To appear in: PART '94, Proceedings of the Australasian Workshop on Parallel and Real-Time Systems,
7–8 July, 1994, Melbourne, Australia. Proceedings published by North-Holland.

The Host-Engine Software Architecture for Parallel Digital Signal Processing

H. John Reekie
Matthias Meyer[1]
School of Electrical Engineering
University of Technology, Sydney
email: {johnr, matthias}@ee.uts.edu.au

Abstract
This paper describes a software architecture for parallel real-time signal processing systems. The architecture consists of two layers: the engine layer, which performs real-time signal processing, and the host layer, responsible for control and configuration operations. Client programs access DSP resources through "objects," which encapsulate physical resources, signal processing algorithms, and support facilities such as scheduling, synchronisation, and inter-processor communications.

The architecture provides a flexible model for construction of real-time parallel DSP systems, and can map onto many different single- and multi-processor configurations. It also provides a basis for hardware-independent access to DSP resources, rapid system construction, re-configurability, and dynamic user control over real-time processing. Two pilot implementations of this architecture are currently under construction.

1 Introduction

Parallel digital signal processing (DSP) systems are increasingly being used to provide the massive computational power needed for many real-time signal processing applications—such as image processing, real-time control, sonar, and digital audio. These systems are typically built with off-the-shelf DSPs (digital signal processors) such as the ADSP-21020 [1], the TMS320C40 [13], and the DSP96002 [8], communicating via shared memory, fast special-purpose busses, or high-speed communications links. They thus belong to the MIMD (multiple-instruction multiple-data) class of parallel processors.

Software support for parallel DSP systems is not plentiful. We do not know of any parallel languages available for parallel DSP systems, although there are a growing number of support products in the form of distributed multi-tasking operating systems [9, 4, 15]. We do not believe, however, that the architecture of these systems, adapted from conventional operating systems, is entirely appropriate for typical hard real-time DSP applications and hardware architectures. We propose therefore a different architecture, which we call the host-engine architecture. We are presently working on two pilot implementations of this architecture: *DASS* (Digital Audio Sub-System) is an implementation for the TMS320C30 processor, while *SPOOK* (Signal Processing Object-Oriented Kernel) is our research project on a network of TMS320C40 processors, aimed at exploring the viability of the host-engine architecture for large DSP networks.

---

1 Visiting from the Technical University of Hamburg-Harburg.

1.1 Motivating examples

Figure 1 and Figure 2 are block diagrams of the type of system in which we are interested. Figure 1 is a simplified audio mixer, such as may be used in a recording studio. A complete studio mixer would have dozens of input and output channels, necessitating the use of many DSPs. Processing functions would include equalisation, dynamic range control, spatial simulation and enhancement, metering, and sample rate conversion. Most of these functions could be adjusted in real-time by the recording engineer, using a control console of some kind. Because different recording situations will require different numbers of channels and types of processing, it must be possible to re-configure the DSP software at any time. The mixer would operate internally at the 48 kHz professional audio sample rate, although input and output at other sample rates may be required.

The digital mixer example is largely a single-sample-rate system, with each audio processing function viewed as a self-contained block. In other applications, a finer-grain viewpoint is appropriate. Figure 2 shows a simple 2-band multi-rate complementary filter (MCF) bank used for audio processing. The input signal is split into a low band from 0 to $\pi/3$ and a high band from $\pi/3$ to π. More sub-bands are easily generated if $H_c(z)$ is itself an MCF. This filter structure features no aliasing errors, strictly linear phase, and perfect signal reconstruction if sub-band signals are not modified.

In our experience, "hard-coded" assembler programs for this type of system are difficult and time-consuming to write—furthermore, any change in the block diagram requires substantial re-structuring of the code. With our host-engine implementations, we hope to be able to construct systems like these examples from separate modules, but without sacrificing real-time performance. Resilience to change and dynamic control over parameters during real-time execution are benefits we hope to realise.

