Constructing Hardware-Software Systems from a Single Description

Ian Page
Programming Research Group,
Oxford University Computing Laboratory,
Parks Road, Oxford, England OX1 3QD

Abstract

The study of computing is split at an early stage between the separate branches that deal with hardware and software; there is also a corresponding split in later professional specialisation. This paper explores the essential unity of the two branches and attempts to point to a common framework within which hardware-software codesigns can be expressed as a single executable specification, reasoned about, and transformed into implementations. We also describe a hardware/software co-design environment which has been built, and we show how designs can be realised within this environment. A rapid development cycle is achieved by using FPGAs to host the hardware components of the system. The architecture of a hardware platform for supporting experimental hardware/software co-designs is presented. A particular example of a real-time video processing application built using this design environment is also described.

1 Introduction.

Our approach to unifying the traditionally separate disciplines of hardware and software design is to find a single framework that has the power to describe accurately the *behaviour* of systems, whatever the nature of their implementations may be. These must range from the largely sequential behaviours of microprocessor implementations of programs expressed in standard programming languages, to the behaviour of highly parallel arrangements of gates and flip-flops that constitute digital hardware implementations. We suggest that there is such a descriptive framework in the language and algebra of CSP [1] and its embodiment as the occam programming language [2].

We believe that a program can have the necessary expressive power to enable it to serve as an executable specification of a system, whether it be entirely in hardware, entirely in software, or in a combination of the two. Moreover, we believe that programs are the only reasonable basis for a single framework which spans both hardware and software implementations.

We use program transformation to map a user, or source, program into a variety of target programs. These programs can then be interpreted as hardware, as software, or as machine code for an application-specific processor together with a hardware description of the processor. Using such transformations, a single application program can be made to exploit the inherent, and widely differing, cost-performance characteristics that each of these forms has. Thus, hardware support can be given to the parts of the application that need it most.

For our hardware implementations, we typically use Field-Programmable Gate Arrays (FPGAs) in conjunction with microprocessors. More specifically, we use Dynamically Programmable Gate Arrays (DPGAs) which allow us to create new hardware systems rapidly by software means alone. Using hardware compilation together with DPGAs enables us to construct realistic, working hardware-software systems in hours, or even minutes. We use the term *systems compilation* to denote this process of implementing programs as a mixture of hardware and software. Systems compilation involves the re-structuring and partitioning of the original program into separate components, followed by either hardware compilation or traditional software compilation of those components into actual implementations.

The long-term goal of our research is to build a working design environment wherein system designs can be formulated, simulated, reasoned about, transformed, and mapped into a wide spectrum of implementations, and where all of these tasks can be accomplished by working with programs alone, avoiding traditional hardware level descriptions completely.

We wish however to allow and support the injection of domain-specific knowledge into the optimisation of designs as they move from specifications to implementations. We envisage, but currently cannot support, electronic engineers and other specialists interacting with particularly important designs before, during and after compilation. We would like to see design environments which support such optimisations whilst maintaining correctness of the overall design and allowing re-use of those optimsations in later designs. This is a large-scale challenge which we have hardly started to address.

There is naturally a great deal of related work, much of which has proven to be very useful in helping us set our own goals. We would refer the interested reader to [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. This is far from an exhaustive list of relevant related work, but it does give give decent coverage of the topics which we see as closely allied to our endeavours. In parallel with the work on compilation of imperative programs described in this paper, there is also significant activity in investigating declarative approaches to hardware compilation [14, 15].

2 A Language for Co-design.

We have as a major goal the implementation of hardware/software systems from a single specification, presented in the form of an executable program. We believe that this starting point places some tight restrictions on any language which might be used for this task.

Until we know much more about how to compile truly high-level programs into both hardware and software, it is necessary to start with a language which is simple and yet expressive enough to deal with real applications. We feel that there should be a close connection between the text of a program and the cost of implementing that program, especially in hardware. Without this connection, it is difficult to see how programmers would be able to make progress when partitioning their programs between hardware and software. An underlying assumption of our work is that automatic partitioning is currently intractable in the general case.

The language must also have expressive power for parallelism. Hardware is only faster than software when its inherent parallelism can be exploited. We believe that we are unlikely to make much progress by trying to extract parallelism from a program written in an inherently sequential language such as C. Many attempts have been made over the years to do precisely this, particularly in the context of compilation for pipelined and parallel computers, but modest success in certain specific cases is perhaps the greatest claim that can generally be made for these attempts.

The CSP specification language, and its programming language counterpart occam, offer two characteristics which are essential for our work and which are not shared by many other languages. Firstly, it has the expressive power to handle parallelism and communication easily. Secondly, there is an associated algebra [16] which allows us to re-write programs from one form into another, with great confidence that the behavioural properties of the program are preserved.

The language we actually use as the basis for much of this work is called Handel [17]. It is essentially a small subset of occam2 with only the minimum of control and data structures. It has however been extended so that we can use integers of arbitrary width. Also, new type conversion operators have been added to enable the programmer to move freely between this richer set of integer types. These extensions are essential so that the programmer can explicitly express the desired representation size of all storage elements. With hardware implementations it is much less reasonable than it is with software implementations to leave the choice of data representation to the compiler alone.

Our development philosophy is that we deliberately place all burdens on the programmer that we do not know how to handle both automatically and with reasonable efficiency in the compilation process. As the work progresses it will, hopefully, become possible to achieve more of the compilation automatically and so we expect the burden on the programmer to decrease. However, we have already reached a stage where interesting applications systems can be implemented with relative ease by mortal programmers! We hope that the application example provided later in this paper justifies this claim.

2.1 The Handel Language.

The basic data types in the language are integers, single-dimensional arrays of integers, and single-dimensional arrays of constants. The corresponding hardware implementations of these are registers, RAMs and ROMs respectively. Each of these basic data types has a sub-type which is the (arbitrary) width, in bits, of the data. In addition, the array types have a further parameter which is the number of elements in the array. The channels which mediate much of the communication between parallel processes are the point-to-point, unbuffered, synchronised channels of CSP/occam; they too have an arbitrary width.

Expressions in the language are formed from variables, constants, and operators. The operators include simple arithmetic and bitwise logical operations, field extraction and concatenation operators, comparison operators, and a conditional choice operator. The language also supports shared, common sub-expressions. These are important so that the number of separate physical copies of a particular expression evaluation circuit can be controlled by the programmer.

The control statements offered in Handel are simultaneous assignment, channel-based input and output, `While` and `Until` loops, `If`, `Case`, `PriAlt`, and sequential and parallel combinators, as well as a primitive form of procedure invocation. Since there is no single 'best' implementation of a procedure call in hardware, Handel offers a set of building blocks which aid whatever implementation of a procedure call is necessary for any particular instance.

The Handel language currently has no concrete syntax; it is defined only as an abstract syntax and its implementation is embedded in the SML programming system [18]. SML is a very supportive programming environment and it is especially useful for abstraction, meta-programming, and program transformation; it has proven indispensable for our work. The ability to write meta-programs allows us to keep the core of the Handel language very small, whilst also having the facilities to build higher level structures on top of the language. For instance the core Handel language has no data types other than arbitrary-width integers and single dimensional arrays of them. However, the SML level allows the language to be extended by the user or by library code to support, say, floating point numbers, tuples, or multi-dimensional arrays. Similarly, the control structures can be extended as desired. In essence, the SML environment acts as a very sophisticated, and fully type-checked, macro-processor which takes user programs and produces Handel code as output.

3 Modelling of Digital Hardware.

We start from the simple observation that most of today's digital hardware operates synchronously. This is no accident, as synchronous systems are far easier to design and implement by comparison with asynchronous systems. It is true that they don't have wonderful scaling properties, but they manifestly suit most of today's problems well. We therefore choose to restrict ourselves to hardware systems with a single clock.

Note there is nothing which insists on this restriction; we make it simply in order that we can build systems both quickly and reliably within the available hardware technology. When the implementation route seems attractive, it is quite simple to replace the compiler's control circuit primitives with asynchronous ones. In a recent collaboration, we have made some progress towards a way of proving the correctness of one with respect to the other using the notion of protocol converters [19].

We make the further simplifying assumption for the purposes of this paper that all flip-flops in the hardware will be updated on the rising edge, say, of every clock cycle. This is not a severe restriction as any hardware which has a more complex synchronous clocking arrangement can easily be represented in this form by the introduction of explicit multiplexing of input and feedback signals around each flip-flop.

The following simple occam program, called a *normal form* program, can thus model the behaviour of our synchronous hardware circuits:

```
BOOL s0, s1, s2, .... :
SEQ
  s0, s1, s2, .... := 0, 0, 0, ....
  WHILE running (s0, s1, s2, ...)
    s0, s1, s2, .... := e0, e1, e2, ....
```

The Boolean variables `s0, s1, ...` represent the flip-flops in the hardware circuit. The initial assignment may be necessary to represent some initial power-on state of the hardware. Our compilation scheme insists that the control state is initially zero, and for simplicity we currently extend this to all the state in the hardware. The Xilinx 3000 series DPGA chips which we generally use support this by resetting all flip-flops on power-up.

