Object Oriented execution Model (OOM)

Nikola Markovic †, Daniel Nemirovsky †, Ruben Gonzalez ł, Osman Unsal †,
Mateo Valero †, Adrian Cristal †‡

Barcelona Supercomputing Center (BSC) †
c/ Jordi Girona, 31 08034, Barcelona (Spain)

Chalmers University of Technology (CUT) ł
SE-412 96 Gothenburg (Sweden)

Artificial Intelligence Research Institut - Spanish National Research Council (IIIA-CSIC) ‡
E-08193 Bellaterra, Catalonia (Spain)

{nikola.markovic,daniel.nemirovsky, osman.unsal,mateo.valero, adrian.cristal}@bsc.es,
rgonzalez@chalmers.se

***Abstract*—This paper considers implementing the Object Oriented 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 a new execution model called Object Oriented execution Model - OOM - that implements the OO Programming Models. All OOM hardware structures are objects and the OOM Instruction Set directly utilizes objects while hiding other complex hardware structures. OOM maintains all high-level programming language information until execution time. This enables efficient extraction of available parallelism in OO serial code at execution time with minimal compiler support. Our results show that OOM utilizes the available parallelism better than the OoO (Out-of-Order) model.**

***Index Terms*—Object-Oriented, parallel and asynchronous execution.**

I. Introduction

THE Chip Multi Processor (CMP) has made a great impact on processor design during the past two decades since CPU design hit the power wall. During the 1990's, computer architects presented several CMP designs [15], [16]. It provides several advantages over the centralized superscalar approach [8]. Firstly, the simple design of CMP enables high clock rate and allows the implementation of fast communication network in each of the processing units. Additionally time-consuming design validation phase is alleviated. Finally, available silicon space is better utilized [6]. In the past ten years, many high-performance processor vendors have introduced designs that can execute multiple threads simultaneously on the same chip through simultaneous multithreading [10], multiple cores [17] or a combination of the two [11]. These designs utilize nonspeculative Thread-Level Parallelism (TLP) smoothly.

Nevertheless, the execution time of a sequential application or of a single thread of a parallelized application still suffers from the same issues as twenty years ago. Lam et al. [9] shows that finding a significant level of parallelism in numerical programs where control flow is not data dependent is a simple task for the compiler, however that is not the case if the control flow is data dependent. The study also points out that in case of non-numerical programs, processors that do not support speculative execution will hardly be efficient in extracting parallelism. Recent approaches that are usually referred to as speculative TLP decrease the execution time of these applications. These approaches execute several speculative threads in parallel. Threads are speculative in the sense that they can be data or control dependent on other threads and therefore their proper execution and commitment are not guaranteed. Two main ways to leverage speculative TLP are reducing the execution time of high-latency instructions by means of side effects with helper threads [3], [4], [13] and parallelizing the application into speculative threads for example in [1], [7], [14] among others.

On the other hand, writing parallel applications for multicore processors still appears to be an extremely difficult task for ordinary programmers [12]. A solution, we believe, is to allow programmers to continue working with the most successful programming model to date, the Object-Oriented model, and to move the complexity of parallelization of sequential applications for multiple homogeneous or heterogeneous cores to hardware. As a way of accomplishing that, we propose a new and unique execution model, leveraging an Object-Oriented

(OO) programming model. We see it as an architecture that ports software concepts into hardware by forming an abstract, virtual hardware layer. This hardware layer is able to preserve semantic information specified by the programmer in a high level language for execution time.

In this paper, we present an Object Oriented execution model. In contrast to previously mentioned proposals that tend to leverage parallelism by extracting blocks of instructions with the least possible amount of data and control dependences and forming separate threads out of them, our OO model exploits parallelism by preserving and leveraging information available at the high level programming language code to hardware. Our model represents data as *objects*, user defined functions as *contexts*, and *operations* - hardware implemented instructions and functions eg. add, sub, cmp, etc. An *object* can store and manage multiple versions of a data value. Each *context* has its own memory space. By having memory space per *context*, we are breaking the traditional stack into smaller pieces. The model also defines a new ISA that consists of three basic instructions and *operations*.

