Interface Specification for Reconfigurable Components

Satnam Singh
Xilinx Inc, 2100 Logic Drive, San Jose, California 95124, USA
Satnam.Singh@xilinx.com

Abstract

This paper presents a way of encoding some kinds of dynamic reconfiguration behaviour in the interface portion of circuit descriptions. This has many advantages. The user of a reconfigurable circuit has some knowledge about the reconfigurable interface of the circuit. Static analysis tools can make better decisions about how to schedule virtual hardware. And most importantly the compiler can automatically synthesize the required interface between reconfigurable portions of the system and the regular portions of the design. Several existing models of dynamic reconfiguration from the literature are captured using our type system extension based on sum types. This is especially important in System-on-Chip (SoC) contexts where a reconfigurable IP block may have to communicate over a non-trivial IP bus like CoreConnect™.

1. Introduction

The primary contribution of this paper is a scheme for describing the types of reconfigurable components which can be realised on reconfigurable platforms like Xilinx's Virtex™-II PRO. Such architectures make it feasible to produce hot-swappable circuit components which can be dynamically attached to and removed from the system and peripheral buses (CoreConnect in the case of Virtex-II PRO). Supporting for describing reconfiguration behavior is required because in order to harness the power of dynamic reconfiguration application developers need high level abstractions for describing reconfigurable circuits and systems. Reconfigurable circuits can change type as they execute and this fact is not adequately captured using existing type systems and interface descriptions. By enhancing the types of reconfigurable circuits with information about their reconfiguration behaviour, high level compilers can exploit this information to perform sophisticated optimizations and to generate code to automate the reconfiguration process e.g. perform scheduling. It is widely accepted that one of the main barriers to the acceptance of reconfigurable computing is the lack of high level tools and languages that support this technique. The work presented here tries to close the semantic gap between painstakingly hand-coded reconfiguration schemes and high level circuit design.

Types have often been neglected in the design of many hardware description languages and it is notable that the type systems of modern programming languages are more powerful and flexible than the type systems of mainstream hardware description languages. This is unfortunate because types can capture important information about the interface and behaviour of reconfigurable circuits. In the software community researchers have even been able to use type checking to ensure run-time security properties in mobile computing environments [5]. It should be possible for dynamically reconfigurable circuits to enjoy some of the same advantages using advanced type systems.

The type system presented here is designed to describe *interacting reconfigurable circuits*. The ability to describe and safely compose interacting reconfigurable circuits allows us to invent new abstractions for dynamic reconfiguration behaviour which would otherwise have been too difficult to model satisfactorily with conventional type systems.

2. Interface Representation

Asked where information about reconfiguration behaviour should be expressed, many would advocate the use of ad hoc annotations to the code, pragmas or special language constructs. All of these techniques have been shown to work but they often result in cluttered code which is hard to read and modify and specialised to a particular mode of operation. Furthermore, by looking at the interface of the circuit one cannot deduce anything about the reconfiguration behaviour of the circuit.

The principal argument of this paper is that reconfiguration behaviour is so important that it should be expressed in the circuit interface. In hardware description languages the circuit interface includes the types of the signals at the interface of cells. We propose that the types say something about how circuits are reconfigured.

We take further the notion of just using types to describe reconfiguration behaviour by showing how types can be used to constrain reconfiguration behaviour of composite circuits. This is accomplished by supplying a type more specific than the most general type.

The remainder of this paper describes in an informal manner some extensions to an existing type system used in the Lava HDL. Why not choose the type systems of VHDL [11] or Verilog? First, the Lava HDL has been designed to directly support the expression of dynamically reconfigurable circuits. However, its type system has been wholly inherited from the Haskell programming language without any regard for the expression of dynamic reconfiguration. Second, it is virtually impossible to express dynamic reconfiguration in VHDL or Verilog and these languages have inflexible and unclean type systems which are not easy to subject to modification and experimentation.

Although the type experiments presented here are expressed using the Lava type system we emphasise that the results of this research can be applied to the development of mainstream hardware description languages and emerging systems like JHDL [2][10] and Pebble [14]. This is where the practical benefits of the largely theoretical results of this paper lie.

0-7803-7607-2/02/$17.00 ©2002 IEEE

The exploration and experimentation described here is guided by the desire to describe a useful subset of coarse-grained reconfigurations by introducing as few extra concepts as possible. We are guided by the restriction which views reconfiguration as a dynamic choice between circuits *which may have different types*. This naturally leads us to a type system which has the notion of choice built-in. We build upon this one extra type feature to describe surprisingly wide and useful modes of dynamic reconfiguration.

