How Programmable is Reconfigurable Hardware? : A Design Model for Reconfigurable Architectures

R. P. Self, M. Fleury, A. C. Downton

University of Essex, Dept. of Electronic Systems Engineering
Wivenhoe Park, Colchester, CO4 3SQ
rpself@essex.ac.uk

1 Introduction

With large-capacity FPGAs, such as the Xilinx Virtex family, complex systems can now be constructed from reconfigurable hardware, and sophisticated designs are more easily implemented through C-language hardware compilers, such as Handel-C [1]. However, improved language support is not enough, as effective system design needs structured methods and high-level design support, which a language by itself cannot provide. This paper address the need for additional application development support by proposing a system-level asynchronous model, based on CSP (Communicating Sequential Processes) [2], and implementing an FPGA runtime environment allowing user-level 'task' objects to communicate via Handel-C channels.

An asynchronous model has been chosen in order to provide timing commonality across hardware and software domains, and thereby overcome the timing disparity that prevents high-level design elements from being easily mapped between hardware and software implementations. To simplify interaction of software modules with FPGA hardware, an on-chip Task Manager has been designed to run hardware tasks as if they were processes residing in software. Employing a standard 'runtime' simplifies design and development because systems are engineered from existing library code and, furthermore, applications are more robust, as development is based on already proven structures, instead of relying on *ad hoc* design. Because the Task Manager forms an integral part of the application, the runtime is designed to occupy a small footprint, which minimises overhead when deploying prototypes to target platforms. This paper considers the potential impact of such Task Manager related overhead on applications by examining a pipelined implementation of the Karhunen-Loève Transform (KLT) algorithm [3].

The rest of this paper is organised as follows. Section 2 presents the asynchronous system design model, while Section 3 discusses the Task Manager and task operation. In Section 4, the implementation cost of conventional, and task variants of the KLT are compared. Finally, Section 5 presents conclusions.

2 System-Level Design

With increased complexity, component architectures [4] become essential to the effective design, development, and management of codesign systems. For flexibility reasons, loosely coupled architectures are desirable, but mainstream digital techniques and tool systems are based on synchronous timing. Although the synchronous timing model results in simple and reliable circuits, functionally separable units become interlocked by the timing dependencies necessary to achieve lock-step operation. To address this issue a two-layer model, shown in Figure 1, has been used that operates asynchronously at the system-level to realise the required flexibility, but functions synchronously at the lower level to retain the efficiency and deterministic behaviour associated with conventional hardware.

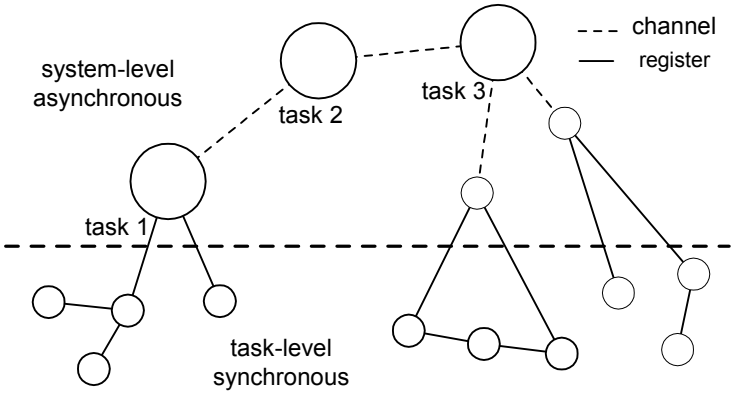

Figure 1: Task-Based Design

The two-layer model comprises:

System-Level View: At the system-level a design is deemed to consist of functionally independent tasks communicating asynchronously via channel objects.

Task-Level View: Tasks are self-contained functional units performing a well-defined operation. At the micro-architectural level, tasks can be decomposed into further sequential and parallel units. Such units can be constructed with synchronous logic in order to maximise performance and ensure predictable behaviour, or employ channels to form more loosely-coupled processing networks.

