First Implementation Set: "Hardware"

Project Deliverable D2.2

Christiaan Baaij [UT]; Jan Kuper [UT]; Javad Zarrin [IT]; Lutz Schubert [HLRS]; Vincent Gramoli [EPFL]; Tommaso Cucinotta [SSSA]; Daniel Rubio Bonilla [HLRS]

Due date: 30/04/2011
Delivery date: 30/04/2011

This work is partially funded by the European Commission under
FP7-ICT-2009.8.1, GA no. 248465

Version History

Version	Date	Change	Author
0.1	06/02/11	First TOC	Christiaan Baaij
0.2	08/02/11	Initial Content (from IR2.1 and wiki)	All
0.3	25/02/11	Refined content, explanatory text	All
0.4	05/03/11	Updated structure	Christiaan Baaij
0.3	18/03/11	Update test on resource descriptions for discovery	Javad Zarrin
0.4	21/03/11	Add section on CλaSH	Christiaan Baaij
0.5	21/03/11	Add cpu description example	Christiaan Baaij
0.6	24/03/11	Add introduction	Christiaan Baaij
0.7	25/03/11	Revise Chapter on MCoreSim, start on section about hardware synthesis	Tommaso Cucinotta, Christiaan Baaij
0.8	28/03/11	Extend section on hardware synthesis and Update section on resource descriptions for discovery	Christiaan Baaij, Javad Zarrin
0.9	29/03/11	Section on resource capabilities	Daniel Rubio Bonilla
0.10	08/04/11	Updated section on CλaSH	Christiaan Baaij
0.11	13/04/11	Revised section on hardware synthesis and resource discovery	Christiaan Baaij, Javad Zarrin
0.12	16/04/11	Finalise section on resource capabilities	Daniel Rubio Bonilla
1.0	30/04/11	Revision and fixes	Christiaan Baaij

I. Table of Contents

1. List of Figures

2. List of Tables

3. Abbreviations

Abbreviation	Description
HDL	Hardware Description Language
S(o)OS	Service-oriented Operating System(s)
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit

I. Introduction

The main objective of WP2 is to develop abstraction mechanisms to express both the functionality and the non-functional properties of hardware architectures in a semantic way. Chapter II introduces the *CλaSH* hardware description environment that focusses on the functional aspects of hardware architectures. It lifts the design process of hardware to a higher level of abstraction than offered by traditional tools and design environments. It allows us to capture and describe complexity of contemporary and future architectures (far) more easily than possible in current design environments.

Moving to Chapter III, we go into dealing with non-functional, derived, aspects of hardware and processor architectures. We discuss a modelling environment that allows us to describe the rough structure of a processor architecture. We can use the same environment to query (non-functional) properties of the processor models, such as whether they are superscalar. We developed a new modelling environment as there was no existing work that allows for easy capability extraction. Using a hierarchical model as opposed to flat data to encode these capabilities is both less error-prone and more general (as we can derive many capabilities through queries as opposed to hard-coding them). We also go into detail how the capability information can be used for code instrumentation.

As we need to make the information of the hardware architecture and capabilities available across nodes (for efficient code/application distribution) in heterogeneous large-scale dynamic environments, we need to categorize information relevant at the different hierarchies. This problem is tackled in Chapter IV. As the systems are potentially highly dynamic (ephemeral connectivity of mobile devices), capability information might have to be transmitted frequently, warranting the need to reduce overhead of capability information exchange. Also for this purpose a hierarchical capability information encoding makes sense, as we can reduce capability information being sent to that which is relevant for the specific level in the hierarchy.

Having to analyse the impact of our design choices relating to the code mapping we do based on capability information (Chapter III), and not having future hardware architectures available, we choose to simulate our systems. Because one of the major aspects of future processing systems [8] is due to the network connecting all the cores, we chose to start our focus of a simulator there. Chapter V discusses the *MCoreSim* NoC-simulator [9] we are developing based on the well-known *OMNeT++* network modelling and simulation framework. We intend to incorporate the topology descriptions from D3.2 [7] into future versions of the *MCoreSim* framework.

II. Hardware Modelling

Hardware description languages (HDLs) have not allowed the productivity of hardware engineers to keep pace with the development of chip technology. While traditional HDLs, like VHDL [4] and Verilog [5], are very good at describing detailed hardware properties such as timing behaviour, they are generally cumbersome in expressing the higher-level abstractions needed for today's large and complex circuit designs.

With increasing complexity in architecture design, it is preferable to lift the design process to a higher, more abstract level. CλaSH [1][2] is a hardware specification environment consisting of a hardware description language (HDL) and a compiler. The HDL has a syntax and semantics similar to the functional programming language Haskell [3]. The compiler translates the CλaSH code into synthesisable VHDL. By using CλaSH, it is possible to implement a complete architecture in a concise way on a high abstraction level while maintaining the possibility to implement parts of the architecture on a lower level, e.g. on the bit level. The designer can thus focus on the desired behaviour of the architecture. Higher level parametrisation is employed to describe the internal functionality of the system.

Functional languages are especially well suited to describe hardware because combinational circuits can be directly modelled as mathematical functions and functional languages are very good at describing and composing these functions.

Important aspects of the high-level approach are polymorphism and higher order functions. Polymorphism means that a function is implemented in a generic way without specifying the concrete type of its operands. Later, this function can be used in different contexts and with different types. Higher order functions offer, among other things, a powerful means to combine components in a generic way.

An alternative way to model hardware is to leverage the tools available in simulation frameworks, such as the MCoreSim tool [9] we are building in S(o)OS. Indeed, in MCoreSim, the modelling of different hardware topologies is possible thanks to the Network Description Language (NED) embedded in the OMNeT++ framework. These aspects are discussed briefly in Section V after an overview of MCoreSim, and specifically in Section V. 8. , where a short example is also shown for modelling a mesh topology.

1. Functional Hardware Specifications

The CλaSH [1][2] compiler can translate descriptions of an hardware architecture given in the CλaSH to RTL-style VHDL. The CλaSH libraries provide a set of predefined types (and corresponding operators), including: bits, booleans, signed and unsigned integers, vectors, and indices. Aside from these predefined types, synthesis of user-defined algebraic data types is also fully supported.

Synchronous hardware is modelled as a Mealy machine, where inputs (*i*) and the current state (*s*) are processed, by a transition function (*f*), to form a new state (*s'*) and output (*o*) of the circuit (see Figure 1). A generic transition function *f* is shown in Text 1.

The *State* keyword is used to indicate which argument, and what part of the return value, is connected to the state memory element of the component. The transition function and the initial value of the memory elements have to be combined to describe the complete component. We use the *lifting function* "⇑" for this purpose: it takes a transition function and an initial state to form a *component*. Components created using "⇑" are of the *Comp i o* type, where *i* indicates the input type, and *o* indicates the output type.

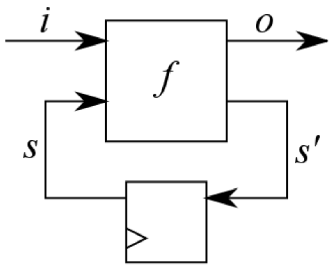


Figure 1: Mealy Machine

```
f (State s) i = (State s',o)
  where
    (s',o) = …
```

Text 1: transition function

The use of "⇑" is illustrated in Text 2: it shows the description of a multiply-accumulate circuit, where the *maccT* (lines 1-3) implements the transition function, and *macc* (lines 4-5), using "⇑", describes the complete component. The *macc* component is also annotated with its type (using "::"), indicating that it is a component with an input consisting of a tuple of 9 bit integers, and a single 9 bit integer output.

```
maccT (State acc) (x,y) = (State acc', acc)
  where
    acc' = acc + x * y
macc :: Comp (Int9,Int9) Int9
macc = maccT ⇑ 0
```

Text 2: Multiply-Accumulate Circuit

We can compose several components using the *arrow-syntax* (⇒ and ⇐); illustrated by the example shown in Error: Reference source not found. We declare a new component *macsum*, that has a four-tuple *(a,b,c,d)* as input, and a single result: the addition of the values r_0 and r_1 (line 1 of Error: Reference source not found). We use "⇒" instead of "=" to indicate that we are defining a *component* and not a *function*. A significant difference between a component and a function is that a function will always return the same value when applied to the same arguments, while this is not necessarily true for a component. The reason for this difference is that a component has an internal state which is remembered through consecutive evaluations of the component, and which typically influences the computation of the result.

```
macsum (a,b,c,d) ⇒ r₀ + r₁
  where
    r₀ ⇐ macc ⇐ (a,b)
    r₁ ⇐ macc ⇐ (c,d)
```

Text 3: composing macc components

The individual *macc* components are instantiated in line 3 and 4, being fed the signal tuples *(a,b)* and *(c,d)*, and returning r_0 and r_1 respectively. Again, we use "⇐" instead of the combination of "=" and function application, because we are dealing with components and not functions.