Throughout this paper when we write "R :: t" this should be understood to be an abbreviation for "circuit R has type t." Types can be concrete e.g. Int8 for an eight bit integer, Bit for a bit type or they can be type variables which allow us to type very general circuits sing *polymorphic* type checking. Concrete types start with a capital letter and polymorphic types start with a lower case letter. For example here is the type of a circuit which swaps its inputs:

```
swap :: (a, b) -> (b, a)
```

This circuit can be called with a pair containing elements of any type. The arrows in the type separate lists of argument types and the final type expression gives the type of the result. For example a circuit take takes as input a clock signal and an eight bit integer and returns a bit value could be given the following type:

```
serialize :: Bit -> Int8 -> Bit
```

This way of writing types allows us to partially apply arguments which results in *residual* circuits. This feature can be exploited to express run-time circuit specialization.

3. Reconfiguration and Time

The type scheme presented in this paper assumes a particular style of reconfiguration where the clocking of sub-circuits can be temporarily suspended for several reasons. This is easily established with clock-enable inputs to registers. One reason for doing this is to hold up logic while some portion of the FPGA is reconfigured. Another reason is to hold up a circuit while it waits for a shared resource like a bus. Thus, when we talk about clocks and ticks and temporal sequences it should be understood that it is the "enabled-clock" that is being referred to. This allows us to abstract away from reconfiguration time and the time it takes to acquire shared resources to give a tractable type system for describing dynamic reconfiguration.

4. Type Extensions

4.1 Dynamic Replacement

One of the most basic forms of reconfiguration that occurs is to replace a circuit A :: t1 -> t2 by another circuit with a totally different type B :: t3 -> t4. This occurs whenever an FPGA is totally reprogrammed with a different circuit. In the context of reconfigurable systems, this usually occurs to a portion of the FPGA fabric, leaving other parts of the system intact and operational. We relate the circuits A and B by composing them to form a "reconfigurable circuit" R which can at any given moment be circuit A or circuit B.

We allow nested reconfigurable circuits so A can be either a static or a reconfigurable circuit.

Exactly how the definition of R is written depends on the particular reconfigurable mechanism or abstraction used and later we provide an example from the literature that fits this model. For the moment we concentrate on the question of R's *type*. At any given moment R has either type t1 -> t2 or t3 -> t4. We propose that the unconstrained type of R should be *both* t1 -> t2 and t3 -> t4. We form a sum of the two types and express this sum type using the + operator:

```
R :: (t1 -> t2) + (t3 -> t4)
```

A user of the circuit R can now immediately tell without looking at the definition of R that this is a reconfigurable circuit. In a realistic scenario, R may have some control inputs which would help to determine which particular type R has at any given moment. An example of dynamic replacement is shown in Figure 1 which shows the same block of reconfigurable fabric being configured to circuits A, B, B, A at successive enabled clock ticks.

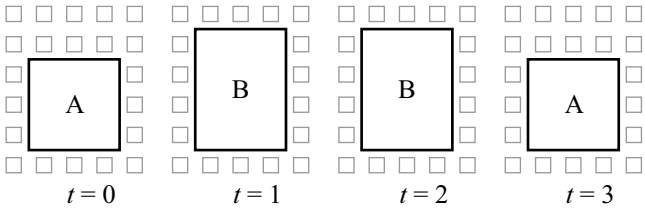

Figure 1. An example of dynamic replacement.

A special case of dynamic replacement occurs when all the circuits in the composition have the same type e.g. t1 -> t2. For such a circuit we could choose a general type like t1 -> t2. However, this does not capture the fact that the circuit described can be reconfigured between two circuits which happen to have the same type. For such circuits we choose the more specific type (t1 -> t2) + (t1 -> t2). A compiler can now examine the body of such a reconfigurable circuit and ensure that the two alternative implementations do indeed have the same type. The fact that a circuit can have two valid types which have no common canonical representation leads to some problems described later in this paper.

4.2 Composition of Dynamic Replacement

When used without any other contextual information, the circuit R presented earlier is free to assume either the type of circuit A or circuit B. However, when R is composed with another dynamically reconfigurable circuit or a static circuit the types interact in an interesting manner.