The Boolean predicate `running` can be used to represent hardware that successfully completes its actions when the predicate becomes false, so that some other behaviour can follow it. Alternatively, it can be replaced by `TRUE` to represent hardware that runs forever without completion. The interesting part of the program is the parallel assignment in which *all* state variables are assigned new values on each iteration of the loop. Each iteration represents one clock cycle and each expression `e0, e1` etc. is a function of the state variables `s0, s1` etc.

Such a normal form program can be interpreted directly as a netlist for the desired hardware. The Boolean variables are interpreted as flip-flops and the expressions are interpreted as combinational Boolean logic. This model obviously neglects many physical properties, such as propagation delays, but it is nevertheless a competent model of the logical behaviour of a collection of gates and flip-flops where there are no combinational cycles.

It is this interpretation of a normal form program as hardware which underlies the claim that software and hardware have been brought into the same framework. It is now possible to use the transformational capabilities of the language so that some parts of a user program can be rendered in normal form and interpreted as hardware, while other parts of it are compiled in the conventional way into machine code.

With no loss of generality, but with a significant gain in expressive power, we can now use integer variables and expressions as well as Boolean ones. This is justifiable because, by a change of representation (data refinement), any arithmetic or other complex operators in the right-hand side expressions can be turned into Boolean expressions, and any structured data objects on the left-hand side (such as integer variables) can be turned into collections of single-bit variables.

4 Compiling Software into Hardware.

4.1 Implementation Philosophy.

There are obviously many different ways to compile a particular user program into hardware, just as there are very many machine code sequences that could be generated from the same high-level program by a conventional compiler. So how should we proceed with choosing an implementation strategy? We chose to make just a few top-level decisions about implementation philosophy which have proven successful and which have enabled many lower-level decisions to be made rather simply.

Since we are motivated by the practical applications of systems compilation, we have to produce hardware/software systems which actually work, and to do so with no input of specialist electronic engineering knowledge during the process. Because of this, we have been strongly influenced by DPGA implementations of hardware circuits. We wish to make working hardware-software systems by purely software means and DPGAs are the obvious candidates for hosting the hardware implementations.

4.1.1 Synchronous versus asynchronous circuits.

The decision to use DPGAs also, for us, forces the decision to use synchronous rather than asynchronous circuits. Despite the fact that asynchronous circuits have many desirable properties, and that synchronous circuits don't scale well, today's DPGAs are not good vehicles for implementing asynchronous circuits. Even if they were, there is the problem that asynchronous circuits typically need more hardware for a given functionality than synchronous ones and today's DPGAs are rather small.

Bundled-data conventions can ameliorate the problems of hardware overhead of asynchronous circuits very considerably. Unfortunately they entail a massive practical problem which is that each sub-circuit needs to have its timing characteristics taken into consideration at the implementation stage. By contrast, using synchronous circuits there is only one major timing decision to be made, namely the selection of the global clock frequency.

Since we are trying to design hardware by software means alone, we do not allow ourselves to be involved in any manual intervention with the hardware implementation. Thus, our DPGA implementations are usually created by the Xilinx automatic place & route tools working directly from the output of the hardware compiler. As a matter of routine, we never intervene with our designs after the automatic compilation stage, to speed them up, or to decrease their size, or to assist in placement. In our view, this would in any case only help with the implementation of one specific circuit, and that work might have to be repeated later, even for a close variant of the same circuit. We fully accept that we lose out on system performance by taking this approach, and we justify it solely on the grounds that we are working towards a 'single button press' design environment rather than trying to create the fastest circuits possible in today's technology.

However, the above does not imply that we expect the first implementation of a co-system will be the final one. Indeed we almost always expect that some manual revision of the hardware part of algorithm, and perhaps the hardware/software boundary itself, may be necessary. In particular, we use the feedback from the place & route phase to reconsider how the hardware circuit can be made smaller or faster. To achieve this, any rewriting of algorithms and recompilation is possible, but we disallow intervention at anything other than the source code level. Since the language is explicitly not a hardware description language, there are no opportunities for directly building particular hardware structures as part of the solution. The effect of these constraints, is some loss of performance and area. However, we believe that this is more than compensated for by the fact that we can in actual practice regard the construction of the total system as a software programming task alone.

Having chosen synchronous implementations, we took a simple decision that the program, as presented to the hardware compiler, would have the parallelism and sequencing inherent in its text faithfully maintained in the hardware implementation. This means that there is a close connection between the text of the program and its spatial and temporal properties when converted to hardware. This is of considerable assistance to the programmer who is trying to create useful hardware/software partitions. This decision certainly does not mean that there can be no automatic shifting of computation between sequential and parallel forms; it simply means that this must be achieved by a program transformation step which happens ahead of hardware compilation.

4.1.2 Timing behaviour of programs.

A key decision in simplifying the compilation process was that all expression evaluation should be done in a single clock cycle. So, no matter how complex any expression in the program, a combinational logic circuit is implemented which calculates the value of that expression. Again, this forces any pipelining of large expressions, in order to achieve reasonable cycle times, to be rendered as program transformation steps ahead of hardware compilation.

```
INT r,a,b,c,d :                 INT r,a,b,c,d,t1,t2,t3 :
r := Abs ((a+b) + (c+d))        SEQ
                                  t1,t2 := a+b, c+d
                                  t3 := t1 + t2
                                  r := Abs t3
```

Figure 1: A Simple Expression Pipelining Transformation

A simple example of this type of program transformation is shown in Fig. 1. In this example, the left-hand program involves a moderately complex expression. In the right-hand program the expression evaluation has been spilt into three sequential steps, each of which involves an arithmetic (ripple-carry) computation.

In this case, it was particularly simple to balance the three steps so that they all take approximately the same amount of real-time to execute. However, even with this simple example, there is the difficult issue of whether to share any variables for the temporary results. A standard scheduling and allocation phase could be invoked here, but the issue is clearly becoming rather technology-dependent. The choice of target architecture could either strongly indicate, or strongly counter-indicate the introduction of additional variables. Currently, we have simply tried to ensure that the language has the competence to denote the major solutions, even though we may not know how to obtain the optimal solution automatically.

Having made the 'single-cycle' assumption for expression evaluation, it then becomes feasible, and beneficial, to ensure that the control circuits for the various language constructs impose no additional clock cycles on the implementation. This is reasonably straightforward to achieve and it results in a delightfully simple timing calculus for the programs. Thus, assignment and ready-to-run communication each take exactly one clock cycle to execute and all other language statements add precisely zero additional clock cycles to execute. This is with the honourable exception of the `Delay` statement whose sole purpose is to wait one clock cycle. It is in the nature of such a parallel hardware implementation that all statements are scheduled for execution as soon as they become ready (where 'ready' includes any necessary synchronisation implied by the program).

The simple timing calculus means that it is easy to look at the text of a program, manually or automatically, and to see how many clock cycles will be needed for any program fragment. This textual clarity is important for a programmer who wants to trade off parallel and sequential elements at a low level in order to achieve some desired level of performance. To assist the programmer further, the compiler offers a simple program transformation which tags each statement with the number of clock cycles it requires, or with bounds on that number if there is some non-determinism involved.

A further important benefit comes from the choice of a synchronous implementation strategy. It allows us to compile a language which has a larger set of laws than CSP/occam. These additional rules directly support more efficient implementations and the rules are readily understandable either by programmers or hardware engineers. The key observation is that all assignments which are scheduled into the same global clock cycle only update their destination registers as an atomic action at the end of that clock cycle. This allows us, for example, to implement a language which has the following as a transformation law:

$$\mathbf{a} := \mathbf{b} \;\|\; \mathbf{c} := \mathbf{d} \quad \Longleftrightarrow \quad \mathbf{a}, \mathbf{c} := \mathbf{b}, \mathbf{d}$$

where $\|$ is an infix notation for the occam `PAR` statement. There are no side conditions about any of the variables being distinct (except for `a` and `c` of course), as there are in occam. An example of the advantage that can be gained from this richer semantics is shown later.

4.2 Mapping Statements into Hardware.

The transformation of an arbitrary user program into normal form can be accomplished by a series of syntax-directed applications of the laws of programming to the program. The theoretical basis for this and the transformation steps themselves can be found in the references [20, 21, 22].

The current Handel compiler does not in fact follow these steps. Instead, the program is transformed directly into a netlist graph, which intentionally achieves the same result, but much more quickly. In fact it is much easier to comprehend the basis of the normal form transformation by looking at the fragments of netlist graph which are generated by each of the language constructs, rather than by looking at the normal form transformation steps themselves. The transformation laws are considerably obscured by the necessity to deal formally with shared use of the datapaths, which are just about the simplest parts of the hardware!

