Designing and Maintaining a High-Performance Embedded Software Architecture

Erik Hons, National Instruments

Well designed software architectures enable better reuse of software components and more rapid development of system variants than monolithic designs. Software architectures are necessarily abstractions, and therefore have to be carefully designed to perform well. This case study follows the design of such an architecture for large distributed embedded control systems with lessons on delivering and maintaining system performance as architectures are ported to new platforms.

Introduction

A software architecture refers to the design of and interactions between software components in a system. An appropriate software architecture will reflect the organizational, technical, and market requirements of the systems it supports [1]. For large distributed embedded systems these requirements typically include being low-overhead, being portable to heterogeneous embedded hardware platforms, and supporting deterministic execution. Designing architectures that support a broad class of applications while retaining these attributes and others is a difficult challenge.

This case study is a research effort within National Instruments to design a software architecture for large distributed embedded control applications. Here we propose an architecture based on modelling data-transfer explicitly as channels, and then separating configuration of the channels from access to channel data. This partitions systems into channel implementations, system composition, and data processing components that can be reused independently within different applications while keeping overhead low.

Approach

National Instruments produces software development tools, rather than complete systems. Software development involves realizing a high-level model of an application on a platform [8]. A properly designed software architecture can provide this platform. Our objective was to produce such an architecture by considering the types of embedded applications to target, and what hardware would be required to support them. We considered how those applications would be specified, and derived a software architecture that allowed this while maximizing end user productivity and performance.

Target Applications

The advent of the “internet of things” is driving embedded applications to become larger, more distributed, and more complex. In addition, future applications require heterogeneous hardware for processing and transfer of data. This includes CPUs of varying types, core numbers, and cache hierarchies [6], FPGAs, GPUs, and bus technologies like PCIe, EtherNET, EtherCat, etc. To this, control applications add a requirement for real-time execution. All of this must be supported efficiently by software development teams with specialized skills sets. This creates a long-term development challenge with consistent requirements that warrants a dedicated software architecture.

To address this, the working group focused on large distributed embedded control applications. The applications were large in terms of the number of IO channels, data-transfer channels and data-processing operations to ensure the architecture could scale appropriately. The applications were distributed, meaning that different data-processing domains were separated by links with high data-transfer latency (high relative to data-processing times). The applications were also embedded in the sense that the processors and other hardware had to fit into small rugged enclosures that could be integrated into a piece of equipment without impeding its operation. The requirement can be taken to mean that cost, power, and size of the processors and storage must be minimized. Integrating control into these systems requires careful coordination of operations to avoid unpredictable interactions that affect determinism.

The result is a problem space with heterogeneous asynchronously operating data-processing and data-transfer elements that need to be tightly coordinated. The appropriate software architecture must make heterogeneous actors behave consistently and in a coordinated manner without incurring a significant overhead.

Observations

In distributed embedded applications, data-transfer has to be modelled explicitly because network communication channels are slow relative to processing and therefore need to operate asynchronously. The challenge in modelling data-transfer explicitly is defining an interface that handles all the possible implementations. Sending data over a TCP link looks very different from reading a memory mapped register.

Fortunately, applications tend to follow patterns based on the way they access and transfer data. Two key properties of those patterns are whether a processing element uses every value from a data source (e.g. FFT of a waveform) or only the latest (e.g. reporting a temperature) and whether the application is most sensitive to data latency or throughput. Based on this, three common "paradigms" were identified and labelled as follows:

- "Tags": read latest value, low latency
- "Streams": read every value, high-throughput
- "Messages": read every value, low latency

Within each paradigm, the access to the data was largely the same. That is the read and write operations were consistent in terms of semantics and parameterization. That suggests that code that only performs reads or writes can be reused with any implementation of the same paradigm.

The configuration of different data-transfer mechanisms tended to be very different even within each paradigm. However, these differences were localized to the part of applications that was least often reused: the system level configuration of data-transfer connections. This means the application specific code can be separated from the reusable data-transfer and data-access components.

Control applications read inputs (feedback signal), process them and write outputs (control signal). Acquisition of real-world inputs and generation of outputs are a form of data-transfer. The transfer is between the digital and real worlds, and so only one "end" of the transfer is visible to the software. Because of this, inputs and outputs can be modelled by the same paradigms above. A control application then naturally divides into data-transfer and data-processing code, components which can be reused in different applications. The missing piece is the system level composition code which is application specific and thus not reusable.

Data-transfer requires drivers and system level development, knowledge of hardware, and knowledge of bus protocols. Data-processing code tends to be application domain specific such as control algorithms. While the skill sets are transferrable, there is reason to want to allow people or teams to specialize. The componentization described above enables this.

Proposed Architecture

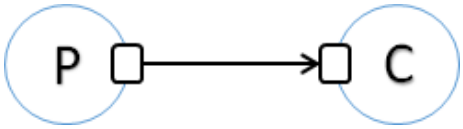

Figure 1: A producer and consumer connected by a channel with endpoints

A specific mechanism for data-transfer is referred to as a **channel**. To reflect the fact that the processing logic at each end of a data-transfer link needs local memory access, each end of a channel must have an **endpoint** which represents a local buffer or copy of the data. The endpoints provide one or more **accessors** which are implementations of the standardized paradigm data-access interfaces. The accessors determine how the endpoints are accessed and ensure the semantics of the particular paradigm are followed. The accessors are primarily read and write operations but include other operations as necessary (e.g. status).

System configuration code creates endpoints and then **links** them to form channels. The channels and endpoints have configuration options specific to the implementation being selected and thus implementation specific configuration interfaces. The system configuration code is thus specific to the particular channels begin used. It is appropriate for system configuration code to be non-portable in this way. To deliver maximum performance, implementation specific options have to be accessible. If desired, higher level abstractions can automate the generation of channel configuration.

