A Design Methodology for Construction of Asynchronous Pipelines with Handel-C

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

Department of Electronic Systems Engineering, University of Essex,

Wivenhoe Park, Colchester, CO4 4SQ, UK

e-mail rpself@essex.ac.uk

Abstract

CSP channels are proposed as a means of developing high-level, asynchronous pipeline architectures over and above existing synchronous logic. Channel-based design allows hardware systems to be designed and constructed using top-down software engineering methods, which have not previously been available within hardware-software co-design. The intention is to enhance support for future large-scale co-designs. The design methodology and its performance implications are demonstrated through an exemplar, pipelined design of the Karhunen-Loève Transform (KLT) algorithm, implemented using the Handel-C silicon compiler applied to dense FPGAs.

1 Introduction

This paper describes a method for asynchronously-timed design based on the CSP (Communicating Sequential Processes) [1] model, using channels for communication. Once the design is fixed it is possible, if desired, to optimise to a synchronous implementation. The intention is to address the need for higher-level software engineering support, absent from some recent hardware-software co-design methodologies. In particular, a higher-level design structure could helpfully be added to Handel-C [2,3]. Handel-C is a language for silicon compilation already equipped with channels, but typically utilised without channels to produce a register-based, time-driven [4] design.

Handel-C is a language combining the familiarity of C software with the parallel programming model implemented in Occam [5]. This paper proposes an asynchronous infrastructure created from user-level 'task' objects that communicate via Handel-C native channels. We demonstrate the approach and its effects on system performance by a small but detailed study based on the KLT algorithm. In practice however, the pay-offs, in terms of clarity of design and productivity, are likely to be most evident in larger-scale designs. This insight is based on previous work on large-scale, continuous-flow pipelined applications [6,7], in which a top-down design approach was found to be an effective way to guide the design process. As Handel-C is an implementation of Hoare's CSP there are potential benefits in terms of formal design in a CSP manner [8].

A software-oriented hardware design [9] seems necessary because hardware developers now face complexity issues well understood and addressed through software engineering. Specifically, developers must meet a seemingly impossible dilemma: on the one hand they see time-to-market pressures continue to eat into the development cycle, and on the other hand they must nevertheless deliver mixed hardware-software designs.

Field Programmable Gate Array (FPGA) devices, such as Xilinx Virtex-II Pro [10], represent an increasingly credible alternative to custom silicon for small to medium production runs as they now possess multi-million gate capacities and support the embedded-processor capability demanded by large-scale ASIC (Application Specific Integrated Circuit) and System-on-Chip (SoC) designs. In addition, FPGAs minimise lead times and costs, as designs are constructed from pre-built silicon, programmable to project requirements. For large production runs (over 100 k devices) ASICs remain the way to meet cost targets, and have significant advantages in terms of power usage, switching speed, and silicon area [11].

Until recently, FPGA designs have been engineered with established digital design tools and hardware description languages (HDLs), such as VHDL [12] and Verilog. The advent of C-language silicon compilers now offers a means of improving developer productivity and simplifying system design by replacing hardware-level descriptions with algorithm-level system specifications coded in the 'C' programming language [13,14]. Furthermore, given the availability of high-capacity, programmable hardware (for example the Virtex-II can incorporate up to four PowerPC cores), there is now both a need and an opportunity to incorporate software methods into hardware design and development. Although C avoids the need to consider much of the low-level logic that is necessary in HDLs, and therefore simplifies many implementation aspects, additional high-level structure to support system-level design is necessary, but not incorporated in current design methods nor supported by C hardware compiler tools.

Defining an asynchronous development model for hardware design removes global timing dependencies that are fundamental to synchronously timed systems. The asynchronous model allows the construction of loosely-coupled systems that facilitate component-based design and incremental development practices, already well established in software engineering [15], but not applied to hardware design. Notice that in general a hardware system, comprising for example an FPGA and other peripheral devices, appears asynchronous by virtue of

independent synchronous clocks on each device.

This paper is organised as follows. Section 2 reviews C- and C++-based languages as a means for hardware design, while Section 3 describes the task system model and incremental development methodology. In Section 4, the role of channels in the creation of asynchronous pipelines is explained, and the mechanism by which asynchronous channels are implemented in Handel-C code is shown. Section 5 illustrates the use of incremental development, with a case study implementation of a Karhunen-Loève Transform [16] pipeline. Section 6 summarises results obtained, whereas Section 7 discusses their implications and draws conclusions.

2 C-Languages for System-Level Design

2.1 Background

Consensus in the hardware design community appears to favour the C and C++ languages as successors to VHDL and Verilog for the purpose of hardware specification and system-level design [17]. For example, the design of the Neon 256-bit accelerator [18] by a team from Compaq and Mitsubishi employed C++ templates for bit-width specification, `C' mathematical libraries, and a fast cycle-accurate simulator written in `C'. The reason for this choice is that SoC designs are expected to dominate the custom chip market and these designs frequently include both embedded processors and custom logic. In such designs, software represents the largest element in development cost, so selecting a software language for system design seems logical, given that most of the development is software related. C and C++ have emerged as the preferred candidate languages because they are implementation efficient, and can directly manipulate device hardware, both requirements of embedded systems, in which system resource and efficiency are prime considerations. Furthermore, C and C++ are well established in engineering, allowing existing code and development skills to be reused in the design of SoC solutions.

