. Pratt. Summary of team ihmc's virtual robotics challenge entry. In *Proceedings of the IEEE-RAS International Conference on Humanoid Robots.*, Atlanta, GA, 2013. IEEE.

[9] B. Miller and R. Pozo. Java scimark 2.0. http://math.nist.gov/scimark2/, 2004. [Online; accessed 2013-06-30].

[10] I. Molnar. Preempt-rt, 2008.

[11] J. Pratt, T. Koolen, T. De Boer, J. Rebula, S. Cotton, J. Carff, M. Johnson, and P. Neuhaus. Capturability-based analysis and control of legged locomotion, part 2: Application to m2v2, a lower body humanoid. *The International Journal of Robotics Research*, page 0278364912452762, 2012.

[12] F. Siebert. Realtime garbage collection in the jamaicavm 3.0. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 94–103. ACM, 2007.

[13] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads, 2011.

[14] R. P. Weicker. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, Oct. 1984.