Hard real-time Control and Coordination of Robot Tasks using Lua

Markus Klotzbuecher
Katholieke Universiteit Leuven
Celestijnenlaan 300B, Leuven, Belgium
markus.klotzbuecher@mech.kuleuven.be

Herman Bruyninckx
Katholieke Universiteit Leuven
Celestijnenlaan 300B, Leuven, Belgium
herman.bruyninckx@mech.kuleuven.be

Abstract

Control and Coordination in industrial robot applications operating under hard real-time constraints is traditionally implemented in languages such as C/C++ or Ada. We present an approach to use Lua, a lightweight and single threaded extension language that has been integrated in the Orocos RTT framework. Using Lua has several advantages: increasing robustness by automatic memory management and preventing pointer related programming errors, supporting inexperienced users by offering a simpler syntax and permitting dynamic changes to running systems. However, to achieve deterministic temporal behavior, the main challenge is dealing with allocation and recuperation of memory. We describe a practical approach to real-time memory management for the use case of Coordination. We carry out several experiments to validate this approach qualitatively and quantitatively and provide robotics engineers the insights and tools to assess the impact of using Lua in their applications.

1 Introduction

This work takes place in the context of component based systems. To construct an application, computational blocks are instantiated and interconnected with anonymous, data-flow based communication. Coordination refers to the process of managing and monitoring these functional computations such that the system behaves as intended. Keeping Coordination separate from Computations increases reusability of the latter blocks as these are not polluted with application specific knowledge. Examples of typical coordination tasks are switching between controllers upon receiving events, reconfiguring computations and dealing with erroneous conditions. A complex example of an robot applications constructed using this paradigm can be found here [1].

To implement coordination we propose to use the Lua [2] extension language. Using an interpreted language for this purpose has several advantages. Firstly, the robustness and hence safety, of the system is increased. This is because scripts, in contrast to C/C++, can not easily crash a process and thereby bring down unrelated computations that are executed in sibling threads. This property is essential for the aspect of coordination, which, as a system level concern, has higher robustness requirements than regular functional computations.

Secondly, the use of a scripting language facilitates less experienced programmers not familiar with C/C++ to construct components. This is important for the robotics domain, where users are often not computer scientists. Moreover, rapid prototyping is encouraged while leaving the option open to convert parts of the code to compiled languages after identification of bottlenecks. At last, the use of an interpreted language permits dynamic changes to a running system such as hot code updates. This is essential for building complex and long running systems that can not afford downtime.

The major challenge of using Lua in a hard real-time context is dealing with allocation and recuperation of memory. Previously we sketched two strategies to address this: either running in a zero-allocation mode and with the garbage collector deactivated or in a mode permitting allocations from a preallocated memory pool using a O(1) allocator and with active but controlled garbage collection [3]. In practice, especially when interacting with C/C++ code it may be inconveniant to entirely avoid collections, hence now we consider it necessary to run the garbage collector.

The rest of this paper is structured as follows. The next section gives an overview over related work. Section 3 describes how we address the issue of memory management in a garbage collected language used for coordination. Section 4 describes four experiments with the two goals of demonstrating the approach and giving an overview of the worst-case timing behavior to be expected. Robustness is discussed in the context of the last experiment, a coordination statechart. We conclude in section 5.

2 Related work

The Orocos RTT framework [4] provides a hard real-time safe scripting language and a simple state machine. While both are much appreciated by the user community, the limited expressivity of the state machine model (e.g. the lack of hierarchical states) and the comparably complex implementation of both scripting language and state machines have been recognized as shortcomings. This work is an effort to address this.

The real-time Java community has broadly addressed the topic of using Java in hard real-time applications [5]. The goal is to use Java as a replacement to C/C++ to build multi-threaded real-time systems. To limit the impact of garbage collection parallel and concurrent collection techniques are used [6]. For our use case of building domain specific coordination languages we chose to avoid this complexity as coordination can be defined without language level concurrency. In return this permits taking advantage of the deterministic behavior of a single threaded scripting language.

The Extensible Embeddable Language (EEL) [7] is a scripting language designed for use in real-time application such as audio processing or control applications. Hence, it seems an interesting alternative to Lua. Lua was ultimately chosen because of its significantly larger user community.