In the main, C-language tools have evolved to address the ASIC and SoC design requirement, as this is where the complexity issues of large-scale hardware and SoC designs are most evident and new tools most urgently needed. Many different C-language variants have been proposed, including SystemC [19], SpecC [20], and OCAPI-xl, [21]. Despite the fact that they differ with respect to language syntax and tool set they share a common objective—the need to provide an efficient implementation route to custom silicon. In practice, this can only be achieved by close cooperation with backend implementation tools, which allow place-and-route and layout to be controlled [22] and clock speed and area constraints to be precisely managed. Consequently, the tool chain is necessarily complex and a broad range of skills is required to deliver the efficiencies that SoC designs [23] require.

However, a simplified tool set is entirely possible for FPGAs, as these devices are constructed from pre-designed architectures and low-level layout is already fixed. Although performance optimised FPGA design still requires some degree of floor-planning, there are many cases in which automatic place-and-route provides adequate results. Simplified development has the advantage that it allows software engineers, who would otherwise be precluded from such development by the need for low-level hardware design experience, to create FPGA designs.

2.2 The Handel-C Language

Handel-C belongs to a group of C-language hardware development tools that are rooted in computer scientists' desire to compile programs directly into hardware, for example [24]. It has therefore evolved semantics that are more closely aligned to those of (parallel) software programs than, for example, tools such as SystemC and OCAPI-xl, which have evolved from the existing HDL tool culture. Handel-C syntax is primarily based on the C-language, but takes its support for parallel programming from Occam. It therefore has a set of parallel programming primitive constructs not commonly found in other C-language systems. It provides `seq` for sequential operation, `par` for parallel composition, and `prialt` alternation

(asynchronous multiplexing but with a predetermined service order). Furthermore, it provides channels as a means by which parallel processes can communicate. There is also a completely asynchronous version of Handel-C in development called BachC [25].

Handel-C has been designed so that there is a direct link between program code and hardware behaviour. Therefore, programmers are able to rationalise about design decisions, since intent expressed in source code is honoured in the hardware generated. Consistent with this policy is the adoption of a simple synchronous timing model—in Handel-C assignment and ready-to-run channel communication takes precisely one clock cycle, while all other statements and expressions are deemed to execute in zero time[1].

Figure 1 gives an example of sequential and parallel code. Figure 1a shows how sequential code is written—it is the same as standard C. Although the `seq` construct could be added in the same format as `par` is used in Figure 1b, this is syntactically redundant as sequential operation is the default case. As assignment takes one clock cycle, the code in Figure 1a takes three cycles to complete.

The code shown in Figure 1b is functionally equivalent to that in Figure 1a, but the lines of code contained by the `par` block run in parallel. In this case, two clock cycles are required for the program to finish. This is because, although the first line (`x=1`) takes one clock cycle, line two requires two, as the enclosing parentheses define new scope, and in this scope, sequential operation is the default. The execution of independent parallel statements within a

[1] The mapping from software statement to hardware logic in Handel-C is direct: assignment requires storage of a result in a register, implying a clock edge, whereas expressions are implemented purely in combinational logic. The penalty for this simple model is that the overall clock speed is defined by the worst-case expression propagation time, encouraging minimum expression complexity per statement as a programming discipline.

`par` block is terminated by a synchronisation barrier: the thread of execution only passes out of the `par` when all enclosed statements have completed.

Channel communication, as with Occam, is expressed through the input `?`, and output `!` operators, for example `myChan ? x`, inputs a value through channel `myChan` into variable x. However, communication can only start at clock edges. Communication internal to the FPGA is fully resolved, as unlike in the CSP model, all hardware behaviour is deterministic. Externally connected channels may, however, take an indeterminate number of clock cycles to become ready.

In general, the channel has been neglected as a communication/synchronisation mechanism [26] (for instance, channels are not a feature of Java), partly because the rendezvous mechanism can lead to deadlock in sequential languages, and partly because buffered communication has been easier to implement. However, channels do not require synchronisation by the programmer, and, because buffering is not employed, implementations can potentially result in fast context switching. CoWare [27], a well-known SoC co-design suite of tools, also bases its communication semantics on channels, potentially making possible exchange of designs at the semantic level.

3 Design Methodology and Development Model

3.1 Task-Based System Architecture

As design complexity grows, component architectures [28] become essential, to support multiple concurrent designers and rapid prototyping, to facilitate design exploration, and to allow independent Intellectual Property (IP) to be easily incorporated into designs. However, the synchronously timed Handel-C model locks functionally independent components together by the explicit timing relationships that are necessary to manage data flow between system nodes.

To break the timing bond a two-layer system model, shown in Figure 2, has been devised that operates asynchronously at the system-level in order to realise the flexibility required, while retaining the efficiency benefits and advantage of deterministic operation inherent in synchronous hardware.

The model comprises two design perspectives, or views:

1. *System-Level View:* At the system-level a design is deemed to consist of a set of functionally independent tasks, which communicate asynchronously via channel objects.

2. *Task-Level View:* Tasks are self-contained functional units performing a well-defined operation. Task themselves can be further decomposed into both sequential and parallel executing units, constructed using register-transfer logic in order to maximise system performance and ensure time-predictable behaviour. Tasks are continuously executing entities. They are activated by the arrival of data on incoming channel ports, perform the required processing, and are then suspended until result data can be sent to successor nodes.

3.2 Incremental Development

Incremental development allows a design to be functionally verified prior to partitioning into parallel modules. Individual modules can then be optimised to achieve the desired area/performance trade off. However, register-based design methods are not well suited to incremental development because the execution times of separate modules are closely interlinked due the global synchronous clocking employed in the FPGA. Consequently, module-level design changes that affect timing can propagate across the system requiring revision to affected modules in an *ad hoc* and often unpredictable manner.

