Architecture for Object-Oriented Programming Model

Nikola Markovic, Ruben Gonzalez, Osman Unsal, Mateo Valero, Adrian Cristal

nikola.markovic@bsc.es, gonzalez@ac.upc.edu, osman.unsal@bsc.es,

mateo.valero@bsc.es, adrian.cristal@bsc.es

Abstract

Current mainstream architectures have ISAs that are not able to maintain all the information provided by the application programmer using a high level programming language. Typically, the information that is lost in compiling to a low-level ISA is related to parallelism and speculation [14]. For example some loops are typically expressed as parallel loops by the programmer but later the processor is not able to determine this level of parallelism; conditional execution might apply control independent execution that at execution time is basically impossible to detect; function and object-level parallelism is lost when code is transformed into a low-level ISA that is oblivious to programmer intentions and high-level programming structures.

Object Oriented Programming Languages are arguably the most successful programming medium because they help the programmer to use well-known practices about data distribution through operations related with the associated data. Therefore object oriented models express data/execution locality more naturally and in an efficient manner. Other OO software mechanisms such as derivation and polymorphism further help the programmer to exploit locality better. Once object oriented programs have been compiled then all information about data/execution locality is completely lost in current assembly code (ISA code). Maintaining this information until runtime is crucial to improve locality and security. Finally, Object Oriented Programming Models maintain the idea of memory (data memory) far from the programmer. These are all desirable qualities that is mostly lost in the compilation to a low-level ISA that is oblivious to the Object-Oriented Programming model.

This report considers implementing the Object Oriented (OO) Programming Model directly in the hardware to serve as a base to exploit object/level parallelism, speculation and heterogeneous computing. Towards this goal, we present new computer architecture that implements the OO Programming Models. All its hardware structures are objects and its Instruction Set directly utilizes objects hiding totally the notion of memory and other complex hardware structures. It also maintains all high-level programming language information until execution time. This enables efficient extraction of available parallelism in OO serial or parallel code at execution time with minimal compiler support. We will demonstrate the potential of this novel computer architecture through several examples.

1 Introduction

Current mainstream architectures have ISAs that are not able to maintain all the information provided by the application programmer using a high level programming language. Typically, the information that is lost in compiling to a low-level ISA is related to parallelism and speculation [14]. For example some loops are typically expressed as parallel loops by the programmer but later the processor is not able to determine this level of parallelism; conditional execution might apply control independent execution that at execution time is basically impossible to detect; function and object-level parallelism is lost when code is transformed into a low-level ISA that is oblivious to programmer intentions and high-level programming structures.

Object Oriented Programming Languages are arguably the most successful programming medium because they help the programmer to use well-known practices about data distribution through operations related with the associated data. Therefore object oriented models express data/execution locality more naturally and in an efficient manner. Other OO software mechanisms such as derivation and polymorphism further help the programmer to exploit locality better. Once object oriented programs have been compiled then all information about data/execution locality is completely lost in current assembly code (ISA code). Maintaining this information until runtime is crucial to improve locality and security.

Finally, Object Oriented Programming Models maintain the idea of memory (data memory) far from programmer. However, after compilation to low-level ISA all object data are bound to memory locations. This low-level memory management limits the processor in a lot of speculative scenarios:

- Some potentially parallel code is executed sequentially because the compiler is not able to determine the speculation risk in the intermediate code and produces overly conservative code. During runtime data references have been transformed to memory addresses and it is impossible to guarantee that the high-level code which has been broken into memory references can be executed in parallel.
- High-level code can exhibit control independent execution in a switch-condition which only modifies one object that it totally independent from others. If the object structure is maintained until execution time, then it will be easier to detect and apply control independence.

A solution, we believe, is to allow programmers to continue working in the most successful programming model to date – the Object-Oriented one, and to move the complexity of parallelization for multiple homogeneous or heterogeneous cores to hardware. As way of accomplishing that, we propose new and unique computer architecture, leveraging Object-Oriented (OO) programming model. It is presented as an architecture that extends software concepts into hardware. We call this architecture Object-Oriented Computer Architecture (OOCA). OOCA is an abstract hardware layer based on asynchronous communication and dataflow with hidden memory. It “short-circuits” the OO programming model directly to hardware by representing every system structure (data, functions, conditions, etc.) as an object on an OOCA Processor (OP). OOCA Processor hides memory management from OOCA ISA.

By not having notion of memory and preserving Programmer’s information from Object-Oriented Code until execution time OOCA model improves locality data/execution, performance for speculative scenarios [1, 3, 6, 9] and level of parallelism (Object Level Parallelism). Object Level Parallelism combines ILP [10, 11], dataflow [2] and method level parallelism (MLP) [14], trying to emphasize on their good sides and suppressing bad.

OOCA model provides asynchronous, control independent and parallel execution model which preserves sequential view of program. Methods are executing asynchronously, in its own context with out any other interferences, where input/output parameters and results are sent in asynchronous ways. It opens possibilities for dynamic optimization, which was not feasible in compile time.

Maintaining Object-Oriented Code information until execution time will open a window to execute operation in heterogeneous target hardware: OOO processor, multiprocessor, embedded processors, FPGA and other alternatives.

This paper describes the Object-Oriented Computer Architecture (OOCA) and its execution model benefits. Section 1 provides an overview of the OOCA. OOCA architecture is described in Section 3. Microarchitecture view of OOCA Processor follows in Section 4. An explanation of OOCA execution model and its benefits is given in Section 5. In Section 6, we compare OOCA with other architectures. In Section 7, we summarize future work and offer concluding remarks.

2 OOCA overview

2.1 OOCA architecture basics

Figure 1 depicts the Object-Oriented Computer Architecture system stack. One major advantage of Object-Oriented Computer Architecture (OOCA) is the transparent way of exposing hardware to Object Oriented Language, without any impact on the programming model. Compiler translates the code written in Object Oriented Language into OOCA Language which preserves OO semantics. Section 3.1 provides a more detailed description of OOCA Language.

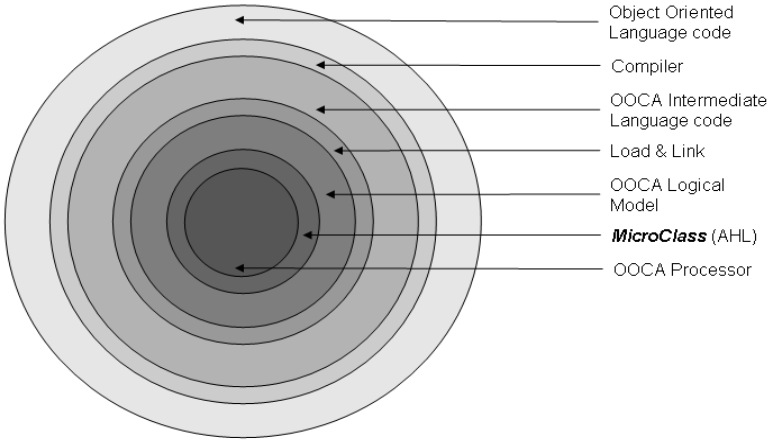


Figure 1: Object-Oriented Computer Architecture model view

The loader is responsible for loading code from OOCA Language into OOCA processor. Descriptor context objects of classes, methods, loops, etc., are created by Loader based on information provided from OOCA Language. *Descriptor context* is an object which contains information about properties the actual *instance contexts* of objects are going to have after creation. *Instance context objects* are created at execution time based on information from *descriptor context object*.