The OO model provides two main advantages. First, better data locality, where *objects* are smaller units of locality. Second, an asynchronous, control independent and parallel execution which preserves the sequential view of program. The methods are executing asynchronously, in their own context without any other interferences, where input/output parameters are sent in an asynchronous way. This opens up possibilities for dynamic optimization, which was not feasible at compile time. We compare the level of available parallelism, on the quicksort algorithm applied on a linked list. Our results show that the ideal OOM utilizes better available parallelism than the ideal OoO (Out-of-Order) model. It is also able to exploit approximately the same level of parallelism for different levels of *operation* granularity, which makes it very flexible compared to the OoO model.

II. Model explanation

In this section, we describe the Object Oriented execution Model. First we describe OOM Instruction Set Architecture (ISA), explaining in detail the most important ISA instructions and their functionalities. Second, we explain the concept of objects in our model. We provide a detailed description of each object's functionality and its role in OOM. Finally we provide an explanation of an asynchronous, control independent and parallel execution flow.

A. OOM ISA

This ISA consists of three basic instructions and *operations*. *Operations* are hardware-implemented instructions or functions e.g. add, sub, cmp, etc. The three basic instructions are: *call* , *send* and *select* . The instruction *call* triggers the execution of a *context* or an *operation*. *send* sends the reference of an *object* to a *context* or *operation*, while *select* fetches references of *objects* encapsulated inside of an *object* - class attributes. A set of these instructions is assigned to each *context* (similar to the instructions inside of a function).

Each *call* instruction inside the list of instructions of a *context* has a unique sequential number. This number is statically assigned at compile time and represents the sequential order in which the *context* or *operation* invoked should be committed inside of the calling *context*, like a sequential order of *operation* inside of the function. While executed *call* instruction also assigns a dynamically calculated ID to the created *context* or *operation*. The *call* instruction calculates the ID based on its parent *context* ID and the sequential number assigned to it inside its parent *context*. This ID represents the position of the invoked *context* or *operation* inside global sequential commit view of the program.

B. OOM Objects

The OO model defines two types of objects: *object* and *context*. User defined functions and methods are represented as a kind of an object called *context*. The *context* has its own memory space where it stores references to *objects* that are argumenta and local values of a user defined function or method. It also has the list of instructions. The heart of the OO model are *objects* which manage and store the data and behave "like hardware threads". All *objects* are hardware implemented. The *objects* can be divided into two different kinds. First kind of *objects* are basic *objects*. These *objects* are for basic types like int, float, bool, char, etc. The basic *object* stores and manages multiple versions of data through a versioning mechanism, explained in Subsection II-B1. The second kind of *objects* are complex *objects*. They are for user-defined types like classes and structures. The complex *object* has a set of references to other basic or complex *objects* e.g. attributes of a class, fields of a structure, etc.

Since each data type has a set of instructions or methods that can be performed on it, for the basic types like int they are the instructions like sub, add, cmp, mov, etc. while for the class they are the methods and the functions of the class, each basic *object* has the set of

operations and complex *object* has the set of *contexts* that can be performed on it.

Both kinds of *objects* behave "like hardware threads" of execution. When an instruction *call* is executed, a new message is sent to an *object* to execute one of its *contexts* or *operation*. *Objects* manage a list of its *contexts* or *operations* scheduled for execution and executes them when it gets the processor.

1) Versioning mechanism: A basic *object* has a table of values. The table of values has two fields: a key and a value. The table key is ID of *operations* that produced the value. Whenever some *operation* produces a new value for the *object*, a pair consisting of the *operation's* ID and the produced value is stored in the basic *object* table. An entry to the table whose corresponding key is the ID of a committed *operation* is called a committed entry. Only one entry at the time can be the committed entry, all other entries are considered to be speculative entries. Allthough the entries may be stored out-of-order in the table, they are always committed in a sequential order when an operation corresponding to the key in the table is committed. As the new entry becomes the committed entry, the previous committed entry is deleted from the table along with any entries stored by mis-speculatively executed *operations*.

C. OOM execution

If the static program control flow is represented as a directed graph, where nodes are *contexts* and *operations* and arcs represent flow from one node to another, then the program execution can be viewed as going through that directed graph. The fact that the traditional stack is broken allows this directed control flow graph of the static program to be represented through smaller directed graphs where each small graph contains only *contexts* that share memory space. Small graphs become nodes of the large graph. Allthough data and control dependencies between small graphs still remain, their instructions and *operations* can be scheduled for parallel execution on different processing units. The fine granularity of *operations* inside *contexts* allows for the execution of only those instructions and *operations* that have their input data ready.