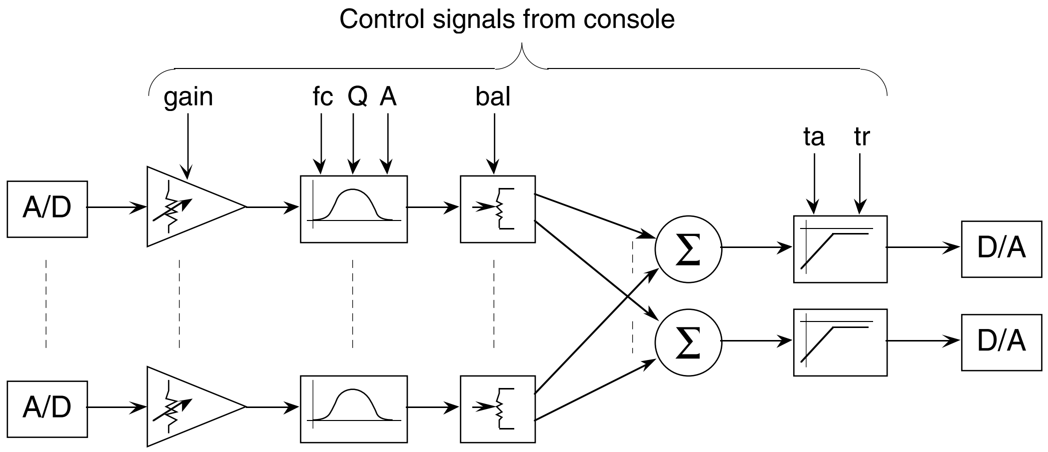

Figure 1. Digital mixer example

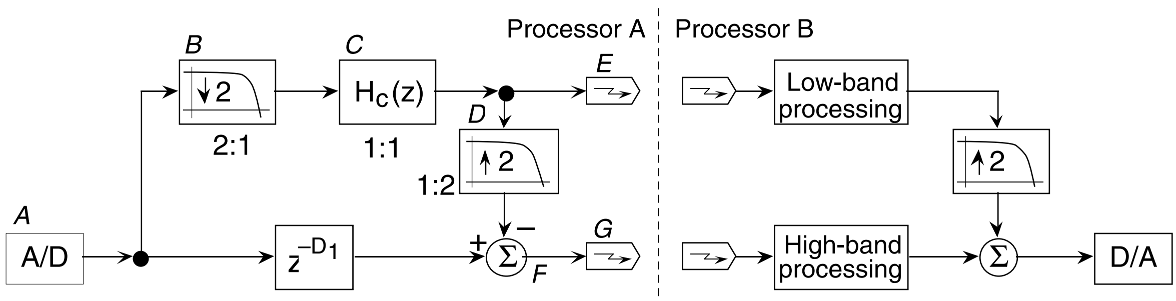

Figure 2. Multi-rate filter example

2 The host-engine architecture

The host-engine architecture models a real-time DSP system at two levels of abstraction: the processor level, and the object level.

2.1 The processor architecture

The processor level of the host-engine architecture is a collection of *processor pairs*. Each pair contains one *host* and one *engine*, as illustrated in Figure 3. Hosts and engines are *virtual*—they do not necessarily correspond directly to hardware processors. Each processor pair consists of a number of smaller components, as shown in Figure 4, each of which can reside on a different physical processor. This gives considerable flexibility in mapping hosts and engines to physical machines.

The engines perform all real-time signal processing: each generally (but not necessarily) corresponds to a DSP operating under hard real-time constraints. They communicate with each other through high-speed statically-routed communications channels with guaranteed speed and response times. Each engine consists of a core, which (conceptually) is the mechanism that performs signal processing computations, an executive, which manages the core and communicates with the host, and a communications interface to other engines.

The hosts communicate with the client computer or computers and control the engines. They do not have hard real-time constraints, and communicate over a dynamically-routed network. Each host consists of an executive, which provides the client with access to its engine's resources, a capabilities knowledge component, an interface to the communications network, and code generation and downloading facilities. The capabilities knowledge component allows the client to adapt to different processors: it contains information such as the type of processor, its clock speed, I/O resources, special hardware (such as, say, an FFT co-processor), and memory sizes and speeds.

The system is thus divided into two distinct layers, each with its own characteristics: the engine layer provides access to the raw computational power and throughput of the DSPs, while the host layer provides intelligent distributed control mechanisms. Although generally hosts will reside on general-purpose computers and engines on DSP processors—so that the virtual and physical processor capabilities are a good match—the architecture allows implementations to support other mappings to physical processors.