All objects (*descriptor and instance contexts*) are derived from OOCA base class, ***MicroClass,*** and have unique reference, hardware identifier (note that hereafter in document word reference relates to this definition). *MicroClass*, which is the OOCA Abstract Hardware Layer, is responsible for managing the objects and executing OOCA primitives, through its methods, ***MicroMethods***. These OOCA primitives (a.k.a. ISA) enable the hardware to work directly with objects. OOCA primitives are oblivious to memory implementation issues such as addresses and memory management. An explanation of OOCA Abstract Hardware Layer (*MicroClass*) is given in Section 3.3.

OOCA hardware is composed of homogeneous or heterogeneous OOCA Processors (OP). However not having a global concept of memory in the traditional sense can significantly simplify OP design. In addition, we can have custom OP's that are optimized for certain types of objects using exactly the same primitives. Instance of an object will reside on OP. All objects have symmetrical access to

hardware through *MicroClass.* Each object will be accessed only through methods, *MicroMethods.* The way the object is stored is a design issue of the OP.

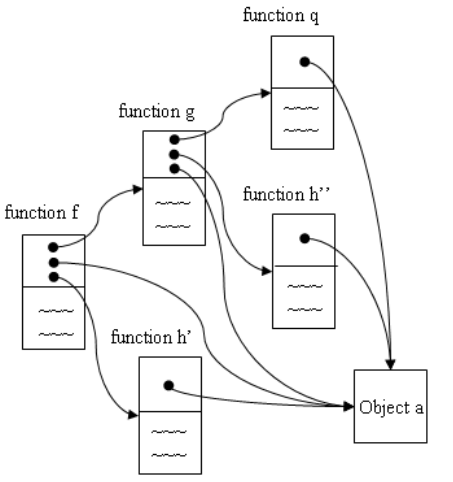
2.2 OOCA execution philosophy

To illustrate the power of OOCA execution model consider the simple example from Figure 2 a).While executing sequential program from Figure 2 a), OOCA processor creates logical model of *instance context object* shown on Figure 2 b).When OOCA Processor starts with execution of *instance context object* of `function f` OOCA primitives. It will create *instance context object* of `data a` and *instance context objects* of `functions g` and `h'`. Execution of *instance context objects* of `functions g` and `h'` OOCA primitives can start immediately on other OOCA processor cores and go in parallel with *instance context object* of `function f` execution. While OOCA primitives of *instance context object* of `function g` are executed *instance context objects* of `functions q` and `h''` are going to be created. Execution of their sets of OOCA primitives can start in parallel with *instance context objects* of `functions f`, `g` and `h'`, if there are available cores on OOCA processor.

```
...
class C {
  ...
}
...
f(...) {
  ...
  C *a = ...;
  g(a);
  h(a);
  ...
}
h(C * a) {
  ...
  c =+ *a;
  ...
}
...
```

```
...
g(C * a) {
  ...
  h(a);
  q(a);
  ...
}
...
q(C * a) {
  ...
  *a =+ b;
  ...
}
...
```

a) Simple example in C++



b) OOCA logical model of *instance context objects*

Figure 2: An example Code segment

If instance context objects of `function h'` during its execution uses a temporally wrong value of *instance context objects* of `data a`, and when instance context objects of `function q` sets new values in *instance context objects* of `data a`, the mechanism of data versioning [12] will be started. It will determine that *instance context objects* of `function h'` has used wrong values of *instance context objects* of `data a` and start "chain" re-execution. This will re-execute only those OOCA primitives that actually need to be re-executed, not all OOCA primitives that have been executed after wrong value of variable `a` has been used.

3 OOCA Architecture view

Using the quicksort algorithm on list shown in Figure 3, in this section we are explaining OOCA architecture. In Section 3.1 we present OOCA Language. OOCA Logical Model is explained in Section 3.2. Abstract Hardware Layer (*MicroClass*) is described in Section 3.3.

```
1.  class Elem{
2.    public:
3.      Elem(int number): number(number) {}
4.      string& getNumber(){return number;}
5.      void setNumber(int& i) {number = i;}
6.      bool compare(Elem * e){
7.        if (getNumbre() <= e->getNumber()) return true;
8.        else return false;
9.      }
10.   private:
11.     int number;
12. };
13. void quickSort(List<Elem*> *lst){
14.   if (lst->empty()) return;
15.   Elem *l = new list <Elem*>(), *r = new list <Elem*>();
16.   Elem* pivot = partition(lst, l, r);
17.   quickSort(l);
18.   quickSort(r);
19.   l->insert(pivot);
20.   l->insert( r);
21.   lst = l;
22. }
```

```
23. Elem* partition(List<Elem*> *lst, List<Elem*> *l,
                    List<Elem*> *r){
24.   Elem* pivotElem = lst->getFirst();
25.   lst->eraseFirst();
26.   while(!lst->empty()){
27.     Elem* e = lst->getFirst();
28.     if(e->compare(pivotElem)) l->insert(e);
29.     else r->insert(e);
30.     lst->eraseFirst();
31.   }
32.   return pivotElem;
33. }
```

Figure 3: An example of quicksort algorithm on the list

3.1 OOCA Language

OOCA Language code describes all classes, loops, conditions and methods. It also contains set of predefined STANDARD IDs to refer to well known classes (integer, float, collection, ...) and methods (add, mull, div, ...) of OOCA. These well known objects are explained in Section 3.2.

The Loader which is a special part of OOCA software/hardware creates *description contexts* based on information from OOCA Intermediate Language code file. *Objects, instances contexts* of classes and methods, are created in execution time based on information from *description contexts*,

3.1.1 Description of classes

Each class from Object Oriented Language is going to be translated to OOCA language. After translation, description of each class in OOCA language format will contain information about its fields, description of class methods and properties of class (Figure 4). This information is required by Loader to create a data structure which forms the *description context* of the observed class.

```
/*OOCA Language description*/

class Elem
  OOCA_atributes
    class int <number> [create]
  OOCA_functiones
    constructor
      OOCA_arguments
        class int <number>
      OOCA_temporal
        function assign r1
      OOCA_return
        class Elem <ret> [create]
      OOCA_code 11
  ...
    setNumber
      OOCA_arguments
        class int <i> [nochange]
      OOCA_temporal
        function assign r1
      OOCA_caller
        class Elem <this>
      OOCA_code 13
    ...
/*OOCA Language ISA code*/
...
13:  begin
       3 call(assign, r1)
       1 send(i, r1)
       2 send(this.number, r1)
     end
...
```

Figure 4: Part of OOCA Language for class Elem

3.1.2 Description of executable blocks

Every executable block consists of all the methods as well as any loop or condition. Each such block is represented with its associated description and a set of OOCA primitives (setNumber in Figure

4). The associated description can contain information about the block properties, local variables, arguments, temporary fields, return value, references to description of other executable blocks and reference to a set of OOCA primitives Based on this information from the OOCA Intermediate Language code, the Loader is able to create *description context of executable block* of observed method, condition, loop, etc.