Circuit composition in the paper is modelled using the serial composition combinator in Lava which connects the output of one circuit to the input of another circuit. However the role of serial composition appears implicitly in other hardware description languages e.g. implicitly using names to wire together nets as is the case in structural VHDL. For this reason the typing scheme we present here can be ported to other languages taking account of the

particular circuit composition mechanisms provided by that language.

The connecting wires for serial composition should have compatible types under unification. This is illustrated in Figure 2 for a very simple composition of two static circuits. The composite NAND gate has a legal type because the result type of the AND gate (Bit) is compatible with the input type of the invertor (also Bit).

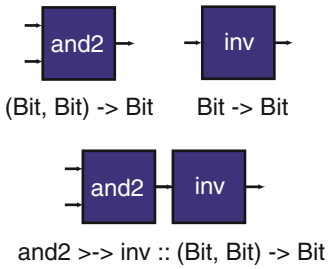

Figure 2. Typing serial composition

Consider first two static circuits P :: t2 -> t5 and Q :: t4 -> t6. What is the type of the Lava serial composition R >-> P? In this case R is no longer free to assume type (t1 -> t2) or (t3 -> t4) at any given moment because at all moments circuit P is expecting an input of type t2. Thus it is possible to use our type system to calculate that the type of R >-> P :: t1 -> t5. This means that we no longer have a reconfigurable circuit and static analysis by the compiler can prune away unwanted circuitry for this composition. Similarly the type of R >-> Q :: t3 -> t6.

Now consider another dynamically reconfigurable circuit with type S :: (t2 -> t7) + (t4 -> t8). The composition R >-> S should have a type which expresses all the legally typed combinations of the permutations of possible types. In this case there are only two legal alternative i.e. t1 -> t7 and t3 -> t8 so the type of R >-> S becomes (t1 -> t7) + (t3 -> t8).

In general the composite reconfigurable type can be calculated by forming the sum type of a cross product of all types of the two constituent circuit types *A* and *B* and then discarding the pairs that can't be composed together with type unification:

$$\sum \{ t_a \rightarrow t_d \,|\, (t_a \rightarrow t_b, t_c \rightarrow t_d) \in A \times B, unifybc \}$$

It is not always the case that when two reconfigurable circuits are composed that the resulting circuit is reconfigurable. Consider another reconfigurable circuit T :: (t2-> t7) + (t1 -> t5). The composition R >-> T has only one legal type i.e. t1 -> t7 which means that R >->T :: t1 -> t7 which is no longer a reconfigurable circuit.

Is it useful to have a type system which can compose reconfigurable circuits to yield non-reconfigurable circuits? We argue that it is important because it allows the design of general circuits like R and their use in specific circumstances (e.g. when only A is required) which can be optimised by removing redundant logic. Why not use A directly? This is because R provides an abstraction for the task performed by A. The performance of R can be modified by replacing A with a better circuit that has the same interface. Other circuits that use this functionality indirectly via R automatically inherit the benefits of the improved A.

4.3 Dynamic Selection

Dynamic replacement expresses the notion that a circuit can be replaced by another circuit of a different type but says nothing about the circumstances under which this occurs (this is expressed in the circuit definition). Dynamic selection describes reconfiguration that is controlled by a specific signal, rather like the control input to a multiplexor.

Such circuits are given a *dependent type*. Dependent types allow the type of an expression to depend on the value of one of the inputs. Dependent types have been successfully used in languages like Cayenne by Augustsson [1].

Consider a reconfigurable circuit U which takes a selection input sel :: Bool which determines which of two circuits A or B (as previously defines) is instantiated. One valid type for this circuit could be:

```
U :: (Bool -> t1 -> t2) + (Bool -> t3 -> t4)
```

which is the type that can be computed if we know nothing about how U is reconfigured. However if we know that sel selects either A or B (implemented by dynamic reconfiguration rather than a multiplexor) then we can deduce a dependent type for U that captures this information:

```
U :: sel::Bool ->  sel == True => (t1 -> t2) +
                   sel == False => (t3 -> t4)
```

Here, each of the reconfigurable type components have a guard which indicates the type of U for a given value of sel. For example the type of the expression (U True) :: t1 -> t2 and the type of the expression (U False) :: t3 -> t4.