2.2 Host-engine objects

The second abstraction of the host-engine architecture is the object abstraction: all DSP resources are encapsulated in software entities called *objects*. For example, an object may encapsulate a particular type of filter, an analog I/O interface, or an inter-processor communications port. Objects reside on

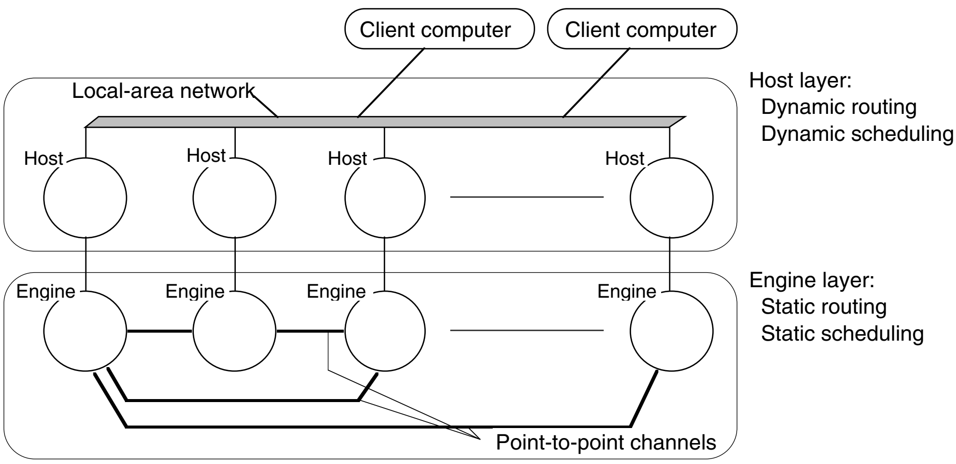

Figure 3. System model

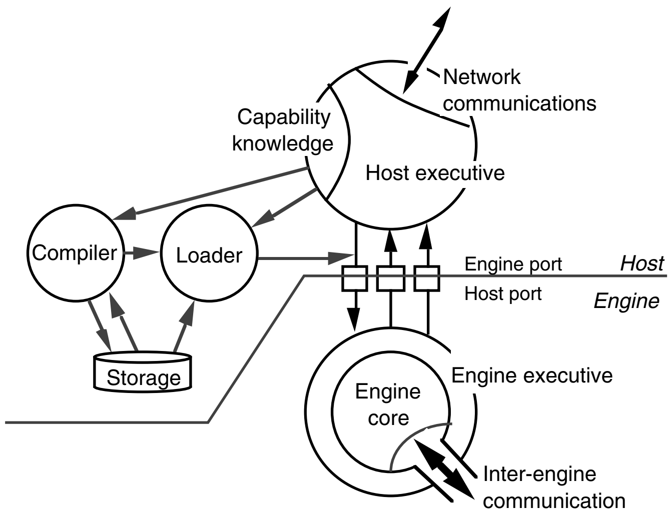

Figure 4. Host-Engine Structure

engines, and are created and deleted at run-time. (Although our use of the term "object" is similar to its use in object-oriented programming languages, we use it here with a very specific and more restricted meaning.)

Each object has a corresponding software entity on its engine's host, called its *proxy*. The proxy is responsible for controlling its object—allocating the resources it needs, communicating with other proxies, and responding to requests from the client program. Although an object and its proxy are physically distinct, their functionality is so inter-twined that it is often helpful to speak of them as just one object—our descriptions of object configurations in following sections will do this. For example, "creating an object" means that the client creates a proxy; the object on the engine is then automatically created by the proxy.

How are objects and proxies used to perform useful work? All we need are two types of attribute:

- Parameters

  A parameter is a value associated with an object, which can be read and modified by the client program. All control over an object, other than creating and deleting it, is exerted by modifying its parameters. Examples of parameters include:
  - Filter characteristics or coefficients
  - An "enable" flag for real-time input and output ports
  - An array of actions in a run-time schedule (see below)

- Actions

  An action is some processing associated with an object. Actions are generally executed by the engine in real time, according to a run-time schedule, but can also be executed in response to a request from the client. For example, all signal processing objects have a "Fire" action, which reads data from input buffers, processes it, and writes it to output buffers.