A channel-based model is able to overcome these limitations because channel communication is asynchronous. Replacing registered module input and output ports with channels removes

the dependency on global clocking because data transfer and per module processing is then automatically controlled through channel handshaking. As the scope of timing changes is localised to within module boundaries, code can be incrementally performance tuned without risk of affecting overall system operation. Furthermore, development is simplified and more highly focused, as the development effort is not diluted by the need to consider the system-level effects of each change made. The benefits of this approach are illustrated by the case study reviewed in Section 5.

4 Asynchronous, Self-Synchronising Pipelines

4.1 Register- and Channel-Based Architectures

The differences between register- and channel-based approaches can be explained by considering the respective models shown in Figure 3. In register-based communication, as indicated in Figure 3a, each system module usually requires separate control and datapath to schedule module execution and process data, respectively. System design often involves the definition of an application-specific controller to start each module and manage local logic operations, synchronise data transfer, and organise tasks such as pipeline pre-fill and flush. Because all activity takes place with respect to a global (hardware) clock module, synchronisation must be designed-in so that result output is presented to successor modules at the correct moment. In many cases, this necessitates the provision of finite-state machine logic or additional 'buffer registers' to handle inter-module timing. The development effort involved can be considerable, especially for complex designs.

Channels perform many of these tasks automatically, as a consequence of the self-scheduling properties of channel handshaking. Channel handshaking is illustrated in Figure 3*b*. Channels are unidirectional; each channel consists of an input (in) and output (out) port, and arbitration (arb) control logic. In order for data transfer to take place, corresponding input and output ports must first be enabled, and ready signals generated. When the arbitration unit

detects the ready signals from both input and output ports it produces an acknowledgement (ack), which ports use to enable local datapath logic. For example, in Figure 3b the enable signal in Module B activates a register to initiate data transfer from module A. Should either port not be ready, the port enable signal is trapped in a flip-flop to ensure that the channel port 'waits' until the corresponding port becomes available.

Because channels have in-built synchronisation and scheduling functionality that would otherwise necessitate custom logic design, channel-based designs are typically simpler and more robust than their register-based counterparts, as the likelihood of design errors is minimised by removing the need for handcrafted controller logic.

4.2 Constructing Pipelines with Handel-C

The Handel-C language supports both synchronous and asynchronous timed systems. Synchronous systems are the default; asynchronous systems are possible because although assignment and channel updates are performed on clock edges, channel transfers are also conditional on data ready signals being asserted at the destination. Channels can therefore be used in conjunction with 'simple' assignment to implement the composite asynchronous and synchronous architectural model described in Section 3.

Figure 4 shows synchronous and asynchronous two-stage pipeline implementations of a Finite-Impulse Response filter. In Figure 4a each of the assignment statements contained in the `par` block corresponds to a pipeline stage. Stages are linked together by variable `pa` and parallel updated on each successive clock cycle. Stages are therefore pipelined because `pa` is updated in one clock cycle and read by the successive pipeline stage in the next.

Figure 4b shows the same pipeline constructed using the asynchronous task model. In this case, each pipeline stage corresponds to a continuously executing task, created by a while loop and with task boundaries defined by the presence of channel statements. Task t1 receives

input on channel `chan_in` into variable `d1` and writes the multiplication result to `tp1` in the next clock cycle. The result held in `tp1` is sent to task 2 in clock cycle three. Data transfer between tasks only takes place if task 2 is able to execute channel input in the same clock cycle as task t1 produces output. For communication to take place, both ends of the channel have to be enabled by their respective tasks. Should either task not be ready the other simply waits until it receives a ready signal.

The problem with the pipeline in Figure 4a is that pipelining is implied by the code rather than being defined by an explicit communication construct. Consequently, the fact that this code represents a pipeline is not self-evident and is easily missed by novice programmers. Furthermore, there is no guarantee that pipeline code is arranged as conveniently as shown; the second stage pipeline statement could be separated from the first by many lines of code, hiding the pipeline completely. In practice, while this approach is implementation efficient, and can prove workable when applied to small, localised micro-pipelines, the ease with which a design and its timing can be understood reduces as the scale of design increases. It is worth noticing that SpecC, together with the design language for the ACM (Adaptive Computing Machine) [11], does include an explicit pipeline construct.

The task-system model exemplified in Figure 4b has the advantage that it retains the system-level structure absent from the synchronous version and this is easily identified from code inspection. The use of HandelC channels to connect the pipeline stages maps directly onto CSP channels, preserving the system-level view of Figure 2. Tasks are readily identifiable, due to the presence of the enclosing ‘while’ block, and task boundaries and interfaces are clearly defined by channel input and output statements. In Figure 4a, tasks are not identifiable by any obvious boundaries, and thus the design style conforms to neither view present in Figure 2. A further benefit is that channel handshaking ensures that tasks are self-scheduling, guaranteeing that pipelines will work irrespective of the clock cycle execution time of individual stages. Using this type of architecture, asynchronous pipelines can be built in

piecemeal fashion and incrementally fine-tuned as necessary to meet throughput and latency constraints, as defined in [6,7].

The only limitation of this approach is that input channel, output channel and computation stages have to execute sequentially with respect to one another, and can, therefore, introduce a fixed single clock cycle overhead for each channel input and output. Consequently, task-based systems are best exploited in substantial designs, where the design-related benefits are greatest and the overhead of up to two clock cycles of channel latency per task is insignificant in relation to overall task execution time. If the circuit must actually perform at an optimal speed and size then Structural Descriptions with placement attributes, at the level of FPGA low-level tools, can be added via the Handel-C design path to Xilinx Foundation tools or VHDL. Even for optimal circuits, introducing channels has benefits in allowing an unknown pipeline stage timing at silicon compilation time to be allowed for, subsequently refining the timing through registers if need be.