3.2 OOCA Logical Model

OOCA Logical Model represents an abstract logical view of *descriptor and instance contexts* and their relations. This model contains all information provided by the programmer in Object Oriented Language. Every object context has its appropriate hardware representation on the Object Processors. OOCA has several different kinds of *descriptor contexts* which can generally be divided into two groups.

One group consists of *descriptor contexts* of well known classes (integer, string, collection, ...) and methods (add, div, mull, print, …). These descriptor contexts are embedded into system hardware. They have pre-assigned STANDARD IDs which are used for specifying these descriptor contexts in OOCA language code file.

The other group consists of *descriptor contexts* of classes and methods defined by user in Object oriented Language. These contexts are created by Loader using the information's from OOCA intermediate language code file. There are two different *descriptor contexts* in this group: *descriptor context* of class, descriptor context of executable block.

3.2.1 Descriptor and instance context of class

Descriptor context of class contains all necessary information, properties, references to descriptor contexts of fields and description of methods, to create *instance context object* of observed class, as it is shown on Figure 5 for class Elem from Figure 4.

Instance context of a class represents specific instance of a class in execution time. It is created based on the information from appropriate *description context of class*. One *instance context* may obtain

many versions of actual data of observed class (data object). Through the mechanism of data versioning, explained in detail later in Section 5.3.1 instance context of class ensures that each instance context of executable block which is using that instance context of class gets the correct version of data.

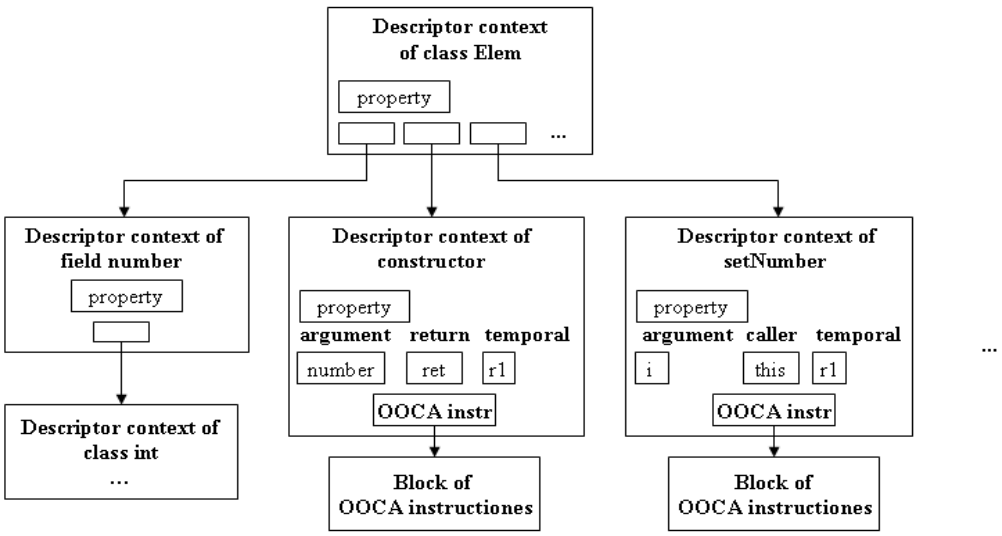


Figure 5: OOCA Logical Model for descriptor context of class Elem

3.2.2 Descriptor and instance context of executable block

Descriptor context of executable block contains the properties of that block, fields for describing the arguments, local variables, temporal variables (for storing temporal products of execution), return value, reference to a set of OOCA primitives, and may contain references to other executable blocks. *Descriptor context of iterators and conditions* are a special form of executable block which consist of the properties, fields and references. This is shown in.Figure 5 for the method setNumber from.Figure 4 *Instance context of executable blocks* are created from *Descriptor context of executable block.*

3.3 *MicroClass* (Abstract Hardware Layer)

MicroClass represents an Abstract Hardware Layer. It consists of a set of methods, ***MicroMethods,*** which are implemented in hardware. *MicroClass* is responsible for managing the context

objects, executing OOCA primitives and resource allocation, through its *MicroMethods*. All context objects have symmetrical access to hardware through *MicroClass*. The memory management is not seen by the *MicroMethods***.** Each object will be accessed only through *MicroMethods*.

All *MicroMethods* (primitives) have two common parameters:

- Hints given by compiler/user. All information generated by compiler that can help runtime. Hints can be avoided. For example: hint *try to execute in parallel* is to give opportunity of parallelism but OP later can execute sequentially.

- Requirements given by compiler/user. All that is required by semantic at runtime. For example: the requirement *do not speculate* must be adhered to by the OP.

MicroClass methods can be groups as: context primitives and executable primitives.

3.3.1 Context Primitives

Context primitives are in service of managing *descriptor context objects*. They are used for loading, registering, destroying and accessing to the fields of descriptor contexts and can not be used out side *descriptor context object*.

3.3.2 Executable Primitives

Executable primitives are used for creating and destroying of instance context objects based on information provided from descriptor context objects, accessing to fields of instance context objects and communication between instance context objects on execution time. Basic executable primitives are: call, send, monoop, douop, threeop create and destroy.

Call primitive is used for creating instance contexts of executable blocks from descriptor contexts of executable blocks and well known methods (add, mull, print, etc.) and preparing it for execution. They are completely independent one from each other, inside and between blocks of instructions, and can be executed in any order if the compiler has not pre-specified some requirement. The send primitive is used

for sending references of instance context objects of classes to instance context objects of executable blocks, iterators and conditions. All sends are also completely independent one form each other, inside and between blocks of instructions, and can be executed in any order. Send primitive can not be executed in case that one of the instance contexts, that send is using reference to, is not created jet. These two primitives are part of OOCA ISA and allow asynchronous method calls and lazy execution; based on previously mentioned properties of call and send primitives.

Monoop (1-op), duoop (2-op) and threeop (3-op) are special purpose operations, intended for carrying out frequent functions, such as addition, subtraction, etc.., on the same type of operands. Monoop is intended for executing primitives with one operand (a++ etc...); douop executes primitives with two operands (a+=c etc...); while threeop executes primitives with three operands (a=b+c etc...).

Create primitive is used for creating instance contexts of objects from descriptor contexts of objects. It returns reference to created object. Destroy primitive is used for destroying instance contexts of object created from descriptor contexts of object. *Instance context* destruction is managed by OOCA processor garbage collector.

Control primitives make special part of executable primitives. They are used for managing the execution of *instance context of iterators and conditions*, which represent control structures from Object Oriented Languages (if then else, for , foreach , forall , while , repeat, …).

