

## Parallel Pipeline: A Pattern for DREs

Gabor Karsai<sup>1</sup>, Ted Bapty<sup>2</sup>  
 Institute for Software-Integrated Systems  
 Vanderbilt University  
 Nashville, TN 37235, USA

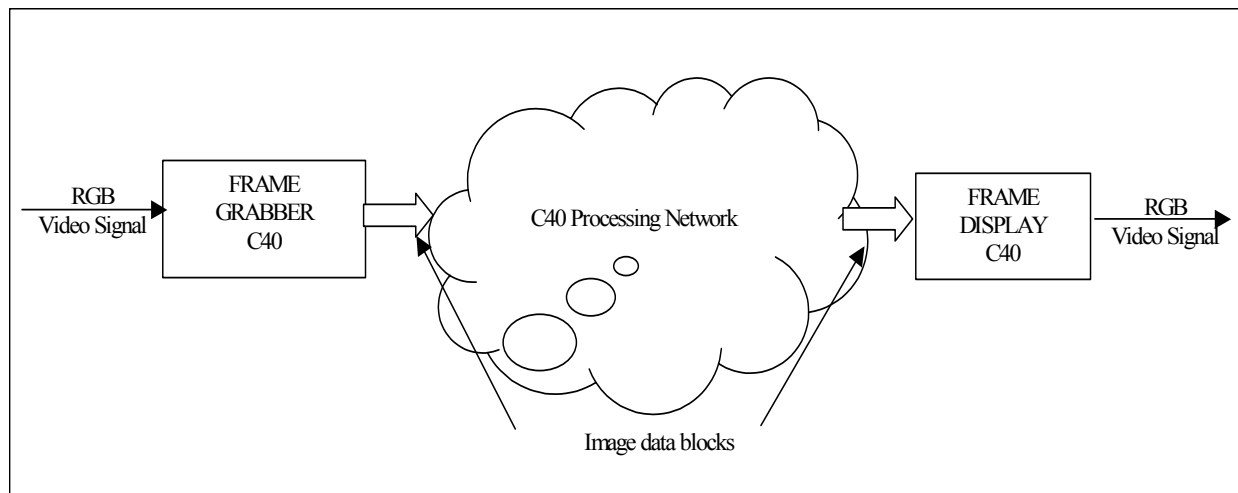
### Abstract

*High-performance DSP applications require remarkably high, sustained data rates. In these systems latency is often traded off for throughput. This paper describes a pattern, which allows implementing systems like these using off-the-shelf hardware components that have limited communication bandwidth and processing power.*

### Introduction

Digital Signal Processing (DSP) on real-time video signals has been around for a number of years. However, initially it was often implemented in special purpose hardware, with operations micro-programmed for a specific hardware architecture. This has changed with the arrival of dedicated DSP chips, which are specialized processors optimized for DSP-like operations. One example processor is Texas Instrument's C67 [1]. The C67 has an integer and floating point engine, with heavily pipelined operations, and direct support for Multiply-and-Add instructions. A 1024 point complex FFT takes about 100 microseconds to compute on this processor. The processor also has 6 communication ports, with about 16Mbytes/sec sustainable data rates. Communication ports can be connected to internal DMA engines that allow transferring data between the communication port and memory and between communication ports.

In a past project [2], we wanted to use this processor for processing real-time video data streams. A 512x512 pixel image was captured with a frame grabber device, DMA-ed into the memory of a dedicated C67, which then shipped it the data block out on the communication port. Subsequent C67 processors had to process this image, and execute various operations on it, for instance convolution with a filter, or 2D FFT. The resulting images had to be displayed on a video device, which also employed a dedicated C67 for taking a image data block, and placing them into a video frame buffer, from which the display circuitry generated the analog video signals. The overall architecture of the system is shown on Figure 1.



**Figure 1: Real-time image processing network**

<sup>1</sup> [gabor@vuse.vanderbilt.edu](mailto:gabor@vuse.vanderbilt.edu)

<sup>2</sup> [bapty@vuse.vanderbilt.edu](mailto:bapty@vuse.vanderbilt.edu)

This system is obviously an example for a distributed, real-time, embedded system: it runs on multiple processors, it has to comply with strict timing requirements (video frames – data blocks – couldn't be dropped, and data blocks had to be produced on time), and it was embedded in a larger physical environment (an on-line jet engine testing system). The problem was that a single C67 was not powerful enough to satisfy all the communication and processing requirements. Capturing the image, executing all the processing on the image, and then copying the result into an output buffer simply took too much time. (Images were sampled at 60Hz, i.e. every 16.67 ms.) While the throughput and reliability requirements were strict, the latency requirement was less stringent, and this allowed us to use an interesting pattern to solve the problem.

### Pattern [3]: Parallel Pipeline

**Intent:** To provide a high sustained processing rate on a multitude of processors equipped with fast communication devices.

