HARPO/L: A LANGUAGE FOR HARDWARE/SOFTWARE CODESIGN.

Theodore Norvell, Xiangwen Li, Dianyong Zhang

Electrical and Computer Engineering
Memorial University

Md. Ashraful Tuhin Alam

Computer Science
University of Calgary

ABSTRACT

We present the main features of a programming language designed to be applicable for parallel computing on a wide range of platforms, including a range of programmable hardware. Our language supports an object oriented, yet statically instantiated style of programming; parallel processing with communication and synchronization by rendezvous; and reusability via generic classes, interfaces, and connections between objects.

1. INTRODUCTION

Advances in digital hardware densities make it feasible to implement large portions of applications on Application Specific Integrated Circuits (ASIC) or programmable hardware such as Field Programmable Gate Arrays (FPGA) and Coarse Grained Reconfigurable Architectures (CGRA). However portions of applications not needing the additional speed of reconfigurable hardware will continue to be implemented using more space-efficient microprocessors (μP), which could be on a different chips, on the same chip, or even virtual processors build on a programmable hardware substrate. We could characterize this split as one of 'hardware' vs. 'software,' but in reality we are just talking about various points in a complex space of programmable hardware.

In current practice different implementation technologies use different programming or hardware description languages. Thus it is necessary to decide on implementation technology early in the design process. If the design is to use multiple implementation technologies, one must decide on how the design will be split among these technologies early in the design process. It is then difficult to migrate future generations of a product to new technologies or to reuse design components where the implementation technolgy is different.

It would be desirable to use a programming language that can be efficiently and effectively targeted toward a variety of implementation platforms. In this way, how a system is distributed among implementation technologies can be decided after the system's behaviour is designed. Indeed different members of a product family might employ different distributions.

This paper explains the design of one such programming language. Our requirements include:

- The language must support good engineering practices such as decomposition and information hiding.
- The language must support reusability and 'programming in the large'.
- The language must support parallel, concurrent, and distributed computing.
- The language must suitable for hardware implementation. Specifically it should not use dynamic memory allocation; all addresses and communication paths should be statically resolvable.
- The language must be simple and have a clearly defined semantics.

The remainder of this paper discusses how our language, HARPO/L (for HARdware Parallel Objects Language), achieves these objectives.

2. STRUCTURE

2.1. Classes and objects

A HARPO/L program is a system of objects, each a member of a class. Classes can be instantiated a number of times to create objects. Superficially this resembles an object-oriented programming language in the sense of Simula-67 and its successors. However there is an important difference. In order to meet the requirements that programs be implementable directly in hardware, we instantiate all objects statically, that is at compile time. Furthermore, to meet the requirement that references and communication paths be statically resolved there is no assignment of object references.

Here (schematically) is a simple HARPO/L program consisting of two classes and two objects.

```
(class Consumer
    public proc put( in i : int8 )
```

This work was supported by the National Science and Engineering Research Council and Memorial University's graduate fellowships program.

```
    ...
class)
(class Producer ...c.put(i)... class)
obj c := new Consumer()
obj p := new Producer()
```

Although the word new may connote heap allocation at run-time, as in Java or C++, in HARPO/L all instantiation is static.

Each instance of class Producer, above, is bound to the object c. This makes this class difficulty to reuse. Instead we can make the binding during instantiation. The slightly improved program is(class Consumer ... class)

```
(class Consumer
    public proc put( in i : int8 )
    ...
class)
(class Producer
    constructor( obj c : Consumer )
    ...c.put(i)...
class)
obj c0 := new Consumer()
obj p0 := new Producer( c0 )
obj c1 := new Consumer()
obj p1 := new Producer( c1 )
```

In this program, producer p0 communicates with consumer c0 while producer p1 communicates with consumer c1. It should be empathized that there is no reassignment of constructor parameters such as c, thus for each object, the communication path is fixed at compile time.