5 Case Study: A Karhunen-Loève Transform Pipeline

5.1 System Architecture

This case study illustrates the use of the CSP channel model and the use of iterative techniques to create a pipelined implementation of the KLT algorithm. The KLT development discussed here is part of a larger hardware-software codesign project, reported elsewhere [29]. Overall, the KLT design comprises three distinct processing phases, which are mapped to CSP tasks: covariance formation, eigen-vector calculation, and eigen-space mapping. Tasks one and three are implemented in hardware to exploit data parallelism, while task two is implemented in software to take advantage of floating-point efficiency available from a general-purpose processor (in our case this was a PC, but in future it might be an embedded processor such as one of the PowerPC cores included in Virtex II Pro).

The case study is concerned with hardware design of the eigen-space mapping task, the architecture of which is shown in Figure 5. Source input and result output data pipeline stages manage the flow of data between external SRAM banks and the KLT pipeline processor. The pipeline processor works as follows: at the normalisation stage, incoming image vectors are normalised by subtracting an image mean vector stored in distributed RAM; the normalised data is then transformed into eigen-space by multiplication with eigen-vector data, also stored in distributed RAM memory. Finally, the summation stage accumulates the partial results generated by the multiplier into a result data element for storage in the result SRAM data bank. Counters, not shown in Figure 5, are used to generate the distributed RAM lookup table addresses and sequencer control logic.

5.2 Developing the KLT Pipeline

As already indicated, selecting the best design configuration can be a non-trivial task because there are often a number of candidate architectures, and these need to be implemented on-chip to determine the most suitable solution. The incremental design philosophy advocated in this paper introduces a step-by-step approach that helps eliminate unnecessary design iterations, thereby optimising the development process. Avoiding redundant design stages minimises the place-and-route iterations needed to establish accurate clock speed and gate usage figures; running the place-and-route tools is, in turn, a time consuming exercise in all except the most trivial of design scenarios, limiting the interactivity of the design process.

Our development methodology iterates from a sequential implementation through to a (possibly) fully parallel one. At the initial stage(s) the focus is on establishing correct functionality, which is more easily completed in sequential mode, while later steps are concerned with performance optimisation. At each step in development, a clock-speed and gate count profile is produced from which an informed judgement can be made regarding the next architectural iteration, if needed. Gate count statistics and maximum clock speed are

produced by output from the Handel-C compiler and Xilinx place-and-route (PAR) tools respectively.

In the specific case of the KLT design, a latency figure for each pipeline stage is also shown in order to determine which stage should be optimised in the next design iteration (the stage with the longest delay). In this simple example, pipeline latency is established by merely counting the number of (register) assignments in the related section of code manually; similar information for more complex algorithms can be determined using automated profiling tools such as Unix `gprof` or `Quantify`.

Step 1 C-Software Reference Design: The KLT was first implemented using ANSI-C running sequentially on a Pentium microprocessor so that the algorithm could be evaluated and baseline source and result datasets created. The ANSI-C version was transferred to SystemC in order to establish candidate pipeline partitioning points. SystemC models tasks by objects that are activated as threads. At compile time, a simulation kernel is linked in to produce an executable specification. From the results of the SystemC simulation, it was concluded that the pipeline should be constructed from Normalisation and Eigen-space Transformation stages, with separate stages for input and output. This corresponds to the top-level model shown in Figure 5. The model was further modified to include SystemC fixed-point modelling to establish optimal register widths from the input data. (As a fixed-point representation has a smaller dynamic range than a floating-point representation (for a given register width) then inaccuracy due to truncation can be introduced. Floating-point resolution varies according to the size of the input numbers, hence the purpose of fixed-point modelling is to find (for given test data) the register widths that introduce the least relative error compared with a floating-point representation.)

Step 2 Hardware Parallel Pipeline: The objective of this step is to verify pipeline operation and generate FPGA-based module timing results to identify which pipeline stage should be speeded up in the next iterative step. Creating a first-cut implementation of the KLT pipeline

involves defining the channel-based modules from the pipeline partitioning exercise concluded in step 1 and performing a cut-and-paste of code from the corresponding parts of the C-software design. In practice, this Handel-C porting exercise is essentially transparent as the Handel-C language supports most ANSI-C statements, although some adjustment is needed to include variable width specifications—in this case, determined by the fixed-point modelling completed earlier.

This initial design was implemented as four pipeline stages, each performing their specific operations sequentially. The stages concerned and stage latency figures relating to the processing time for a single data element appear in Table 1. From these figures it can be seen that the Eigen-Space Transformation stage has the largest latency, and therefore defines the overall pipeline throughput (the reciprocal of the worst-case stage latency). As it dominates pipeline performance, this stage is the subject for optimisation in the next design step.

Eigen-Space Transformation consists of two sub-operations, vector transformation and partial result accumulation. In the revised design, an Eigen Mapper task has been created that performs both of these operations. Performance is improved both by converting sequential code to parallel operation, and by implementing each sub-operation as a separate pipeline task. The Eigen Mapper task can be implemented using either register or channel-based assignment to interconnect pipeline stages. In order to provide a comparison of the merits of each approach, both alternatives have been implemented. These are examined in *Step3a* and *Step3b*, below.