4 OOCA Processor microarchitecture (Open HW Layer)

OOCA Processors are basic processing element of OOCA architecture. OOCA Processor can, either be processor designed to execute OOCA primitives *(Specialized OOCA Processor)*, or contains an execution unit (EU) and knows how to control and execute OOCA primitives, *MicroMethods*, on EU (Figure 6). OOCA Processor hides memory system from OOCA primitives. Instances of *descriptor contexts* and *instance contexts* of all objects will reside on OOCA Processor. Execution Unit of General

Purpose Processor can be any existing processor; OOO (Out-of-Order processor), Vector processor, FPGR, multiprocessor, etc.

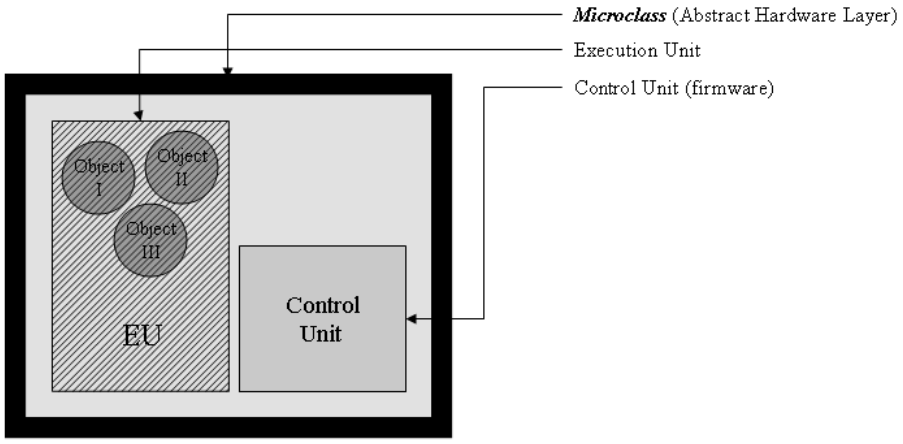


Figure 6: OOCA processor

This flexible hardware model allows easy incorporation of any kind of acceleration units inside of execution unit. Depending on OOCA Processor actual implementation, it will be able to exploit more or less of OOCA advantages (speculation, object versioning, etc.). Open Hardware Layer can contain one or more heterogeneous or homogeneous OOCA Processors (Figure 7).

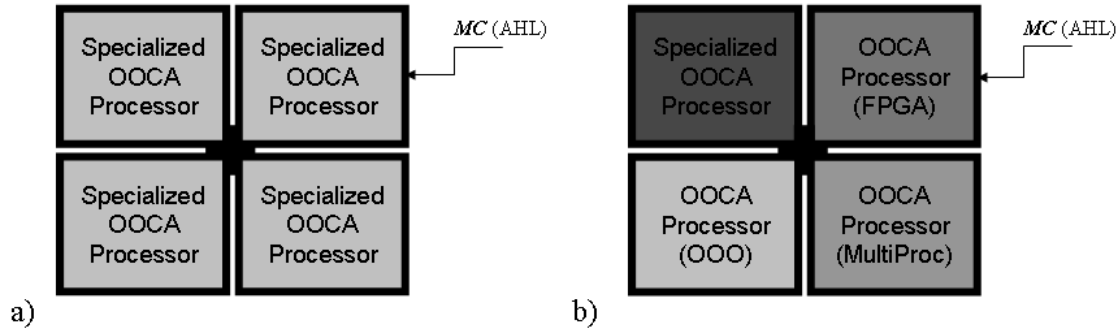


Figure 7: Homogeneous & Heterogeneous OOCA Machine

5 OOCA Execution Model

In this Section we represent OOCA execution model and its three main benefits; better locality (all information are inside object), control independence and efficient recovery mechanism ("chain re-execution") and object level parallelism. We show how blocks of OOCA primitives can be executed

independently one from each other in Section 5.1, how possible parallelism can be extracted using Object Level Parallelism in Section 5.2 and how to incorporate control independent Object-Oriented execution and data versioning in Section 5.3.

5.1 Independent blocks of OOCA primitives

As mentioned in section 3.3.2; all *call* primitives are independent one from another and can be executed in any order, unless a requirement has been set by compiler for them to be performed in a certain order. They prepare and execute *instance context of executable block*. The *instance context of executable block* will be executed when it gets a core of the OOCA Processor. The send primitive dispatches references of *instance context of data objects* to *instance context of executable blocks. They* are completely independent one from another and can not be initiated just in the case that some of the *instance contexts* haven´t been created yet. Primitives monoop, duoop and threeop prepare *instance context of executable block* for execution, and send arguments to it in the same time, and therefore can not be performed in case that *instance context* some argument has not been created yet. They are also independent one from another unless the compiler has set the requirement that they have to be carried out in a certain order.

In each block of OOCA primitives we have two orders: execution order and commit order. Execution order is the actual dataflow order of OOCA primitives inside the block while commit order of OOCA primitives inside the block is marked with the sequence number denoting intra-block commit order of primitives.

Mechanism of data versioning, explained latter in Section 5.3.1 allows different executable blocks to be executed independently one from another. In Figure 8 we show sets of OOCA ISA primitives for functions quicksort and partition. Each OOCA primitive, inside block of OOCA primitives, has associated ***sequence number*** by compiler which represents OOCA primitives sequential commit order. Based on the previously mentioned facts two sets of OOCA primitives can be executed independently.

```
/*OOCA ISA code for partition function*/
begin
  2  call(lst.getFirst, r1)
  1  send(pivotElem, r1.ret)
  3  call(lst.eraseFirst, r2)
  8  call(while_iterator, r3)
  4  send(lst, r3.lst)
  5  send(l, r3.l)
  6  send(r, r3.r)
  7  send(pivotElem, r9.pivotElem)
  9  douop(assign, pivotElem, ret)
End

/*OOCA ISA code for quicksort function*/
begin
  2  call(empty_list_if, r1)
  1  send(lst, r1.lst)
  4  call(List{Elem}.constructor, r2)
  3  send(l, r2.ret)
  6  call(List{Elem}.constructor, r3)
  5  send(r, r3.ret)
 10  call(partition, r4)
  6  send(lst, r4.lst)
  7  send(l, r4.l)
  8  send(r, r4.r)
  9  send(pivot, r4.ret)
 12  call(quicksort, r5)
 11  send(l, r5.l)
 14  call(quicksort, r6)
 13  send(r, r6.l)
 16  call(l.insert, r7)
 15  send(pivot, r7.caller)
 18  call(l.insert, r8)
 17  send(r, r8.caller)
 19  douop(assign, l, lst)
end
```

Figure 8: OOCA ISA primitives for functions quicksort and partition

5.2 Object Level Parallelism

In previous sections we have shown that; block of OOCA instructions doesn't contain branch instructions, almost all instructions inside one block are independent amongst each other and instructions within different blocks are independent one from each other.