**Motivation:** Processing of large data blocks arriving with a high sustained rate may not be feasible on a single processor. However, if the processors can overlap I/O operations with data processing, the parallel pipeline can provide a solution, at the expense of latency.

**Applicability:** The pattern is applicable in DRE-s where high-speed communication is available, and can be overlapped with processing.

**Structure:** The pattern is partly a hardware architecture, partly a software solution. The hardware is a pipeline of processors, each processor having an input communication port and an output communication port. The processor can (1) receive from the input port into memory, (2) process a data block from memory to memory, (3) output from memory to the output port, and (4) forward from the input port to the output port directly (bypassing the memory completely). Note that (1), (2), and (3) are sequential, but (4) can proceed in parallel with (2). The hardware architecture is show on Figure 2.

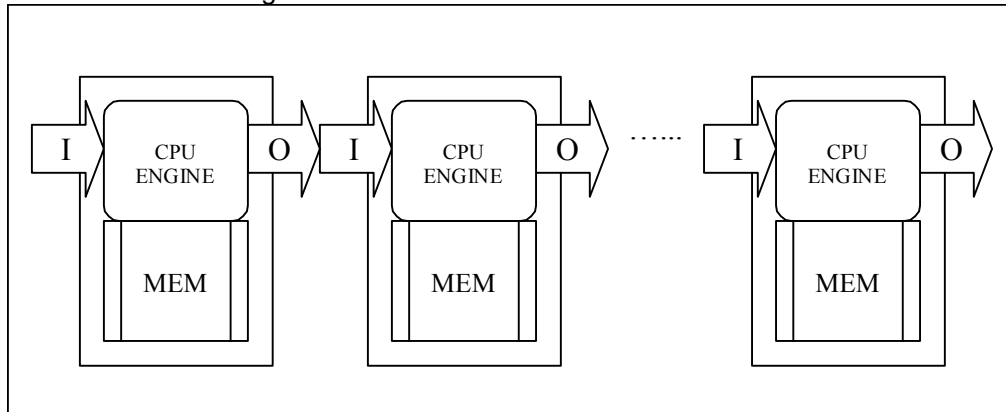
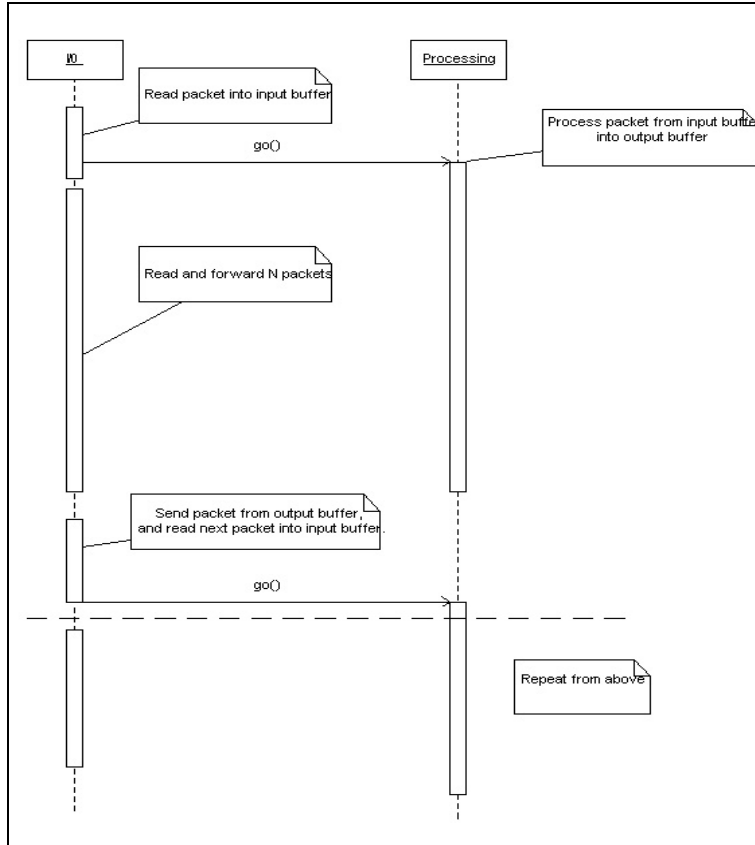


Figure 2: Hardware architecture: processing pipeline

**Participants:** Each CPU runs the same code, consisting of two threads: an **I/O** thread and a **Processing** thread, synchronized through semaphores. The I/O thread can further be decomposed into a separate input and output thread.

**Collaborations:** Figure 3 below shows the collaboration between the I/O and the processing. The following activities take place: First, the I/O thread reads one data packet from the input and places it into the an input buffer, and signals the processing thread to start its processing. The processing thread is started and it executes the necessary algorithms on the data packet and places the result into the output buffer. While the processing is executed, the I/O thread receives and forwards N data packets (without any processing), where N is the number of processors in the pipeline. Once the N data packets have been forwarded, the I/O thread sends the contents of the output buffer to the output. (Note that we are assuming here that the processing has finished by the time N packets have been forwarded!) Then, the I/O thread reads the next input packet into the input buffer, signals the processing thread to continue, etc. These activities are then repeated forever: forward N packet and process in parallel, then send output and get next input and signal the processing.



