A Multi-Threaded Asynchronous Language

Hervé Paulino[1], Pedro Marques[2], Luís Lopes[2],
Vasco Vasconcelos[3], and Fernando Silva[2]

[1] Department of Informatics, New University of Lisbon, Portugal
herve@di.fct.unl.pt
[2] Department of Computer Science, University of Oporto, Portugal
pmarques@med.up.pt, {lblopes, fds}@ncc.up.pt
[3] Department of Informatics, University of Lisbon, Portugal
vv@di.fc.ul.pt

Abstract. We describe a reference implementation of a multi-threaded run-time system for a core programming language based on a process calculus. The core language features processes running in parallel and communicating through asynchronous messages as the fundamental abstractions. The programming style is fully declarative, focusing on the interaction patterns between processes. The parallelism, implicit in the syntax of the programs, is effectively extracted by the language compiler and explored by the run-time system.

1 Introduction

Dataflow architectures represent an alternative to the mainstream von Neumann architectures. In von Neumann architectures, the order in which the instructions in a program are executed is established at compile time, when the executable is produced. A special purpose register, the program counter, keeps track of the flow of execution. In a dataflow architecture, in contrast, instructions are executed as soon as their arguments are available (the so called *firing rule*) and regardless of any pre-established order. This makes the model totally asynchronous and the instructions self scheduling.

Multi-threaded architectures attempt to improve the performance of classic von Neumann architectures by introducing some features from dataflow architectures such as out-of-order execution and fine grained context switching, usually supported by the microprocessor hardware. These additions aim to provide high processor utilization in the presence of large memory or interprocessor communication latency.

The current generation of superscalar microprocessors requires great amounts of fine grained parallelism to fully explore their aggressive dynamic dispatch capabilities, multiple functional units and, in some cases, rather long pipelines. In this context, support for multi-threading at the hardware level may help to avoid pipeline hazards in current von Neumman implementations, thus eliminating the need for complex forwarding and branch prediction logic.

Despite these interesting possibilities, single-thread performance in multi-threaded architectures is typically low, and this has a negative impact on the

performance of individual applications. The ideal situation would call for applications themselves to be partitioned into several fine grained threads by a compiler. A multi-threaded microprocessor would then overlap the multiple threads from that single application, improving performance. In particular, languages that allow efficient compilation from high-level constructs into low-level, fine grained, code blocks, easily mapped into threads at run-time, may potentially profit from multi-threaded hardware.

Programming languages with compiler support for parallel execution have been extensively researched in the past, namely for dataflow architectures [1, 2, 5] However, the recent introduction of process calculi [3, 11] as the reference models for parallel computations provides an interesting alternative. In fact, process calculi are, in a way, a natural choice since they model systems with processes running in parallel and communicating through message passing. Their compact formal definition and well understood semantics may potentially diminish the usual gap between the semantics of a programming language and that of the corresponding implementations.

In this paper, we describe a multi-threaded run-time system for a programming language based on the TyCO (Typed Concurrent Objects) process calculus [6]. The run-time system is based on a specification previously proposed by the authors and formally demonstrated to be sound relative to the base process calculus [4].

The remainder of the paper is organized as follows. Section 2 presents the core programming language. In section 3 we present the specification and describe the implementation of the language's multi-threaded run-time system. Finally, in section 4, we discuss some issues for future research.

2 The TyCO Programming Language

Our source programming language is called TyCO [6]. The language is based on a process calculus in the lines of the asynchronous π-calculus. The main abstractions are communication channels, objects (collections of methods that wait for incoming messages at channels) and asynchronous messages (method invocations targeted at channels). It is also possible to define process templates, parameterized on a series of variables, that may be instantiated anywhere in the program (this allows for unbounded behavior). The syntax for the language kernel is as follows:

P	::=	$\mathbf{0}$	terminated process
	|	$P \mid P$	concurrent composition
	|	**new** $x\ P$	new local variable
	|	$x!l[\tilde{e}]$	asynchronous message
	|	$x?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	object
	|	**def** $X_1(\tilde{x}_1) = P_1\ \ldots\ X_n(\tilde{x}_n) = P_n$ **in** P	definition
	|	$X[\tilde{e}]$	instantiation
	|	**if** e **then** P **else** Q	conditional

where x represents a variable, e an expression over integers, booleans, strings or channels [7], X an identifier for a process template, and l a method name.