All variables in the user program are mapped to hardware registers which are constructed from sets of J-K flip-flops. The registers have input multiplexors if they have multiple sources. This happens when they are the target of more than one assignment or communication in the program. The control circuits for the statements in the program generate the multiplexor control signals and the clock enable signals for destination registers. As previously explained, all expressions are implemented as combinational logic.

In fact, this is about all there is to say about the datapath generation strategy, which is very straightforward. The datapath generated by this strategy thus exactly matches the dataflow graph of the original user program. If the programmer specifically wants some other form of datapath, it is his responsibility to transform the program into a form which exhibits the datapath architecture required, although we have implemented some automatic transformations of this nature for particular forms of datapath. The conversion of a user program into the combination of a machine code program and an application-specific microprocessor is a good example of such a transformation [23].

Each control construct in the program maps onto a control circuit in the hardware. We use a pair of control handshake signals (`start` and `finish`) for each circuit. A handshake signal which is active, is simply a signal which is high at the rising edge of the global clock. We also make an *environment assumption* that a start signal will never be given to a control circuit if there has not been a corresponding finish signal from any previous start signal. The individual circuits are designed to maintain this environment guarantee to any nested control circuits. Since the entire program is only a single statement at the top-level, this translates into an assumption that the environment will start the hardware program running just once and will not attempt to start it again before the program has completed.

To improve readability in the control circuit diagrams which follow, we use a box with a triangle in the upper-left corner to represent a single instance of a control circuit. Each control circuit has a single input and a single output signal which implements the control handshake protocol. The connections between these circuits and the datapath is fairly obvious and is thus not shown in any detail here.

4.2.1 Assignment.

Because of the method of handling expressions, the assignment statement is particularly simple to implement and its control circuit is shown in Fig. 2. The `start` signal forms the clock-enable signal for the destination register(s) of the assignment. At the end of the cycle in which the assignment is scheduled, the expression hardware has calculated the new value (by assumption), and it is thus loaded into the destination register. The `start` signal is delayed by a single cycle to provide the `finish` signal, since an assignment is always scheduled immediately, and it always completes in exactly one clock cycle (again by assumption).

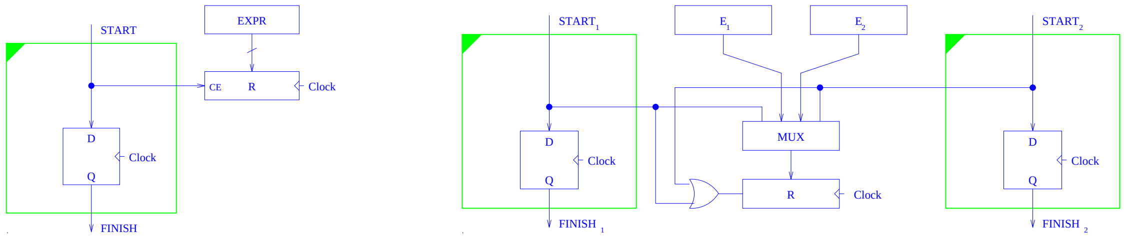

Figure 2: Assignment: Control and Data Multiplexing.

The left-hand part of Fig. 2 shows the circuitry generated by a single assignment of the form `R := EXPR`. The right-hand part of Fig. 2 shows the portion of datapath generated by two assignment statements which target the same register. An active `start` signal steers the appropriate expression value through the multiplexor to the input of the destination register. Whichever `start` signal is active also enables the destination register via the `OR` gate.

If the rules of occam programming are enforced, then any two assignments to the same variable cannot be in separate arms of a `Par` statement, so the issue of what happens when the two assignments are scheduled at the same time simply does not arise. If the user wants to use the additional transformational rules of Handel programs to allow such assignments into two parallel processes, then there is a proof obligation which must show that the two assignments cannot be scheduled in the same clock cycle, and the user must also shoulder the additional burden of arguing what the semantics of his particular use of such shared store is.

4.2.2 Sequential Composition.

The control circuit for sequential composition is trivially simple. The start and finish signals of the component processes are connected together together in a 'daisy-chain' as shown in the left-hand part of Fig. 3. It is particularly clear from this diagram that the control strategy is basically that of 'one-hot' control state encoding. This particular scheme appears to be well-suited to DPGA implementations since it requires little in the way of wiring resources when compared to encoded representations of control state.

4.2.3 Parallel Composition.

The right-hand side of Fig. 3 shows the control circuitry for parallel composition. This circuit passes control simultaneously to all the parallel statements to initiate parallel execution. The `PAR` statement of occam is defined to be synchronised, so that the whole construct terminates only when all of the constituent components have terminated. Thus, the `PAR` control circuit collects together the separate finish signals in a set of flip-flops and produces its own finish signal as soon as the last finish signal is generated. In addition, this signal resets the synchroniser flip-flops ready for the next time that this circuit is used.

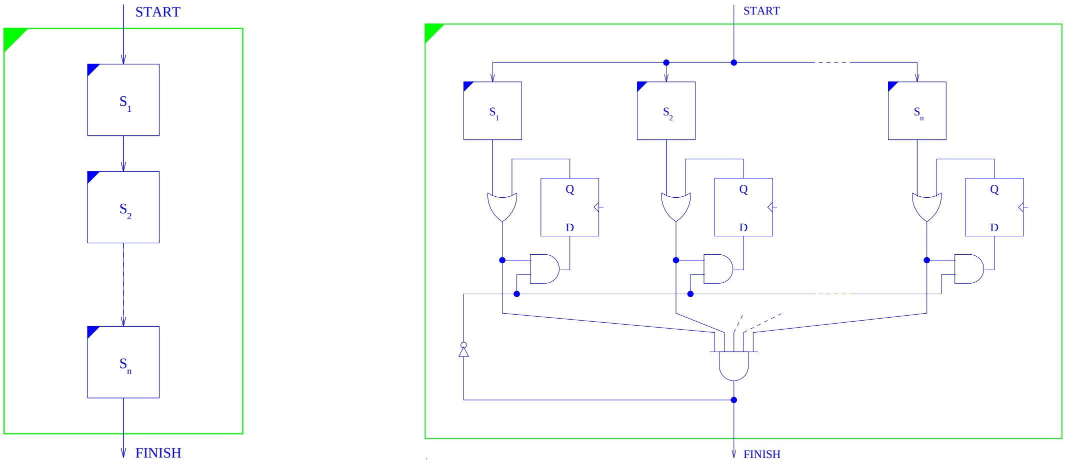

Figure 3: Sequential and Parallel Composition.

Optimisations are routinely performed to remove such synchronisers, wherever a textual analysis of the program can demonstrate a partial ordering on the termination times of the individual processes.

4.2.4 Miscellaneous Constructs.

Fig. 4 shows the control circuitry for some minor control constructs. The `Skip` and `Delay` constructs have no effect on the state of the computation. They take exactly 0 and 1 clock cycles respectively to complete.

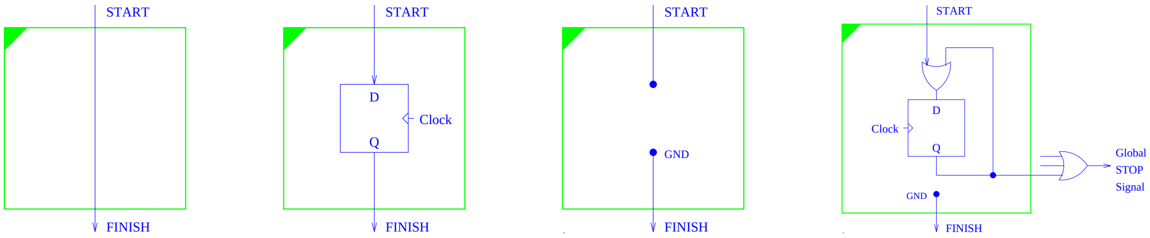

Figure 4: Skip, Delay, Stop1, Stop2.

The `Stop1` and `Stop2` circuits show two different refinements of the `Stop` construct. `Stop` represents a broken computation ('bottom' in the semantics). The `Stop1` circuit simply refuses to pass on the handshake signal (though of course it could do anything at all!) The `Stop2` circuit uses a local latch to remember that a particular instance of the `Stop` command was activated and the state of this latch can be monitored by external hardware for debugging purposes, for example.

4.2.5 Channel Input and Output.

Fig. 5 shows the control circuitry for synchronised channel input and output. The left-hand circuit is the same for either and input or an output command. Synchronisation is achieved by looping back the `start` signal through a flip-flop and a multiplexor which is controlled by the `transfer` signal. The control token is trapped in this feedback loop until activation of the `transfer` signal.