Synchronous circuits described in CλaSH can be *simulated* using a Haskell interpreter or compiler, because CλaSH circuit descriptions, and the CλaSH libraries, are also valid Haskell code. Although CλaSH is similar to Haskell, there are several constructs in Haskell that are not accepted by the CλaSH compiler, as their synthesis requires the analysis of time-variant behaviour. These constructs include dynamic data-structures and structures for which the size can not be determined at compile-time.

A) Polymorphic Circuits

CλaSH has three important language features that enable highly parametric designs: polymorphism, higher order functions, and type derivation. As a result of polymorphism a designer can describe the general flow of data, without being restricted to any specific type of data. A component where polymorphism is of great aid is a FIFO buffer, which does not manipulate the data that it stores, and its structure is the same for all types of data.

There are two ways to make concrete instantiations of polymorphic functions: 1) make a clone and annotate the clone with a concrete type, or 2) let the type be inferred from the context at the instantiation. We elaborate both cases using a two-position FIFO buffer as an example.

```
data FS a = FS { m0 :: a , w0 :: Bool
               , m1 :: a , w1 :: Bool
               }
```

Text 4: Record type for FIFO state

We start with the definition of a new record-type *FS* that will encode the state of the FIFO (Text 4). The record has four fields *(m0,w0,m1,w1)*, where *m0* and *m1* are the memory locations, and *w0* and *w1* are two flags indicating whether the respective memory locations are written. While the fields *w0* and *w1* are of type *Bool*, the types *m0* and *m1* both of the same polymorphic type *a*. So where *FS a* is the polymorphic type, e.g., *FS Int9* and *FS (Bool,Bool)* are two concrete instantiations of *FS a*.

The transition function for the FIFO, describing the behaviour, is given in Text 5.

```
fifo (State (FS {..})) (rd,nv) =
  (State (FS m0' w0' m1' w1'),(full,ov))
  where
    (m0',w0',m1',w1')
      | rd && w0      = (nv, True, m0, w0  )
      | rd || not w1  = (m0,   w0, nv, True)
      | not w0        = (nv, True, m1, w1  )
      | otherwise     = (m0,   w0, m1, w1  )

    full  = w0 && w1
    ov    = m1
```

Text 5: Polymorphic 2-position FIFO Buffer

The *fifo* function has two inputs, read (*rd*) and new value (*nv*), and two outputs, a full flag (*full*) and a data output (*ov*). The *rd* input indicates if a value is being read from the FIFO, *nv* is the new value to be stored, *full* is an output flag that indicates if the FIFO is full, and *ov* is the value exiting the FIFO. The code *(FS {..})* in line 1 automatically brings the fields of the *FS* record into scope. The code *(FS m0' w0' m1' w1')* assigns *m0'* to the first field of *FS* (which is *m0*), *w0'* to the second field, etc. Lines 3 to 8, the assignment of *(m0',w0',m1',w1')*, is a *guarded* expression. A *guard* is the Boolean expression between the "|" and the "=". The expression for which its guard evaluates to T*rue* is used for the assignment. The first expression for which its guard evaluates to *True* is used when multiple guards evaluate to *True*. The *otherwise* guard always evaluates to *True*.

We can now make a concretely typed clone of the FIFO transition function by annotating the clone with a type. For example, a FIFO that stores 9-bit integers (Text 6):

```
fifo9 :: State (FS Int9) -> (Bool, Int9)
  -> (State (FS Int9), (Bool, Int9))
fifo9 = fifo
```

Text 6: Cloned 9-bit Integer FIFO

For the second case we use the *fifo* transition function in a concrete context and let the type inference mechanism determine the correct type for that specific instance. We define a concrete initial state s_0 that stores 9 bit integers, by replacing the type parameter *a* of the *FS* record with the concrete type *Int9*. An instance of *fifo* is then lifted with s_0 to form the concretely typed component *fifo9c* (Text 7).

```
s₀ :: FS Int9
s₀ = FS 0 False 0 False

fifo9c = fifo ⇑ s₀
```

Text 7: Derived 9-bit Integer FIFO

The type of the instance of *fifo* in Text 7 can be made concrete through automated type-derivation because: 1) the type of the initial state of the memory elements is known, and 2) the data in- and outputs are of the same type as the memory elements.

B) HIGHER ORDER CIRCUITS

Having highlighted the parametrisation potential offered by polymorphism and type derivation, we now move to the third feature of CλaSH that enables natural parametrisation: higher order functions. Higher order functions are essentially no different from `normal' (first-order) functions, in that they process and return values. We call functions higher order however when (some of) the values they process and/or return are *functions* themselves. A well known higher order function is *vmap*, whose type signature is shown in Text 9.

```
vmap :: (a -> b) -> V s a -> V s b
vmap f as = ...
```

Text 8: Type signature of vmap

The first argument of *vmap* is a *function* (*f*) with input type *a* and output type *b*. The second argument is a vector of length *s* and element type *a*, and the return type is again a vector of length *s* but with element type *b*. What the *vmap* function does is apply the function *f* to every element in the vector. Text 9 shows the usage of the *vmap* function by negating a vector of Boolean values.

```
negateAll :: V s Bool -> V s Bool
negateAll v = vmap not v
```

Text 9: Negating a vector of Booleans

Because there are no data dependencies between the function applications that *vmap* introduces, all function instances can be executed in parallel. The CλaSH compiler will therefore translate the *negateAll* function to a set of concurrently executing components. A visual interpretation of the resulting hardware when applying *negateAll* to a 12 element vector is shown in Figure 2.

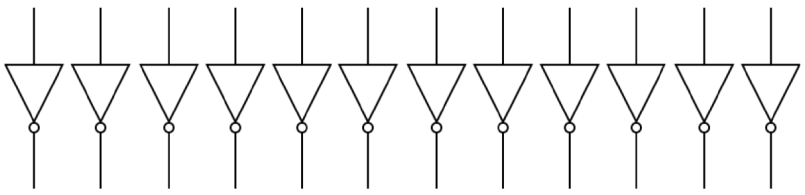


Figure 2: negateAll - derived hardware

A common circuit where the use of higher order functions leads to a concise and natural description is a crossbar. The code for the description of such a circuit is shown in Text 10.

```
crossbar inputs selects = vmap (mux inputs) selects
  where
    mux inp x = inp ! x
```

Text 10: Crossbar circuit

The *crossbar* function selects those values from *inputs* that are indexed by the *selects* vector. Besides the fact that the *vmap* function again takes another function as an argument, we also see that the *mux* function is only applied to one argument while the definition suggests it has two arguments. This is another instance of higher-order functionality: by applying *mux* to only a single argument, the function will *return a function* that still expects the original second argument. This coincides with the type signature of *vmap*. The crossbar is polymorphic in the width of the input (defined by the length of *inputs*), the width of the output (defined by the length of *selects*), and the signal type (defined by the element type of *inputs* vector). We (also) remark that the element type of the *selects* vector is automatically inferred to be *Index*; following from the use of the indexing operator for vectors (!).

c) Example: Higher Order CPU

This section discusses a somewhat more elaborate example in which user-defined higher-order function, partial application, lambda expressions, and pattern matching are exploited. The example concerns a *CPU* which consists of four function units, *fu*$_0$ … *fu*$_3$, (see Figure 3) that each perform some binary operation.