From an operational point of view, TyCO computations evolve for two reasons: object-message reduction (i.e., the execution of a method in an object in response to the reception of a message) and, template instantiation. These actions can be described more precisely as follows (where v is the result of the evaluation of an expression e, either an integer, a boolean, a string or a channel):

$$x?\{\ldots, l(\tilde{x}) = P, \ldots\} \mid x!l[\tilde{v}] \rightarrow \{\tilde{v}/\tilde{x}\}P$$

The message $x!l[\tilde{v}]$ targeted to channel x, invokes the method l in an object $x?\{\ldots, l(\tilde{x}) = P, \ldots\}$ at channel x. The result is the body of the method P running with the parameters $\tilde{x}$ replaced by the arguments $\tilde{v}$. For instantiations we have something similar:

$$\textbf{def } \ldots\ X(\tilde{x}) = P\ \ldots\ \textbf{in } X[\tilde{v}] \mid Q \rightarrow \textbf{def } \ldots\ X(\tilde{x}) = P\ \ldots\ \textbf{in } \{\tilde{v}/\tilde{x}\}P \mid Q$$

A new instance $X[\tilde{v}]$ of the template process bound to X is created. The result is a new process with the same body as the definition but with the parameters $\tilde{x}$ replaced by the arguments $\tilde{v}$ given in the instantiation.

This kernel language constitutes a kind of assembly language upon which higher level programming abstractions can be implemented as derived constructs. In the example below, we use two such constructs for sequential execution of processes (;) and for synchronous method calls (**let**/**in**) as defined in [8]. The programming example illustrates the use of these primitives and derived constructs. We begin by defining a simple template for a bank account, and creating an account with 100 euro.

```
def Account(self, balance) =
   self ? {
      deposit(amount,replyto) =
         replyto![] | Account[self, balance+amount]
      balance(replyto) =
         replyto![balance] | Account[self, balance]
      withdraw(amount, replyto) =
         if amount >= balance then
            replyto!overdraft[] | Account[self, balance]
         else
            replyto!dispense[] | Account[self, balance-amount]
   }
in new myAccount Account[myAccount, 100]
```

To deposit a further 100 euro and then get our account balance place the following processes running in parallel with the above code:

```
myAccount!deposit[100] ;
let x = myAccount!balance[] in
   io!puts["your account balance is:"] ; io!printi[x]
```

The **let**/**in** construct calls the method balance at channel myAccount and waits for a reply value. On arrival, the reply triggers the execution of the process after the **in** keyword and prints the current value of the balance attribute.

3 The Virtual Machine

The TyCO source code is compiled into a small and compact language, the TyCO Intermediate Language (TyCOIL) [9], that is given as input to the TyCO's runtime system. This language features simple and fast instructions, thus sharing the advantages of RISC machines.

3.1 The TyCO Intermediate Language

The TyCO virtual machine uses a variable number of general purpose registers ($r_0, \ldots, r_{n-1}$), where n, the number of registers, is specified by the TyCOIL directive registers. A thread's execution begins by placing its activation record in register r_0. Registers may contain integers (primitive types integer and boolean) or heap references (for strings and channels).

TyCOIL Program

```
registers 9

string string#1 "your account balance is:"

code main_code {
    – code for def Account
    – code for new myAccount
    – code for Account[myAccount, 100]
    – code for myAccount!deposit[100]
    – code for first semicolon
}
code Account_code {
    – code for self ? {
        – deposit ...
        – balance ...
        – withdraw ...
    – }
}
data self_table {
    self_deposit_code
    self_balance_code
    self_withdraw_code
}
code self_deposit_code {
    – code for replyto![]
    – code for Account[self, balance + amount]
}
code self_balance_code {
    – code for replyto![balance]
    – code for Account[self, balance]
}
code self_withdraw_code {
    – code for if amount >= balance ...
}
code continuation_for_let {
    – code for io!puts[string#1]
    – code for second semicolon
}
code continuation_for_first_semicolon {
    – code for let x = ...
}
code continuation_for_second_semicolon {
    – code for io!printi[x]
}
```

Fig. 1. A sketch of a TyCOIL program

TyCOIL programs are composed of three kinds of labeled fragments: code, data, and string. code is a sequence of instructions terminated by schedule, also referred to as a thread. data fragments describe initialized data that may be used, for example, for method tables. string fragments are used to hold string constants. Figure 1 illustrates the structure of the TyCOIL program corresponding to the TyCO example in section 2.