Dynamic selection can be used to model the RC_MUX dynamic reconfiguration abstraction proposed by Luk et. al reported in [12] and to a lesser extent in [13]. The dynamic selection scheme we present here is a little more general than Luk's RC_MUX mechanism, which requires all the alternatives to have inputs of the same type and an output of the same type (at the bit level) as shown in Figure 3. Our scheme can describe dynamic selection between circuits having totally different types.

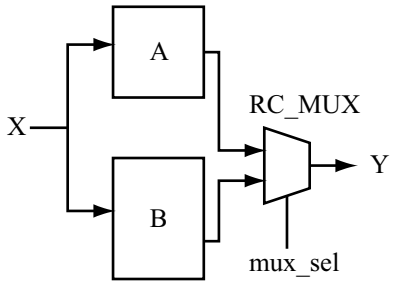

Figure 3. The RC_MUX dynamic reconfiguration abstraction

The design of Flexible Array Blocks (FABs) [9] and Reduced Flexible Array Blocks (RFABs) [18] can be expressed using dynamic selection because the reconfiguration behaviour of these circuits is controlled by four configuration bits, which are inputs to the blocks and essentially effect a dynamic selection.

It may be tempting to apply the dynamic selection type to the active 4x4 ATM switch described by Dollas et. al. [6] since the control inputs to the switch effectively drive multiplexors which effect selection. However this circuit does not have a dynamic reconfiguration type because the selection occurs between static circuits (not by reconfiguration) so this circuit can be adequately typed by the base type system (or regular type systems of languages like VHDL or Verilog).

4.4 Temporal Interleaving

Sometimes a reconfiguration occurs in a specific sequence e.g. alternating A, B, A, B, A, B... relative to some clock signal as shown in Figure 4.

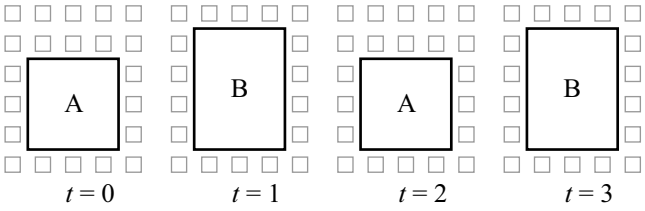

Figure 4. Temporal interleaving

Such behaviour is captured by specifying a sequence that can be repeated any number of times (including zero):

V :: clk:{t1 -> t2, t3 -> t4}+

Valid execution traces of the types of the reconfigurable circuit V include {}, {t1 -> t2}, {t1 -> t2, t3 -> t4}, {t1 -> t2, t3 -> t4, t1 -> t2} relative to a clock signal called clk.

The execution type traces of V are equivalent to the execution type trace of U where the control signal is a clock pulse clk i.e. U clk. The clk control signal switches between the two alternatives giving arise to the same sequence. This observation allows us to implement the temporal interleaving type construct by translation into the dependent type mechanism for dynamic switching.

We illustrate how temporal interleaving can be viewed as 'syntactic sugar' i.e. a new construct that can be re-expressed in terms of existing language constructs (rather like a macro). Here is the rule for translating a two-element sequence:

```
clk:{t1 -> t2, t3 -> t4}+ ⟹
  clk::Bool ->   tick clk 'mod' 2 == 0 => (t1 -> t2) +
                 tick clk 'mod' 2 == 1 => (t3 -> t4)
```

where the symbol ⟹ means "is implemented by translation into". This type definition makes use of the function tick which returns the sequence number of an alternating signal i.e. it counts ticks on the rising edge. This allows us to provide syntactic sugar for larger sequences e.g.

```
clk:{t1 -> t2, t3 -> t4, t5 -> t6}+ ⟹
  clk::Bool ->   tick clk 'mod' 3 == 0 => (t1 -> t2) +
                 tick clk 'mod' 3 == 1 => (t3 -> t4) +
                 tick clk 'mod' 3 == 2 => (t5 -> t6)
```

Compiler tools can exploit the information given in the type about the sequencing of reconfigurable circuit to implement optimizations. They can also help check that the definition body does indeed build a circuit in a way that is faithful to the given type and thus capturing some design errors at compile time. The type also gives the user of a library of reconfigurable intellectual property some extra valuable information about the reconfiguration behaviour of a given circuit.