If we observe example of execution on OOCA Processor of quicksort algorithm on a linked list from, **Figure 9** we can see that while execution of first function quicksort (Q1) instruction block is in progress we will have ready for parallel execution partition function (Q1.p) instruction block and quicksort function (Q2 and Q3) .

As soon as partition function (Q1.p) picks up pivot element (element 3) and puts the first element (element 7) in list that is going to be passed to partition function (Q2.p), partition function (Q2.p) can choose it as pivot element and proceed with processing other elements as they are inserted into list by partition function (Q1.p) (elements 4, 12, 18, etc.). The same execution model can be applied further on partition functions (Q2.p) and (Q4.p and Q5.p), etc.

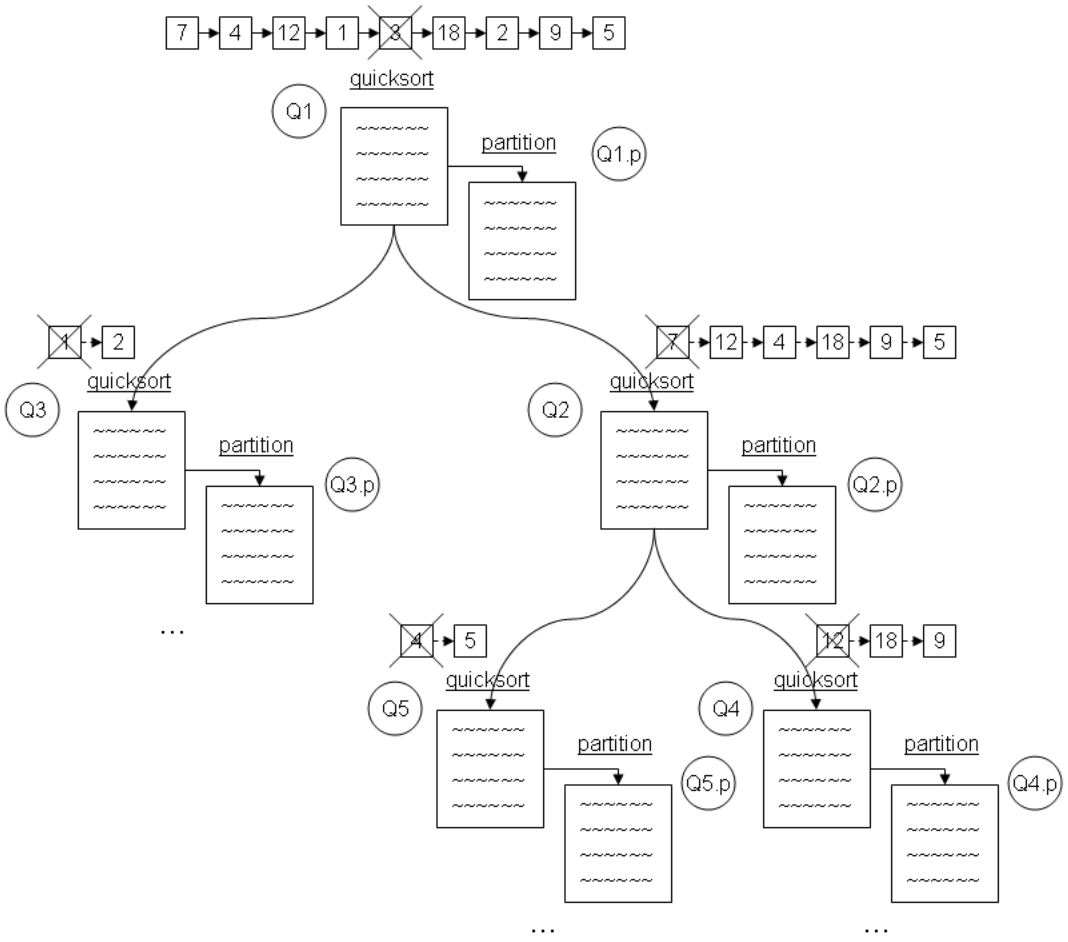


Figure 9: An example of execution of quicksort algorithm on list on OOCA processor

Note that the above observation apply further down the block execution path tree. How many of these blocks will be executed in parallel (how many instructions will be executed in parallel) depends on actual number of cores of OOCA Processor. If OOCA Processor has for example four cores (Figure 10) we could execute in parallel instructions from blocks of following functions quicksort (Q1), (Q2) and (Q3) and partition function (Q1.p), while executing functions quicksort (Q2) and (Q3) instruction blocks, instruction blocks for functions quicksort (Q4) and (Q5) and partition functions (Q2.p) and (Q3.p) would be ready for execution. They would be placed in the ready queue to wait until scheduler assigns them to a core of OOCA Processor.

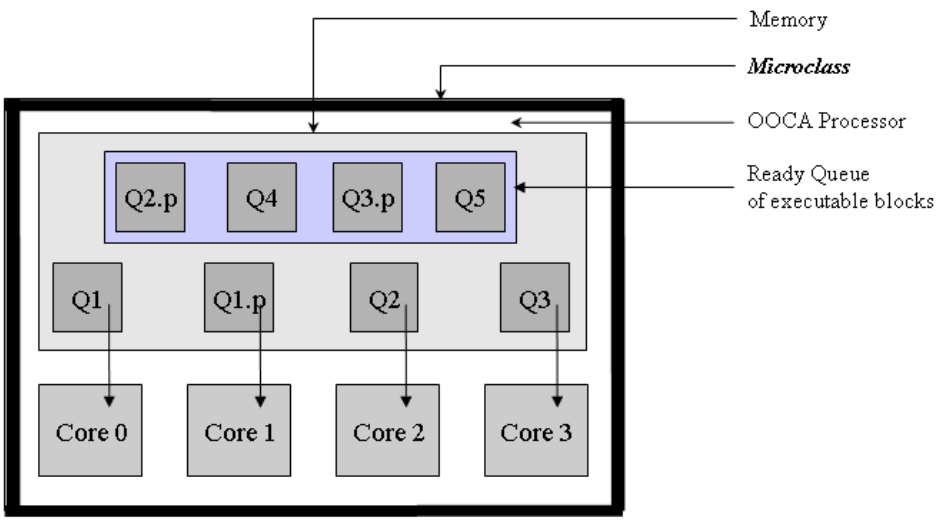


Figure 10: OOCA Processor with four cores

5.3 Object-Oriented Speculative Execution

5.3.1 Object-Oriented Data Versioning

Data versioning mechanism is based on set of *signatures* which are located in *instance context of data object*. One *Signature* consists of several fields. Those fields are: reference to the *parent signature* (psig in Figure 11), *signature* of a method from which observed method is called (note that operations like add are also methods in OOCA); *sequence number* (seq in Figure 11), explained in previous section 5.1.; *depth* (dep in Figure 11), number which represents depth of call (if parent depth is n, then depth of observed method is n+1); *if* (if in Figure 11), number that distinguishes then and else branch of condition; *iteration* (iter in Figure 11), number that distinguishes iterations of a loop; *r/w* (r/w in Figure 11), marks weather data value is read or modified; *commit* (cm in Figure 11), marks weather that signature refers to last committed data value; *value* (val in Figure 11), reference to actual data. New *signature* is inserted into *instance context of data* when value of data is read or changed. When method completes its execution, including all methods that it called, data value associated with that methods signature is committed (Figure 11 b)).