Code for replyto![balance]	**Code for** self ? {deposit ... balance ... withdraw ...}
– channel replyto in register r3 – balance in register r4 **r3.lock()** r1 := r3.getStatus() if r1 <= 0 jump enqueue r1 := r3.dequeue() r1[2] := r4 – Fill arguments launch r1 jump done enqueue: – Message frame at r1 r1 := malloc 3 r1[2] := r4 – Fill arguments – End of message frame r3.enqueueMsg(r1) done: **r3.unlock()**	– channel self in register r4 **r4.lock()** r5 := r4.getStatus () if r5 >= 0 jump enqueue r5 := r4.dequeue () r6 := r5 [1] – Get offset of the method r6 := self_table [r6] r5 [1] := r6 – Fill address of thread code r5 [2] := r3 – Fill closure launch r5 jump done enqueue: – Object frame at r5 r5 := malloc 4 r5 [1] := self_table – Fill code r5 [2] := r3 – Fill closure – End of object frame r4.enqueueObj (r5) done: **r4.unlock()**

Fig. 2. TyCOIL code example

The TyCOIL instructions can be divided into six categories: memory allocation, channel (communication queue) manipulation, thread manipulation, external service execution, arithmetic, and program flow. Since the last two are common to most of the languages, we will focus our description on the other instructions.

The memory allocation instruction malloc allocates a new, uninitialized, frame in the heap. Frames that contain a thread's execution environment (activation records) are of a special kind: they must start with a slot (used by the machine to enqueue the frame), followed by the address of the code to be executed.

Another category of instructions manipulate channels: newChannel allocates a frame from the heap to serve as a communication channel. enqueueObj and enqueueMsg place a frame at the end of the channel's queue, updating its status accordingly. dequeue retrieves and removes a frame from the front of the channel's queue, updating the channel's status. getStatus retrieves the channel's current status: zero for the empty queue, a negative number, -n, for a queue containing n messages, a positive number, n, for a queue containing n objects.

The thread manipulation instructions operate on the virtual machine's run-queue: launch places a new task in the run-queue; and schedule frees the processor, allowing the machine to dequeue and execute a thread from the run-queue.

TyCO allows the definition of services external to the machine's core features. These are invoked through the external instruction that executes the service synchronously. Examples of external services are input/output and string operations.

Figure 2 shows a small example of the TyCOIL code corresponding to a message and an object taken from the example in section 2.

The code for the message starts by querying the channel on r3 for its status. If there are no objects on the queue, it jumps to enqueue to insert a newly created frame, representing the message, in the channel's queue. When the channel has objects the code retrieves a frame from the queue, fills slots with message's argument, and launches the frame as a thread ready for execution, in the run-queue.

The object's code is symmetric, the main difference lies in the frame inserted in the queue, that contains the code to execute if a reduction occurs, code-for-io!prints[x], rather than the argument, as was the case with the code for the message.

3.2 The Multi-Threaded Virtual Machine

The virtual machine's implementation consists on a org.tyco.vm Java package, that includes the following subpackages:

org.tyco.vm.core - the core virtual machine implementation;
org.tyco.vm.values - values assignable to a general purpose register;
org.tyco.vm.assemble - construction of a fragments table containing object representations for each fragment in the TyCOIL program;
org.tyco.vm.externals - supplies the org.tyco.vm.externals.External Java class, whose extension is required to implement services external to the machine.

The virtual machine is initialized with the program's table of fragments, the number of registers required to run the program, and an external services' table.

The multi-threaded architecture, illustrated in figure 3, contains several concurrent threads. Each thread has its own set of general purpose registers, and a program counter that points to the code fragment it is executing. The remainder of the data structures are shared by all threads, namely: the tables for fragments and external services; the heap, that stores all the frames and channels in current use; and the run-queue holding the tasks ready to execute.