4.5 Dynamic Specialization

One optimization technique available to reconfigurable circuits, which is not available in static implementations, is the ability to dynamically specialize a circuit when one input varies far less frequently than another input. For example, consider a multiplier where one input changes every 10,000th tick but the other input changes every tick. This circuit can be optimised by dynamically computing a new constant coefficient multiplier circuit (or swapping one in) every 10,000th ticks. Such on-line techniques have been implemented in the Lava system, Luk's group [13] and others [15]. There are also off-line variants too Figure 20. Such techniques can be used to epitomise reconfigurable intellectual property cores at run-time using suitable verification technology.

Consider the example of a general multiplier circuit in the left hand side of Figure 5. At time *t*=0 the circuit specializes on the circuit input 13 to produce a constant coefficient multiplier which is smaller (and ignores the second input). This multiplier is used until tick 356 when the second input changes to 17 (causing the enabled clock to the circuit to be held up while a specialized circuit is computed or swapped in) and the fabric is reconfigured with a different constant coefficient multiplier.

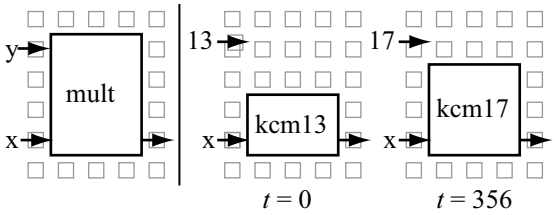

Figure 5. Dynamic specialisation

A first attempt at computing a type for such a circuit might produce something like:

```
mult :: ((Int8, Int8) -> Int16) + (Int8 -> Int16)
```

This type reflects the fact that this circuit, at any given moment, can have the type of a general multiplier `(Int8, Int8) -> Int16` or the specialized constant coefficient multiplier `Int8 -> Int8`. However, externally this circuit always has the type of a general multiplier which suggests the type:

```
mult :: (Int8, Int8) -> Int16
```

but this says nothing about the reconfigurable behaviour of the circuit. Somehow we would like to express the fact this circuit has a family of types of all the possible reconfigurations:

```
mult :: (0, Int8) -> Int16 +
        (1, Int8) -> Int16 +
        (2, Int8) -> Int16 +
        ...
        (255, Int8) -> Int16
```

where the types of constants are denoted by the constants themselves. The solution we adopt is to nominate the specializing input explicitly in the type. This can be seen as syntactic sugar since the corresponding sum type can always be evaluated for each type that is finitely enumerable (and only such types are valid for representing signals). The specialising input is enclosed in angle brackets:

```
mult :: (Int8, <Int8>) -> Int16
```

To make the translation tractable, we impose the restriction that only one specialisation can be specified in a single type signature. Specialization on multiple inputs can be expressed by nesting definitions. Furthermore, we restrict the nomination of specialization signals to primitive types rather than composite types. This helps to ensure we always have a finite translation into the underlying ‘sum of types’ feature used to implement the dynamic specialization type. It is satisfying that we can describe dynamic specialization without having to introduce any special type notions beyond reconfigurable type sums.

4.6 Reconfigurable Interconnect

One important class of reconfiguration works by re-routing wires. In a static circuit such functionality is established by using multiplexors to route signals depending on some control inputs. In a reconfigurable circuit the fabric is reprogrammed when the control inputs change to establish a direct connection. Of course in a particular fabric such “hard wires” may still be implemented by wires that pass through multiplexors but at the user’s abstraction level such wires appear as a direct connection (which may be routable on a faster and longer wire than the equivalent multiplexor based implementation).

In conventional hardware description languages wires (or nets) are treated separately from “circuits” like AND gates and registers and the operations available on circuits and nets are quite different. Often one has far less flexibility to manipulate wires. We argue that wiring circuits is as valid a circuit as an AND gate! This lets us treat wiring circuits as first class objects (just like regular circuits) and wiring expressions have types (and reconfigurable types) just like regular circuits. By removing the unnecessary special treatment of nets in hardware description languages we end up with a simpler and more powerful and systematic treatment of circuits in general.

A consequence of treating wiring as regular circuits is that the reconfigurable type schemes presented so far are equally applicable to writing circuits and reconfigurable interconnect. At the most abstract level the type of reconfigurable interconnect is calculated by taking the sum type of all the possible interconnections supported by a particular interconnect.

In specific applications the reconfigurable interconnect can be analysed to calculate a more specific type. For example if the reconfigurable interconnect is used to “hard wire” a general datapath to perform a particular calculation then the type of the hardwired circuit is obtained by analysing the types of the components of the data-path. This type of reconfiguration had been used to specialize ALU-type data-paths.