The `ready` signal goes to the arbitration circuit and returns as the `transfer` signal to indicate when the partner to this communication is also ready to run. When both circuits are ready to communicate, the lower-left circuit is activated, which is simply the assignment circuit seen earlier. It can be clearly seen from these diagrams that communication is just distributed, synchronised assignment.

As with assignment, the scope and usage rules of occam guarantee that there can be no more than one input and one output command scheduled for the same channel at the same time, hence the very simple arbitration circuit on the right-hand side of the figure.

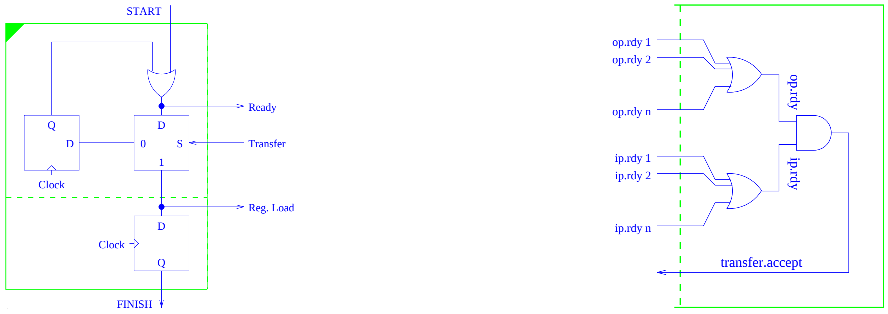

Figure 5: Communication: Synchronisation and Arbitration

4.2.6 Binary Choice.

The left-hand side of Fig. 6 shows the control circuitry for the binary choice, or `If`, statement. The `start` signal is steered to trigger just one of the guarded commands under the control of the guard expression. Since only one command can be active, the `finish` signals of the two arms can be simply `OR`ed together to derive the completion handshake signal for the entire construct.

Figure 6: If and While Constructs.

4.2.7 Guarded Iteration.

The right-hand side of Fig. 6 shows the control circuitry for the `While` loop. It is somewhat similar to the `If` circuit except that here the start `signal` is steered either to the feedback loop or to form the `finish` signal, under the control of the circuit which implements the Boolean guard expression. The control token is trapped in the feedback loop until the controlling Boolean expression, `BE`, evaluates to `false`.

A very similar circuit exists for the `Until` loop which always executes at least once and tests the guard expression at the end of the loop. This form of the loop doesn't exist in occam but it is sometimes useful in hardware compilation since it avoids the need for the duplicated hardware (or procedure call) which would be necessary to implement the program `Until x Do y` as the program `y; While Not x Do y`.

There is a tricky design issue involved with these iteration circuits. The particular implementation shown here is not at all the obvious one, and without further steps being taken it is actually capable of failing. This implementation was pursued despite these problems because it leads to a simpler and more reasonable timing calculus than the alternatives.

The control circuit fails if the controlled statement `S` has any path through it which executes in zero time. This is because there is then a combinational path through the box `S`, which then creates a combinational loop in the hardware because of the feedback loop in the `While` control circuit. For instance, either of the programs `While x Do Skip`, or `While x Do (If a Then b Else Skip)`, would create a loop in the combinational circuitry because of the decision that a false-guarded loop should not take a clock cycle to execute.

The simplest way of implementing the desired semantics without introducing any undesirable combinational loops would be to insist that evaluating the guard expression should take one clock cycle itself, thus breaking the combinational loop. However with hardware compilation we will often be interested in writing loops which execute a single iteration in a single clock cycle. Here it would be unacceptable to introduce even one extra cycle into the loop execution as it would halve system performance.

The scheme adopted in Handel is to use the time-efficient circuit and to make the compiler perform a pre-transformation on the program to replace certain instances of `Skip` with `Delay` so that no zero-time loop bodies exist in the program.

4.2.8 Channel Protocol Converters.

There will usually be some channel-based data transfer across the boundary of the Handel program so that it can interact with its environment. For example, the program may wish to interact with an external RAM, a microprocessor bus, or a communications chip. Our choice was to make all such communications look like ordinary channel communications to the Handel program; sometimes using a bundle of channels for some complex interfaces.

However, there is no reason to expect that these external components will want to communicate with the hardware-compiled circuit using the particular channel protocol imposed by the compiler. We solve this problem by introducing a library of Channel Protocol Converters, CPCs, which mediate the communications between the hardware-compiled circuit and the external devices. Some of these CPCs are very simple, many of them containing only between 0-5 gates, latches etc. Others are considerably more complex. Because Handel is *not* a hardware description language and such CPCs sometimes require very careful hardware design, we believe it is unreasonable to expect the compiler to generate such CPC circuits automatically from program language statements in the user's program. Instead, a library of parametrised, pre-defined components seems the right solution here.

Currently, our CPCs are designed at the level of gates and flip-flops and are thus not subject to any automatic checks. The major reason why they cannot yet be brought within the language framework is that some CPCs, such as that between the DPGA and the microprocessor, must handle truly asynchronous communications. This necessitates metastability resolvers and the other paraphernalia of communications between separately clocked systems.

Note that because of physically unavoidable metastability problems, such circuits cannot be proven correct - because they simply are *not* correct. It would take a new notion of provable correctness, perhaps based on statistics or on specifications which explicitly encompass failure, to address such issues. One possible way to handle the issue would be to introduce into the language channels which are allowed to fail non-deterministically; it would then be safe to compile these to circuits with metastability hazards. It would perhaps be a rare programmer who was willing to include constructs in his program which can fail outside his control, but at least it is an honest way of dealing with the problem.

These considerations introduce a major implementation issue, even after accepting the inevitable problems posed by metastability itself. There is naturally a problem with any CPCs that cross clock boundaries. For example, if a clocked DPGA circuit is interfaced with the bus of a separately-clocked microprocessor, then there is an inevitable problem of clocking and metastability in the interface circuit. To build such interfaces successfully requires more than a definition of the logic circuitry alone; it is necessary to specify real-time guarantees on certain aspects of the circuit's performance.

With today's vendor tools for DPGAs, this is impossible to achieve when automatic place & route is used for circuit layout. Our solution is to generate placement and routing information for such critical CPCs as well as the circuitry. This needs a great deal of work to set up the library of parametrised interface circuits, but is the only way of achieving flexibility while still guaranteeing very tight timing constraints, at least until vendor tools are capable of dealing with such constraints.

With the Xilinx software that we most commonly use, it is possible to specify the placement of logic blocks as part of the XNF netlist file format that we use as our interface into Xilinx DPGA technology. Unfortunately, their netlist format does not allow the user to specify the routing as well. For that reason, our CPC macro generator also produces a separate file of Xilinx editor commands, which allows us to specify the exact routing resources needed as well as the circuitry and placement information.

It is unfortunate that so much technology-specific work has to be put into this part of the compilation system, but at least it means that virtually the whole of the rest of the system can be technology independent. We look forward to vendor tools that can take away some of the burden from the user here by offering implementations which meet user-provided constraints, on timing performance at least.

4.3 The Compilation Process.

The compiler parses the user's program by recursive descent of the abstract syntax tree representation. It generates the netlist for one of the above control circuits each time it encounters a control construct in the program. The compiler also adds sets of flip-flops to the netlist when it encounters any variable declarations. Channel declarations generate the arbitration circuitry only, as the synchronisers are generated by the communication commands themselves. The compiler also adds multiplexors to any multiply-sourced destinations to implement the simple datapath strategy outlined earlier.

The association of a CPC with an external channel calls up the appropriate CPC macro generator which adds the CPC circuitry to the netlist, and may also generate some technology-specific files if it involves physical pre-placement of circuitry.

The compiler's netlist is generally technology independent. There are back-end netlist transformation routines which massage the netlist so that it is suitable for the different DPGA's that we use. These transformations might impose a ceiling on the number of inputs to any gate, for example, or might implement one of the compiler's rather complex abstract flip-flops with one of a small number of physical circuits, depending on the target technology. Although we mostly use Xilinx 3000-series DPGAs, we have also successfully produced designs for Atmel and Algotronix (now Xilinx) DPGAs [24]. The netlist can also be produced in VHDL form for simulation purposes, or to access other implementation routes.

The resulting netlist could be implemented by any available means, such as MSI TTL, gate arrays, ASICs, or DPGAs. However, because of our interest in fast system development and dynamic systems we use DPGA implementations exclusively. The DPGA vendor's place & route software eventually produces a configuration bitmap from our netlist. This bitmap is typically then converted into the form of an occam array declaration. In this textual form, it is integrated with the microprocessor program, which is then responsible for downloading the bitmap into the DPGA at system initiation time. The output of the place & route software also gives an estimate for the worst-case delay through the combinational circuitry, and this is used to set the frequency synthesiser which generates the DPGA clock speed. This information is also incorporated as code directly into the program which boots up the DPGA. Virtually the whole of the compilation and implementation process is completely automatic apart from having to manually submit the routing edits file to the Xilinx XACT software since there seems to be no way to make this software run as a background (batch) process.