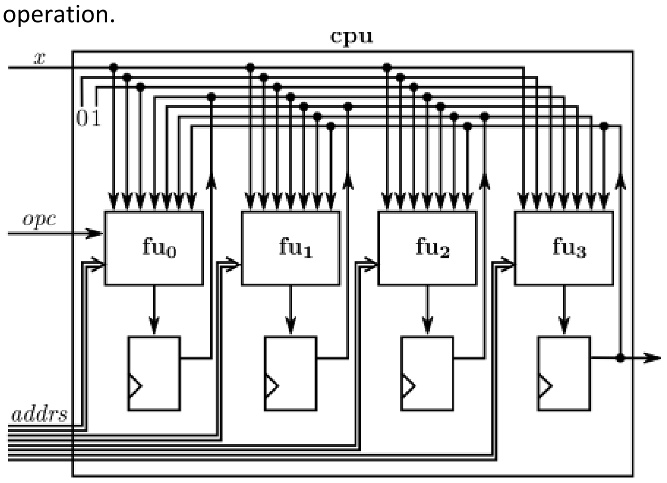


Figure 3: CPU with higher-order Function Units

Every function unit has seven data inputs (of type *Int16*), and two address inputs (of type *Index D7*). The latter two addresses indicate which of the seven data inputs are to be used as operands for the binary operation the function unit performs.

These seven data inputs consist of one external input *x*, two fixed initialization values (0 and 1), and the previous outputs of the four function units. The output of the *CPU* as a whole is the previous output of *fu*$_3$.

Function units *fu*$_1$, *fu*$_2$, and *fu*$_3$ can perform a fixed binary operation, whereas *fu*$_0$ has an additional input for an opcode to choose a binary operation out of a few possibilities. Each function unit outputs its result into a register, i.e., the state of the CPU. This state can e.g. be defined as follows:

```
type CpuState = State (Vector D4 Int16)
```

i.e., the state consists of a vector of four elements of type *Int16*. The type of the *CPU* as a whole can now be defined as (*Opcode* will be defined later):

```
cpu :: CpuState
  -> (Int16, Opcode, Vector D4 (Index D7, Index D7))
  -> (CpuState, Int16)
```

Note that this type fits the requirements of the *run* function. Every function unit can be defined by the following higher-order function, *fu*, which takes three arguments: the operation *op* that the function unit should perform, the seven *inputs*, and the address pair *(a_0,a_1)*. It selects two inputs, based on the addresses, and applies the given operation to them, returning the result:

```
fu op inputs (a₀,a₁) =
  op (inputs!a₀) (inputs!a₁)
```

Using partial application we now define:

```
fu₁ = fu add
fu₂ = fu sub
fu₃ = fu mul
```

Note that the types of these functions can be derived from the type of the *cpu* function, thus determining what component instantiations are needed. For example, the function *add* should take two *Int16* values and also deliver a *Int16* value.

In order to define *fu_0*, the *Opcode* type and the *multiop* function that chooses a specific operation given the opcode, are defined first. It is assumed that the binary functions *shiftR* (where *shiftR a b* shifts *a* to the right by the number of bits indicated by *b*) and *xor* (for the bitwise *xor*) exist.

```
data Opcode = ShiftR | Xor | Equal

multiop ShiftR = shiftR
multiop Xor    = xor
multiop Equal = \a b -> if a == b then 1 else 0
```

Note that the result of *multiop* is a binary function from two *Int16* values into one *Int16* value; this is supported by CλaSH. The complete definition of *fu_0*, which takes an opcode as additional argument, is:

```
fu₀ c = fu (multiop c)
```

The complete definition of the *cpu* function is (note that *addrs* contains four address pairs):

```
cpu (State s) (x,opc,addrs) = (State s', out)
  where
    inputs = x +> 0 +> 1 +> s
    s'     = (fu₀ opc inputs (addrs!0)) +>
               (fu₁     inputs (addrs!1)) +>
               (fu₂     inputs (addrs!2)) +>
               (fu₃     inputs (addrs!3)) +>
               empty
    out    = last s
```

Due to space restrictions, Figure 3 does not show the internals of each function unit. We remark that CλaSH generates e.g. *multiop* as a subcomponent of *fu_0*. The combined code fragments of this simple CPU can be found in Appendix C: Simple CPU. While the *CPU* has a simple (and maybe not very useful) design, it illustrates some possibilities that CλaSH offers and suggests how to write actual designs.

2. *MAPPING DESCRIPTIONS TO HARDWARE*

We want to use the same resource description for different purposes to avoid semantic gaps between the description used for one purpose (for example simulation) and another description for a different purpose (for example synthesis). The history of hardware description languages has also shown that engineers, people who use these description languages, do not want to repeatedly describe the same system for different purposes. VHDL, these days commonly associated with hardware synthesis, was originally meant to be a specification language when it was first standardized. These specifications were meant to be the documentation for the designed hardware systems. Simulation tools for these VHDL specifications were developed quickly afterwards, followed by automated synthesis tools. This has however led to the situation where most (but not entirely) of the latest VHDL specification [4] is accepted by simulators, but only a small subset is accepted by automated synthesis tools.

Because CλaSH is (currently) a subset of Haskell, simulation of circuit specification comes for free in terms of existing Haskell compilers and interpreters. CλaSH specifications are also completely synthesisable using the CλaSH compiler which translates to the specification to that subset of VHDL which is accepted by all VHDL synthesis tools. We note that any netlist-like format could have been used, however, VHDL was chosen because it is a standardized language and has many supporting tools from different vendors. Because these tools use different (dialects of) netlist representation languages, choosing a specific one for CλaSH would have meant that we could not use all the available tools. Familiarity with VHDL by the CλaSH developers also added greatly in verifying the correctness of the implementation.

The abstractions used in CλaSH specifications, such as polymorphism and higher order functions, are however not representable in these netlist formats. As a result we have to transform potential polymorphic, higher order circuit descriptions to monomorphic, first order functions. Something we can only do if the top-level entity is monomorphic and first-order. We already mentioned this restriction in the simple cpu example of subsection II. 1. c) . Another example where a CλaSH description does not exactly match a netlist format, is that CλaSH descriptions usually lack the explicit naming of an output port. These discrepancies between the original descriptions and a netlist format lead to the definition of a 'normal form', which states the form a CλaSH description should have to be directly translatable to a netlist format.

This normal form is however defined at the System F (type lambda-calculus) level, a language to which all CλaSH descriptions can be de-sugared to, and which has far fewer language constructs. Because System F has fewer language constructs, a normal form is far easier to define. And, as a result, the transformations to bring a System F description into a normal form, are also easier to define. The transformation system applies all transformations exhaustively until the description is in the desired normal form. The soundness, termination, completeness, and confluence of this transformation system are currently not formally proven. However, extensive use of the CλaSH system indicates that many of these properties hold. We do however intend to formally prove these desired properties in future iterations.

III. Resource Capability Descriptions

In addition to the structural and architectural information of the resources and their relationships to each other (topology), the actual capabilities of the resource are needed for adaptation and mapping of code properties – in other words, to exploit the specific capabilities of the resource –. In the simplest case, the common denominator can be assumed, as this will remain compatible with most types from a functional perspective, even if it will not help in how to map resources to improve performance. This way the specific capabilities are not exploited (for example, a GPU will be considered as a single core machine without vector units).

We want to describe the resource capabilities in a way that allows us to:

- identify the requirements of the code
- adapt the code to make use of the hardware capabilities

In this chapter we will expose how to identify the hardware resources requirements from the source code and how the resource capabilities can be described and derived from hardware descriptions.

1. Identifying Resource Requirements from Code

Code does not in general explicitly state resource requirements (besides for some programming extensions), instead, the requirements for optimal performance have to be derived indirectly from the code behaviour and structure analysis.

Resource requirements are therefore not only derived from the code, but also must be considered "boundary" criteria for segmentation and adaptation. In case that a very heterogeneous system is given, then multiple iterations with different boundary may need to be mapped to fit better the capabilities of each resource.

In summary, we can assure that the size and amount of the segments can vary significantly depending on the hardware resources available at each moment.