3 Approach

To achieve deterministic allocations, Lua was configured to use the Two-Level Segregate Fit (TLSF) [8] O(1) memory allocator. This way memory allocations are served from a pre-allocated, fixed pool. Naturally, this raises the issue of how to determine the required pool size such that the interpreter will not run out of memory. We address this in two ways. Firstly, by examining memory management statistics the worst case memory consumption of a particular application can be determined and an appropriate size set. Due to the single threaded nature of Lua a simple coverage test can give high confidence that this value will not be exceeded in subsequent runs. Furthermore, to achieve robust behavior the current memory use is monitored online and appropriate actions are defined for the (unlikely) case of a memory shortage. What actions are appropriate depends on the respective application.

This leads to the second challenge for using Lua in a hard real-time context, namely garbage collection. In previous work [3] we suggested to avoid garbage collection entirely by excluding a set of operations that resulted in allocations. However, in practical applications that transfer data between the scripting language and C/C++ this is not always possible. Consequently the garbage collector can not be disabled for long periods and must be either automatically or manually invoked to prevent running out of memory. For achieving high determinism, it is necessary to stop automatic collections and to explicitly invoke incremental collection steps when the respective application permits this. Only this way it can be avoided that an automatic collection takes place at an undesirable time.

The Lua garbage collector is incremental, meaning that it may execute the garbage collection cycle in smaller steps. This is a necessary prerequisite for achieving low garbage collection latencies, although of course no guarantee; ultimately the latency depends on various factors such as the amount of live data, the properties of the live data[1] and the amount of memory to be freed. The control and coordination applications we have in mind generally tend to produce little garbage because the scripting language is primarily used to combine calls to C/C++ code in meaningful ways. Even though, to achieve high

[1] In Lua, for instance, tables are collected atomically. Hence large tables will increase the worst-case duration of an incremental collection step.

robustness the worst-case duration of the collection steps can be monitored to deal robustly with possible timing violations.

The following summarizes the basic approach. First, the desired functionality is implemented and executed with a freely running garbage collector. This serves to determine the maximum memory use from which the necessary memory pool size can be inferred by adding a safety margin (e.g. the maximum use times 2). Next, the program is optimized to stop the garbage collector in critical paths and incremental steps are executed explicitly. The worst case timing of these steps is benchmarked, as is the overall memory consumption. The program is then executed again with the goal to confirm that the explicitly executed garbage collection is sufficient to not run low on memory.

4 Experiments

In this section we describe the experiments carried out to assess worst-case latencies and overhead of Lua compared to using C/C++ implementations. All tests are executed using Xenomai [9] (v2.5.6 on Linux-2.6.37) on a Dell Latitude E6410 with an Intel i7 quad core CPU and 8 GiB of RAM, with real-time priorities, current and future memory locked in RAM and under load.[2] Apart from the cyclictest all tests are implemented using the Orocos RTT [4] framework. The source code is available here [15].

4.1 Lua Cyclictest

The first test is a Lua implementation of the well known cyclictest [10]. This test measures the latency between scheduled and real wake up time of a thread after a request to sleep using `clock_nanosleep(2)`. The test is repeated with different, absolute sleep times. For the Lua version, the test is run with three different garbage collector modes: `Free`, `Off` or `Controlled`. `Free` means the garbage collector is not stopped and hence automatically reclaims memory (the Lua default). `Off` means the allocator is stopped completely[3] by calling `collectgarbage('stop')`. `Controlled` means that the collector is stopped and an incremental garbage collection step is executed after computing the wake up time statistics (this way the step does not add to the latency as long as the collection completes before the next wake up).

The purpose of this test is to compare the average and worst case latencies between the Lua and C version and to investigate the impact of the garbage collector in different modes.

Results The following table summarizes the results of the cyclictest experiments. Each field contains two values, the average ("a") and worst case ("w") latency given in microseconds, that were obtained after fifteen minutes of execution.

sleep time	500	1000	2000	5000	10000
	a, w	a, w	a, w	a, w	a, w
C	0, 35	0, 31	0, 45	1, 35	1, 30
Lua/free	2, 41	2, 39	3, 39	3, 45	5, 46
Lua/off	2, 38	2, 39	3, 38	3, 43	5, 38
Lua/ctrl	2, 38	2, 42	3, 37	3, 36	5, 46

Comparing the C cyclictest with the Lua variants as expected indicates that there is an overhead of using the scripting language. The difference between the three garbage collection modes are less visible. The table below shows the average of the worst case latencies in microseconds and expressed as a ratio to the average worst case of C. Note that the average of a worst-case latency is only meaningful for revealing the *differences* between the four tests, but not in *absolute* terms. A better approach might be to base the average on the 20% worst-case values.

test	WC avg (us)	ratio to C
C	35.2	1
Lua/free	42	1.19
Lua/off	39.2	1.11
Lua/ctrl	39.8	1.13

The above table shows that a freely running garbage collector will introduce additional overhead in critical paths. Running with the garbage collector off or triggered manually at points where it will not interfere add approximately 11% and 13% respectively compared to the C implementation. Of course the first option is only sustainable for finite periods. 13% of overhead does not seem much for using a scripting language, however it should be noted that this is largely the result of *only* traversing the boundary to C twice: first for returning from the sleep system call and secondly for requesting the current time.