5 Implementing Software.

The software part of a hardware/software system can be compiled using standard compiler tools into machine code for commercially available microprocessors. The software end of the hardware/software interface can be provided quite reasonably by a library of procedures. This is how we handle this interface, but with the additional wrinkle that the code for the interface procedures is held in parametrised form and produced as program text as a side-effect of instantiating particular CPCs.

It is interesting to note that the full problem of implementing software is, counter-intuitively, somewhat harder than implementing hardware. The reason is the complexity of the underlying execution mechanism. For hardware this is simply a collection of gates and flip-flops, whereas for software it is a complete computer - which itself has to be implemented as hardware. If we are interested in high reliability implementations of software, it is necessary (i) to include a formal description of the processor on which the machine code is to run as an integral part of the derivation, and (ii) demonstrate the correctness of the compiler.

Fortunately, our approach allows us to solve both these problems simultaneously. Using the laws of programming, we can take a user program and massage it into the form of an Instruction Set Simulator (ISS) program together with the appropriate machine code program. As the laws are correctness-preserving, we know that any ISS program designed by this method must be correct, as must the machine code program that it runs. Such ISS programs are commonly used as a formal description of processor behaviour;

our common programming framework ensures that we can simply put such an ISS program through our hardware compiler in order to get an implementation of the processor.

We have implemented a number of systems which take a user program and automatically design a microprocessor to execute that particular program. The approach is essentially that of parametrising an abstract description of a processor. Related work at Oxford has produced methods of compiling programs into machine code with provable correctness. For further details of these aspects of our work, the reader is directed to [23, 25, 21, 26, 27, 28].

Although the issues of parametrised processors and provably correct compilers are interesting, for the purposes of this paper we will ignore them and regard the compilation of programs into machine code as a trivial problem which can be solved by using off-the-shelf compilation tools.

6 Implementation of Hardware/Software Systems.

In order to experiment with actual hardware-software co-designs we have constructed a hardware platform, known as HARP1 [29], which is now licensed for commercial production [30]. As shown in Fig. 7, HARP1 consists of a 32-bit RISC microprocessor closely coupled with a Xilinx XC3195A DPGA [31] with its own local memory. The microprocessor can load an arbitrary hardware configuration into the DPGA via its bus. A processor-controlled 100MHz frequency synthesiser allows the DPGA to be clocked at its highest rate, which depends on the depth of combinational hardware and DPGA wiring delays.

The HARP1 printed circuit board uses surface mount technology and is 90mm $\times$ 168mm in size. It is a (size 6) industry-standard TRAM module [32] so that it can be incorporated easily into a wide range of off-the-shelf hardware and interfaced to a variety of host computing systems. The commercial version now under construction has a similar specification but using a size-4 TRAM module (90mm $\times$ 112mm).

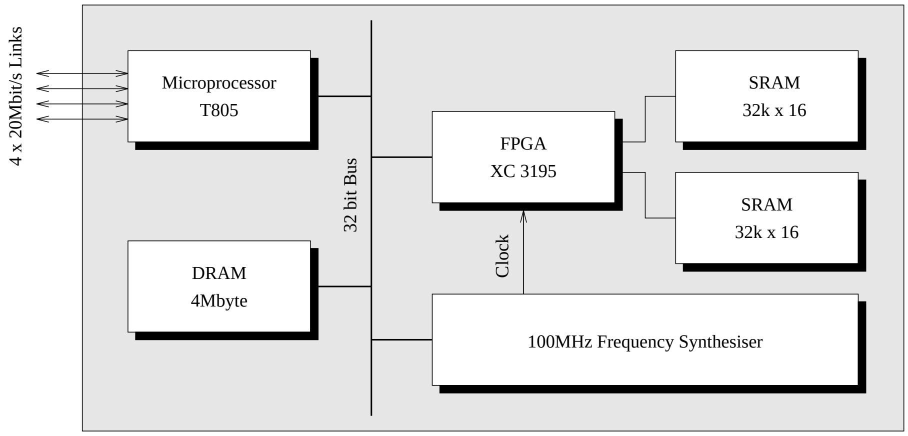

Figure 7: The HARP1 Reconfigurable Computer Module

The main input/output channels for the board are the 4 $\times$ 20Mbit/sec serial links offered by the microprocessor. This makes it very easy to link these boards together or to link them with other available TRAM modules, such as A/D converters, frame grabbers and other microprocessors and DSP systems. However, the bandwidth is rather limited and it is planned that the next board will have much increased input/output bandwidth so that multimedia applications, for example, can realistically be targetted.

7 An Example Hardware-Software Codesign.

Here we show an example of a hardware-software system that uses the processor and DPGA chip on the HARP1 board. It is necessarily a simple example in order to show it in detail. Although the whole application does nothing interesting it is clearly trivial to replace the kernel algorithm with something more complex, leaving the system structure exactly as presented here. Of course, the methods presented here apply to much more interesting applications, and one of these will be presented later, but naturally not with the same level of detail.

The complete system is the parallel combination of two processes, one of which is implemented by hardware and the other by software.

7.1 The Hardware Process

The hardware part of the system is shown in Fig. 8 and is a program which fills one bank of the HARP1 static ram with numbers, and then acts as a 'peek' process controlled by the microprocessor. The second loop in the hardware program repeatedly inputs an address from the T805 microprocessor, looks up the corresponding value in external SRAM and sends the value back to the microprocessor. This is a trivial, 'manufactured' example so that there are no details of the application to obscure the structure of the implementation.

In this particular program, the hardware process communicates with the microprocessor using a pair of 16-bit channels. It also communicates with an external hardware component, namely one of the banks of 32k x 16-bit SRAM on the HARP1 board.

Communication with the SRAM, as with all external communications is modelled by occam channels. Reading from the SRAM is done by a parallel combination of an output of the address and an input of the data. Writing is similarly accomplished with two output statements in parallel. Note that this program might easily have been produced from a higher-level program which simply declared the array and used it as if it were an internal resource rather than an external process as here. Similarly we normally use a language construct which packages up the two parallel communication statements into a single array reference. They are shown in expanded form here to aid the explanation of the implementation.

Communication with the microprocessor is usually via its bus and is consequently quite fast. Commercially-available occam compilers do not support the notion of Channel Protocol Converters, and instead they support just a small number of communications protocols (namely links and within-processor channels). Rather than trying to change the vendor-supplied occam compiler, the communications primitives at the microprocessor end appear as procedure calls in the code. Thus, an output statement from the microprocessor to the DPGA process might appear in the program as `DPGA.write.chan (Output_Channel, expr)`, rather than `Output_Channel ! expr`. The necessary channel synchronisation can be achieved either by polling or by interrupt, according to which Channel Protocol Converter is specified by the programmer.

On looking at the program text, an occam programmer would probably only be surprised to see the `a <- 15` expression which delivers the least significant 15 bits of `a`, and the `INT15` type which declares a 15-bit integer. The same programmer might however be shocked to see the multiple use of the value of `a` in the first parallel statement since this violates occam rules. A version of this program which respected the occam rules could have been written, but would have introduced some unnecessary sequentiality into the program. In our example, this sequentiality has been optimised away by applying the richer set of laws of Handel programs.

The optimisation can be explained by reference to the synchronous (single-cycle) behaviour of the language and its implementations. The CPC for the SRAM makes a guarantee to the compiler that it will always run without being delayed by the environment, so that all three arms of the parallel construct will certainly be scheduled at exactly the same time and will complete simultaneously after a single clock tick. The statement is therefore not only meaningful, but also useful, since it halves the execution time that would be enforced by a straightforward interpretation of the corresponding sequential occam program.

For interest, some of the statistics output by the hardware compiler for the program in Fig. 8 are shown here:

```
After compilation  : 86 LATCHES, 462 GATES
After optimisation : 84 LATCHES, 267 GATES
```

7.2 The Software Process

The software process in Fig. 9 relies on a number of procedures not detailed here, as they are of little interest. Some of these, such as `so.write.string`, are standard occam library procedures for communicating with a host computer (usually a 486 PC in our case, if one exists at all in the implementation).

```
PROC main_hw (
  CPC_Tputer.PAIR.IN     : CHAN OF INT16 FromT805,
  CPC_Tputer.PAIR.OUT    : CHAN OF INT16 ToT805,
  CPC_SRam.RAM.ADDR      : CHAN OF INT15 SRH_Addr,
  CPC_SRam.RAM.IN        : CHAN OF INT16 SRH_Din,
  CPC_SRam.RAM.OUT       : CHAN OF INT16 SRH_Dout)
  INT16 a, d :
  SEQ
    UNTIL a = 32768(INT16)
      PAR
        SRH_Addr ! a <- 15
        SRH_Dout ! 0(INT16) - a
        a := a + 1(INT16)
    WHILE TRUE
      SEQ
        FromT805 ? a
        PAR
          SRH_Addr ! a <- 15
          SRH_Din ? d
        ToT805 ! d
:
```

Figure 8: The Program for Hardware Implementation

There are also the `DPGA.read` and `DPGA.write` procedures which are specific to our HARP1 system. They use occam PORT input/output commands to communicate over the transputer bus with a Channel Protocol Converter in the DPGA in order to support standard occam, synchronised channel communication between the software and hardware processes. These are the procedures referred to earlier whose bodies are automatically produced by the CPC generator.