(By convention, object types and action names begin with an upper-case letter, while parameter names begin with a lower-case letter.)

Objects can also maintain some internal state. Executing an action on an object may cause it to update its state—since clients cannot see the state, however, we do not concern ourselves with it any further.

Figure 5 illustrates an equaliser object and its proxy. The engine object implements a bi-quadratic filter section, with a single action, *Fire*. Its proxy allows the client to set the buffer pointers and fire count. It also allows the client to set and read the equaliser characteristic: centre frequency, band width, and gain. When set, the proxy limits them to reasonable values, then calculates the filter coefficients and sends them to the engine object. The client can read back the current equaliser characteristic at any time.

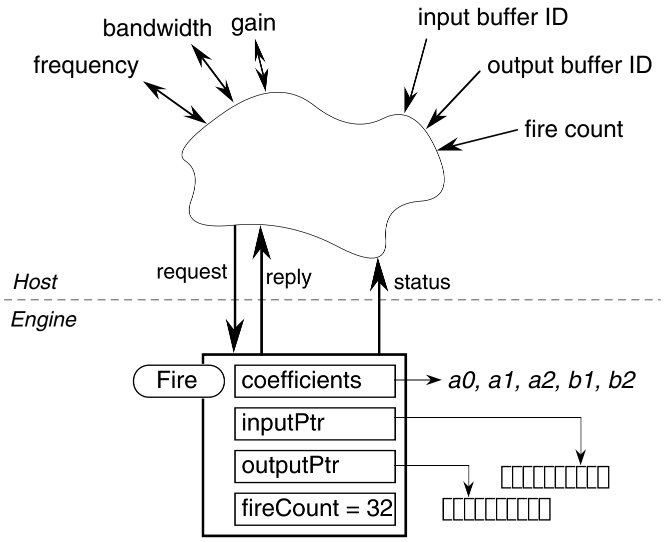

Figure 5. An object and its proxy

3 Buffering and scheduling

Objects are of two basic types: signal processing objects, which are those that are visible on block diagrams such as Figure 1 and Figure 2; and control objects, which perform functions such as scheduling and buffering. Because the block diagram represents a large-grain dataflow graph, signal processing objects are also referred to as “nodes” that are “fired” to cause them to process some portion of a signal. In our architecture, a node is fired by executing its “Fire” action. The order in which nodes are fired is called a *schedule* [6].

In this section, we examine the alternatives available for buffering data between objects and scheduling them for execution.

3.1 Buffering

Two choices are available to us for buffering data between objects: linear buffering and circular buffering. Circular buffers—that is, FIFO queues of limited length—are often used in real-time DSP, and are very efficiently supported by the instruction sets of most modern DSPs. They have several other advantages:

- Multi-rate processing is easily supported
- FIR filters can use them for their state vector with no additional cost
- Small delays can be realised with no additional cost

Linear buffers are simply arrays of elements of an appropriate size. In a real-time DSP system, they are read or written in entirety each time a node is fired. They have the following advantages:

- Buffer pointers do not need to be saved after firing a node
- Implementation in a high-level language may be more efficient

Of our pilot implementations, DASS uses linear buffering, while SPOOK uses circular buffering. This will enable us to compare both approaches. In SPOOK, we have had to work around some hardware limitations: the TMS320C40 has only a single register to specify circular buffer lengths, complicating buffer allocation; also, the DMA co-processor does not support circular buffering, so for inter-processor communications we must use linear buffering and insert extra copying nodes.

3.2 Scheduling

There are also two choices available for scheduling actions: dynamic scheduling, in which the order of action execution is determined at run-time, based on the availability of data; and static scheduling, in which the order of execution is pre-determined. Static scheduling is more efficient because nodes do not need to check that buffers contain enough input data or output space.

A node is said to be synchronous if it consumes and produces a fixed and known number of samples each time it is fired. Static scheduling can only be used on a synchronous dataflow graph—that is, one containing only synchronous nodes [6]. Both of our example systems are synchronous. For example, one possible schedule for Figure 2, if we wanted to process 64 input samples at a time, is:

64A, 32B, 32C, 32D, 64F, 64G, 64A, 32B, 32C, 64E, 32D, 64F, 64G

where, for example, *32B* indicates that node *B* is fired 32 times; since it is a 2:1 down-sampling node, it will consume 64 input samples and produce 32 output samples. Note that the delay node is implemented by initial data in the buffer between A and F.