3 Task Management Runtime

3.1 Overview

Tasks form the means by which FPGAs expose processing resources to application software. Each task has a Task Id to uniquely identify it across hardware and software boundaries. Application software executes hardware tasks by calling a `RunTask` function with a parameter specifying the `'taskid'` of the requested task. Communication between application and task is achieved with a simple command-response messaging protocol, implemented as library code in software, and controlled by a Task Manager runtime on the FPGA.

A messaging protocol is employed in order to support higher-level functions than can readily be supported by simple 'bit-and-wire' connectivity, and improve the robustness of task execution. In particular, task operation, shown in Figure 2, uses a three-phase protocol so task status can be tracked in software through the arrival of 'task running' and 'task complete' messages sent from the FPGA. Implementing a high-level protocol allows system software to take abortive action if (say) a FPGA task become deadlocked, and was unable to respond to a start command. Furthermore, development errors are more easily localised, as acknowledgement messages are triggered from known application boundary points in task code.

Although messaging is a flexible means for the provision of an FPGA runtime, task-level programming can be simplified and implementation made more efficient if lightweight signalling is employed. For this reason, the FPGA runtime employs a message-based protocol to communicate between software host and FPGA, but converts messages to so-called 'events' when communicating with tasks. At the implementation level, events are mapped onto Handel-C channels, as they provide a simple and efficient way to achieving asynchronous behaviour in synchronous hardware.

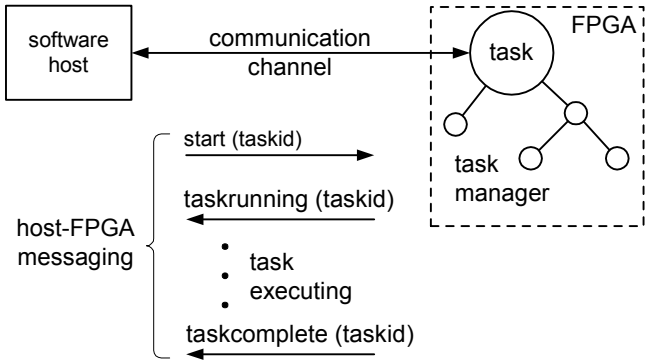

Figure 2: Host-FPGA Task Messaging

3.2 Task Manager

The Task Manager runtime architecture is shown in Figure 3, it comprises a set of fully parallel executing components, interconnected by channels. The system is partitioned into From-Host and ToHost processes (not shown) responsible for handling low-level control and status port connections to the host processor, a Task Manager for system control, and the task(s) containing application code.

The Task Manager is responsible for message handling, converting host messages to task events, and updating task state during task execution. To promote design modularity and allow for protocol extensibility the Task Manager devolves message-event translation to individual 'start', 'task running', and 'task complete' event handlers. Event handlers are also in charge of updating the task state in response to message and event flows within the runtime. Storing state in this way allows the Task Manager, for example, to reject activation requests destined for already running tasks.

Each task consists of Task Controller and DoTask units, created to separate task control from application code. This architectural division allows 'task running' events to be generated by the Task Controller, instead of the DoTask module, preventing application code from becoming intermixed with generic runtime control. The direction arrows in Figure 3 depict the flow of host-manager message and manager-task event flows.

To illustrate task operation a Handel-C implementation of a task-based Finite Impulse Response (FIR) filter [5] is shown in Figure 4. The code comprises a while loop implementing a process, channel variables for task-to-runtime connections, and FIR code (located between the channel input (?) and output (!) statements). The variables `chan_iStart` and `chan_oComplete` correspond to the incoming and outgoing channels, connecting task to Task Manager.