2. Identifying Resource Capabilities

In order to identify:

- how well the resource performs given specific algorithm types
- how the code and/or algorithm needs can be adapted to make best use of the resource's specifics

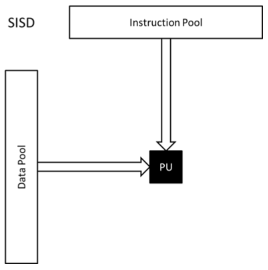
some information about its capabilities and restrictions are needed:

a) Processor Classes

The processor class provides information about the code-data relationship of the processor, in particular regarding its parallelism, i.e. whether the processor can only execute a single instruction on a single data (SISD), or on the other hand execute multiple instructions at the same time on different data (MIMD), or a single instruction on multiple data in parallel (SIMD) which effectively is the GPU structure – this is closely related to what became known as Flynn's taxonomy [10]:

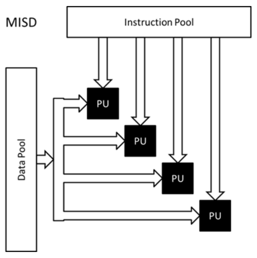
	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SISD Processors



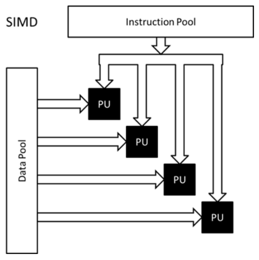
- Plain sequential processor (can have pipelining)
- Can only operate one single datum (load/store) and instruction at a time
- It can be used in particular for sequential code with little degree of parallelism
- notable architecture feature:
    - single data-PU-channel
    - single instruction-PU-channel

MISD Processors



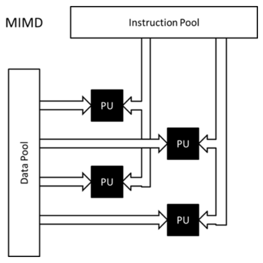
- Execute multiple operations on a single datum
- lead to serious consistency problems
- so far, no reasonable, let alone commercial realisation
- notable architecture features:
    - single data-PU channel
    - multiple instruction-PU channels

SIMD Processors



- Execute the same operation on multiple data
- each datum can thereby regarded as a dimension in a vector, a SIMD with vector-length 8 can therefore handle 8 dimensional vectors
- sometimes used as coprocessors
- related to GPUs
- notable architecture features:
    - multiple data-PU channels
    - single instruction-PU channels

MIMD Processors



- Represents processors with multiple SIMD ways, normally pipelined
- many classes: VLIW, superscalar
- typical general architecture today
- as opposed to multi-processor SIMD, instructions and data are related

- general architecture features:
    - multiple data channels
    - multiple instruction channels

  but, the relationship between data / code to PU may vary
- limited by:
    - the degree of intrinsic parallelism in the instruction stream, the amount of instruction-level parallelism
    - the complexity and time cost of the dispatcher and associated dependency checking logic
    - the branch instruction processing
    - the size and capabilities (i.e. load overtake stores) of the out of order execution in case of superscalar processors
    - the quality of the prefetch unit

Identification criteria in the resource description:

Type	**Data-PU Channels**	**Instruction-PU Channels**	**Number of PUs**
SISD	1	1	1
MISD	1	>1	=#instruction-PU channels
SIMD	>1	1	=# data-PU channels
MIMD	>1	>1	Different versions • #DPU • #IPU (<#DPU)

But that information on itself is not enough. It is needed to distinguish between different intentions of the models. Even though SIMD, MIMD etc. reflect architectural notions of a processor description, they do not necessarily incorporate capability information that is related to the conceptual architecture of the processor. For example, given a certain ISA with a 4 dimensional vector SIMD instruction, one hardware implementation can use 4 ALUs and calculate it simultaneously while in others one ALU can be used to calculate the 4 data, one after the other, with at least a 4 time increase in processing time while compared to computing a single data. In the case of 4 ALUs computing simultaneously, the execution time will be the same for a “package” of for 4 data or for just 1 datum.

Often, a similar situation is found when operating with floating point units and its precision. A double precision operand instruction can usually be processed in a single precision FPU, but the time needed usually varies from 4 to 16 times more. And even floating point operations can be performed in a *only* integer ALU, but with a much high processing time cost (of the order of hundreds of integer instructions). This last technique called *softfp* is often used in microcontrollers and other kind of less performance (and small transistor count) processors.

We can note that the processors are also denoted particularly by the following characteristics:

1. the lines toward the data pool:
    1. number of lines (number of data that can be processed simultaneously)

        2. width of a line (bit size of each data, calculation precision)

    Modern processors can usually trade the numbers of lines for the width of the lines, that means that more precision data can be calculated if the number of data is reduced. For example, a 128 bit wide vector FPU could operate 2 dimensional 64 bit data or 4 dimensional 32 bit data.

2. the type of ALUs and their capabilities:
    1. functionalities: ADD, MULT, DIV etc.
    2. type: floating point, integer or both.

    Many modern architectures have specialised ALUs that can only execute a set of specific instructions and respectively only operate on specific data types (it is very common to have many ALUs that can perform simple instructions like ADD or AG (Address Generation) and very few for complex rare instructions like DIV). A “complete” processor has a set of ALUs that allows it to execute all the instructions of its ISA. Therefore, ALUs of different types in a single processor either lead to a reduced number of "complete" ALUs or to multiple "specific" SIMDs that share the instruction line. This information can not be easily specified or identified, as often is not clearly released by the manufacturers.

3. the lines to the instruction pool
    1. number of lines is the maximum number of instructions issued per cycle (in the SISD and SIMD case that's automatically 1, in MIMD it is more than 1). The real number IPC (Instructions per Cycle) is usually considerable inferior than the maximum theoretical – but it is heavily affected by the type of code that is being executed.

4. Registers
    1. ISA specific registers: these are the registers usable by the programmer (or the compiler). They differ in number and bit width size. There can also be many different register banks, as for example for the integer ALU, for the FPU or for the vectorial units.
    2. Hidden registers: these registers are not accessible for the user, but are used by the hardware inside the CPU for different purposes like register renaming, pipeline instruction overtaking etc..
    3. Register window: if there is and its size, which has a big impact on function calls.
    4. Register duplication: some processors have duplicated register banks speed up the thread context switching. Some processors have this technique improved like in the case of Intel's HyperThreading Technology.

This information about the different kind of Processing Units is necessary as a too abstract description does not provide meaningful information about how well a segment of code will behave in a particular hardware configuration.

The first filtering has to be to check if the hardware can execute the particular segment of code that the OS wants to place, although nowadays almost all processors (does not matter if it is a CPU or a GPU) can in theory run any kind of code, we might still find some that can not. After it we have to look into particularities of the code. If it has many floating point instructions, then we should look for a processor with a FPU or a vectorial unit with FP capabilities that fulfil the minimum needed precision. Another important check is if there are special operations (like DIV, SQRT, dot product instruction...) because a processor with special instructions for this operations can execute code faster than a hardware that does not have them, but that would behave faster in a more generic code. As for superscalar processors, not all pipelines are the same or have the same capabilities and this should be taking into account (a code with many DIV operations will probably perform better in a superscalar CPU with only two pipelines but both with dedicated hardware DIV ALUs, than in a superscalar CPU with 4 pipelines but with only one of them fitted with a hardware DIV capable ALU).

The previous text tries to justify why all this information is needed, and it is difficult to make a right balance between the level of detail that is useful and when too many details only complicate the code segmentation mapping without any improvement in performance (or probably even making it worse).

B) Configurable Hardware: Example of Single Instruction Multiple Data (SIMD) Processor/Pipeline

Previously it has been explained the difficulties in determining the level of detail of the hardware description that is useful and meaningful for the segmentation and code segment on hardware mapping algorithm. To this other hardware characteristics have to be added, like the flexibility of hardware configuration. This flexibility is another factor that has to be considered and that increases the algorithm complexity. Many of today's hardware can be reconfigured to make use of its resources in very different way.