**Figure 3: Sequence chart for one CPU of the parallel pipeline**

It is easy to see that once the pipeline is filled up, it will process the data blocks in the correct sequence, and will sustain the required data rate. The crucial metric is  $N$ : the number of processors in the pipeline. It has to be determined based on the time required for processing one packet.

There are a number of timing constraints the system has to satisfy. Using the following notation:  $t_{IN}$  : time required to receive one packet,  $t_{OUT}$  : time required to send one packet,  $t_P$  : time required to process one packet,  $t_F$  : time between two packets (“frame-time”), the following inequalities must hold:

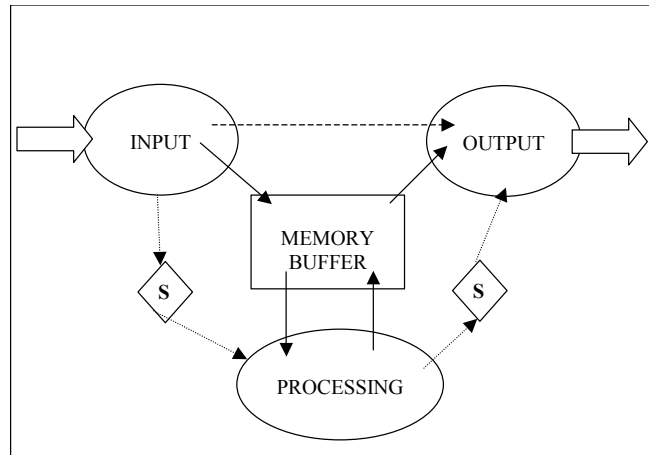
$$t_{IN} + t_{OUT} \leq t_F \quad (1)$$

$$N * t_F \geq t_P \quad (2)$$

If there are separate input and output threads, the requirements are:

$$t_{IN} \leq t_F, \text{ and } t_{OUT} \leq t_F \quad (3)$$

Note that the processing thread is not signaling the I/O thread about its finishing the processing. We made the assumption here that the processing finishes before the  $N$ -th packet is received on the input. Figure 4 shows the various activities on one processor: the input and output activities, together with the processing. Semaphores (or any other suitable technique) can be used to synchronize the processing to the I/O operations.



**Figure 4: Activities in one processor**

**Consequences:** The pattern will yield the required data rates (assuming that the I/O facility on the processor is fast enough), but at the expense of latency. Obviously, the longer the processing chain is, the longer it takes for all packets to propagate through. If this latency is not acceptable, another solution must be sought.

**Implementation:** The pattern described above can be implemented on processing nodes that (1) have high-speed communication interfaces, and (2) are capable of overlapping I/O forwarding operations with processing. Typically, hardware support is required to do this (in the form of DMA channels).

**Known uses:** Real-time video processors and other high-speed DSP applications [2][4].

## Summary, conclusions

The pattern described above provides a design solution for implementing high-data-rate applications on a network of processors. It trades off latency for throughput, and has been used in a number of applications. It is applicable whenever the basic resources: high-speed communication interfaces and overlapped I/O and processing, are available. If the communication interface is not fast enough, other hardware topologies could be used. For instance, a node can be dedicated as a “splitter” that divides the data block into smaller pieces and sends them on different communication links to other processors.

In general, the pattern shows that for DRE-s it is often necessary to think in terms of concurrency and hardware abstractions, which are inherently parallel. In the above pattern, the essential assumption was that forwarding and processing can happen in parallel because independent hardware resources are available for both. Hence, implicit parallelism provided by the hardware can be utilized in a pattern. Another important aspect is time: in order to ensure that the system operates with the required data rates, one must carefully measure timing parameters, and check if those satisfy the design assumptions. Current pattern techniques do not handle time well, timing is often an afterthought. We argue that time and timing are essential in a DRE, hence our patterns should take them into consideration.

The need for capturing parallelism and the time-domain properties indicates that patterns used in DRE-s should perhaps be described using these notions, and that the pattern descriptions should be extended this way.

## Acknowledgement

The USAF AEDC and Sverdrup Technologies have supported, in part, the activities described in this paper.

## References

- [1] Texas Instruments, [www.ti.com](http://www.ti.com)
- [2] Nichols J., Moore M.: An Adaptable, Cost Effective Image Processing System, The 10th JANNAF Non-destructive Evaluation Sub Committee, pp. 1-5, Salt Lake City, UT, March, 1998.
- [3] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
- [4] Moore M., Sztipanovits J., Karsai G., Nichols J.: A Model-Integrated Program Synthesis Environment for Parallel/Real-Time Image Processing, SPIE Conference on Parallel and Distributed Methods for Image Processing, pp 31-45, San Diego, CA, June, 1997.