Normally a task is idle, blocked on channel `chan_iStart`, waiting for a start event. The task is activated by the arrival of an event, produced by the Task Manager following a start message from the software host. The task then accepts the event by completing a read on the channel. This triggers the waiting Task Controller to sending a 'task running' event to the Task Manager, and allows the task to begin executing application code. Finally, once the task has finished processing it sends a task completion event on `chan_oComplete` and returns to a waiting state, listening for new start events on its incoming channel

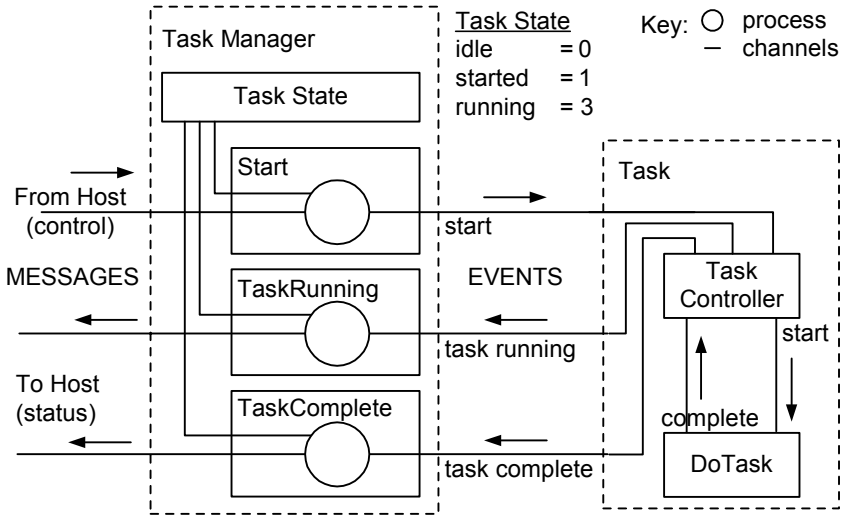

Figure 3: The Task Manager Runtime

4 Performance and Implementation

4.1 Reference Design

The design examined herein is a pipelined implementation of the KLT algorithm. The KLT transforms zero-mean data into eigen-space representation on the basis of the statistical properties of input data.

The implementation is shown in Figure 5. The design comprises three pipelined stages; stage one and three are implemented in hardware, while stage two is in software. The KLT has been partitioned in this manner because stages one and three operations are highly data parallel and benefit from hardware parallelisation. However, stage two processing involves only a small dataset (stage one results) and requires floating-point arithmetic so this is more efficiently performed in software.

```
// create a continuous process
while(1){

    // wait for host
    chan_iStart ? start;

    // pipelined FIR filter
    par{
        s[1] = c[0] * d;
        s[2] = s[1] + c[1] * d;

        ...
        res = s[7] + c[7] * d;
    }

    // confirm task has completed
    chan_oComplete ! 0;
}
```

Figure 4: Handel-C Implementation of a Task

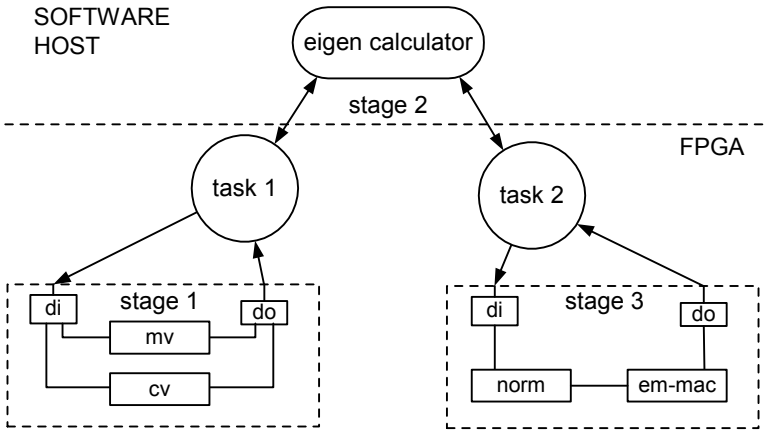

Figure 5: Task Implementation of a KLT Pipeline

The KLT pipeline work as follows. The first stage forms the mean vector (mv) and covariance matrix (cv) from input data, and sends these results to the eigen calculator in preparation for stage two. Next, stage two calculates the mean- and eigen-vectors from the results provided. Finally, in stage three, the data is normalised (norm) and mapped into eigen-space (em-mac) using the mean- and eigen-vector results produced in stage 2.

