Dual-WAM Control

Barrett Technology (DC) – April 1st, 2012

This document describes several software architectures that support controlling two WAM-systems from one PC using a 2-port PEAK PCI CAN card and the Xenomai real time co-kernel. Timing statistics for one of the implementation strategies are also presented.

Overview

To run a single WAM, the following steps must occur in order during each "execution cycle" (one iteration of the real time control loop):

1. Send a request for position updates (75 µs)
2. Receive a position reply from each axis (525 µs)
3. Perform control calculations
4. Send torque command messages (250 µs)

Out of every execution cycle, roughly 850 µs is spent sending or receiving CAN messages (for a 7-DOF WAM). A typical loop-rate is 500 Hz; this leaves a maximum of 1.15 ms in which to perform real time control calculations.

When controlling two WAMs from one PC, each of the four execution phases must be performed on each WAM. Since such a large portion of each execution cycle is spent sending and receiving CAN messages, it is desirable to parallelize CAN communications. This can be accomplished is several ways with varying degrees of concurrency, synchronization, and coordination overhead.

Approach #1: Single-threaded

Dual-WAM control can be accomplished by interleaving the execution phases for each WAM in a single real time control thread. During each execution cycle, the following steps are performed:

1. Send requests for position updates to both WAMs (75 µs)
2. Receive position replies from both WAMs (525 µs)
3. Perform coordinated control calculations
4. Send torque command messages to both WAMs (250 µs)

Pros:
With this approach, it is very easy to have a high degree of real time coordination between the two arms. There will also be very little relative-jitter in the timing of messages sent to each arm.

Cons:
Assuming that controlling two WAMs is more than twice as computationally intensive as controlling one WAM, this approach may be processor limited.

Approach #2: Simple multi-threaded

Another approach is to spin off two copies of a standard (single-WAM) real time control thread. Each

thread has the same real time priority. When one thread is blocked waiting for an interrupt from the CAN card, the other thread may be runnable. If both threads are blocked, then the Xenomai scheduler may give control to the “root thread” that represents the Linux operating system and all non-real time tasks.

Pros:
All execution phases (including control calculations) are performed in parallel.

Cons:
Instead of explicitly interleaving the execution phases for each WAM, the Xenomai scheduler (which may have unobservable state and may be sensitive to initial conditions) is responsible for correctly prioritizing and switching between the real time tasks. This is expected to increase jitter in control loop execution time and may allow the two execution cycles to drift relative to each other. Also, real time coordination may be more difficult and have more overhead as it will require synchronization.

Approach #3: Multi-threaded with synchronization

It is possible to modify Approach #2 by adding barriers at various points in each execution cycle. This approach takes some control back from the Xenomai scheduler by manually enforcing synchronization at the points in the execution cycle where the barriers are used.

Pros:
Relative to Approach #2, we would expect no drift, reduced jitter, and reduced sensitivity to scheduler state and initial conditions. As the threads are already synchronized, real time coordination may be somewhat easier.

Cons:
There is some amount of overhead incurred by synchronization.

Hardware versus software concurrency

Barrett has not yet experimented with hardware concurrency.

Xenomai does support assigning a processor affinity to a real time thread. By default, all real time threads share a single processor. Although hardware concurrency is certainly preferable, it is possible that low-level conflicts or Xenomai implementation limitations might prevent two real time threads that are running on different processors from properly sharing the single IRQ line from a PCI CAN card.

This is an important avenue to explore.

Implementation of Approach #2

We used the libbarrett controls library to implement Approach #2. (Given the existing code base, this approach was the easiest to implement.) We ran two 7-DOF WAMs (no FTS, no BarrettHands) from a standard Barrett WAM-PC. Below are timing statistics for:

- A baseline program running one WAM while performing minimal real time control calculations.

http://web.barrett.com/svn/libbarrett/trunk/examples/ex01_initialize_wam.cpp

- A program implementing Approach #2 and performing point-to-point moves with both WAMs simultaneously. Data is reported separately for each of the real time control threads. http://web.barrett.com/svn/libbarrett/trunk/sandbox/two_wams.cpp

	Mean execution cycle duration (µs)	**Stdev of execution cycle duration (µs)**	**Total number of execution cycles**	**Number of missed release points**	**Number of overruns**
Baseline	990	5.2	5692	0	0
Dual-WAM 1	1300	49	9792	1	1
Dual-WAM 2	860	97	11189	1	1

All three control loops were run with a target period of 2000 µs. An “overrun” occurs when an execution cycle takes more than 2000 µs to complete. The start of an execution cycle is signaled to Xenomai by an interrupt from the motherboard's Programmable Interrupt Controller (PIC). A “missed release point” occurs when a thread does not have to wait for a PIC interrupt because one has already occurred. If a thread is not scheduled properly, it may miss a release point even though it did not overrun.

As expected, the jitter is much greater in the dual-WAM case than it is for the baseline. Scheduler conflicts seem to explain why one of the (identical) dual-WAM threads consistently took longer than the other. We are unable to explain why one of the dual-WAM threads took, on average, *less* time to execute than the baseline. The difference between the total number of execution cycles might be explained by the fact that the control loops were started and stopped manually with keypresses.

The dual-WAM systems seemed to behave normally; there were no audible, visible, or tangible differences in the way the systems moved or operated. Though one overrun did occur in each of the dual-WAM control threads, it does not seem to be a systemic issue; perhaps the overrun resulted in a more favorable scheduler configuration that then ran stably from that point forward. Or the overrun could have been a transient condition related to starting or stopping the control thread.

Conclusion

The simple approach (from a software standpoint) seems to “just work” without any undesirable side-effects. If field-testing reveals critical performance drawbacks of this approach, there are several other avenues to explore that may increase control loop performance.