4.7 Reconfigurable Components on Buses

Normally when two circuits are composed the connection is implemented by dedicated wires that directly connect together the ports of the two circuits. Another important type of communication that occurs between circuits is via a bus which does not provide a dedicated communication channel but which is instead shared with other circuits by arbitration. Such bus structures are likely to be common in high-end reprogrammable architectures. Any high level scheme for describing dynamic reconfiguration should address how dynamically reconfigurable circuits compose with other circuits over a bus structure rather than dedicated wires.

For the purposes of this paper we abstract away the details of the operation of a specific bus architecture. The key features we acknowledge is the ability to send and receive data over the bus where there is a possibly infinite delay before a (language level) data item can be sent and a possibly infinite delay before a data item can be received.

A system may have more than one type of bus, each with its own protocol for communication. For example IBM’s CoreConnect IP bus actually has three bus components with different protocols. We model a particular type of bus protocols with type constructors which takes as parameters the type of data being communicated. For example a circuit `M` which has an 8-bit integer input and which

sends a 16-bit integer on the CoreConnect processor local bus (PLB) would have the following type:

```
M :: Int8 -> PLB Int16
```

Consider another peripheral circuit N which accepts a 16-bit integer over the peripheral bus called the OPB and has a single bit output:

```
N :: OPB Int16 -> Bit
```

Now when we compose these two circuits using serial composition i.e. M >-> N we would hope the composite circuit will type check even though the output type of one circuit is not the same as the input type of the other circuit. The underlying data-item is type consistent i.e, is an Int16. What is different is how the data-item is communicated. We can exploit the existing class system by using systematic overloading to define an M and N that can communicate over any set of buses for which a conversion protocol is known. Assuming that a class called CoreConnect has been defined with methods for the buses PLB and OPB to send and receive data, we can give more suitable types to M and N:

```
M :: CoreConnect bus => Int8 -> bus Int16
N :: CoreConnect bus => bus Int16 -> Bit
```

We have now abstracted away the particular bus used in the communication. We can use any bus that belongs to the CoreConnect class. Similar techniques can be employed in object-oriented languages like C++ and Java.

We can now build upon this type scheme for bus-based communication to describe the communication between reconfigurable circuits. Consider a reconfigurable circuit K that can send either Int8 or Int16 over a bus depending on how it is reconfigured. Similarly imagine a reconfigurable circuit L that can receive either Int8 or Int16 over a bus depending on how it is reconfigured. The circuit K has a direct (non-bus) input type of either Int3 or Int4. The circuit L has a direct output type of either Int5 or Int 6. The types of the two circuits are:

```
K :: CoreConnect bus =>
     (Int3 -> bus Int8) + (Int4 -> bus Int16)
L :: CoreConnect bus =>
     (bus Int8 -> Int5) + (bus Int16 -> Int6)
```

Now what is the type of the composition of K and L i.e. K >-> L? We can systematically apply the rules given earlier for our type scheme without introducing any new notions and we get a reasonable type for the composition:

```
K >-> L :: (Int3 -> Int5) + (Int4 -> Int6)
```

The composite circuit does not connect to any buses on its ports as this is nicely reflected in the type that has been calculated.

The mechanism for describing loosely coupled communication over buses also extends to the typing of channel based communication that is referred to as *task-parallel programming of reconfigurable systems* by Weinhardt and Luk [19]. The types abstract away the exact nature of the communication i.e. shared or distributed memories. In this case one type system is used to describe both the hardware *and* software and this can help to check the integrity of the co-designed system. However the C-based language used in the task-parallel research is not easily amenable to the type treatment here. To benefit from our type scheme the task-parallel scheme would need to be translated into a Lava-esque framework.

The typing for bus-based communication here does not capture the full range of communication behaviour available over a bus. The type system describes the type of information that flows between two circuits connected to the bus, but it does not describe the ability of a master circuit on the bus to communicate with any other master or slave circuit connected to the bus. This type of communication would require either a further extension to the type system or the formation of a sum type that collects together the types of all the circuits on the bus with which the master may wish to communicate.

5. Related Work

The author is not aware of any substantial existing work which tries to exploit the power of types to describe dynamic reconfiguration. Few languages provide high level support for expressing dynamic reconfiguration. Lava is a notable exception and is best placed to benefit immediately from the work described here.