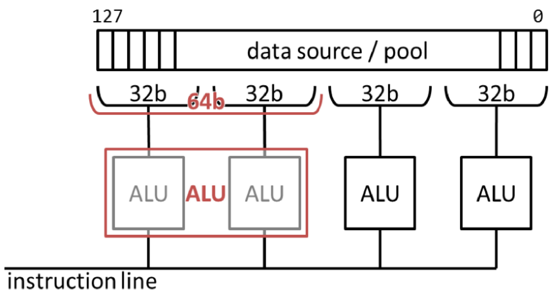


Figure 4: Configurable SIMD processor/pipeline diagram

As shown in Figure 4, SIMD is quite a "generic" type in Flynn's category: many SIMD can "unify" their ALUs into fewer amount but with a wider dataline. If all the ALUs inside the processor (or the pipeline) are unified into a a single ALU then it becomes a SISD. At the same time many SIMD pipelines can be unified sharing the register banks which leads to a MIMD processor.

C) Using Capabilities in Code / Execution

There are a few levels of information sources that need to be matched in order to execute the code in a fashion most suitable to the hardware:

- on the software side:
  - code annotations provided by the user
  - information automatically derived from code analysis
  - profiling of previous executions
- on the hardware side:
  - parametrisation of hardware through automatic tests (ping, automatic benchmarks etc.)
  - hardware self description (flags in cpuinfo etc.)

- ○ database of description provided by manufacturer and the information derived from this

The main point of gathering all this information about the description of the hardware and needs of the software consists in providing a flexible means to query for resources with specific capabilities with the aim to try to map the software segments as good as possible.

As opposed to pure typing or look up table, a description that is closer to the hardware architecture offers more freedom in retrieving capabilities. As such, additional information can be retrieved from the description, if the user writes an according query.

Appendix A: FlynnCatQuery describes the datatype with which to describe a processor in a hierarchical fashion, allowing for the earlier mentioned queries for capabilities. We see in Text 11 the textual representation of the SIMD machine shown in Figure 4, according to the datatype described in Appendix A: FlynnCatQuery.

```
SIMD_CPU il0 il1 = PU { alus = [alu0,alu1]
                      , registers = [reg0] }
 where
  alu0 = ALU { function = "ADD, MUL", aluType = "FP"
             , register = reg0, dl = dl0, il = il0}
  alu1 = ALU { function = "ADD, MUL", aluType = "FP"
             , register = reg0, dl = dl1, il = il1}
  dl0   = DL { pool = pool0, address = 0, width = 32}
  dl1   = DL { pool = pool0, address = 32, width = 32}
  pool0 = DataPool { size = 64, poolId = "0" }
  reg0  = Register { rId = "0" }
```

Text 11: SIMD description using FlynnCatQuery

When we use the *flynnCategory* query (also described in Appendix A: FlynnCatQuery) on a concrete instantiation, where *il0* and *il1*, will refer to the same instruction line, then we indeed get the answer that Text 11 indeed describes a SIMD machine.

IV. Resource Descriptions for Discovery

An important aspect of resource discovery and management in a network with multi-core nodes is the way of describing the processing resources. From the network and service discovery point of view we need to have a more abstract and high-level model of resource description. Resource description should include useful, effective and short information which would be appropriate for services advertisement by sending messages. Due to the network overhead and traffic, we cannot put a lot of information about the resource in the related message for the service announcement. On the other hand, for resource mapping and management; it is required to have a set of criteria and parameters for resource evaluation and service matching according to the requirements of the queries. Thus, resource description should support explicit evaluation criteria.

There are two important performance parameters for the processors, which are time of execution and throughput (FLOPS/IOPS). These two depend on many other parameters such as cache size, bandwidth, latency, interconnection and even the type of code which is so complicated. The alternative evaluation criteria is using benchmarks which in case of S(o)OS could be neither practical nor helpful .

To achieve a more efficient and abstract resource model, the description of computing resources in a heterogeneous network with multi chips and multi core nodes necessitates being divided into below levels including a mechanism for the streaming of the descriptions with a particular functionality which can diminish low level detailed information and absorb new attributes and predicates of the next upper level architectures:

1. Inter -Cores Level (Network on Chip (NoC) Level): resource description in this level has been characterized according to THE type of information which is directly related to the core specifications. The most typical information in this level is floating point and integer operation per cycle and private cache size.
2. Inter -Chips Level: describes the specifications of the edges for the core communications, interconnection networks and memory hierarchy which include information such as: bandwidth, latency, shared or global cache size and number of cores.
3. Inter -Nodes Level: Resource description covers a total overview of all the communications, memory and processing features and aspects in the node such as memory size and number of processors.

Due to the above classification resource description is defined in three different layers .The coded information ranging through the resource description from the first level to the last one changes from hardware details to more abstract information. The unnecessary information which gets reduced moving toward the last layer is encrypted into specific fields in the upper layers.

In the resource discovery process, firstly the description of all nodes are read , then the nodes which can provide the query with the minimum requirements are selected and some specific encrypted fields are decoded for them in the lower layers , and the information contained in them is extracted . This selection and extraction operation in the resource description continues to the last layer until the minimum requirements for the query is reached. The allocation of resource to query is conducted via Distributed Hash Tables (DHT) and Queue Management System (QMS).

We can use the aforementioned multi-layer resource description model to describe resources, but we need to improve or complete these models in the ways such as the followings:

1. The performance score can be extracted from all the information provided by the resource description, which requires us to have an explicit algorithm to calculate this item with.
2. Another alternative is using Mico High Performance Computing (MHPC) Benchmarks such as Sandia Micro-Benchmark Suite (SMB) or MIBA Micro-Benchmark Suite for real-time evaluating and benchmarking.
3. The typical processor benchmarks are real applications which could vary depending on the use case and the end user applications. To provide an evaluation criteria, we can use a real-time benchmarking model according to the past execution performance.
4. Using one of the solutions 1 or 2 with combination of 3

Figure 5 shows the architecture of the multi-layer resource description which keep an abstract of all the resource specifications related to computing and communicating capabilities of the nodes and their components. We have defined layers to avoid unnecessary data transmission and communication overhead as the consequence of resource discovery operation in a large multi-core system . For instance, in the process of discovery operation among two cores on a same tile, discovery messages are not required to carry node communication behaviours.

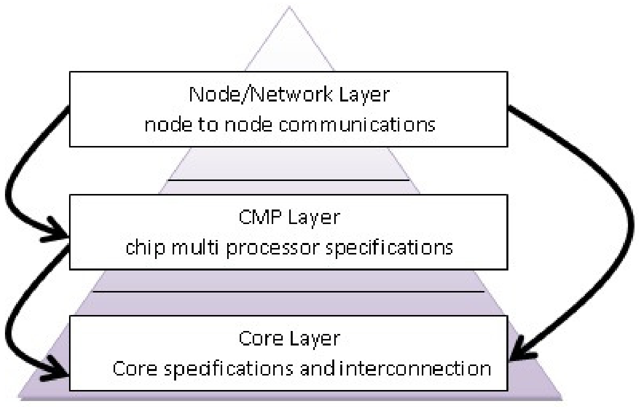

Node/Network Layer
node to node communications
CMP Layer
chip multi processor specifications
Core Layer
Core specifications and interconnection

Figure 5: Architecture of the multi-layer resource description

In the above hierarchical architecture containing computational and communicational properties and behaviours , most describing links and interconnection characteristics are located at the top of the pyramid, and computational resource descriptions are at the bottom . Due to the hierarchical structure of the resource description model, it could provide a comprehensive perspective of all other neighbouring resources in the vicinity and in the network for each processing unit independently ,besides, the model is able to find out detailed information for each node or chip multi -processor.

Multi-Layer resource description acts like fisheye's model (Figure 6). It means that every processing unit (core) in a cluster doesn't need to keep resource descriptions about all other processors . In this model descriptions of resources include accurate, predictable and unclear data. the accurate data comes from closer cores which are placed in the same tile or node . In the process of resource discovery for a query originated from a particular core, processors with the minimum latency have priority to be nominated as members of qualified resource groups, therefore, for a core, it is not required to know about far away cores precisely.