[2] `ping -f localhost`, and `while true; do ls -R /; done`.

[3] This is possible because the allocations are so few that the system does not run out of memory within the duration of the test.

4.2 Event messages round trip

The second experiment measures the timing of timestamped event messages sent from a requester to a responder component, as shown in Figure 1. The test simulates a simple yet common coordination scenario in which a Coordinator reacts to an incoming event by raising a response event, and serves to measure the overhead of calls into the Lua interpreter. The test is constructed using the Orocos RTT framework and is implemented using event driven ports connected by lock free connections. Both components are deployed in different threads. Three timestamps are recorded: the first before sending the message, the second at the responder side and the third on the requester side after receiving the response. The test is executed using two different responder components implemented in Lua and C++.

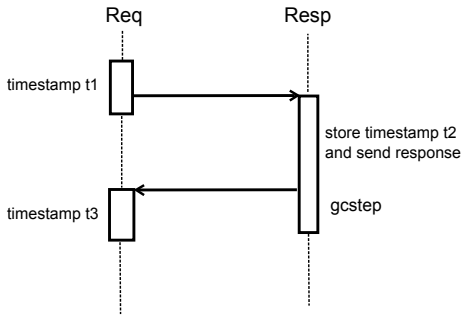

FIGURE 1: *Sequence diagram of event round trip test.*

For the Lua responder, this application takes advantage of the fact that the requester component will wait for 500us before sending the next message and executes an incremental garbage collection step after sending each response. If this assumption could not be made, the worst-case garbage collection delay would have to be added to the response time (as is the case for experiment 4.3).

Results The following table summarizes the average ("a") and worst-case ("w") duration of this experiment for the request $(t2 - t1)$, response $(t3 - t2)$ and total round trip time $(t3 - t1)$; all values in microseconds.

	req	resp	total	Lua/C (total)
	a, w	a, w	a, w	a, w
C	9, 37	7, 18	16, 50	-
Lua	15, 47	11, 59	26, 106	1.63, 2.12

On average, the time for receiving a response from the Lua component is 1.6 times slower than using the C responder. The worst case is 2.2 times slower. Of the 1 MiB memory pool, a maximum of 34% was used. It is worth noting that for the initial version of this benchmark, the response times were approximately eight times slower. Profiling revealed that this was caused by inefficient access to the timestamp message; switching to a faster foreign function interface yielded the presented results.

4.3 Cartesian Position Tracker

The following two experiments illustrate more practical use cases. The first experiment compares both a Lua and C++ implementation of a so-called "Cartesian position tracker", typical in robotics, and running at 1KHz, by measuring the duration of the controller update function. In contrast to the previous example the incremental garbage collection step is executed during the controller update and hence contributes to its worst case execution time.

The following listing shows the simplified code of the update function. Note that `diff` function is a call to the Kinematics and Dynamics Library (KDL) [11] C++ library, hence the controller is not implemented in pure Lua. This is perfectly acceptable, as the goal is not to replace compiled languages but to improve the simplicity and flexibility of using the primitives these offer.

```
pos_msr = rtt.Variable("KDL.Frame")
pos_dsr = rtt.Variable("KDL.Frame")
vel_out = rtt.Variable("KDL.Twist")
local vel, rot = vel_out.vel, vel_out.rot

function updateHook()
  if pos_msr:read(pos_msr) == 'NoData' or
     pos_dsr:read(pos_dsr) == 'NoData' then
    return
  end

  diff(pos_msr, pos_dsr, vel_out, 1)

  vel.X = vel.X * K[0]
  vel.Y = vel.Y * K[1]
  vel.Z = vel.Z * K[2]
  rot.X = rot.X * K[3]
  rot.Y = rot.Y * K[4]
  rot.Z = rot.Z * K[5]

  vel_out:write(vel_out)
  luagc.step()
end
```

Note that for Lua versions prior to 5.2 invoking the incremental garbage collector (`collectgarbage('step')`) restarts automatic collection, hence `collectgarbage('stop')` must be invoked immediately after the first statement. The custom `luagc.step` function executes both statements.