Step 3a Eigen Mapper – Register-Based: The Eigen Mapper task, shown in Figure 6, comprises data input, eigen-space transformation, partial result summation, and result output operations implemented as separate input, micropipeline, and output logic blocks that execute sequentially with respect to each other. The micropipeline performs all task computation; it implements the transform and summation operations as parallel-pipelined operations thereby reducing what would otherwise require two clock cycles of processing time to a single clock

cycle. In the description that follows it should be noted that the data stream processed by the Eigen Mapper consists of six-element vectors presented in serial form.

Normalised data vector elements are received from the previous stage into register `din` and then processed in the transform stage of the micropipeline where they are multiplied by eigen-vector data and stored in register `pa`. The role of the summation second stage of the pipeline is to accumulate the transformed data present in `pa` into register `ac` until a complete six-element vector has been processed. The summation stage therefore has a sequencer to schedule the accumulator and time the despatch of a complete result into register `dout` for subsequent storage in external RAM banks.

As the input, micropipeline and output logic block processes execute sequentially and each takes one clock cycle to output a result once the pipeline has been filled, the Eigen Mapper has an effective throughput time of three clock cycles. Although this significantly improves over the eight clock cycles required in the initial design iteration the clock speed of the design has now dropped from the (sequential) baseline figure of 28.3 MHz to 20 MHz (Table 2).

In general, such performance problems can either be solved by optimising device layout through floorplanning or by the insertion of additional 'buffering' registers, which reduces device interconnect path lengths, minimising propagation delays and thus improving clock speed. Because Handel-C is aimed at rapid application development, it achieves simplicity-of-use and shortened development time by reducing the ability to optimise place-and-route through the supply of additional placement information. Consequently, in Handel-C performance tuning relies on inserting registers to break up delay paths.

In practice, this exercise requires an iterative approach whereby: registers are first added; a place-and-route performed; and the revised clock speed noted from place-and-route reports. In the case of this design, the source of the problem was found to be associated with the eigen-vector RAM block. The solution was to insert an additional register between the eigen-vector

and multiplier logic blocks. A further register was added to the data input path to the multiplier in order to equalise the throughput path lengths within the transform section of the pipeline. The revised configuration is shown in Figure 7.

Although this modification gives an improved clock speed of 30.58 MHz, this design is in many respects unsatisfactory. Inserting the registers increases the pipeline start-up latency by one clock cycle and requires a change to the sequencer logic to account for increased pipeline start-up delay. In particular, the need to manually retime the pipeline is both time consuming and error prone. The channel-based design (Step 3b) addresses these concerns and simplifies the design because such data-path retiming logic is unnecessary, as in-built channel handshaking deals with such issues automatically. Therefore, except for the purpose of illustrating register-based design in this paper, we do not advocate this approach with Handel-C, as it does not systematically produce improvements.

Step 3b Eigen Mapper – Channel-Based: This design is functionally equivalent to the previous one, but is implemented using channels as opposed to register communication. In this configuration, the two-section micropipeline is replaced with three parallel executing 'micro-tasks' connected via channels. The revised architecture is shown in Figure 8; it consists of eigen data, transform, and summation tasks. The transform and summation tasks are each partitioned into single cycle (channel-based) sequential input and output stages, and a single cycle processing stage, and therefore take three clock cycles to process a data element. The eigen-data task is constructed in similar fashion, but only needs two clock cycles to complete, as there is no data input stage.

The unequal execution times of the transform (three cycles) and eigen-data (two cycles) task operations would, in the case of a register-based design, necessitate the insertion of additional registers to synchronise input from registers `din` and `vec`. Channels, however, handle this requirement automatically, as is explained by the Handel-C source code below.

The code shown in Figure 9 employs while loops to implement two continuously running tasks, executing in parallel. The `par` block in the transform task synchronises the flow of input data and eigen vector elements into the multiplier. Because the thread of execution only passes out of a `par` block once all enclosed statements have completed, data input to the multiplier is correctly synchronised, irrespective of the relative execution time of the respective tasks.

This design improves over the register-based design both in terms of performance and design resilience. With regard to performance the clock-speed is increased from 30.58 MHz to 31.44 MHz and pipeline start-up latency is reduced by one clock cycle, as channel handshake auto-scheduling avoids the need for additional (redundant) datapath retiming registers. As can be seen from Table 2, gate count is increased from 14,921 to 15,099 gates, but in practice this difference is insignificant, given the million-gate plus capacity of present-day FPGA devices.

6 Results

Table 2 shows pipeline latency, gate-count and clock speed statistics produced for each iteration scenario discussed in Section 5.2 (steps 2-3b). In order to provide a full comparison between channel-based pipelines and their register-based equivalents, a fully synchronous register-only pipeline was created in which channel communication between the three main tasks (as described in Section 5.1) was also replaced by register assignment (step 0). Pipeline latency is determined through code inspection and clock information produced during simulation in Handel-C, while the gate count is reproduced from Handel-C compiler output and Xilinx place-and-route reports, targeting a Virtex XCV1000-4 FPGA. Xilinx place-and-route reports also provide the maximum clock-speed figure shown.

From the results it can be seen that channel-based designs (steps 2-3b) are marginally more costly in terms of gates used than equivalent register-based designs (steps 0, 3a), but this difference is relatively insignificant. However, the channel-based approach has the

disadvantage of introducing additional pipeline latency, which will affect overall pipeline throughput if it is in the slowest pipeline stage. Each channel input or output operation adds one clock cycle of latency, thereby increasing the total stage latency to $n + 2$ cycles; n for computation plus two for input and output. As the channel-associated latency for a pipeline stage is fixed, the single cycle computation employed in this case study represents a worst-case scenario. Therefore, in Table 2, though latency in the final channel-based design has increased from one to three clock cycles, if, as would be the case for large-scale designs, n were very large then the asymptotic effect of the per stage addition of just two extra cycles for channel communication would fade into insignificance.