In general, however, we wish to be able to support asynchronous systems—that is, systems containing nodes that do not consume or produce fixed numbers of samples on each firing. In DASS, dynamic scheduling is performed by using a simple “demand-driven” [2] interface between sub-systems running at different sample rates, similar to that used in Ptolemy’s dynamic dataflow domain [5]. In SPOOK, we are taking a more general approach: the system is divided into synchronous sub-graphs embedded in an asynchronous graph. The top-level scheduler on each processor is dynamic—when it encounters a synchronous sub-graph, it runs that sub-graph’s static schedule. The first version of the dynamic scheduler will use data-driven execution, since naive demand-driven execution does not work well across processor boundaries.

4 Using objects

Unlike other approaches to parallel DSP run-time systems—such as multi-tasking kernels—there are no scheduling, buffering, or communications mechanisms inherent in our architecture. Instead, these functions are provided by control objects, giving our architecture considerable flexibility and adaptability to different scheduling and communications schemes. In this section, we will describe how control objects are used to construct a working system.

4.1 A simple static schedule

Figure 6 shows some of the objects required for a digital mixer that uses circular buffering. Other than the signal processing objects, we require several other types of object: *Analog*, which represents an A/D/A converter, *Circle*, which manages a circular buffer, *Fence*, which waits until a buffer contains some specified amount of data, *Schedule*, which contains a static schedule, and *Engine*, which represents the engine itself.

In Figure 6, *port* executes in the background, adding a new sample to *inbuf* on each interrupt. In the foreground, the *WaitWhileLower* action of *fence* waits until *inbuf* contains some preset number of samples (typically half the size of the buffer), and then continues. *gain* and *eq* can then fire, followed by other nodes. The order in which these actions are fired is specified by the *schedule* parameter of the *sched* object.

To specify the top-level schedule of an engine, the client must set the *taskAction* and *taskObject* parameters of the *Engine* object. The main loop of the program running on the engine simply executes *taskAction* on *taskObject* in an infinite loop. In this examples, we have shown these parameters set to a *Scheduler* object and its *Execute* action—in other words, the top-level scheduler is static. If dynamic scheduling is required, these parameters would be set to a dynamic scheduler object.