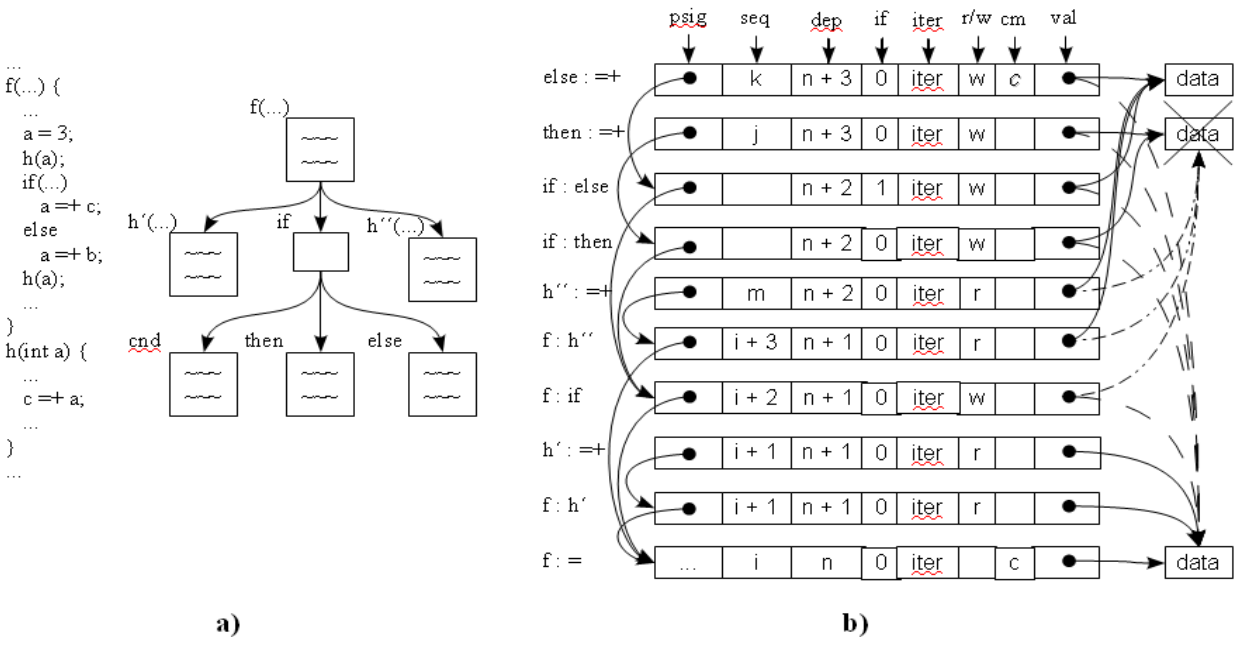


Figure 11: An example with conditional execution

On the Figure 11 a) we show simple example of modifying and reading variable a in `functions f` and `h` and `condition`. Through this example we are showing how information about dataflow are gathered inside *instance context* of `object a` Sets of OOCA primitives for `functions f` and `h` and `blocks` (`cnd`, `then` and `else`) of condition can be executed in parallel. When function `h´` and `h´´` and `condition` are called and `value a` passed to them new *signatures* are inserted into *instance context* of variable a for each one of them, `f:h´`, `f:if` and `f:h´´` in Figure 11 b). Inside `method h`, execution of operation =+ will insert new *signature* in *instance context* of `variable a` (`h´`:=+ and `h´´`:=+). Calling of condition creates blocks then and else and passes variable a to them, this inserts two new *signatures* into *instance context* of variable a, `if:then` and `if:else` in Figure 11 b). During execution of then and else blocks of condition in parallel two new *signatures* will be inserted into instance context of `variable a` marking that variable a has been modified, and each signature will set pointer to a new value.

5.3.2 Control Independent Object-Oriented Execution

If we presume that, in example from previous section (Figure 11), the `then block` finished its execution first and that that `branch` is predicted taken, the `function h´´` would use data values produced in it for further execution. Afterwards, the `cnd block` finishes its execution and determines that `else branch` is taken, it will re-reference data pointer in `f:if` signature and trigger versioning control system, which will commit new value for variable a and go through signatures, checking them, to see where wrong value of variable a was used. In this case, it will find out that OOCA primitive `h´´:=+` used wrong value of variable a. It will trigger re-execution of this primitive only, not of whole set of primitives of `method h` (re-execution of this primitive might trigger re-execution of some other primitives ("chain" re-execution), but never whole primitive sets of some method). By localizing information about dataflow inside an *instance context* of data object we never have need for squashing a set of instructions and re-executing whole set in case of miss-prediction.

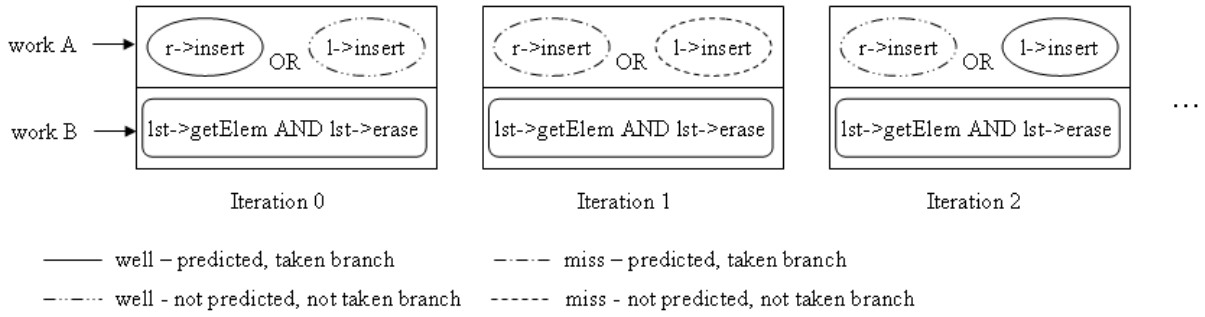


Figure 12: OOCA execution of while loop from Figure 3

Consider the while loop shown in Figure 3. In line 26, the first element of the list is taken (27th line in Figure 3), the condition is evaluated (28th line in Figure 3), depending on the condition resolution element is inserted into one of the other two lists (lines 28 and 29 in Figure 3) and then it is erased from the first list (30th line in Figure 3). Actual work done by each of the loop iterations can be divided into two parts: getting the first element from the head of the list and erasing it after (work B in Figure 12) followed by inserting that element into list l or r depending on condition (work A in Figure 12). If loop iterations are done in parallel and miss-speculation happens in iteration 1 Figure 12 OOCA processor would need to

re-execute only work A in each loop iteration starting from iteration 1, because of “chain” re-execution mechanism.