We can further improve the reusability of the Producer class by use of an interface. The concept closely parallels the interfaces of UML or Java, as illustrated below.

```
(interface ConsumerIntf
    public proc put( in i : int8 )
interface)
(class Consumer0 implements ConsumerIntf
    ...
class)
(class Consumer1 implements ConsumerIntf
    ...
class)
(class Producer
    constructor( obj c : ConsumerIntf )
    ...c.put(i)...
class)
obj c0 := new Consumer0()
obj p0 := new Producer( c0 )
obj c1 := new Consumer1()
obj p1 := new Producer( c1 )
```

In this program, the same producer class is used to communicate with objects of different consumer classes.

2.2. Fields

Objects form a part-whole hierarchy by means of fields which may be public or private.

2.3. Arrays

Arrays of objects can be used to represent repetitive structures. For example

```
obj source := new Source(stage(0))
obj sink := new Sink()
const N := 10
obj stage : Stage[N] :=
    (for i:N do
        (if i=N-1 then new Stage( sink )
        else new Stage( stage(i+1) ) if) for)
```

constructs an array of 10 Stage objects, each connecting to the next, except the last which connects to the sink object.

2.4. Trees

To some extent other structures can be created via recursion. For example a tree of nodes can be constructed using fields

```
(interface Node ... )

(class Leaf implements Node ... )
(class Branch implements Node
    constructor( in i : int32 )
    private obj leftChild : Node :=
        (if i=0 then new Leaf()
        else new Branch(i-1) if)
    private obj rightChild : Node :=
        (if i=0 then new Leaf()
        else new Branch(i-1) if)
    ...
class)
obj tree := new Branch(5)
```

However, whereas array members may be dynamically selected using array indexing, there is no simple way to represent a pointer to a tree node.

```
(class BoundedBuffer{ type T extends Primitive }
    public proc put( in i : T )
    public proc get( out i : T )
    constructor( in capacity : int32 )
    private obj buffer : T[ capacity ] :=
        (for capacity do new T())
    private obj head : int32 := 0
    private obj size : int32 := 0
    (thread
      (wh true
        (accept
          put( in i : T ) when size < capacity
            buffer[ (head + size) mod capacity ] := i
            size := size + 1
        |
          get( out i : T ) when size > 0
            i := buffer[ head ]
            head := (head+1) mod capacity
            size := size - 1 ) ) ) )
```

Fig. 1. A full example

3. BEHAVIOUR

3.1. Threads

All objects work concurrently. Each object contains zero or more threads which execute a fairly standard repertoire of commands: assignment (of primitive types only), alternation, repetition, sequential composition, as well as parallel composition and parallel loops. The semantics of parallelism is that of interleaving of atomic actions (reads, writes, and local computations).

Interobject communication is by shared memory or by rendezvous.

3.2. Rendezvous

Rendezvous [1, 2] is a mechanism for thread synchronization and communication pioneered in the Ada programming language. From the point of view of a client object, rendezvous looks much like a method call in an language such as Java or C++. In the client thread we might have

```
c.put(i)
```

which calls method put in server object c with parameter i. The method is declared as public in the server class and is implemented by an accept statement within one of that class's threads.

```
(class Consumer
    public proc put( in i : int8 )
    (thread ...
        (accept put( in i : int8 )
            ... // Do something with i
        accept)
        ...
    thread)
class)
```

The rendezvous mechanism in HARPO/L allows servers to offer clients a selection of services and allows guarding (i.e. selective enabling) of services. A classic example illustrating both these facilities is that of a bounded buffer, shown in Figure 1.

Although rendezvous is the language primitive, it can be used to model other communication and synchronization mechanism such as channels and semaphores. Library classes for these mechanisms could be implemented natively.