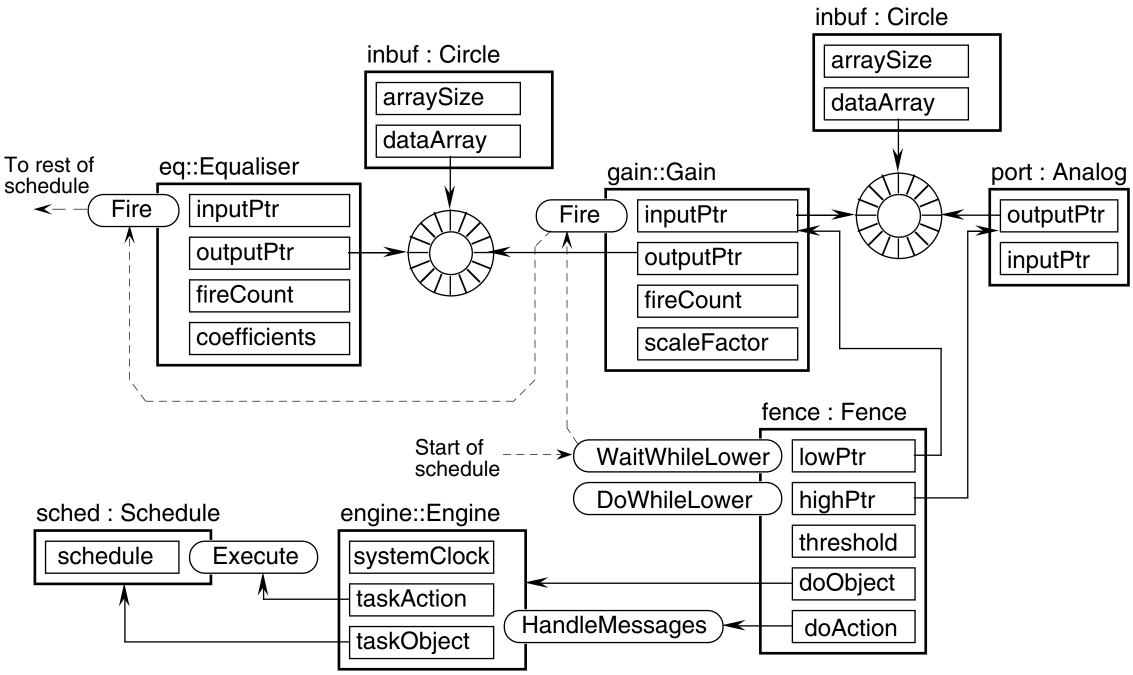

Figure 6. Example object configuration

4.2 Background processing

The *WaitWhileLower* action executed in Figure 6 waits idly until sufficient input data is available. This is time that could usefully be employed to perform background processing: reclaiming unused memory and forwarding messages to other processors are two background tasks that DASS will perform. To utilise this time, the schedule can contain the *DoWhileLower* action: instead of busy-waiting, this action repeatedly executes the action specified by the *doAction* parameter. In the configuration shown in Figure 6, the engine's *HandleMessage* action is repeatedly executed while waiting for data, so that messages from the host are processed in the background. Since this time is slack time anyway (assuming a deterministic real-time application), we have gained useful processing time for almost nothing!

5 Layers of functionality

The primary goal of a host-engine implementation is to provide efficient access to and control over real-time hardware. However, it provides a base upon which more sophisticated systems can be built.

Exactly what functionality is implemented on top of the basic host-engine functionality depends on the target application. DASS, for example, has a fairly straight-forward layer built onto it, which provides the client program with a virtual audio channel interface. There are a small number of audio channel configurations "hard-wired" into this layer.

In a more ambitious system intended for prototyping or for highly-configurable applications, however, some degree of automatic configuration of control objects and support for object-to-processor allocation, is required. Such a system may contain several layers as illustrated in Figure 7: a) a "block-diagram" layer, which contains only signal processing objects, some of which may be hierarchical; b) a "flat" layer, in which all hierarchical objects have been expanded fully; and c) a processor-graph layer, in which a sub-graph has been assigned to each processor and IPC nodes added. Underneath all of this sits the host-engine layer, which also contain buffering and scheduling objects. Layer (c) in particular is of interest for real-time system construction: it is in this layer that automatic

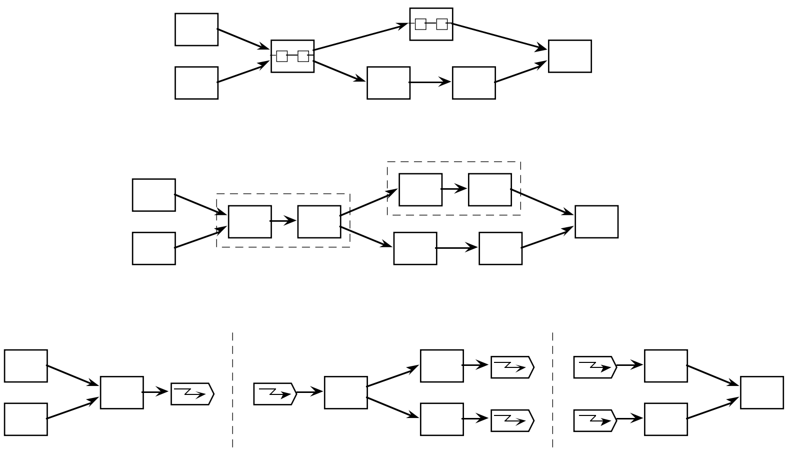

Figure 7. Illustrating the layered development system

schedule generation [6, 10] and automatic buffer allocation schemes [3] could be implemented to relieve the client programmer from the need to manually specify schedules and buffering.

A host-engine implementation could also be used as the run-time system for a parallel DSP language compiler, which generates host-engine objects from very-high-level source code. [12] describes one possible high-level source language.

6 The pilot implementations

As mentioned earlier, we are working on two pilot implementations of the host-engine architecture.