As the computational complexity of each task grows, the impact of channel latency on pipeline performance becomes insignificant. Conversely, channel-based designs are more highly registered than synchronous designs, resulting in improved place-and-route performance and correspondingly higher clock speeds. Channel-based pipelines are, therefore, particularly advantageous for complex co-designs where the impact of channel-related latency on performance is small, and improved clock speed and design productivity are of greater importance. The reader should note that the KLT exemplar requires a further large-scale study to confirm the utility of the design approach in general terms.

7 Conclusions

Languages such as Handel-C and Bach-C principally simplify design by extending semantics to encompass a more software-orientated model. Specifically, sequential and parallel implementations can be specified with equal ease by the use of `seq` and `par` constructs respectively, allowing easy iterative construction of design variants. Although HDLs provide parallel assignment operators for implementing parallelism, sequential operation is much more difficult. In HDLs, a finite state machine must be designed to control the sequence of operations performed by the datapath. In contrast, with languages such as Handel-C, the flow

of control is simply implied by the relative order of statements in code. Incremental development is therefore impractical in HDL-based designs because of the need for complex rework at each design iteration, but is a realistic proposition with silicon compilers as the effort required to switch from sequential to parallel mode is minimised by the provision of specific language constructs.

This paper has demonstrated, through an exemplar implementation of the Karhunen-Loève Transform algorithm, the use of CSP channels to create hardware systems based on an asynchronously timed model, instead of the conventional synchronous one currently used in established design methods. Clock-driven designs result in tightly-coupled systems that undermine design flexibility and reuse strategies. Our alternative asynchronous channel-based approach overcomes these limitations and in so doing establishes a mechanism that is exploited in the development methodology described here to facilitate a top-down, incremental development approach more commonly associated with software systems.

In addition to these improvements to the overall design process, channel-based design provides automatic task scheduling and data flow, which would otherwise have to be explicitly engineered using conventional methods. Furthermore, designs are more robust because system communication is derived from standard structures, rather than based on *ad-hoc* logic. Although channel-based pipelines increase per-stage latency by two clock cycles, this cost is fixed and, for non-trivial designs, will be outweighed by the benefits afforded by simplified design and improved clock speed.

8 Acknowledgements

This work is being carried out with assistance from an EPSRC/DERA CASE studentship 9930329X.

9 References

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

[2] Page, I.: 'Constructing Hardware-Software from a Single Description', Journal of VLSI Signal Processing, 1996, **12** pp. 87–107

[3] Bowen, M.: 'Handel-C Language Reference Manual', Celoxica Inc., Didcot, UK, 2001

[4] Arato, P., Visgrady, T., and Jankovits, I.: 'High Level Synthesis of Pipelined Datapaths' (Wiley, Chichester, 2001)

[5] 'Occam 2 Reference Manual' (Inmos Ltd., Prentice Hall, New York, 1988)

[6] Downton, A. C., Tregidgo, R. W. S., and Çuhadar, A.: 'Top-down Structured Parallelisation of Embedded Image Processing Applications', IEE Proceedings I (Vision Image and Signal Processing), December 1994, **141** (6) pp. 431–437

[7] Fleury, M., and Downton, A. C.: 'Pipelined Processor Farms: Structured Design for Embedded Parallel Systems' (John Wiley & Sons, Inc. New York, March 2001, ISBN 0-471-38860-2) pp. 305.

[8] Nissanke, N.: `Realtime Systems' (Prentice Hall, London, 1997, ISBN 0-13-651274-7)

[9] Fleury, M., Self, R. P. and Downton, A. C.: 'Hardware Compilation for Software Engineers: An ATM Example', IEE Proc. Software, 2001, **148** (1) pp. 31–42

[10] 'Virtex-II Pro™ Platform FPGA Handbook', Xilinx Inc., http://www.xilinx.com/publications/ products/v2pro/handbook/index.htm, 2002

[11] Masters, P.: ACM Technology Guide, Chapter 1: Problems with Rigid Computing, 27 pages available from (Nov. 2002) http://www.silvertech.com.

[12] Chang, K. C.: 'Digital Systems Design with VHDL and Synthesis: An Integrated Approach' (IEEE Computer Society, 1999)

[13] Zhu, X. and Lin, B.: 'Hardware Compilation for FPGA-based Configurable Computing Machines,' Proceedings of DAC '99, June 1999, pp. 697–702

[14] Snider, G., Shackleford, B., and Carter, R. J.: 'Attacking the Semantic Gap Between Application Programming Languages and Configurable Hardware', Proceedings of 9th International Symposium on Field Programmable Gate Arrays, 2001, pp. 115–124

[15] Szyperski, C.: 'Component Software: Beyond Object-Oriented Programming', (Addison-Wesley, Harlow, UK, 1998)

[16] Fleury, M., Downton, A. C., and Clark, A. F.: 'Karhunen-Loève Transform: An Exercise in Simple Image-Processing Pipelines', Computers and Artificial Intelligence, 2000, **19** (1) pp. 19–36

[17] Verkest, D., Kunkel, J. and Schirrmeister, F.: 'System-Level Design Using C++', Proceedings of DATE '00, March 2000, pp. 74–80

[18] McCormack, J., McNamara, R., Gianos, C., Jouppi, N. P., Dutton, T., Zurawski, J., Seler, L., and Correll, K.: Impementing Neon: A 256-Bit Graphics Accelerator, IEEE Micro, **19** (2) pp. 58-69

[19] Liao, S. Y.:' Towards a New Standard for System Level Design', 8th International Workshop on Hardware/Software Codesign, CODES 2000, May 2000, San Diego, California, pp. 2–7

[20] Dömer, R.: 'The SpecC System-Level Design Language and Methodology, Parts1 & 2', Embedded Systems Conference, March 2002

[21] Vanmeerbeeck, G., Schaumont, P., Vernalde, S., Engels, M., and Bolsens, I.: 'Hardware/Software Partitioning of Embedded System in OCAPI-xl', Proceedings of CODES '01, April 2001, pp. 25–27

[22] Wakerly, J. F.: 'Digital Design: Principles and Practice' (Prentice Hall, London, 2000 3rd edn.)