6 Related work

In the 1980s many experimental and commercial systems were produced with the aim of embedding high-level language features directly in hardware. In this section, we non-exhaustively review relevant research from this period focusing mainly on efforts aimed at incorporating object-oriented language features in the processor. These efforts, which were ahead of their time, weren’t able to translate the designs into efficient implementations mostly because the semiconductor technology of the time was severely limited in available transistor budget and memory capabilities.

Intel’s iAPX 432 processor [8] has many hardware structures for object support. In this processor, consisting of a three chip module, the data manipulation can operate over characters, ordinals, integers, floating-point variables, bit strings, arrays, records, or "objects." Note that in OOCA all data manipulations are done through “objects”, even for characters, integers, etc. Objects are treated as single entities and their internal organization is hidden and protected from other hardware procedures. OOCA does not have instruction pointer for executing instructions and function call are totally asynchronous, also all functions are represented as special “objects" in unlike the iAPX 432. Furthermore, the iAPX 432 has specific memory organization, while OOCA does not have a global concept of memory in the traditional sense, it is relaying on memory organization of actual processing unit which represents execution unit of OOCA Object Processor.

On the other camp are processors that were designed for particular OO languages. Swamp [4] is such an example for the Smaltalk-80. Among other innovations, Swamp introduced specialized caches for methods and contexts. Each context is only partly initialized

when created, and has no memory allocated for it until absolutely necessary. We adopt the same approach in OOCA. Another example for OO language/specific processors is PicoJava-I [5] which supports Java runtime by executing Java byte code directly in hardware resulting in up to a 20X speedup over pure software implementations. PicoJava-I core includes a RISC-style pipeline and straightforward instruction set. It contains regular processor blocks such as cache, pipeline, stack, floating-point unit, however these are optimized to execute Java byte code efficiently. More recently, SUN disclosed the details of an object-aware memory architecture design [13]. It includes a special address space for objects which are then accessed using object IDs mapped by a translator to physical addresses. To support this, the system includes object-addressed caches. Although similar, OOCA extends this concept, in our case all memory accesses has to be through an object.

Earlier research on Aleph, Accent and Mach [7] kernels introduced inter-process communication to move data between programs and the kernel, so applications could transparently access resources on any machine on the local area network. Accent utilized copy-on-write in which only the portions of the data that actually changed were actually copied, conceptually this is similar to OOCA miss-prediction recovery mechanism in which only the data that is not consistent is re-executed. OOCA also using message passing mechanism, through primitives of AHL implemented in HW, but on much finer grain level, for passing arguments between executable blocks (functions, loops, conditions).

7 Conclusions

The goal of this paper has been to propose a new architecture that moves the high-level programming language paradigm closer to hardware. It is based on the Object Oriented Language Model, as one of the most successful models, extracts better data and execution locality and helps to apply hardware optimizations related to parallelism and speculation more effectively.

Other papers like [14] have demonstrated that current architectures are limiting the speedup due to architectural deficiencies. OOCA can be a new way to overcome some of this limitation because it preserves the information given by the programmer in the high-level language. That can help the hardware to execute the program more efficiently.

OOCA Model offers a novelty asynchronous control independent parallel object aware execution model. In this model, methods are executed asynchronously, in its own context, where all input/output communication between them is done in asynchronous way. This OOCA model offers very aggressive approach on speculative scenarios.

The OOCA processor does not require deterministic hardware. Its execution unit can use anything from simple execution unit, FPGA, OOO Processor, Multithreaded Processor, …, to a complex CMP. While OOCA Control Unit manages the control/execution/interpret of OOCA ISA code through a special software/hardware layer called MicroClass. This fact opens interesting window for heterogeneous execution where some objects are bound to specific hardware.

Finally, we have demonstrated that OOCA can totally or partially some solve problematic scenarios that remain unsolved by current architectures like control independence and object speculation parallelism.

8 References

1. H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO '98)*, pages 226–236. IEEE Computer Society, Dec. 1998
2. D. Burger, S.W. Keckler, K.S. McKinley, "Scaling to the End of Silicon with EDGE Architectures", et al. *IEEE Computer*, 37 (7), pp. 44-55, July, 2004.
3. L. Codrescu and D. S. Wills. Architecture of the atlas chipmultiprocessor: Dynamically parallelizing irregular applications. In *Proceedings of the 1999 International Conference on Computer Design (ICCD '99)*, pages 428–435. IEEE Computer Society, Oct. 1999.
4. D. M. Lewis , D. R. Galloway , R. J. Francis , B. W. Thomson, "Swamp: a fast processor for Smalltalk-80, " Proceedings of Conference on Object-oriented Programming Systems, Languages and Applications, OOPSLA, p.131-139, September 1986.

5. J. M. O'Connor, M. Tremblay, Marc, "PicoJava-I: The Java Virtual Machine in Hardware", IEEE Micro, Volume 17, Issue 2: pp. 45–53, March/April 1997.
6. J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques(PACT '99)*, pages 303–313. IEEE Computer Society, Oct. 1999.
7. R. F. Rashid, "From RIG to Accent to Mach: the evolution of a network operating system," Proceedings of 1986 ACM Fall Joint Computer Conference, 1986.
8. J. Rattner, "Hardware/software cooperation in the iAPX-432," ACM SIGARCH Computer Architecture News, Volume 10, Issue 2, March 1982.
9. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO '97)*, pages 138–148. IEEE Computer Society, Dec. 1997.
10. Smith, J.E. Sohi, G.S., "The microarchitecture of superscalar processors", Dept. of Electr. & Comput. Eng., Wisconsin Univ., Madison, WI, USA; This paper appears in: Proceedings of the IEEE**,** Dec. 1995, Volume: 83 , Issue: 12 ,On page(s): 1609 – 1624
11. Sohi, G.S.; Breach, S.E.; Vijaykumar, T.N.; "Multiscalar processors", Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on 22-24 Jun 1995 Page(s):414 – 425
12. Vijaykumar, T.N.; Gopal, S.; Smith, J.E.; Sohi, G.; "Speculative Versioning Cache", High-Performance Computer Architecture (HPCA) 1998, pages 58-69
13. G. Wright, M. L. Seidl, M. Wolczko, "An Object-Aware Memory Architecture," SUN Microsystems Technical Report, TR-2005-143, 2005.
14. Warg Fredrik, Stenström Per, "Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms", PACT 2001. Pages: 221 - 230.