DASS (Digital Audio Sub-System) is an implementation for digital audio applications. The target hardware contains a single TMS320C30 processor for each audio channel, and the amount of inter-engine communication is fairly small. A single host CPU implements all (virtual) hosts and runs the user interface. DASS uses linear buffering and static scheduling. Both layers are implemented in C. DASS is running in prototype form.

SPOOK (Signal Processing Object-Oriented Kernel) is a research project aimed at exploring the viability of the host-engine architecture for large processor networks. Our current target hardware is a network of six TMS320C40s, based on TIM-40 motherboards and modules controlled by a single IBM-compatible PC. It uses circular buffering and a combination of dynamic and static scheduling. The engine layer is implemented in C, and the host layer in C++. We intend to make SPOOK available on the Internet when we have completed it to a useable stage, perhaps late in 1994.

SPOOK is founded on a reasonably general-purpose buffered message-passing layer. Static routing is used—partly for efficiency, but also to ensure that messages arrive in the same order as they were sent. The message-passing layer takes advantage of the TMS320C40's powerful on-chip DMA (direct memory access) co-processor for minimum CPU overhead. Message buffers are allocated dynamically by a very fast memory allocator—the allocator works with lists of power-of-two-sized memory blocks, and can allocate a block of memory in only a few instruction cycles. The host PC uses the same message-passing layer, but with all messages routed through the root processor.

7 Comparison with other systems

7.1 Comparison with block diagram systems

There are an increasing number of “block-diagram” systems becoming available, for DSP and discrete-event simulation [7], instrumentation, and image processing [11]. Although objects in the host-engine architecture appear at first to be at a similar conceptual level to “blocks” in a block-diagram system, the host-engine architecture and block-diagram systems have a very different focus.

The focus of block-diagram systems is generally on providing a conceptually-familiar interface to users, allowing rapid development and instrumentation of simulations. Research systems such as Ptolemy [7] aim also to study different models of computation. Some systems can generate real-time code, but with restrictions on target hardware or software. In contrast, the focus of the host-engine architecture is on support systems for real-time execution on diverse hardware architectures. Portability, reconfigurability, and support for dynamic user control are features of host-engine implementations lacking in current block-diagram systems. The two approaches are not at odds: a block diagram interface would be an ideal user interface for a host-engine system—it would appear as the highest layer of the complete architecture discussed in section 5.

7.2 Comparison with multi-tasking architectures

There are several products for the TMS320C40 that fall into the category of distributed multi-tasking operating systems [4, 9, 15], many of which have migrated from the Transputer market. Although multi-tasking kernels are well-understood, they are—in real-time DSP terms—quite expensive. Unlike the Transputer, typical DSPs have many registers and little hardware support for multi-tasking—their strengths lie elsewhere. For example, Virtuoso's micro-kernel running on a 40 MHz TMS320C40 takes 27 µs to perform four semaphore operations and two context switches [15]. Given that a typical digital audio application must process several audio samples every 20.8 µs (at 48 kHz sample rate), context-switching could amount to a substantial overhead.

Recognising the drawbacks of standard multi-tasking approaches to parallel DSP, Virtuoso also has a "nano-kernel." For the same operations as above, the nano-kernel takes 5.8 µs, although a single context switch takes less than a micro-second. The light-weight context necessary to achieve this requires that nano-kernel processes be coded in assembler: they are thus probably better viewed as a mechanism for implementing run-time support systems than as a general-purpose programmer-level abstraction.

But is multi-tasking essential for real-time parallel DSP? Probably not. A high proportion of DSP applications are deterministic: that is, they have a predictable execution order and timing. Multi-tasking systems tend to assume worst case operating conditions, and are not able to effectively take advantage of the determinism inherent in typical DSP applications. Although direct performance comparisons are difficult, we note that in a host-engine implementation, there is no context-switching between tasks, but merely a transfer of control from one node to another—in other words, a branch instruction or function call. The total overhead of a statically-scheduled, linear-buffering *Gain* node, for example, is about 0.5 µs. The comparison is not entirely fair, however, as circular buffering and particularly dynamic scheduling will require much larger overhead. In effect, full dynamic scheduling amounts to assuming worst-case operating conditions. We expect that any significant performance advantage of SPOOK over a nano-kernel system will come about through use of static scheduling. One of our research goals is to compare the performance and programming style of the two approaches once SPOOK is running.