[23] Rajsuman, R.: ‘System-on-a-Chip: Design and Test’ (Artech House, Boston, MA, 2000)

[24] Hilfinger, P.: ‘A High-Level Language and Silicon Compiler for Digital Signal Processing’, Proceedings of the IEEE Custom Integrated Circuit Conference, 1985, pp. 213–216

[25] Kambe, T., Yamada, A., Nishida, K., Okada, K., Ohnishi, M., Kay, A., Boca, P., Zammit, V., and Nomura, T.: ‘A C-based Synthesis System, Bach, and its Applications’, Proceedings of ASP-DAC, February 2001, pp. 151–155

[26] Hilderink, G., Broenink, J., Vervoort, W., and Bakkers, A.,: ‘Communicating Java Threads’, 20th WoTUG conference, 1997, pp. 48–76

[27] Van Rompaey, K., Verkest, D., Bolsens, I, and De Man, H.: ‘CoWare – A Design Environment for Heterogeneous Hardware/Software Systems’, Proceedings of EURO-DAC ’96, 1996, pp. 252–257

[28] Wolf, W.: ‘Computers as Components: Principles of Embedded Computing System Design’, (Morgan Kaufman, San Francisco, 2001)

[29] Self, R.P., Fleury, M and Downton A. C.: ‘A Run-Time Executive on a Platform FPGA (Submitted to IEEE Design and Test)

Tables:

Pipeline Stage	Stage Latency (clock cycles)
Source Data	2
Data Normalisation	3
Eigen-Space Transformation	8
Result Storage	2

Table 1 Pipeline Latency

Step	Development Scenario	Latency (cycles)	Gates		Speed (MHz)
			Handel-C	Xilinx	
0	Register-Only Pipeline	1	10,819	14,378	25.893
2	Channel Sequential	8	11,130	14,981	28.380
3a	Eigen Mapper: Register V1	3	11,041	14,721	20.056
3a	Eigen Mapper: Register V2	3	11,193	14,921	30.581
3b	Channel Eigen Mapper	3	11,322	15,099	31.444

Table 2 Gate Count and Clock Speed Performance

List of captions to illustrations:

```
int x, y;

x = 1;
y = 1;
x = x + 1;
```

a) Sequential

```
int x, y;

par{
    x = 1;
    {y = 1; x = x + 1; }
}
```

b) Parallel

Figure 1 Sequential and Parallel Composition

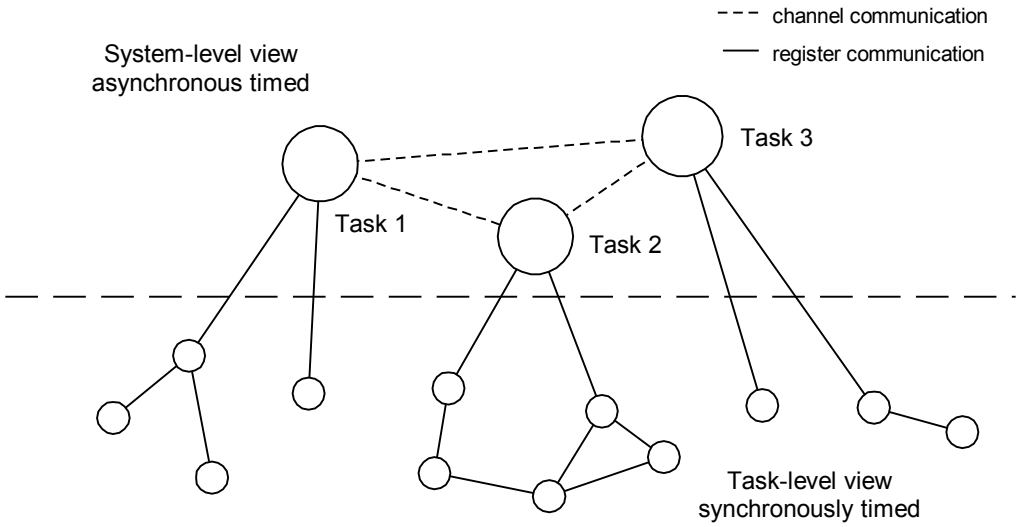

Figure 2 System Development Model

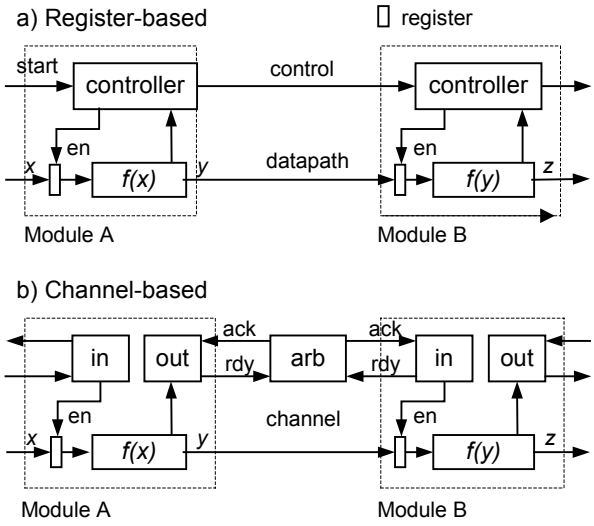

Figure 3 Register- and Channel-based Communication

```
int di, do, pa;

while(1){
    par{
        // pipeline stage 1
        pa = coef[0] * di;

        // pipeline stage 2
        do = coef[1] * pa;
    }
}
```