7.3 The Complete System

The program fragments in Figs. 8 and 9 are simply two parallel processes from the full application. Suppressing much detail, the top-level structure is simply:

```
PAR
  Hardware_Process (in_chan, out_chan)
  Software_Process (out_chan, in_chan)
```

In addition, such top-level user processes are tagged with the identity of the compiler that is to be called to implement them. In this way, the code for the software process is targetted onto the occam compiler, and the hardware process code is passed to the Handel compiler. The top-level compiler process takes the parts produced by these compilers and builds a composite program. This composite program is essentially the original software program, slightly extended with the code necessary to bootstrap the hardware process. The following shows an extract of the automatically modified version of Fig. 9:

```
PROC main_sw ()
  INT s :
  INT16 r :
  VAL config IS
      [#35F004FF,#FFFBECF3,#FBDBFFC3,
       #FBFFFFFF,#FFF7FFFF,#F7F7F7FF,
       ....
  SEQ
```

```
PROC main_sw ()
  INT s :
  INT16 r :
  SEQ
    s := 0(INT)
    WHILE s <= 32768(INT)
      SEQ
        DPGA.write (INT s)
        DPGA.read.16 (r)
        so.write.string (fs, ts, "*n result = ")
        so.write.int (fs, ts, INT s, 12)
        so.write.int (fs, ts, INT r, 12)
        s := s + 1(INT)
:
```

Figure 9: The Program for Software Implementation

```
PAR
  load.fgpa.configuration (config)
  set.clock (10*MHz)

s := 0(INT)
....
```

The `load.fgpa.configuration` procedure is a HARP1-specific procedure for loading a given configuration bitmap into the DPGA. The `config` array is simply the output of the Xilinx place & route tools, rendered automatically into the form of an occam array declaration so that it can be compiled into this top-level program. An alternate scheme is available to read the configuration at run-time from the Xilinx binary output file.

The `set.clock` procedure is again HARP1-specific; it sets the desired global clock frequency by programming the frequency synthesiser. The clock frequency can be determined automatically from examining the timing predictions produced by the Xilinx place & route software, or by other means.

The mapping of the complete system onto the HARP1 board and the associated host processor is shown diagrammatically in Fig. 10. An industry-standard motherboard plugs into the PC bus and supports the HARP1 daughter module. In this instance the mother board does nothing other than offer mechanical support to HARP1, and provide the 20Mbit/sec UART to interface the PC to a transputer serial link. Some applications may not use the PC for user interface, file store or network support, so the HARP1 is then self-supporting, drawing only power from the PC bus, and its microprocessor clock from the motherboard.

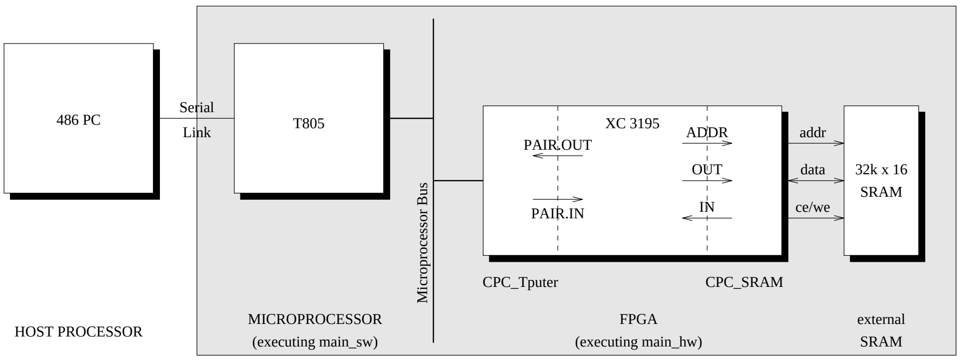

Figure 10: Mapping the program onto HARP1

```
WHILE true
  SEQ
    -- Input new image and form difference image
    WHILE receiving_new_image
      SEQ
        from_host_micro ? new_pixel
        difference_image [x,y] := threshold (abs(new_pixel - image[x,y]))
        image [x,y] := new_pixel
        update_x_and_y

    -- Find bounding boxes of changed regions
    WHILE image_not_fully_explored
      SEQ
        find_an_edge_pixel_of_a_new_region
        WHILE not_back_to_starting_edge_pixel
          SEQ
            find_new_edge_pixel
            update_bounding_box
        to_host_micro ! bounding_box
```

Figure 11: The Hardware Object Tracker : Pseudocode

8 A Larger Example : Video Object Tracking.

This application demonstrates a particular use of our compilation environment. It also represents about the limit of complexity that we have explored to date on a single HARP1 card. We believe it would be difficult to get much more into the single XC3195 DPGA than shown here, without having to resort to manual intervention in the place & route process.

The basic task is to extract from a real-time video source, a frame-by-frame list of the bounding boxes of objects which are moving in the scene. The imagined context of this application is an automatic security camera system which will usually relay a wide-angle view of the entire visual field, but which will pan and zoom the camera onto any moving objects within the scene. For instance, an extended version of the system might be used to record full-frame images of all moving objects onto videotape or to alert a human operator that a particular sort of moving object had been detected, perhaps an intruder.

A transputer-based frame-grabber digitises the full video image and then subsamples it to produce a $128\times128\times8$-bit grayscale image stream along its output link. This video stream is taken into the HARP module via one of the transputer's links and is immediately passed to the DPGA, using channel communication implemented via the transputer bus. A bottleneck in the current implementation is the transputer's inability to move the images quickly enough over its serial links - hence the limitation to 128×128 images. This type of application really needs a hardware environment where the video images are directly accessible to the DPGA. We hope to design and build an advanced version of the HARP system with multi-media capability which will be a much better host for this type of application. However, the performance we are achieving from a single HARP board is still very respectable in cost/performance terms even though it was never envisaged that we should even attempt to support video algorithms on it.

The application works by differencing successive digitised images and then thresholding the resulting difference image. One of the SRAM banks holds a copy of the latest image as well as the latest difference image. A region-filling style of algorithm then explores the over-threshold changes in order to determine the boundaries of the changed regions. A flawed but inexpensive algorithm is used here which is essentially a flood-fill algorithm based on horizontal spans. The algorithm proceeds normally except that it doesn't recurse when the horizontal span splits. It thus completely misses out some parts of concave regions. In practice, the cost of performing full edge-following is not justified by the modest increase in performance. Regions which fall below a given area threshold are discarded.

The boundaries of the above-threshold regions will hopefully correspond with moving objects in the scene. The changed regions are then represented by their bounding rectangles alone. These are actually obtained by computing the maximum and minimum **x** and **y** values during the region-filling process itself. Objects which are smaller than a given area threshold are also discarded.

To help ensure that the bounding boxes do indeed correspond with moving objects, and to increase the noise-immunity of the system, the bounding boxes are modelled and tracked over time by a multiple-object, predictive Kalman filter. This filter both models and predicts the temporal behaviour of the bounding rectangles in terms of their position and size, and the first derivatives of these quantities. Thus, the filter would 'expect' that a person walking behind an obscuring object would emerge from the other side and the movement might well be tracked through such an interruption in visibility. The filter maintains a 'confidence level' for each of the rectangles which models the likelihood that this corresponds to a moving object in the scene. An object description is deleted when the filter's confidence level for that object drops below a given level, when the object stops moving, for instance. Similarly, the filter creates an object description when a new moving object is detected.

8.1 Implementation

The algorithm is partitioned so that the pixel-based operations run in hardware, while the Kalman filter and user interface run in software. This is a good partition since the hardware is best suited to low-complexity, high-bandwidth computation, while the microprocessor is best suited to high-complexity, low-bandwidth computation. Thus, the frame differencing, thresholding, region-filling, and bounding rectangle computation are all performed in the single DPGA chip, which also handles communications with the microprocessor. Pseudocode for this part of the algorithm is shown in Fig. 11.

The DPGA accepts a sequence of $128\times128\times8$-bit frames from the microprocessor. Each frame is transferred to the DPGA by a single microprocessor block-move instruction to the port address associated with the input channel for the hardware process. For each frame, the DPGA calculates and returns an arbitrary length list of bounding rectangle descriptors to the microprocessor.