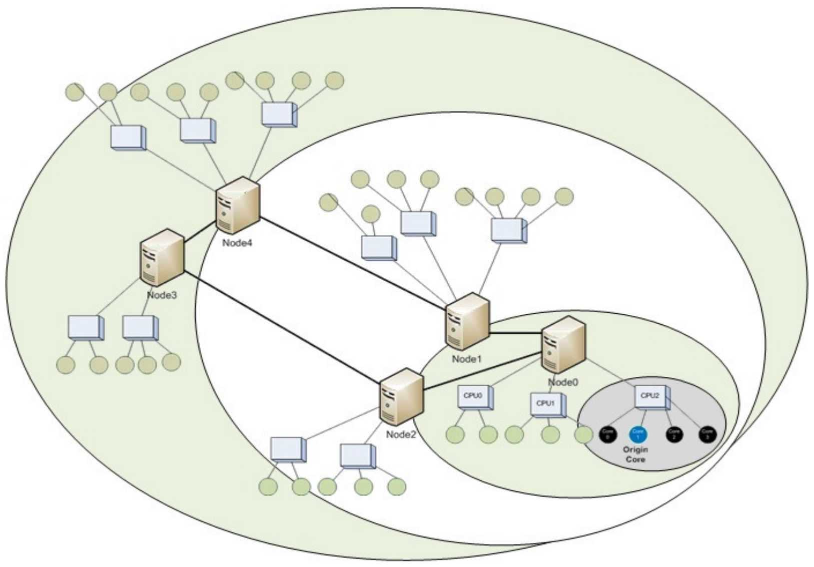


Figure 6: Fisheyes's model of resource description for resource discovery

Table 1 demonstrates set of description parameters that are defined in each layer

Layers	**Name**	**Description Parameters**
Layer1	Node Layer or Network Layer	Network topology description (topology type), network bandwidth and latency, node specifications (node ID, node Type , no. of CPUs,etc)
Layer2	Chip Multi-processor Layer	Inter-chips communications characteristics (rated_bus_speed) , architecture type (Flynn_taxonomy) , CPU specifications (CPU name, CPU ID, no. of cores,etc) , memory and cache hierarchy (shared and global cache, hierarchy,size and latency, DRAM size)
Layer3	Core Layer	Processor's characteristics (core ID, clock rate, instruction set, pipeline, MIPS, private cache L1 and L2,etc.), Memory access (cache and memory latency) , Inter-cores communications behaviours (interconnection_bus , inter-cores latency matrix)

Table 1: Resource description parameters for discovery

The aforementioned resource description model can be considered as a node description model which has built-in network topology and link descriptions. However, complicated network topologies description for the nodes is not included. We are using COTSon and SimNow simulators to evaluate our resource discovery protocol which is discussed in D3.2 in detail. COTSon uses BSDL to describe resources in a simulated cluster. The implementation of resource discovery protocol must be supported by resource descriptions in the most efficient and easy way, therefore we developed a model for resource description and a definition compatible with the required resource information in discovery process.

V. NoC Simulator

This chapter describes MCoreSim [9], the Many-Core Modeling and Simulation framework we are developing in the S(o)OS project for the purpose of comparing different possible alternatives in future many-core, distributed and massively parallel systems. One of the main innovation elements in these kind of systems is foreseen [8][6] to be the presence of a Network-on-a-chip for interconnecting the many cores present on the same chip, and the possible lack of a globally coherent memory. Therefore, the simulator was born with the main objective of simulating the impact of the NoC latencies for the purpose of evaluating different strategies for common operations that are recurrently performed by the software stack, either through the Operating System, or by application-level middleware. These operations may include basic low-level synchronization and communication primitives for Inter-Process Communications (IPC), as well as scheduling and deployment policies which have the responsibility to decide where to deploy each individual software component (i.e., process, thread, code segment). Therefore, MCoreSim does not emulate any Instruction-Set Architecture of any real processor, but rather it supports so far a very basic set of primitive operations needed to move data across the computing and memory elements in a future many-core, tile-based chip architecture.

MCoreSim is based on the well-known OMNeT++ framework for network modelling and simulation. Therefore, in the following we provide a quick overview of OMNeT++ before proceeding to describe our MCoreSim simulator. MCoreSim is still to be considered in a work-in-progress status, and its features and capabilities will improve and expand during the next months.

1. Background on OMNeT++

OMNeT++[1] is a component-based discrete-event simulation library and framework for simulating networked systems. It provides developers with the capability to define highly reusable modules, which can be instantiated and interconnected in different designs. Modules have gates for interaction with other modules and can be combined with each other at arbitrary nesting levels. Modules communicate through message passing via defined gates and channels (connection between modules). Modules, channels and gates can be parametrized by leveraging the Network Description (NED) Language, and used for customizing the behaviour according to specific application scenarios.

2. MCoreSim Simulator Architecture

Due to the difficulties in maintaining a globally coherent cache/memory view in many-core systems, the massage passing paradigm is the most suitable one to be used for communications. In many-core systems, the network-on-a-chip paradigm requires message passing to be implemented in hardware by the on-chip interconnection logic. MCoreSim models a tile-based networked system.

There are two major components: the tile and the connection strategy (topology). The architecture of a single tile is summarized in Figure 7. Each tile includes:

- a CPU, which we call processing element (PE)
- a local memory element (Local Store)
- a Memory Management Unit (MMU) redirecting memory operations towards the local memory element or some remote tile through the local router

1 More information is available at: http://www.omnetpp.org/.

- a local cache memory
- a Network Interface, realizing reliable communications with other tiles

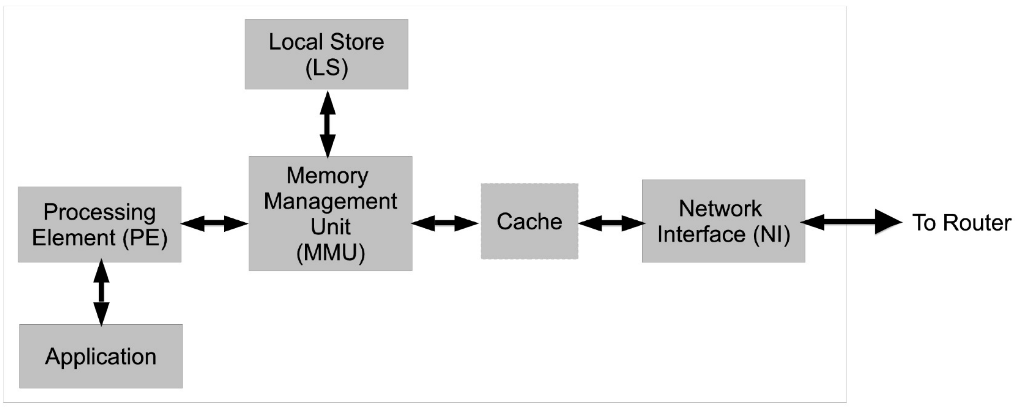


Figure 7: Internal architecture of a tile.

Parameter name	Description
topology	Topology type (mesh or torus)
datarate	datarate of a channel
delay	delay in channel
flitSize	size of flit packet
numX	number of nodes in X direction
numY	number of nodes in Y direction
bufferSize	buffer space available in router
NIbufferSize	buffer space available in NI
router type	routing algorithm for topology
clockRate	clock rate of processing element
memoryClockRate	clock rate of memory element
numVC	number of virtual channels per input port

Table 2: Configuration parameters for MCoreSim.

Logically, a Processing Element (PE) can model a general purpose or special purpose processing unit. Its main function is to fetch the instructions from the application program and simulate the time needed to carry out the involved computations interacting with the necessary local or remote memory elements and/or other cores.

The Network Interface (NI) enables the transfer of data (or instructions) to/from other tiles and/or I/O peripherals. First, it stores the packet to be transmitted in a local buffer then it involves the communication unit for realizing the actual transfer to the destination processing element or memory or I/O controller.

The software that runs over each PE is also modelled in OMNeT++ as a module, which we called generally Application. This is a module that provides the set of instructions to be executed on a computing unit.

The actual topology simulated in MCoreSim may be customized via a set of configuration options. The most important of them are summarized in Table 1. The parameters meaning will become clear in the following description of the main simulator components.