a) Synchronous Pipeline

```
Chan unsigned chan_in, chan_out;
chan unsigned chan_t2;
unsigned d1, d2, tp1, tp2;

par{
    // task t1, pipeline stage 1
    while(1){
        chan_in ? d1;
        tp1 = coef[0] * d1;
        chan_t2 ! tp1;
    }

    // task t2, pipeline stage 2
    while(1){
        chan_ t2 ? d2;
        tp2 = coef[1] * d2;
        chan_out ! tp2
    }
}
```

b) Asynchronous Pipeline

Figure 4 Synchronous and Asynchronous Pipelines

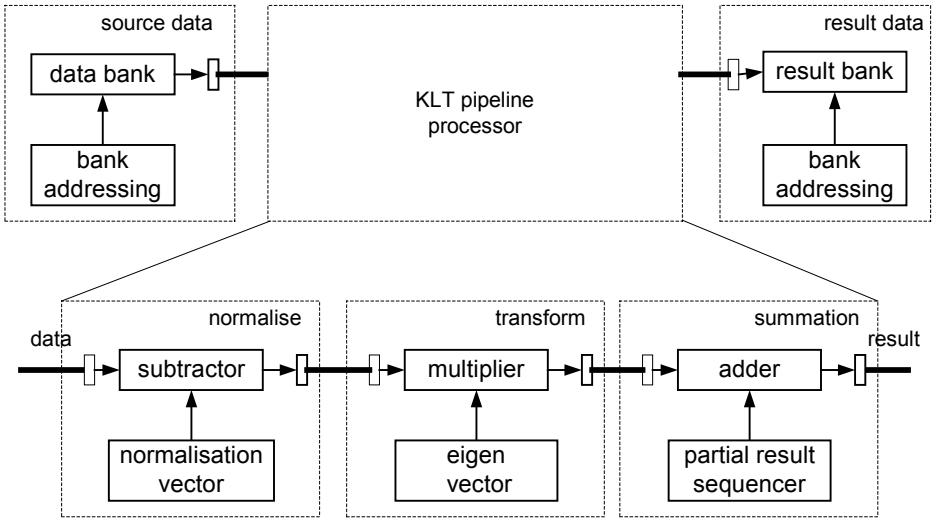

Figure 5 Normalisation and Eigen-Space Transformation Pipeline

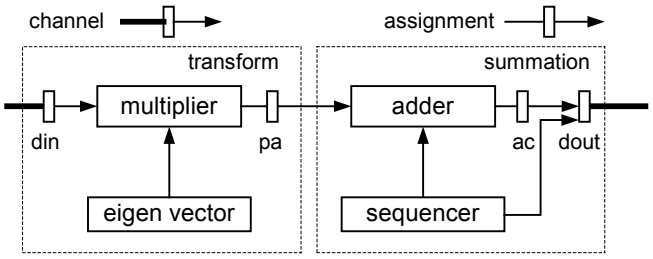

Figure 6 Register-Based Eigen Mapper Task

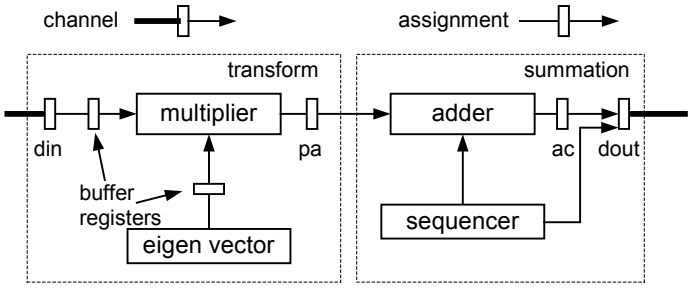

Figure 7 Revised Eigen Mapper Task

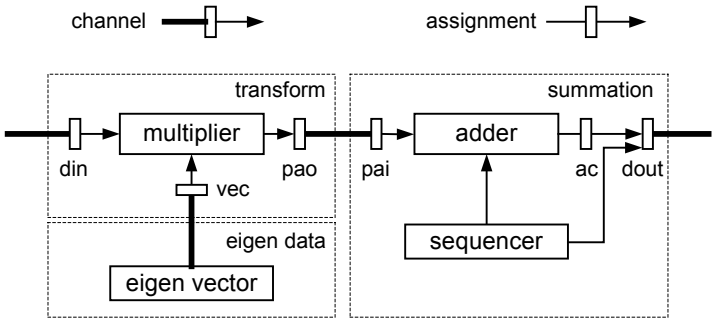

Figure 8 Channel-Based Eigen Mapper Task

```
par{
    // eigen data task
    while(TRUE){
        vchan ! eigenram[addr];
        addr = addr == 35 ? 0 : addr + 1;
    }

    // transform task
    while(TRUE){

        // read data into multiplier input registers
        par{
            vchan ? vec;
            dchan ? din;
        }

        // eigen-space transformation
        pao = din * vec;

        // eigen-space partial result output to summation stage
        ochan ! pao;
    }

    // summation task code
    ...
}
```

Figure 9 Eigen Mapper Code