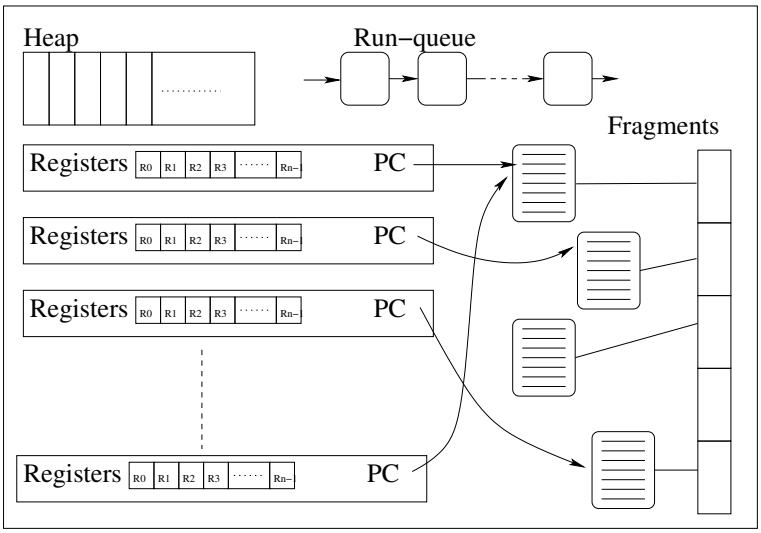

Fig. 3. The multi-thread virtual machine's architecture

The heap is currently managed by the Java runtime itself, while the interactions with the run-queue are performed through the org.tyco.vm.core.RunQueue class, more explicitly, through its enqueue and dequeue methods.

The TyCO virtual machine starts by creating the run-queue and spawning a designated number of threads. Once thread execution is triggered, each of them tries, concurrently, to retrieve a new task from the run-queue. This concurrent behavior implies taking run-queue access control measures to insure data consistency and to prevent race conditions. The code blocks of the org.tyco.vm.core.Runqueue's enqueue and dequeue methods are critical sections as they can change the shared run-queue status. Access to these methods is made mutually exclusive by declaring the methods synchronized.

Every time a thread executes a schedule instruction it tries to retrieve another task from the run-queue. In order to avoid a polling loop, continuously checking for new work when the run-queue is empty, a wait/notify mechanism is used to control access to the run-queue. Thus, if the run-queue is empty, any eager work-seeking thread is put on hold.

After a new task is added to the run-queue, waiting threads are notified that work is available. The machine halts when the run-queue is empty and all threads reach a waiting status.

As seen in the example in figure 2 a sequence of instructions is required to retrieve or add a frame to a channel's queue. This action consists of getting or adding a new frame to the channel's queue and setting the channel's status accordingly. On the multi-threaded virtual machine channels can be used by any running thread. The need for exclusive channel access during this operations is imperative. This is achieved by using the TyCOIL instructions lock and unlock

(bold in figure 2) that explicitly tell the virtual machine the limits of a critical region.

4 Future Work

Work in the TyCO language and runtime system is ongoing. At the virtual machine level we plan to experiment with hardware platforms supporting multi-threading (e.g., Intel's Pentium IV Hyper-threading feature) to evaluate the system's performance.

TyCO is also being used as the building block for the development of a language for programming distributed systems with support for code mobility [10]. The development of the multi-threaded virtual machine is of great importance as it provides the run-time for the nodes of such a system.

Acknowledgement. This work was supported by projects Mikado (IST2001-32222) and MIMO (POSI/CHS/39789/2001), and the CITI research center.

References

1. Blumofe R. D. and Joerg C. F. et.al. Cilk: an efficient multithreaded runtime system. *ACM SigPlan Notices*, 30(8):207–216, August 1995.
2. McGraw J. and Skedzielewski S. et.al. *The SISAL Language Reference Manual – Version 1.2*, March 1985.
3. Honda K. and Tokoro M. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.
4. Lopes L., Vasconcelos V., and Silva F. Fine grained multithreading with process calculi. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'00)*, pages 217–226. IEEE Computer Society Press, October 2000.
5. Nikhil R. The Parallel Programming Language Id and its Compilation for Parallel Machines. *International Journal of High Speed Computing*, 5:171–223, 1993.
6. Vasconcelos V. Typed Concurrent Objects. In *European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *LNCS*, pages 100–117. Springer-Verlag, July 1994.
7. Vasconcelos V. Core-TyCO, appendix to the language definition, yielding version 0.2. DI/FCUL TR 01–5, Departamento de Informática da Faculdade de Ciências de Lisboa, July 2001.
8. Vasconcelos V. TyCO Gently. DI/FCUL TR 01–4, Departamento de Informática da Faculdade de Ciências de Lisboa, July 2001.
9. Vasconcelos V. and Lopes L. The TyCO Intermediate Language. To appear.
10. Vasconcelos V., Lopes L., and Silva F. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.
11. Vasconcelos V. and Tokoro M. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.