System configuration code then requests accessors which it passes to the processing components. By restricting processing components to using the access APIs, it is possible to ensure reuse of the processing code without sacrificing configurability of the data-transfer mechanism, and thus with the performance of the system. As long as the accessor semantics are followed, the processing logic will be unaffected by changes to this configuration.

Processing elements communicate by writing to and reading from endpoints on a channel via the accessors and relying on asynchronous data-transfer channels to move the data. To provide tight coordination, a timing interface has to be added that configures when the data-transfer and data-processing components operate. Generating this configuration is the responsibility of the system configuration code.

Examples of Channels

The simplest possible channel is a local memory tag. In this case, the endpoints are a shared region of memory and the channel is virtual; the behavior of the memory is sufficient. To ensure updates are coherent, the accessor can rely on capabilities of the CPU such as interlocked memory instructions, or can implement a software coherency protocol such as **Fehler! Verweisquelle konnte nicht gefunden werden.**. This

choice is determined by machine capabilities and the size of the data, options that are specific to the particular system configuration and not the processing element accessing the tag. These would be selected automatically by the channel implementation or explicitly with a channel specific interface used only when composing the system and not in the processing element.

Two processing elements using tags over a packet network would rely on a channel using a network protocol. The endpoints would be data-buffers for packet oriented stacks. The channel would comprise the protocol stack and network hardware on and between each network end station. The selection of protocol would determine the specific implementation of the data-transfer, but the semantics would be the same. Updates of the tag would be propagated over the network from one endpoint to the other. The protocol could easily be changed without affecting the processing logic.

The ultimate source and destination of most data is I/O, which itself can be modelled as a data-transfer channel. I/O can be seen as transferring data between the real world and the digital domain with the I/O device itself acting as the channel. In the proposed architecture, the choice of I/O conversion device is a system composition decision that doesn't impact the processing elements as long as the semantics of the data accessors are respected.

Any I/O for which only the latest value matters can be modelled as a tag. Control systems that sample at fixed intervals fit this model well. Any I/O device that can make it's I/O available as a CPU accessible register or write directly to RAM can the modelled as one end of a tag channel.

Here the reuse possibilities begin to be apparent: The processing logic need not concern itself with whether it is talking directly to an I/O device. Instead, any chain of channels and processing loops could be the data source and determined when the system is composed, without modifying the processing elements themselves.

Examples of Systems

Many reuse scenarios are enabled by this architecture. Here, three are highlighted:

During development, it is preferable to validate individual units of an application before composing them into a system. This architecture allows a simulated channel to take the place of a real one, by making the semantics to be emulated explicit. Test sequences of data can be applied to a processing element and the output verified in a way that closely matches the eventual behavior of a channel, provided the channel implementations being used have been properly verified to follow the same semantics.

For an application to scale, it is often necessary for the output of a processing element to be propagated to additional destinations. In this architecture, channels would be free to support multiple receive endpoints which would take advantage of multi-cast protocols. If that isn't possible in a particular topology, a processing element can be added that takes one input and produces the required number of outputs. This processing element itself would be reusable.

Often, as an application changes and I/O needs evolve, it can be necessary to introduce filtering. Filtering can be added to an existing system by adding a processing step between an existing channel and the processing elements that reads from it. Again, this filtering operation itself becomes reusable

Conclusion and Future Work

The research has revealed a useful software architecture for embedded systems. To realize this effectively, hardware support for the channels must be specified and implemented in standard components. One technology that seems to fit well here is

time sensitive networking as defined by the IEEE 802.1 TSN working group. In this case, time needs to be incorporated into the channel model in a way that allows the channels and processing elements to remain independent. This adds to the most complex part of system design when using this architecture: the system composition and configuration code. Future work will include techniques to simplify and automate generating this code using system modelling languages.

References

[1] Eeles, Peter. (2006, February 15). What is a software architecture? http://www.ibm.com/developerworks/rational/library/feb06/eeles/

[2] Chandhoke S., Hayles T., Kodosky J., Guoqiang Wang, "A model based methodology of programming cyber-physical systems," in Proceedings of 7th International Conference on Wireless Communications and Mobile Computing (IWCMC), 2011, pp. 1654-1659.

[3] Bakic, A.M.; Mutka, Matt W., "A compiler-based approach to design and engineering of complex real-time systems," Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on , vol., no., pp.306,313, 1999.

[4] Chandhoke S., Sescila G., Hons E., "Controlling Data Transfer to build Predictable Real Time Systems," unpublished.

[5] Graham, S.; Baliga, G.; Kumar, P.R., "Abstractions, Architecture, Mechanisms, and a Middleware for Networked Control," Automatic Control, IEEE Transactions on , vol.54, no.7, pp.1490,1503, July 2009.

[6] Molka, D.; Hackenberg, D.; Schone, R.; Muller, M.S., "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on , vol., no., pp.261,270, 12-16 Sept. 2009.

[7] Alan Burns and David Griffin. Predictability as an emergent behaviour. In the 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems, 2011.

[8] Alberto Sangiovanni-Vincentelli, Defining platform-based design in EETimes, 5 Feb. 2002.

[9] Simpson, H.R., "Four-slot fully asynchronous communication mechanism," Computers and Digital Techniques, IEE Proceedings E , vol.137, no.1, pp.17,30, Jan 1990

Author

Erik Hons is a principal software architect within National Instruments Research and Development team. For more than ten years he has focused on embedded system architectures, FPGAs and software development. Hons holds a bachelor's degree in Mathematics and Computer Science from Waterloo University. In addition, he received his Master of Science degree in digital communications. Previously he has been a speaker at technical conferences, most recently at Embedded World.

Contact

Internet: www.ni.com
Email: erik.hons@ni.com