3. Interconnection

In OMNeT++, interconnection of modules is modelled by using the `network` keyword in the NED language. This construct is exploited in MCoreSim, allowing for the specification of a configurable 2D mesh and torus. MCoreSim defines a link between each pair of tiles, modelled as a physical channel that has configurable properties like delay, data-rate, etc., that are defined by extending the `ned.DatarateChannel` channel model available in OMNeT++.

The router provides the inter-connectivity of tiles. A typical router architecture consists of a buffer for input packet storage and a computing unit that performs buffer management, routing, packet scheduling, port arbitration and a switch to output port.

In MCoreSim, each of this functionality is defined as a module, such as the Route Computation (RC), Virtual channel Allocation (VA), Switch Arbitration (SA), and Switch (ST) modules. Based on the topology details, the number of buffers can be parametrized (see Figure 8).

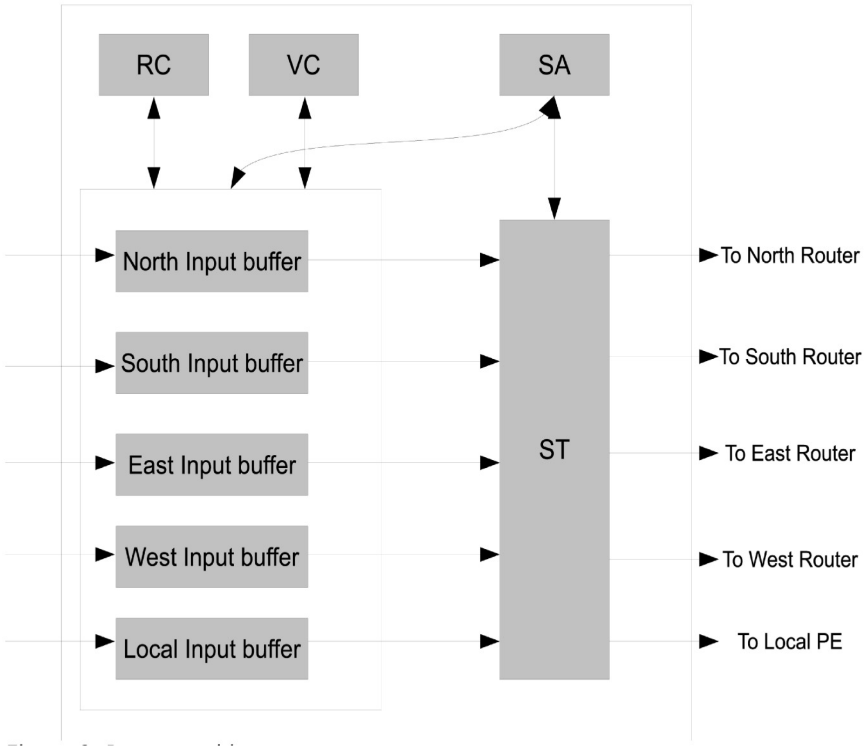


Figure 8: Router architecture.

4. Communications

The MCoreSim protocol library provides the protocols for simulating communications and computations. Up to now the library implements protocols related to a basic communication and computational model, including mesh and torus regular topologies with Dimension-Order Routing (DOR), wormhole switching and a credit-based Link-level Flow Control scheme [8]. Other schemes can easily be plugged in the future into the framework, as required. Due to the close relationship with in-chip communications, further details on these elements are provided in the S(o)OS Project Deliverable D3.2.

5. Computation Plane

The Computation Plane has three major components: processing plane, memory and I/O plane, application modelling. In MCoreSim, the Processing Plane consists of several Processing Elements that constitute an entire chip. A Processing Element may be general-purpose, simulating a typical "CPU" behaviour, or specialized, like GPUs. Each general-purpose PE works by following the traditional fetch-execute cycle. It processes mainly three types of instruction: ALU operations, memory operations, massage passing instructions. In an ALU operation, the PE executes instructions by using its ALU unit, however only the delay needed for this execution is simulated in MCoreSim. In a memory operation, the PE routes the memory access towards a local or remote memory element with the help of the memory management unit (MMU). If the access is local, then it is routed to the Local Store, otherwise the PE generates a Message Passing Instruction towards a remote destination. Message Passing Instructions are routed to their destination tile via the Communication Plane through the local Network Interface.

There are various memory hierarchy types available for many-core systems. MCoreSim implements the Memory Plane as a globally shared memory. In this global memory every PE has its own private memory space called Local Store. Each PE can directly access the Local store but for remote access uses the Communication Plane. The MMU provides the mapping between the addresses generated by the PEs and the either local or remote memory elements.

6. Application Modeling

As of now, an MCoreSim Application supports only three types of Instruction: memory, ALU, message-passing based. Furthermore, these applications are pinned to specific cores at configuration time. In the future, the Application Plane will support a richer set of instructions (but the focus of our simulation will anyway keep the number of supported instructions at the bare minimum), and it will provide dynamic application mapping and scheduling across the available computing infrastructure, so as to better emulate the software stack and OS services.

An element we are working on is the simulation of specific scheduling and deployment algorithms in MCoreSim. This feature will allow to simulate the migration of tasks among the available cores, evaluating the impact of the data locality/remoteness on the performance of simple application workloads. One of the critical workload we will focus on is the one of real-time and multimedia applications.

7. Performance Analysis

In OMNeT++ there is a concept of signal. This is used for collecting statistical information out of a simulation run. With the help of signals, MCoreSim can seamlessly collect statistics for various events that occur during the simulation.

In the Communication Plane, useful performance metrics are the average and/or maximum packet latency, the bisection bandwidth and the network throughput. For the Processing Plane, useful performance metrics are the average/maximum execution time for a given Application, which may be varying depending on the latencies experienced due to the underlying topology.

8. Hardware Topology Modeling

In MCoreSim, the modelling of different hardware topologies is possible thanks to the Network Description Language (NED) embedded into the OMNeT++ framework. As an example, in Figure Figure 9 we report an excerpt of the NED description of a mesh topology for a tile-based many-core architecture.

```
network Mesh {
    parameters:
        int numX;
        int numY;
    types:
        channel CtoC extends ned.DatarateChannel {
            parameters:
                delay = 0.1ns;
                datarate = 50Gbps;
        }
    submodules:
        tile[numX*numY]: Tile {
            parameters:
                numX = numX;
                numY = numY;
         }
    connections:
        for i=0..numX-2, for j=0..numY-1 {
            tile[i+j*numX].eastIn <--  tile[(i+1)+j*numX].westOut;
            tile[i+j*numX].eastOut -->  tile[(i+1)+j*numX].westIn;
        }

        for i=0..numX-1, for j=0..numY-2 {
            tile[i+j*numX].southIn <--  tile[i+
(j+1)*numX].northOut;
            tile[i+j*numX].southOut -->  tile[i+
(j+1)*numX].northIn;
     }
}
```

Figure 9: Definition of a parametric mesh in MCoreSim.

As can be seen, the language allows for an easy creation of parametric topologies. In the shown example, we exploit this feature to have a generically configurable number of tiles along the X and Y axes of the mesh. These parameters are used to instantiate an array of Tile modules, which have been defined elsewhere in MCoreSim. Also, powerful iterative constructs allow for an easy connection of the close tiles with each other, according to their respective positioning (north/south, east/west).

References

[1] Baaij, C.P.R. and Kooijman, M. and Kuper, J. and Boeijink, W.A. and Gerards, M.E.T. (2010) **CλaSH: Structural Descriptions of Synchronous Hardware using Haskell**. In: *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD 2010), 1-3 Sep 2010, Lille, France*. pp. 714-721. IEEE Computer Society, Los Alamitos, CA, USA. ISBN 978-0-7695-4171-6. http://eprints.eemcs.utwente.nl/18376/

[2] Kuper, J. and Baaij, C. and Kooijman, M. and Gerards, M. (2010) **Exercises in architecture specifications using CλaSH**. In: *Proceedings of the 2010 Forum on Specification & Design Languages (FDL 2010), 14-16 Sep 2010, Southampton, UK*. pp. 178-183. ISSN 1636-9874.