4.2 FPGA Implementation

Table 1 shows result data produced by synthesising design configurations consisting of one, two, three, and four parallel executing tasks (no application logic). Table 2 shows the result produced, firstly, when implementing the KLT conventionally (baseline) and, secondly, using the Task Manager runtime. The KLT design cases show the cost of implementing each KLT stage separately, and then in combination. The result data are taken from area and timing output generated by the Xilinx place-and-route targeting a Virtex XCV1000-4 FPGA.

In the case of 'minimal' tasks (Table 1) the area occupied comprises a fixed part relating to the Task Manager (60 slices), and a per-task overhead (20-25 slice) associated with each additional task. The worst-case clock speed is approximately 51 MHz. From the KLT results (Table 2), the difference in area between baseline and Task Manger designs for a single stage is 81 slices, and this rises to 94 for the full, two-stage case. The performance of the Task Manager based designs are marginally improved compared to baseline implements.

4.3 Discussion

The 'minimal' task results (Table 1) provide a constraint profile against which the impact of implementing an application using the Task Manger runtime can be judged. Specifically, the per-task slice usage figures can be used to estimate the area overhead relative to target application, while the per-task clock speed figure sets the clock speed budget for a given design. For example, a single task application would be limited to 71 MHz, while a three task one 54 MHz. However, in practice, neither of these constraints is likely to be of importance for realistic applications, as Task Manager worst-case area usage is almost insignificant for the dense FPGAs targeted by Task Manager based designs (less than 2% for a XCV1000-4). Furthermore, from experience, Handel-C design often run at less then 40 MHz (also see [6]), which is also well within the worst case 50 MHz performance budget shown above. For these reasons, it is believed that the Task Manager runtime can be successfully applied to a broad range of application scenarios.

No of Tasks	Slices	Speed (MHz)
1	80	71.958
2	98	65.308
3	115	51.438
4	153	54.702

Table 1: 'Minimal' Task Area and Performance

	Baseline		Task Manager		Difference	
Design	Slices	Speed (MHz)	Slices	Speed (MHz)	Slices	Speed
Stage 1	555	32.936	633	32.347	14% (78)	1.8% (0.589)
Stage 3	402	24.704	483	25.897	20% (81)	4.8% (1.193)
Stage 1 & Stage 3	970	23.449	1,064	23.493	9.7% (94)	0.18% (0.044)

Table 2: Comparison of KLT Area and Performance

5 Conclusions

This paper has proposed the use of an asynchronously timed, task-orientated model as the basis for simplifying the design and development of complex codesign system featuring FPGAs. An on-FPGA Task Manger runtime environment has been presented that allows hardware 'tasks' to be started from host processor software using function call semantics. The codesign methodology described, simplifies development by replacing *ad hoc* methods with a structured approach and improves application robustness, as systems are constructed from a standard runtime.

A comparative review of task and non-task based developments of the KLT algorithm has demonstrated that clock speed performance of Task Manager based applications are unaffected by the addition of the runtime system, and area overhead is minimal, and almost insignificant when targeting dense FPGA devices, such as Xilinx Virtex series.

References

[1] *Handel-C 3.0 Reference Manual.* Celoxica Ltd, 2001.

[2] C. A. R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, 21 (8), pages 666–677, 1978.

[3] M. Fleury and A. C. Downton. *Pipelined Processor Farms.* Wiley, 2001.

[4] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufmann, 2001.

[5] K. K. Parhi. *VLSI Digital Signal Processing Systems*. Wiley, 1999.

[6] M. Weinhardt and W. Luk, Task-Parallel Programming of Reconfigurable Systems, In *Field Programmable Logic and Applications (FPL 2001)*, LCNS 2147, pages 172–181, 2001.