Results The following table summarizes the results of the worst case execution times in microseconds. The average execution time is approximately 14 times, the worst case duration 7 times slower than the C version. The worst case garbage collection time measured was 29us, of the 1MiB memory pool size a maximum of 34% was in use.

type	duration (avg, max)	Lua/C (total)
	a, w	a, w
C	5, 19	-
Lua	68, 128	13.6, 6.7

In the current implementation the majority of both execution time spent and amount of garbage generated results from the multiplication of the `K` gains with the output velocity. If performance needed to be optimized, moving this operation to C++ would yield the largest improvement.

4.4 Coordination Statechart

The second real-world example is a coordination Statechart that is implemented using the Reduced Finite State Machine (rFSM) domain specific language [12], a lightweight Statechart execution engine implemented in pure Lua. The goal is to coordinate the operation of grasping an object in an uncertain position. The grasping consists of two stages: approaching the object in velocity control mode and switching to force control for the actual grasp operation when contact is made. This statechart is shown in Figure 2.

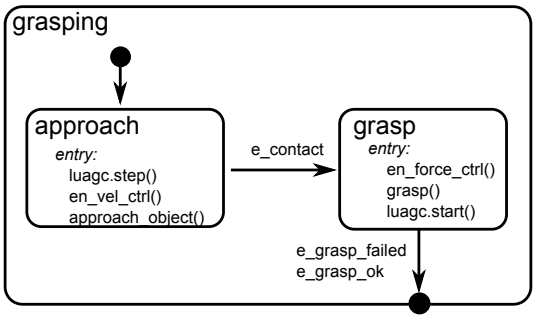

FIGURE 2: *Coordinating the grasping of an object.*

The real-time constraints of this example depend largely on the approach velocity: if the transition to the grasp state is taken too late, the object might have been knocked over. To avoid the overhead of garbage collection in this hot path, the collector is disabled when entering the `approach` state and enabled again in `grasp` after the respective controllers have been enabled.

Besides the actual grasping it is necessary to monitor the memory use to avoid running out of memory. With an appropriately sized memory pool and sufficient garbage collection steps, such a shortage should not occur. Nevertheless, to guarantee robust and safe behavior this condition must be taken into account and the robot put into a safe state. This is shown in Figure 3.

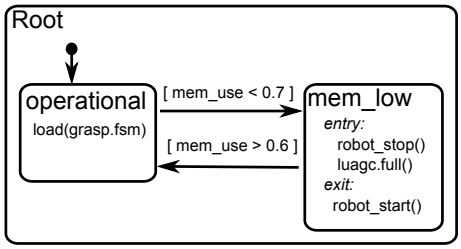

FIGURE 3: *Dealing with low memory.*

As the grasping task can only take place while enough memory is available, it is defined as a substate of `operational`. The structural priority rule of the Statechart model [13] then guarantees that the transition to `mem_low` has always higher priority than any transitions in the grasping state machine.

Identifying the required memory pool size has currently to be done by measuring empirically the maximum required memory of a state machine and adding a safety margin. To avoid this, it would be desirable to infer the expected memory use from the state machine description. Predicting the static memory used by the state machine graph is straightforward; also the run-time memory use of the rFSM core is predictable [4] as it depends on few factors such as the longest possible transition and the maximum number of events to be expected within a time step. However, predicting the memory use of the user supplied programs would require a more detailed analysis/simulation, which is currently out of the scope of this work; but in robotics, most user supplied programs are in C/C++ anyway.

Results The previously described grasping coordination Statecharts are tested by raising the events that effect the transitions from `grasping`, `approach` to `grasp`. The timing is measured from receiving the `e_contact` event until completing the entry of the `grasp` state. After this, the same sequence of events is repeated. The functions for enabling the controller

[4] It consists mainly of traversing and transforming the FSM graph.

are left empty, hence the pure overhead of the FSM execution is measured. Running the test repeatedly for five minutes indicates a worst-case transition duration between `approach` and `grasp` of 180us. The memory pool size was set to 1 MiB and the TLSF statistics report a maximum use of 58%. To test the handling of low memory conditions, in a second experiment the collector is not started in the `grasp` state. As a result no memory is recovered, eventually leading to a low memory condition and a transition to the `mem_low` state. For this test the worst case maximum memory use was as expected 70%.

This test does not take into account the latencies of transporting an event to the state machine. For example, when using the Orocos RTT event driven ports, the experiments from Section 4.2 can complement this one. Moreover it should be noted that so far no efforts have been put into minimizing rFSM transitions latencies; we expect some improvement by optimizing these in future work.