9 Appendix A

Complete OOCA ISA code for the example form Figure 3.

```
/*OOCA Language description for class Elem*/
class Elem
  OOCA_atributes
    class int <number> [create]
  OOCA_functiones
    constructor
      OOCA_arguments
        class int <number>
      OOCA_temporal
        function assign r1
      OOCA_return
        class Elem <ret> [create]
```

```
  OOCA_code l1
getNumber
  OOCA_return
    class int <ret>
  OOCA_temporal
    function assign r1
  OOCA_caller
    class Elem <this>
  OOCA code l2
setNumber
  OOCA_arguments
    class int <i> [nochange]
  OOCA_temporal
    function assign r1
  OOCA_caller
    class Elem <this>
  OOCA_code l3
compare
  OOCA_argumets
    class Elem <e> [change]
  OOCA_return
    class bool <ret>
  OOCA_caller
    class Elem <this>
  OOCA_temporals
    cmp_if <r1>
  OOCA_code l4
cmp_if derived from if
  OOCA_arguments
    class Elem <this> [share OOCA_cond]
    class Elem <e> [share OOCA_cond]
    class bool <ret> [change], [share OOCA_then, OOCA_else]
  OOCA_variables
    class bool <condition> [create]
  OOCA_blocks
    OOCA_cond
      OOCA_temporals
        function Elem.getNumber <r1>, <r2>
        class int <r3>, <r4>
        function int.<= <r5>
      OOCA_code l5
    OOCA_then
      OOCA_variables
        class boolean <r1> [create] = true, [cosnt]
      OOCA_code l6
    OOCA_else
```

```
        OOCA_ variables
          class boolean <r1> [create] = false, [cosnt]
        OOCA_code l7
```

/*OOCA Language ISA code for class Elem*/

```
l1:  begin
       1 duoop(assign, number, ret.number)
     end
l2:  begin
       1 duoop(assign, this.number, ret)
     end
l3:  begin
       3 call(assign, r1)
       1 send(i, r1)
       2 send(this.number, r1)
     end
l4:  begin
       4 call(cmp_if, r1)
       1 send(this, r1.this)
       2 send(e, r1.e)
       3 send(ret, r1.ret)
     end
l5:  begin
       2 call(this.getNumber, r1)
       1 send(r3, r1)
       4 call(e.getNumber, r2)
       3 send(r4, r2)
       7 call(r3.<=, r5)
       5 send(r4, r5)
       6 send(condition, r5)
     end
l6:  begin
       1 duoop(assign, r1, ret)
     End
l7:  begin
       1 duoop(assign, r1, ret)
     end
```

/*OOCA Language description code for partition function*/
partition

```
  OOCA_arfunemts
    class List {Elem} <lst>, <l>, <r> [change]
  OOCA_return
    class Elem <ret>
  OOCA_variables
```

```
    class Elem <pivotElem>
  OOCA_temporals
    function List.getFirst <r1>
    function List.eraseFirst <r2>
    iterator <r3>
  OOCA_code l3
while_iterator generate from iterator
  OOCA_arguments
    class List {Elem} <lst> [share OOCA_condition, OOCA_block]
    class List {Elem} <l> [share OOCA_block send r2.l]
    class List {Elem} <r> [share OOCA_block send r2.r]
    class Elem <pivotElem> [share OOCA_block send r2.pivotElem]
  OOCA_variables
    class bool <condition> [create]
  OOCA_blocks
    OOCA_condition
      OOCA_temporals
        function List.empty <r1>
        class bool <r2>
      OOCA_code l5
    OOCA_block
      OOCA_temporals
        function List.getFirst <r1>
        while_block_if <r2> [create]
        class Elem <r4>
        function List.eraseFirst <r5>
      OOCA code l4

while_block_if generate from if
  OOCA_arguments
    class List {Elem} <l>, <r> [share OOCA_then, OOCA_else]
    class Elem <pivotElem> [share OOCA_condition, OOCA_then, OOCA_else]
    class Elem <r1> [share OOCA_condition, OOCA_then, OOCA_else]
  OOCA_variables
    class bool <condition> [create]
  OOCA_blocks
    OOCA_condition
      OOCA_temporals
        function Elem.compare <r2>
      OOCA code l6
    OOCA_then
      OOCA_temporals
        function List.insert <r2>
      OOCA_code l7
    OOCA_else
      OOCA_temporals
```

```
        function List.insert <r2>
      OOCA_code l8
```

/*OOCA ISA code for partition function*/

```
l3: begin
      1  call(lst.getFirst, r1)
      2  send(pivotElem, r1.ret)
      3  call(lst.eraseFirst, r2)
      8  call(while_iterator, r3)
      4  send(lst, r3.lst)
      5  send(l, r3.l)
      6  send(r, r3.r)
      7  send(pivotElem, r9.pivotElem)
      9  douop(assign, pivotElem, ret)
     end
l4: begin
      3  call(lst.getFirst, r1)
      1  send(r4, r1)
      2  send(r4, r2.r1)
      4  call(lst.eraseFirst, r5)
     end
l5: begin
      2  call(lst.empty, r1)
      1  send(r2, r1.ret)
      3  duoop(r2.!, r2)
      4  send(condition, r2)
     end
l6: begin
      3  call(r1.compare, r2)
      1  send(pivotElem, r2.arg1)
      2 send(condition, r2.ret)
     end
l7: begin
      2  call(l.insert, r2)
      1  send(r1, r2.arg1)
     end
l8: begin
      2 call(r.insert, r2)
      1 send(r1, r2.arg1)
     end
```

/* OOCA Language description code for quicksort function */
quicksort

```
  OOCA_argunemts
    class List {Elem} <lst> [change]
```

```
  OOCA_variables
    class List {Elem} <l>
    class List {Elem} <r>
    class Elem <pivot>
  OOCA_temporals
    empty_list_if <r1>
    function List {Elem}.constructor <r2>
    function List {Elem}.constructor <r3>
    function partition <r4>
    function quicksort <r5>
    function quicksort <r6>
    function List.insert <r7>
    function List.insert <r8>
  OOCA_code l3

empty_list_if generate from if
  OOCA_arguments
    class List {Elem} <lst> [share OOCA_condition]
    class Elem <ret> [share OOCA_then]
  OOCA_variables
    class bool <condition> [create]
  OOCA_blocks
    OOCA_condition
      OOCA_temporals
        function List.empty <r1>
      OOCA code l1
    OOCA_then
      OOCA_variable
        class Elem <r1> [create]
      OOCA code l2
```

/*OOCA ISA code for partition function*/

```
l1:  begin
      2  call(lst.empty, r1)
      1 send(condition, r1.ret)
     end
l2: begin
      1 duoop(assign,r1, ret) [stop further execution]
     end
l3: begin
     2   call(empty_list_if, r1)
     1   send(lst, r1.lst)
     4   call(List {Elem}.constructor, r2)
     3   send(l, r2.ret)
     6   call(List {Elem}.constructor, r3)
```

```
 5   send(r, r3.ret)
 11 call(partition, r4)
 7   send(lst, r4.lst)
 8   send(l, r4.l)
 9   send(r, r4.r)
 10 send(pivot, r4.ret)
 13 call(quicksort, r5)
 12 send(l, r5.l)
 15 call(quicksort, r6)
 14 send(r, r6.l)
 17 call(l.insert, r7)
 16 send(pivot, r7.caller)
 19 call(l.insert, r8)
 18 send(r, r8.caller)
 20 duoop(assign, l, lst)
end
```