For types, the closest related work is Cayenne language, which is the first programming language to implement a dependent type system. Originally dependent types were mainly used in proof systems. We expect the development of experimental hardware description languages and their type systems to benefit from the on-going work in the theory of type systems.

Many reconfigurable architectures come with specialised languages which are often C-based, making them difficult for adapt for strong typing. The single assignment language DIL used for pipelined reconfigurable fabrics [4] like RaPiD [7][8] has foundations that make it suitable for a functional-style type system although it has been cast in a C style.

6. Limitations and Future Work

One problem with the type system presented here is that it is not *decidable*. Given two arbitrary type expressions, it is undecidable whether they mean the same thing because we no longer have a canonical (or normal) form for type expressions (as was the case with the base type system that we started with).

In practice, languages with undecidable type systems can be made to work using suitable approximations and by “timing out” the type checking algorithm if it takes too long. Examples of such languages include Cayenne, Quest and Gofer. As future work, researchers may wish to seek out a set of extensions that do have normal forms.

The type system presented here does not describe all the possible types of reconfiguration in a convenient manner. In the most extreme case dynamic reconfiguration can involve making any arbitrary change to a circuit. In the general case we cannot discern any kind of structure or method that can be captured by a meaningful type. General reconfiguration can be described by the system

presented here using an infinite type, which is the sum of all possible circuits, but this is hardly useful in practice (this corresponds to an untyped circuit).

However there are many important and useful types of reconfiguration abstractions presented in the literature and the work here has attempted to relate the types of several different reconfiguration techniques using a single scheme. This discipline also helps to compare and contrast different types of reconfiguration in a systematic way.

Wormhole routing as described by Bittner and Athanas [3] is a useful reconfiguration technique which is not easily captured by our type system. This is because the nature of the reconfiguration is not naturally expressible as a choice between alternatives. However we speculate that dependent types can be exploited yet again to produce a system for describing wormhole reconfiguration (as supporting by the Colt architecture) because the way a unit is reconfigured depends on data which is passed along in a stream which appears as a data input to the circuit. Since a data item determines the type (after reconfiguration) then dependent types should be powerful enough to capture this mechanism. Our type scheme does not present a suitable way to *compose* such types.

The type scheme presented here is statically checkable to ensure that at run-time there are no type errors. Although this restriction buys security, the cost is that some interesting circuits which have dynamically-typed behaviour cannot be type checked and are rejected by our system. We make this choice because in general we can not assume that there is a run-time type checker available to check the communication between reconfigurable circuits. In the special case where there is a closely coupled microprocessor available or when it is possible to compile the dynamic type checking algorithm into gates, it may be interesting to explore dynamically typed extensions to the type scheme that we have presented here. However, by using dependent types we have realised some of the advantages of dynamic types whilst still retaining static type checking and run-time type security.

7. Conclusions

We have described several theoretical experiments here which show how powerful type systems can be applied to the description of certain kinds of reconfigurable circuits. These concepts can be re-applied to the design of mainstream languages that need to be modified to support high level language constructs for expressing dynamic reconfiguration. This in turn is essential if reconfigurable applications are to emerge from the university and research lab ghetto and gain acceptance by non-specialists in dynamic reconfiguration.

One useful aspect of the way we have designed the type extensions presented here is that it is always apparent if a circuit is reconfigurable (or contains reconfigurable components). This is established by the presence of at least one “+’ symbol in the type or angle brackets. This helps to identify which portions of a circuit are static and which portions are reconfigurable.

At the heart of our characterisation is the observation of choice. For static circuits the choice is over *space* (e.g. selecting via a multiplexor between two circuits). For reconfigurable circuits the choice is over *time* (e.g. swapping one circuit out and replacing it by another circuit).

This work has tried to add as little as possible to an existing type scheme and then tried to see just how many different types of dynamic reconfiguration we can describe and classify. Adding the notion of a reconfiguration sum type allows us to express many useful kinds of reconfiguration including dynamic replacement, dynamic selection, dynamic specialization, and loosely-coupled communication between reconfigurable components.

An implemention of a system that supports the reconfiguration interface specification method presented here is under construction. This system will allow us to evaluate how feasible this method is for specifying large reconfigurable platform-based systems. A key benefit over existing systems will be the ability to perform interface synthesis by generating apporpriate bus-interface logic for reconfigurable components.