8 Discussion

The host-engine architecture provides a model on which implementations of real-time user-controllable signal processing systems can be based. It is applicable to a wide range of physical architectures, is inherently reconfigurable and adaptable, and can support different forms of buffering, communication, and scheduling.

Because host-engine implementations are based on the notion of "objects," software re-useability is a real benefit of the architecture—once an object has been created, it can be used in other systems as well. Rapid construction of real-time systems is also feasible once a small library of general-purpose objects has been built, requiring that only application-specific signal processing objects be built. We will be testing how well these ideas work in practice by building a small library of generally-useful objects for I/O, data conversion, buffering, and filtering.

The host-engine architecture also supports target-independent programming and portability. If interfaces to objects are rigidly specified, then a client program could be portable to a range of hardware platforms, provided that all the required object types are supported by the host-engine implementation on each platform. If the interface between the engine objects and their proxies is also carefully specified, then the host-engine layer itself is resilient to changes in engine hardware. Heterogenous systems (containing different types of processors) could also be supported; a client program could transparently take advantage of specialised resources, such as, for example, an FFT processor.

The host-engine architecture is also amenable to simulation and development on single-processor computers. By locating an engine or engines on the same processor as their corresponding host or hosts, a host-engine system can be simulated on a conventional computer. The first running version of DASS, for example, was a single host and engine on one computer, which used files to simulate real-time I/O.

Acknowledgments

We gratefully acknowledge the assistance of Professor Warren Yates, our academic supervisor. Professor Edward Lee corrected a misconception of ours with regard to the need for dynamic scheduling. The TMS320C40 system was purchased with the assistance of an ARC Infrastructure B grant and with the support of the University Program of Traquair Data Systems, Ithaca, New York. John Reekie's research was supported by an Australian Postgraduate Research Award.

References

[1] *ADSP-21020 User's Manual*, Analog Devices, 1991.

[2] E.A. Ashcroft, *Eazyflow Architecture*, SRI Technical Report, CSL-147, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, April 1985, reprinted in [14].

[3] S.S. Bhattacharya and E.A. Lee, "Memory Management for Dataflow Programming of Multirate Signal Processing Algorithms," *IEEE Transactions on Signal Processing*, to appear.

[4] A.D. Culloch, "Porting the 3L Parallel C Environment to the Texas Instruments TMS320C40," in *Transputer Research and Applications 5,* A. Veronis and Y. Paker, IOS Press, 1992.

[5] S. Ha, "DDF Domain," in *The Ptolemy Almagest*. Dept of EECS, Univ. of California at Berkeley, 1990/91/92, chap. 3, Available by anonymous ftp from ptolemy.eecs.berkeley.edu.

[6] E.A. Lee and D.G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, 1235-1245, September 1987.

[7] E.A. Lee and D.G.Messerschmitt and twenty-two others, "An Overview of the Ptolemy Project," March 1994, Available by anonymous ftp from the Ptolemy distribution site, ptolemy.eecs.berkeley.edu.

[8] *DSP96002 IEEE Floating-Point Dual-Port Processor User's Manual*, Motorola Inc., 1989.

[9] *Helios*, Perihelion Software.

[10] J.L. Pino, T.M. Parks, and E.A. Lee, "Automatic Code Generation for Heterogenous Processors," in *ICASSP 94,* April 1994, pp. II-445–II-448.

[11] J. Rasure and M. Young, "Dataflow visual languages," *IEEE Potentials*, vol. 11, no. 2, pp. 30–33, April 1992.

[12] H.J. Reekie, *Towards Effective Programming for Parallel Digital Signal Processing*, Report 92.1, Key Centre for Advanced Computing Sciences, University of Technology, Sydney, May 1992.

[13] *TMS320C4x User's Guide*, Texas Instruments Inc., 1991, Literature number SPRU063.

[14] S.S. Thakkar (ed.), *Selected Reprints on Dataflow and Reduction Architectures*. IEEE Computer Society Press, 1987.

[15] E. Verhulst, "Meeting the Parallel DSP Challenge with the Real-Time Virtuoso Programming System," *DSP Applications*, pp. 41–56, January 1994.