4. GENERICS

To support software reusability, classes can be generic over type parameters. As with Java [3, 4], generic classes are based on F-bounded polymorphism [5]. Generic classes and interfaces have one or more type parameters which are bounded by super-type, for example

```
(interface GenericConsumerIntf{ type T
                                 extends Primitive }
    public proc put( in i : T )
interface)
(class GenericProducer{ type T extends Primitive }
    constructor( obj c : GenericConsumerIntf{T} )
    ...
class)
(class Int8Consumer
    implements GenericConsumerIntf{int8}
    ...
class)
obj c0 := new Int8Consumer()
obj p0 := new GenericProducer{int8}( c0 )
```

5. SPECIFICATION AND IMPLEMENTATION

Our group has been investigating several aspects of implementation of HARPO/L and like languages, especially with an eye to implementation on CGRAs. This work is or will be reported elsewhere and here we present only the barest skeleton.

- Detailed specification of the syntax, type system, and semantics of the language. The type system is presented as a set of type inference rules. The operational

semantics is presented as a translation to coloured Petri nets.

- Compiler front end. The front end implements parsing, type checking, and object instantiation.
- Translation to C. The C back end converts the data structure produced by the front end to C code employing calls to the Pthreads library.
- Intermediate representations for explicitly parallel code. We have been investigating representations based on Concurrent Single Static Assignment and translation to executable data-flow graphs [6, 7, 8].
- Optimization of parallel code [8].
- Scheduling, placement, and routing for implementation on CGRAs. We have been investigating scheduling, placement and routing methods based on graph embedding [9].

6. CONCLUSION

To end we review our goals and look at how they are achieved:

- The language must support good engineering practices such as decomposition and information hiding.
  - We employ an object based approach
- The language must support reusability and 'programming in the large'.
  - Generic classes, interfaces and constructor parameters allow reusability. Constructor parameters support architectural level design.
- The language must support parallel, concurrent, and distributed computing.
  - The language is explicitly parallel. Compilers can determine what memory needs to be accessible by which objects, as there are no pointers, which allows distribution of objects according to their sharing of memory. The communication primitive is suitable for both shared memory and distributed computing.
- The language must suitable for hardware implementation. Specifically it should not use dynamic memory allocation all addresses and communication paths should be statically resolvable.
  - The language replaces dynamic memory allocation with a flexible, statically executed sublanguage for object instantiation and connection.
- The language must be simple and have a clearly defined semantics.
  - We have kept the language features simple and consistent. We have formally defined both the static semantics (using inference rules) and the dynamic semantics (using coloured Petri nets).

7. REFERENCES

[1] Jean D. Ichbiah, Bernd Krieg-Brueckner, Brian A. Wichmann, John G. P. Barnes, Olivier Roubine, and Jean-Claude Heliard, "Rationale for the design of the Ada programming language," *SIGPLAN Not.*, vol. 14, no. 6b, pp. 1–261, 1979.

[2] Gregory R. Andrews, *Foundations of Multithreaded, Parallel, and distributed programming*, Addison Wesley Longman, 2000.

[3] Martin Odersky and Philip Wadler, "Pizza into java: translating theory into practice," in *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 1997, pp. 146–159, ACM.

[4] Gilad Bracha, Norman Cohen, Cristian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutmire, Kreten Thorup, and Philip Wadler, "Adding generics to the java programming language: Participant draft specification," 2001.

[5] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell, "F-bounded polymorphism for object-oriented programming," in *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, New York, NY, USA, 1989, pp. 273–280, ACM.

[6] Jaejin Lee, David A. Padua, and Samuel P. Midkiff, "Basic compiler algorithms for parallel programs," in *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 1999, pp. 1–12, ACM.

[7] John Teifel and Rajit Manohar, "Static tokens: Using dataflow to automate concurrent pipeline synthesis," in *Proceedings 10th International Symposium on Asynchronous Circuits and Systems, 2004.*, 2004, pp. 17– –27.

[8] Dianyong Zhang, "Intermediate representations for compiling parallel languages to cgras," M.S. thesis, Memorial Universty of Newfoundland, 2008, Check title.

[9] Mohammed Ashraful Alam Tuhin, "Compiling parallel applications to coarse-grained reconfigurable architectures," M.S. thesis, Memorial University of Newfoundland, 2007.