There are many types of reconfiguration that are not amenable to a satisfactory description with our scheme. Much more work needs to be done to help use types to check and compose dynamically reconfigurable systems. This paper represents the first steps in that direction.

References

[1] Lennart Augustsson. *Cayenne: a language with dependent types.* In Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, pages 239-250, 1998.

[2] Peter Bellows and Brad Hutchings. *JHDL - An HDL for Reconfigurable Systems*. FCCM’98. IEEE Computer Society. 1998.

[3] R.A. Bittner and P.M. Athanas, *Wormhole Run-time Reconfiguration*. In Proc. 5th Int'l. Symp. on Field Programmable Gate Arrays, Monterey, California, 1997.

[4] Mihai Budiu and Seth Copen Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. FPGA’99. ACM Press. 1999.

[5] Luca Cardelli and Andrew Gordon. *Types for mobile ambients*. In Conference Record of the ACM Symposium on Principles of Programming Languages, San Antonio, January 1999. ACM Press. 1999.

[6] Apostolos Dollas, Dionisios Pnevmatikatos, Nikolaos Aslanides, Stamatios Kavvadias, Euripides Sotiriades, Sotirios Zogopoulos, Kyprianos Papdemetriou, Nikolaos Chrysos, Konstantinos Harteros, Emanouil Antonidakis, Nikolaos Petrakis. *Architecture and Application of PLATO, a Reconfigurable Active Network Platform*. FCCM’01. IEEE Computer Society. 2001.

[7] C. Ebeling, D. C. Cronquist and P. Franklin. *RaPiD - reconfigurable pipelined datapath*. In the 6th International Workshop on Field-Programmable Logic and Compilers. LNCS. Springer Verlag. 1996.

[8] C. Ebeling, D. C. Cronquist, P. Franklin and S. Berg. *Mapping applications to the rapid configurable architecture*. FCCM’97. IEEE Computer Society. 1997.

[9] Simon D. Haynes and Peter Y. K. Cheung. *A Reconfigurable Multiplier Array For Video Image Processing Tasks, Suitable*

for Embedding In An FPGA Structure. FCCM'98. IEEE Computer Society. 1998.

[10] Brad Hutchings, Peter Bellows, Joseph Hawkings, Scott Hemmert, Brent Nelson, Mike Rytting. *A CAD Suite for High-Performance FPGA Design*. FCCM'99. IEEE Computer Society. 1999.

[11] IEEE Std. 1076-1987. *IEEE Standard VHDL Reference Manual*. 1997.IEEE Computer Society. 1999

[12] Wayne Luk, Nabeel Shirzi and Peter Y. K. Cheung. *Modelling and Optimising Run-Time Reconfigurable Systems. IEEE Symposium on FPGAs for Custom Computing Machines '96*. Eds. K.L. Pocek and J.M. Arnold IEEE Computer Society Press, 1996.

[13] Wayne Luk, Nabeel Shirazi and Peter Y. K. Cheung. Compilation Tools for Run-Time Reconfigurable Designs. FCCM'97. IEEE Computer Society. 1997.

[14] W. Luk and S. McKeever. *Pebble: a language for parameterised and reconfigurable hardware design. Field-Programmable Logic and Applications*. LNCS 1482. Springer-Verlag. 1998.

[15] John MacBeth and Patrick Lysaght. *Dynamically Reconfigurable Cores*. FPL 2001. LNCS 2147. Springer-Verlag. 2001.

[16] S. McMillan and S. Guccione. *Partial Run-Time Reconfiguration Using JRTR*. Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications, LNCS 1896, 2000.

[17] Herman Schmit. *Incremental Reconfiguration for Pipelined Applications*. FCCM'97. IEEE Computer Society. 1997.

[18] Chakkapas Visavakul, Peter Y. K. Cheung, Wayne Luk. A Digit-Serial Structure for Reconfigurable Multipliers. Field-Programmable Logic and Applications. Belfast, UK. Springer-Verlag. 2001.

[19] Markus Weinhardt and Wayne Luk. *Task-Parallel Programming of Reconfigurable Systems*. Field-Programmable Logic and Applications. Belfast, UK. Springer-Verlag. 2001

[20] Michael J. Wirthlin and Brian McMurtrey. Efficient Constant Coefficient Multiplication Using Advanced FPGA Architectures. Field-Programmable Logic and Applications. Belfast, UK. Springer-Verlag. 2001.