Robustness considerations As described, basic robustness of coordination state machines is achieved by monitoring of memory and current real-time latencies. However, the system level concern of coordination unfortunately combines the two characteristics of (i) requiring higher robustness than functional computations and (ii) being subject to frequent late modifications during system integration, the latter of course being susceptible to introduce new errors. The combination of scripting language and rFSM model can mitigate this effect in two ways. Firstly the scripting language inherently prevents fatal errors caused by memory corruption, thereby making it impossible to crash the application. Secondly, rFSM statecharts execute Lua user code in safe mode[5]. This way errors are caught and converted to events that again can be used to stop the robot in a safe way.

5 Conclusions

We have described how the Lua programming language can be used for hard real-time coordination and control by making use of an O(1) memory allocator, experimentally determining worst-case memory use and manually optimizing garbage collection to not interfere in critical paths. Several experiments are carried out to determine worst-case latencies.

As usual, benchmark results should be judged with caution and mainly serve to remind that appropriate validation should be repeated for each critical use. In particular when real-time allocation and collection is involved, run time validation of real-time constraints must be considered as an integral part the application.

The major shortcoming of the current approach is that worst-case memory use can be difficult to predict. To deal with this we currently allocate additional safety margins. As the overall memory usage of the Lua language is comparably small, such a measure will be acceptable for many systems, save the very resource constrained.

To conclude, we believe the results demonstrate the feasibility of our approach to use a scripting language for hard real-time control and coordination that permits to significantly improve robustness and safety of a system. The price of these improvements are (i) increased yet bounded worst-case latencies, (ii) computational overhead, as well as (iii) requiring additional precautions such as manual scheduling of garbage collection. In summary, we believe this constitutes a modern and practical approach to building hard real-time systems that shifts the focus from lowest possible latency to sufficient latency while maximizing reliability.

Future work will take place in two directions. On the high level we are investigating how to automatically generate executable domain specific languages from formal descriptions. Implementationwise we intend to investigate if and how the presented worst case timing behavior can be improved by using the luajit [14] implementation, a high performance just-in-time compiler for Lua.

Acknowledgments This research was funded by the European Community under grant agreements FP7-ICT-231940 (Best Practice in Robotics), and FP7-ICT-230902(*ROSETTA*), and by K.U.Leuven's Concerted Research Action *Global real-time optimal control of autonomous robots and mechatronic systems.* The authors also gratefully acknowledge the support by Willow Garage, in the context of the *PR2 Beta program.*

References

[1] R. Smits et al., "Constraint-based motion specification application using two robots.", `http://www.orocos.org/orocos/constraint-based-motion-specification-application-using-two-robots`, 2008.

[5] Using the Lua `pcall` function

[2] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua – an extensible extension language", Softw. Pract. Exper., vol. 26, no. 6, pp. 635652, 1996.

[3] M. Klotzbuecher, P. Soetens, and H. Bruyninckx. "OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages." In International Workshop on Dynamic languages for RObotic and Sensors, pages 284289, 2010.

[4] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, May 2006, http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf.

[5] "Real-time specification for Java (RTSJ)", version 1.0.2, http://www.rtsj.org/specjavadoc/book_index.html, 2006.

[6] Sun Microsystems. "Memory Management in the Java HotSpotTM Virtual Machine.", 2006, http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf

[7] D. Olofson, "The Extensible Embeddable Language", http://eel.olofson.net/, 2005.

[8] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant- time dynamic storage allocator for real-time systems", Real-Time Syst., vol. 40, no. 2, pp. 149179, 2008.

[9] P. Gerum. "Xenomai - implementing a rtos emulation framework on GNU/Linux", 2004.

[10] T. Gleixner. https://rt.wiki.kernel.org/index.php/Cyclictest

[11] R. Smits. "KDL: Kinematics and Dynamics Library", http://www.orocos.org/kdl/, 2001.

[12] M. Klotzbuecher. "rFSM Coordination Statecharts", https://github.com/kmarkus/rFSM, 2011.

[13] D. Harel and A. Naamad. "The STATEMATE semantics of statecharts.", ACM Trans. on Software Engineering Methodolody, 5(4):293333, 1996.

[14] M. Pall, "The LuaJIT Just-In-Time Compiler", 2011, http://luajit.org/

[15] Experiments source code. http://people.mech.kuleuven.be/~mklotzbucher/2011-09-19-rtlws2011/source.tar.bz2