III. OOM ARCHITECTURE

In Fig. 1. we present a possible architecture for the Object Oriented execution Model. The Object Creation and Allocation Unit creates *objects* and assigns them to the Object Processors. Each Object Processor is associated with a scratch pad memory called Object Memory. Several *objects* can be mapped on the same Object Processor. Shared Object References is a scratch pad memory shared by all Object Processors in the system. It keeps information where each *object* is located.

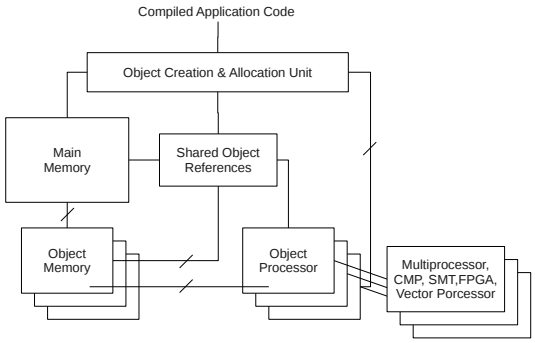

Fig. 1. Object-Oriented execution Model architecture

An Object Processor is the minimum execution unit. The OOM Architecture is a group of Object Processors collaborating to do execution. This Architecture implements a Virtual Hardware layer (Object Oriented execution Model). Each Object Processor is internally flexible. It can support different hardware implementations. Flexible Internal Hardware Model of the Object Processor can be managed by many possible cores e.g. Out-of-Order processor, In-order processor (embedded), Multiprocessor, Vector processor, Processor + FPGA (Reconfigurable Architectures), Data flow processor, etc.

IV. RESULTS

To depict the available parallelism for the quicksort algorithm on the linked list we used DDG (Dynamic Dependency Graph) analysis methodology of the algorithm presented in [2]. The DDG of the program is a partially ordered, directed, acyclic graph where the nodes of the graph represent computation that occurred during the execution of the program, and the edges represent dependencies that force a specific order on the execution of the instructions. To give an upper bound on the available parallelism, we analyze a DDG containing only true data dependencies. For the ideal OoO model we used the Pin Tool to generate traces, while for the ideal OOM we implemented the functional level simulator to generate traces. For the ideal models we take that all instructions, memory accesses and messages sent through interconnection network take one cycle.

Fig. 2. shows the available parallelism (note that horizontal axis represents the list size as the number of elements in the list). For the ideal OOM, we present

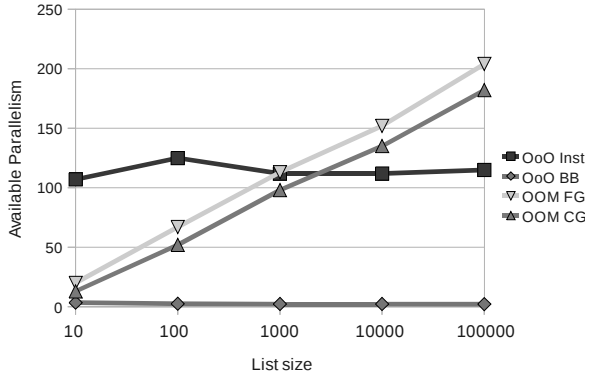

Fig. 2. Available parallelism of the quicksort algorithm on the linked list - for the ideal OoO model on the instruction and basic block level and for the ideal OOM model on fine and coarse grained level

results for two levels of *operation* granularity. First one is fine granularity where the *operations* are like sub, add, cmp, etc. instructions of the basic types int, float, etc. The second one is coarse grained where the *operations* are whole functions like function that adds an element to the list, function that pops an element form the list, etc. of the class types list. Results from Fig. 2. show that while the ideal OoO model always extracts approximately the same level of parallelism, on the other hand our ideal OOM is able to extract more parallelism as the number of object and the instructions executed in system increases, and even outpacing the OoO model, because it exploits data locality in a natural way. Fig. 2. also shows that our model is able to extract almost the same level of available parallelism for different granularity of *operations*, which is not the case for the OoO model.

V. Conclusion and future work

The goal of this paper has been to propose a new execution model that moves the high-level programming language paradigm closer to hardware. It is based on the Object Oriented Language Model, being 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.