[3] Jones, S.P. (Editor), **Haskell 98 language and libraries**, *Journal of Functional Programming*, 2003, vol. 13, no. 1.

[4] VHDL Language Reference Manual, IEEE Std. 1076-2008, 2008

[5] Verilog Hardware Description Languages, IEEE Std. 1365-2005, 2005.

[6] Tommaso Cucinotta, Giuseppe Lipari, Rui Aguiar, João Paulo Barraca, Bruno Santos, Jan Kuper, Christiaan Baaij, Lutz Schubert, Hans-Martin Kreuz (2010) **Definition of Future Requirements – Project Deliverable D5.2**, July 2010

[7] Bruno Santos, Javad Zarrin, João Paulo Barraca, Rui Aguiar, Christiaan Baaij, Jan Kuper, Dario Faggioli, Giuseppe Lipari, Tommaso Cucinota, Alexander Dragojevic, Vasileios Trigonakis, Vincent Gramoli, Daniel Rubio Boninalla, Lutz Schubert (2011) **First Implementation Set: Protocols – Project Deliverable D3.2**

[8] Tommaso Cucinotta, Giuseppe Lipari, Dario Faggioli, Fabio Checconi and Sunil Kumar, Rui Aguiar, João Paulo Barraca, Bruno Santos and Javad Zarrin, Jan Kuper and Christiaan Baaij, Lutz Schubert and Hans-Martin Kreuz, Vincent Gramoli (2010). **State of the Art – S(o)OS Project Deliverable D5.1** , July 2010

[9] Sunil Kumar, Tommaso Cucinotta, Giuseppe Lipari, **A Latency Simulator for Many-core Systems,** *in Proceedings of the 44th Annual Simulation Symposium (ANSS 2011), part of the Spring Simulation Multiconference (SpringSim'11).* Boston, April 2011.

[10] Michael J. Flynn, **Some Computer Organizations and Their Effectiveness**. IEEE Trans. Comput., 1972

APPENDIX A: FLYNNCATQUERY

```haskell
module FlynnCatQuery where

data PU  = PU { alus      :: [ALU]
              , registers :: [Register]
              }

data ALU = ALU { function    :: ALUfunction
               , aluType     :: ALUtype
               , dl          :: DataLine
               , il          :: InstructionLine
               , register    :: Register
               }

type ALUfunction = String
type ALUtype     = String

data DataPool = DataPool { size    :: Int
                         , poolId  :: String
                         }

data Register = Register { rId     :: String }

data DataLine = DL { pool    :: DataPool
                   , address :: Int
                   , width   :: Int
                   }

data Memory = Memory { memSize :: Int }

data InstructionLine = IL { ilId :: String }

data FlynnCategory = SISD | SIMD | MISD | MIMD
  deriving (Eq,Ord,Show)

superscalar :: PU -> Bool
superscalar (PU {alus=alus}) = (flynnCategory alus == MIMD) &&
                               shareAllRegisters alus

shareAllRegisters = allEqual . (map (rId . register))

flynnCategory :: [ALU] -> FlynnCategory
flynnCategory [a]                                              = SISD
flynnCategory alus | sameIL alus && dataShared alus            = SISD
                   | sameIL alus                               = SIMD
                   | not (sameIL alus) && dataShared alus      = MISD
                   | not (sameIL alus) && not (dataShared alus) = MIMD

sameIL = allEqual . (map (ilId . il))

dataShared alus = (allEqual $ map (poolId . pool . dl) alus) &&
                  (allEqual $ map (address . dl)       alus)
```

Appendix B: Object Model for Multi-Layer Resource Descriptions

```
#include <iostream>
using namespace std;
#ifndef _layers_H
#define _layers_H
#define Topology_Type_Def Ring;
#define Bandwidth_Left_Def_Value 10;
#define BandwidthRight_Def_Value 10;
#define Latency_Left_Def_Value 5;
#define Latency_Righ_Left_Def_Value 5;
#endif  /* _layers_H */
enum topologyType {Ring, Bus, Star, Mesh, Hybrid};
enum architectureType {SISD,SIMD,MISD,MIMD};
class topologyLayer {
    topologyType topology;
public:
    ~topologyLayer(){topology= Topology_Type_Def}
};
class nodeLayer {
private:
    int NodeLeftID;
    int NodeRightID;
    double BandwidthLeft ;
    double BandwidthRight ;
    double LatencyLeft;
    double LatencyRight;
public:
    int nodeId;
    double rated_bus_speed;
    nodeLayer();
    nodeLayer(double,double,double,double);
    void setNodeLeftID(int lid){NodeLeftID=lid;};
    void setNodeRightID(int rid){NodeRightID=rid;};
    int getNodeLeftID(){return NodeLeftID;};
    int getNodeRightID(){return NodeRightID;};
    void setBandwidthLeft(double bwl){BandwidthLeft=bwl;};
    void setBandwidthRight(double bwr){BandwidthRight=bwr;};
    double getBandwidthLeft(){return BandwidthLeft;};
    double getBandwidthRight(){return BandwidthRight;};
    void setLatencyLeft(double lal){ LatencyLeft=lal;};
    void setLatencyRight(double lar){LatencyRight=lar;};
    double getLatencyLeft(){return  LatencyLeft;};
    double getLatencyRight(){return LatencyRight;};
    void configNodeLayer(int,double,double,int,double,double);
};
nodeLayer::nodeLayer(){
BandwidthLeft=10; BandwidthRight=10; LatencyLeft=5; LatencyRight=5;
}
nodeLayer::nodeLayer(double BwL,double BwR,double LaL,double LaR){
BandwidthLeft=BwL; BandwidthRight=BwR; LatencyLeft=LaL; LatencyRight=LaR;
}
 void nodeLayer::configNodeLayer(int lid, double bwl, double lal, int rid,
double bwr, double lar){
NodeLeftID=lid;
BandwidthLeft=bwl;
LatencyLeft=lal;
NodeRightID=rid;
BandwidthRight=bwr;
LatencyRight=lar;
}
class CMPLayer : public nodeLayer {
public:
    int CPU_ID;
    architectureType Flynn_taxonomy;
    double DRAM_size;
    struct L23 {
      double cache_size;
      double cache_line;
    };
    union L2_shared_cache_type { bool has_shared_L2; struct L23 data; }
L2_shared;
    union L3_global_cache_type { bool has_shared_L3; struct L23 data; }
L3_global;
};
class coreLayer : public CMPLayer {
public:
    double interconnection_bus;
    double * latency_map_cores;
};
class Core : public coreLayer {
public:
    int coreID;
    double frequency;
    double L1_Instruction_cache_size;
    double L1_data_cache_size;
    double mips;
    double L1_latency;
    struct L2 {
      double cache_size;
      double latency;
    };
    union L2_private_cache_type { bool has_private_L2; struct L2 data; }
L2_private;
    double memory_latency;
    double l2_shared_latency;
    double l3_global_latency;
};
class CMP : public CMPLayer{
public:
    int no_of_cores;
    Core * cores;
};
class MPNode : public nodeLayer {
public:
    int no_of_CMPs;
    double * latency_cmp_cmp;
    CMP * cmps;
};
```

APPENDIX C: SIMPLE CPU

```
type CpuState = State (Vector D4 Int16)

cpu :: CpuState
  -> (Int16, Opcode, Vector D4 (Index D7, Index D7))
  -> (CpuState, Int16)

fu op inputs (a_0,a_1) =
  op (inputs!a_0) (inputs!a_1)

fu_1 = fu add
fu_2 = fu sub
fu_3 = fu mul

data Opcode = ShiftR | Xor | Equal

multiop ShiftR = shiftR
multiop Xor    = xor
multiop Equal = \a b -> if a == b then 1 else 0

fu_0 c = fu (multiop c)

cpu (State s) (x,opc,addrs) = (State s', out)
  where
    inputs = x +> 0 +> 1 +> s
    s'     = (fu_0 opc inputs (addrs!0)) +>
             (fu_1     inputs (addrs!1)) +>
             (fu_2     inputs (addrs!2)) +>
             (fu_3     inputs (addrs!3)) +>
             empty
    out    = last s
```