The entire hardware program is rather too large to be included here. It consists of 402 lines of abstract Handel-in-SML code, including comments and file-folding overhead. When this SML code is evaluated, it generates the hardware program in the abstract syntax form suitable for the hardware compiler. When this abstract syntax program is pretty-printed in concrete, occam-style, syntax, it appears as 126 lines of code (uncommented). A fragment of this program is shown in Fig. 12 simply to give the reader some feel for the level of the language.

The other two Handel bit-field operators which were not introduced earlier appear in this fragment. They are `a \\ n` which returns the least significant `n` bits of the expression `a`, and `a ^ b` which returns the two expressions concatenated bitwise (with `a` at the least significant end). Also, `FALSE` and `TRUE` are the representations of the 1-bit constants 0, and 1 respectively. The pretty print suppresses some of the detail that goes into the construction of the CPCs which are listed in the program heading. These details include the assignments of the channel data and control signals to DPGA pins. The names of the protocol converters should be enough to explain their role in the program.

About ten lines of the concrete-syntax program would not easily be comprehended by a human reader as they were macro-generated by the programmer's SML meta-program. None of these lines are shown in Fig. 12. In practice, we rarely look at the concrete syntax program; it is merely an intermediate representation which is sometimes useful for reference or documentation purposes.

For interest, some of the statistics generated by the hardware compiler for the program in Fig. 11 are shown here:

```
After compilation   : 195 LATCHES, 1337 GATES
After optimisation  : 184 LATCHES,  758 GATES
After fan-in adjust : 184 LATCHES,  793 GATES
```

The Kalman filter, image stream transfer, and menu-based user interface are implemented in occam software running on the HARP transputer as shown in Fig. 13. Although the hardware is hosted in a 486 PC box, this has no part in the running application; it is however used to bootstrap the HARP board and the other two TRAM modules.

```
PROC main (
  CPC_TPChanInOut.PAIR.IN  : CHAN OF INT8  d_data,
  CPC_TPChanInOut.PAIR.OUT : CHAN OF INT9  d_res,
  CPC_SRam.RAM.ADDR        : CHAN OF INT15 Addr0,
  CPC_SRam.RAM.IN          : CHAN OF INT8  Din0,
  CPC_SRam.RAM.OUT         : CHAN OF INT8  Dout0)
  INT14 c_area :
  INT8  d_old, c_typos, c_txpos, c_cypos, c_cxpos :
  INT7  c_xpos, c_ypos, c_minx, c_maxx, c_maxy :
  BOOL  d_bank, c_pass, c_lr, c_flag :
  ....
  WHILE TRUE
    INT15 FUNCTION inccxcye() IS ((c_cxpos <- 7) ^ c_cypos) + 1 :
    INT14 FUNCTION incxye()   IS (c_xpos ^ c_ypos) + 1 :
    PROC addrout()    IS  Addr0 ! ((c_cxpos <- 7) ^ (c_cypos <- 7)) ^ d_bank :
    PROC endofframe() IS  d_res ! 1 ^ TRUE :
    PROC getnext()    IS  d_data ? d_old :
    PROC resetcxcy()  IS  c_cxpos, c_cypos := 0, 0 :
    PROC inccxcy()    IS  c_cxpos, c_cypos := inccxcye() <- 8, inccxcye()\\7 :
    ....
    SEQ
      SEQ
        endofframe ()
        PAR
          d_bank := TRUE
          resetcxcy ()
        getnext ()
        UNTIL (MSbit c_cypos)
          PAR
            getnext ()
            W0gen0 ()
            inccxcy ()
        PAR
          resetcxcy ()
          d_bank := d_old <- 1
        UNTIL (MSbit c_cypos
          INT5 FUNCTION diff() IS ((ABS((d_new^FALSE) -
                                   (d_old\\2)^FALSE)))\\1)\\1 :
          SEQ
            ....
```

Figure 12: The Hardware Object Tracker : Pretty-Printed Extract

```
PAR
  -- Command interpreter
  WHILE true
  SEQ
    keyboard ? user_command
    execute (user_command)

  -- Data input/output and Kalman filter
  WHILE true
    SEQ
      frame_grabber ? image
      to_dpga ! image
      WHILE more_data_from_dpga
        SEQ
          from_dpga ? bounding_box
          update_image (bounding_box)
          match_and_update_filter_predictions (bounding_box)
      transform_if_needed (image)
      to_frame_buffer ! image
:
```

Figure 13: The Software Part : Pseudocode

8.2 Performance

The final output of the system as it currently stands, is a stream of video images passed from the HARP board to a transputer-based framebuffer for viewing on a monitor. The output video images can be presented in a number of ways, one of these being with the imposition of the bounding boxes as coloured overlays. The system performance that this application achieves is between 12-23 frames/sec, depending on what is happening in the scene. This seems to us to be fairly good performance, considering that it is obtained from two general-purpose chips, and has been programmed in a general-purpose language, with absolutely no intervention being made by the programmer below the level of the programs he wrote.

When this same application is run on the microprocessor alone, its performance drops by a factor of six. This ratio could be even greater if the hardware platform allowed video input/output into the memory which directly accessible to the DPGA, rather than being routed via the microprocessor as here. To match this performance in software alone would obviously require the use of at least six similar microprocessors. However, it is not at all clear how the computation could be partitioned to achieve this easily. Hence, the application gains a very real advantage from this hardware/software implementation; the addition of a single DPGA chip has had at least the benefit that would have been achieved by parallelising the software and using another five microprocessors. Again, this seems like a good reward for the modest increase in hardware and programming effort that was entailed.

We have implemented a demonstration mode in the program which displays the contents of just one of the Kalman-tracked windows and performs a software zoom on the image to bring it up to a fixed size. This gives a good idea of what the system would look like when connected to a video camera with computer-controlled pan and zoom. Picture quality is necessarily sacrificed since we are operating from a locked-off camera with a software zoom implemented by pixel replication. The software zoom also slows down the frame rate to about ten frames/sec. It is clear however from observing the system in operation in this mode, that it is tantalisingly close to being an exploitable commercial product, though this was not a goal of the project.

8.3 Summary

Perhaps the most impressive feature of the system is that this fully operational, high-bandwidth combination of hardware and software was constructed by an undergraduate programmer, who used no knowledge of hardware in its construction [33]. Indeed, the programmer still (almost proudly!) proclaims that he has no knowledge whatever of hardware. It is literally true that no-one examined, or even thought about, any circuits or layouts during the building of this system.

To judge the amount of effort that went into its production, this was originally set as a 100-hour, third year undergraduate project. This produced an interesting, though impractical, design for the object tracker. Because of the potential of the design, the student was hired to work for twelve weeks over the summer vacation and he completed the whole system in rather less than that time. Indeed the Kalman filter system and user interface were added because the implementation had proceeded so quickly. The hardware program went through twenty iterations, not all of which progressed as far as the place & route stage. A compilation would typically take only moments, gate-level optimisation rather longer, with the place & route taking almost all the time and usually being handled as an overnight run on a Sparc 5 workstation.

However, in exchange for not having to learn anything about hardware the student did have to learn a few new programming tricks. Most of them were learned via a 16-hour lecture course on CSP, and a five-day course on occam programming which together taught him the fundamentals of parallel processing. The other additional skill which was necessary was to understand the timing semantics of Handel, but as this is very simple this took him very little time to grasp. After that it was a matter of iterating through the design/implement/modify cycle until a satisfactory result was obtained.

It is certainly true that writing software and having to think about its detailed temporal behaviour is considerably harder than when such considerations can be ignored. It is however, not so hard as to be impractical as this example demonstrates. In any case, in actual practice it is almost always the case that the programmer has to think very carefully about the time and space implications of the key algorithms in his applications even if writing in C on a conventional processor. Our route makes the time and space overheads very explicit and quite simple to understand, and offers tangible rewards to the programmer for considering them in detail.

In many ways our route is similar to using a behavioural HDL and programming language route, perhaps VHDL plus Ada. However, the author believes that starting from the premise that one is not describing hardware, but rather describing computations, results in a different mind-set and a different approach to problems. In particular, it does seem to remove a large number of worries from the shoulders of the implementor, while substituting only a few, more easily managed problems.

We feel that the achievements of this particular project underline the ultimate realism of our goal to turn hardware/software implementation into just another smart compilation process, although many interesting challenges still remain.

9 Conclusions.

We hope that we have demonstrated the practicality of representing the hardware and software components of a system within a single framework. With this framework, implementations of programs can be generated which have amounts of software and hardware corresponding with desired cost-performance characteristics. We believe that this framework will eventually result in such designs being provably correct, and that moreover such proofs will be a natural side-effect of the compilation process.

Although the current hardware and software languages are actually different, they are very close in spirit. It is our intention to combine them both into a single language when time and our understanding permit. Our claim that this application has been implemented within a single framework relies on the fact that we are not over-concerned (at least for the purposes of this paper) with the differences between the languages nor in the details of the implementation of channel communication between the software and hardware. It is at least true that these differences do not overly concern the programmers who actually use the system.