The OO execution Model offers a novel asynchronous, control independent, parallel and object-aware execution model. In this model, methods are executed asynchronously in their own context, where all input/output communication between them is done in an asynchronous way. This OO execution model can offer a very aggressive approach in speculative scenarios. The processor based on OOM does not require deterministic hardware. Its execution unit can use anything from a simple execution unit, FPGA, OoO Processor, Multithreaded Processor, to a complex CMP. The OO execution model manages the execution of OOM ISA code through a special software/hardware virtual layer. This opportunity opens interesting windows for heterogeneous execution where some objects can be bound to specific hardware.

Finally, we have demonstrated that the ideal OOM utilizes the available parallelism better than the ideal OoO model and that it is able to extract almost the same level of parallelism on fine and coarse grained level while the OoO model fails to do so. We will continue working on our functional level simulator, to extend it and port it to an FPGA board using Bluespec environment.

References

[1] H. Akkary, M. Driscoll, "A Dynamic Multithreading Processor", *Proc. 31st Int'l Symp. on Microarchitecture*, pp. 226, 1998.

[2] T. Austin, G. Sohi, "Dynamic Dependency Analysis of Ordinary Programs", *The 19th Int'l Symp. on Computer Architecture*, pp. 342, 1992.

[3] R. Chappel, J. Stark, S. Kim, S. Reinhardt, Y. Patt, "Simultaneous Subordinate Microthreading (SSMT)", *Proc. 26th Int'l Symp. Computer Architecture*, pp. 186-195, 1999.

[4] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, J. Shen, "Speculative Precomputation: Long Range Prefetching of Delinquent Loads", *Proc. 28th Int'l Symp. Computer Architecture*, pp. 14, 2001.

[5] M. Fillo, S. Keckler, W. Daly, N. Carter, A. Chang, Y. Gurevich, W. Lee, "The M-Machine Multicomputer", *Proc. 28th Int'l Symp. Computer Microarchitecture*, pp. 146, 1995.

[6] L. Hammond, B. Neyfeh, K. Olukotun, "A Single-Chip Multiprocessor", *Computer*,vol. 30, no. 9, pp. 79, 1997.

[7] L. Hammond, M. Willey, K. Olukotun, "Data Speculation Support for a Chip Multiprocessor", *Proc. 8th Int'l Conf. Architectural support for Programming Languages and Operating Systems*, 1998.

[8] M. Johnson, "Superscalar Microprocessor Design", *Prentice Hall*, 1990.

[9] M. Lam, R. Wilson, "Limits of Control Flow on Parallelism", *Proc. 19th Int'l Symp. Computer Architecture*, pp. 46, 1992.

[10] T. Marr et al., "Hyperthreading Technology Architecture and Microarchitecture", *Intel Technology J.*,vol. 6, no. 1, 2002.

[11] A. Mendelson et al., "CMP implementation in the Intel Core Duo Processor", *Intel Technology J.*, vol. 10, no. 2, 2006.

[12] D.Patterson et al., "The parallel computing landscape: a Berkeley view", *Low Power Electronics and Design*, pp. 231, 2007.

[13] A. Roth, G. Sohi, "Spevulative data-driven Multithreading", *Proc. 7th Int'l Symp. High-Performance Computer Architecture*, pp. 37, 2001.

[14] S. Sarangi, W. Liu, J. Torrellas, Y. Zhou, "ReSlice: Selective Re-Execution of Long-Retired Misspeculated Instructions Using Forward Slicing", *Proc. 38stInt'l Symp. on Microarchitecture*, pp. 257, 2005.

[15] J. Smith, S. Vajapeyam, "Trace Processors: Moving to Fourth Generation Microarchitectures", *IEEE Computer*,vol. 30, no. 9, pp. 68, 1997.

[16] G. Sohi, S. Breach, T. Vijaykumar, "Multiscalar Processors", *Proc. 22nd Int'l Symp. Computer Architecture*, pp. 414, 1995.

[17] S. Storino, D. Borkenhagen, "A Multithreaded 64-bit PowerPC Commercial RISC Processor Design", *Proc. 11th Int'l Conf. High-Performance Chips*, 1999.

[18] D. Tullsen, S. Eggers, H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Proc. 22nd Int'l Symp. Computer Architecture*, pp. 392, 1995.