Much exciting work remains to be done. For example: the design of languages for hardware-software co-design, the design and implementation of hardware and software compilers for the languages, optimisation of hardware implementations, asynchronous hardware implementations, automatic design and implementation of microprocessors, knowledge-driven and language-driven placement strategies for silicon implementations, new architectures for DPGAs.

It seems likely to us that the combination of high-level, programming-based approaches to hardware-software codesign together with reconfigurable hardware will fundamentally change the ways in which the digital systems of the future are designed and built. If true, this has far-reaching implications for the necessary skills basis that industry must adopt and nurture to be successful in the future. Hardware engineers, who already have an excellent understanding of the nature of parallelism, will have to become more programming-literate, and programmers will have to become more aware of parallelism and the time and space implications of the programs that they write.

Further details of our work can be found in [25, 34, 35, 23] together with other references given earlier. The world-wide web pages based at

`http://www.comlab.ox.ac.uk/oucl/hwcomp.html`

give access to some of the documents mentioned here and to other details of our hardware compilation work.

Acknowledgements.

I wish to thank Matthew Bowen for his implementation of the video tracking system, Robin Watts for his improvements to the hardware compilation environment, and Geoffrey Randall for producing the diagrams in this paper. I would also like to thank both the anonymous reviewer and Wayne Luk for some very useful comments. This work has been supported in part by the European Union (Esprit/OMI programme), the U.K. Engineering and Physical Sciences Research Council (EPSRC), Advanced Risc Machines Ltd. (ARM), Sharp Laboratories of Europe (SLE), Inmos Ltd. (SGS Thompson), Atmel Corp., Xilinx Development Corp., Hewlett Packard, Music Semiconductor, and Oxford University Computing Laboratory.

References

[1] C.A.R. Hoare, *Communicating Sequential Processes*, International Series in Computer Science, Prentice-Hall, 1985.

[2] Inmos, *The* occam2 *Programming Manual*, Prentice-Hall International, 1988.

[3] Erik Brunvard, *Translating Concurrent Communicating Programs into Asynchronous Circuits*, Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, September 1991.

[4] F Hanna, M Longley, and N Daeche, 'Formal systhesis of digital systems', in *Formal VLSI Specification and Synthesis*, ed., L Claesen, number I in VLSI Design Methods, pp. 153 – 169. Elsevier-Science, (1990).

[5] Patrice Bertin, Didier Roncin, and Jean Vuillemin, 'Introduction to programmable active memories', Technical report, DEC Paris Research Laboratory, (June 1989).

[6] Gerard Berry, 'A hardware implementation of pure esterel', Technical report, DEC Paris Research Laboratory, (July 1991).

[7] Peter Barrie, Paul Cockshot, George Milne, and Paul Shaw, 'Design and verification of a highly concurrent machine', *Microprocessors and Microsystems*, **16**(3), 115 – 124, (1992).

[8] Paul Cockshot, Paul Shaw, Peter Barrie, and George Milne", 'A scalable cellular array computer'. University of Strathclyde, Draft Report, December 1991.

[9] David Lewis, Mark van Ierssel, and Daniel Wong, 'A field programmable accelerator for compiled-code applications', (1993). Department of Electrical Engineering, University of Toronto.

[10] Sam Weber, Bard Bloom, and Geoffrey Brown, 'Compiling joy to silicon', (July 1991). Cornell University, submitted to POPL '91.

[11] Kees van Berkel, Joep Kesels, Marly Roncken, Ronald Saeijs, and Frits Schalij, 'The vlsi-programming language Tangram and its translation into handshake circuits', (1991). Philips Research Laboratories, Eindhoven, submitted to EDAC 91.

[12] David May and Catherine Keane, 'Compiling occam into silicon', in *Communicating Process Architecture*, Prentice Hall and Inmos, (1988).

[13] A. Wenban, J. O'Leary, and G.M. Brown, 'Codesign of communication protocols', *IEEE Computer*, **26**(12), 46–52, (Dec 1993).

[14] Wayne Luk, Teddy Wu, and Ian Page, 'Hardware-software codesign of multidimensional algorithms', in *FPGAs for Custom Computing Machines*, IEEE, (1994).

[15] Wayne Luk and Teddy Wu, 'Towards a declarative framework for hardware-software codesign', in *Proc. Third International Workshop on Hardware/Software Codesign*, 181–188, IEEE Computer Society Press, (1994).

[16] A.W. Roscoe and C.A.R. Hoare, 'Laws of occam programming', *Theoretical Computer Science*, **60**, 177–229, (1988).

[17] Michael Spivey and Ian Page, 'How to program in handel', Technical report, see http://www.comlab.ox.ac.uk/oucl/hwcomp.html, Oxford University Computing Laboratory, (1993).

[18] Laurence Paulson, *ML for the Working Programmer*, CUP, 1991.

[19] G. Brown, W. Luk, and J. O'Leary, 'Retargeting a hardware compiler proof using protocol converters', *Formal Aspects of Computing*, (to appear).

[20] J.P. Bowen and He Jifeng, 'Programs to hardware', in *Tutorial Material, Formal Methods Europe '93, Industrial-Strength Formal Methods*, ed., P.G. Larsen, pp. 437–450, (1993). In A.P. Ravn (ed.), *Provably Correct Systems (ProCoS)* tutorial.

[21] Jifeng He, Ian Page, and Jonathan Bowen, 'Towards a provably correct hardware implementation of Occam', in *Correct Hardware Design and Verification Methods, Proc. IFIP WG10.2 Advanced Research Working Conference, CHARME'93*, eds., G.J. Milne and L. Pierre, volume 683 of *Lecture notes in Computer Science*, pp. 214–225. Springer-Verlag, (1993).

[22] Jonathan Bowen, Jifeng He, and Ian Page, 'Hardware compilation', in *Towards Verified Systems*, ed., J.P. Bowen, Real-time Safety-Critical Systems, chapter 10, 193–207, Elsevier, (1994).

[23] Ian Page, 'Automatic design and implementation of microprocessors', in *Proceedings of WoTUG-17*, pp. 190–204, Amsterdam, (April 1994). IOS Press. ISBN 90-5199-1630.

[24] Wayne Luk, David Ferguson, and Ian Page, 'Structured hardware compilation of parallel programs', in *More FPGAs*, eds., Will Moore and Wayne Luk, Abingdon EE&CS books, (1994).

[25] C.A.R. Hoare and Ian Page, 'Hardware and software : The closing gap', *Transputer Communications*, **2(2)**, 69–90, (June 1994).

[26] C.A.R. Hoare, 'Refinement algebra proves correctness of compiling specifications', in *3rd Refinement Workshop*, eds., C.C. Morgan and J.C.P. Woodcock, Workshops in Computer Science, pp. 33–48. Springer–Verlag, (1991).

[27] C.A.R. Hoare and He Jifeng, 'Refinement algebra proves correctness of a compiler', in *Programming and Mathematical Method: International Summer School directed by F.L. Bauer, M. Broy, E.W. Dijkstra, C.A.R. Hoare*, ed., M. Broy, volume 88 of *NATO ASI Series F: Computer and Systems Sciences*, 245–269, Springer–Verlag, (1992).

[28] J.P. Bowen, He Jifeng, and P.K. Pandya, 'An approach to verifiable compiling specification and prototyping', in *Programming Language Implementation and Logic Programming (PLILP'90)*, eds., P. Deransart and J. Małuszyński, volume 456 of *Lecture Notes in Computer Science*, pp. 45–59. Springer–Verlag, (1990).

[29] Adrian Lawrence, Andrew Kay, Wayne Luk, Toshio Nomura, and Ian Page, 'Using reconfigurable hardware to speed up product development and performance', in *FPL95*, eds., W. Luk and W. Moore, Lecture Notes in Computer Science. Springer Verlag, (1995).

[30] Sundance Multiprocessor Technology Ltd., 4 Market Square, Amersham, Bucks HP7 0DQ, U.K., *Product Overview*, 1995.

[31] Xilinx, San Jose, CA 95124, *The Programmable Gate Array Data Book (1993).*

[32] Inmos, *The Transputer Development and iq systems Databook*, Inmos Ltd., 1991.

[33] Matthew Bowen, 'Video motion tracking', Undergraduate project report, see http://www.comlab.ox.ac.uk/oucl/hwcomp.html, Oxford University Computing Laboratory, (1994).

[34] Ian Page and Wayne Luk, 'Compiling occam into FPGAs', in *FPGAs*, eds., Will Moore and Wayne Luk, 271–283, Abingdon EE&CS books, (1991).

[35] Ian Page, Wayne Luk, and Henry Lau, 'Hardware compilation for FPGAs: Imperative and declarative approaches for a robotics interface', in *Proc. IEE Colloquium on Field-Programmable Gate Arrays – Technology and Applications, Ref. 1993/037*, pp. 9.1–9.4. IEE, (1993).