A Virtual Machine for a Process Calculus

Luís Lopes[1], Fernando Silva[1], and Vasco T. Vasconcelos[2]

[1] DCC-FC & LIACC, Universidade do Porto,
Rua do Campo Alegre, 823, 4150 Porto, Portugal
{lblopes,fds}@ncc.up.pt
[2] DI-FC, Universidade de Lisboa,
Campo Grande, 1700 Lisboa, Portugal
vv@di.fc.ul.pt

Abstract. Despite extensive theoretical work on process-calculi, virtual machine specifications and implementations of actual computational models are still scarce.
This paper presents a virtual machine for a strongly typed, polymorphic, concurrent, object-oriented programming language based on the TyCO process calculus. The system runs byte-code files, assembled from an intermediate assembly language representation, which is in turn generated by a compiler. Code optimizations are provided by the compiler coupled with a type-inference system. The design and implementation of the virtual machine focuses on performance, compactness, and architecture independence with a view to mobile computing. The assembly code emphasizes readability and efficient byte code generation. The byte code has a simple layout and is a compromise between size and performance. We present some performance results and compare them to other languages such as Pict, Oz, and JoCaml.
Keywords: Process-Calculus, Concurrency, Abstract-Machine, Implementation.

1 Introduction

In recent years researchers have devoted a great effort in providing semantics for pure concurrent programming languages within the realm of process-calculi. Milner, Parrow and Walker's π-calculus or an equivalent asynchronous formulation due to Honda and Tokoro has been the starting point for most of these attempts [9, 17].

In this paper we use Vasconcelos' Typed Concurrent Objects to define a strongly typed, polymorphic, concurrent, object-oriented language named TyCO [23, 25]. Typed Concurrent Objects is a form of the asynchronous π-calculus featuring first class objects, asynchronous messages, and template definitions. The calculus formally describes the concurrent interaction of ephemeral objects through asynchronous communication. Synchronous communication can be implemented with continuations. Templates are specifications of processes abstracted on a sequence of variables allowing, for example, for classes to be modeled.

Unbounded behavior is modeled through explicit instantiation of recursive templates. A type system assigns monomorphic types to variables and polymorphic types to template variables [25]. Other type systems have been proposed that support non-uniform object interfaces [20]. The calculus is reminiscent of the Abadi and Cardelli's ς-calculus in the sense that objects are sums of labeled methods attached to names, the *self* parameters, and messages can be seen as asynchronous method invocations [3].

TyCO is a very low-level programming language with a few derived constructs and constitutes a building block for higher level idioms. We are interested in using TyCO to study the issues involved in the design and implementation of languages with run-time support for distribution and code mobility. In this paper we focus on the architecture and implementation of a sequential run-time system for TyCO. Introducing distribution and mobility is the focus of a cooperating project [24]. Our long term objectives led us to the following design principles:

1. the system should have a compact implementation and be self-contained;
2. it should be efficient, running close to languages such as Pict [19], Oz [16] or JoCaml [2];
3. the executable programs must to have a compact, architecture independent, format.

The architecture of the run-time system is a compact byte-code emulator with a heap for dynamic data-structures, a run-queue for fair scheduling of byte-code and two stacks for keeping local variable bindings and for evaluating expressions. Our previous experience in parallel computing makes us believe that more compact designs are better suited for concurrent object-oriented languages, whether we want to explore local and typically very fine grained parallelism or concurrency, or evolve to mobile computations over fast heterogeneous networks where the latencies must be kept to the lowest possible.

The remainder of the paper is organized as follows: section 2 introduces the TyCO language; sections 3 describes the design and some implementation details of the run-time system; section 4 describes the optimizations implemented in the current implementation; section 5 presents some performance figures obtained with the current implementation, and finally; sections 6 and 7, respectively overview some related work, present some conclusions and future research issues.

2 Introducing TyCO

TyCO is a strongly, implicitly typed concurrent object-oriented programming language based on a predicative polymorphic calculus of objects [23, 25]. TyCO is a kernel language for the calculus, and grows from it by adding primitive types with a set of basic operations, and a rudimentary I/O system.

In the sequel we introduce the syntax and semantics of TyCO. The discussion is much abbreviated due to space constraints. For a fully detailed description the reader may refer to the language definition [23].

Syntax Overview The basic syntactic categories are: *constants* (booleans and integers), ranged over by $c, c', \ldots$; *value variables*, ranged over by $x, y, \ldots$; *expressions*, ranged over by $e, e', \ldots$; *labels*, ranged over by $l, l', \ldots$, and; *template variables*, ranged over by $X, Y, \ldots$. Let $\tilde{x}$ denote the sequence $x_1 \cdots x_k$, with $k \geq 0$, of pairwise distinct variables (and similarly for $\tilde{e}$ where the expressions need not be distinct). Then the set of processes, ranged by $P, Q, \ldots$ is given by the following grammar.

P	::=	$\texttt{inaction}$	terminated process
	|	$P \mid P$	concurrent composition
	|	$\texttt{new}\ \tilde{x}\ P$	channel declaration
	|	$x\ !\ l[\tilde{e}]$	message
	|	$x\ ?\ M$	object
	|	$X[\tilde{e}]$	instance
	|	$\texttt{def}\ D\ \texttt{in}\ P$	recursion
	|	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P$	conditional
	|	(P)	grouping
D	::=	$X_1(\tilde{x}_1) = P_1\ \texttt{and} \ldots\ X_k(\tilde{x}_k) = P_k$	template declaration
M	::=	$\{l_1(\tilde{x}_1) = P_1\ , \ldots,\ l_k(\tilde{x}_k) = P_k\}$	methods
e	::=	$e_1\ op\ e_2 \mid op\ e \mid x \mid c \mid (e)$	expressions

Some restrictions apply to the above grammar, namely: a) no collection of methods may contain the same label twice; b) no sequence of variables in a template declaration or collection of methods may contain the same variable twice, and; c) no declaration may contain the same template variable twice.

The method l_i in an object $x?\{l_1(\tilde{x}_1) = P_1, \ldots, l_k(\tilde{x}_k) = P_k\}$ is invoked with a message $x!l_i[\tilde{e}]$; the result is the process P_i where the variables in $\tilde{x}_i$ are replaced by the values of the expressions in $\tilde{e}$. This form of reduction is called *communication*. Similarly, an instance $X_i[\tilde{e}]$ selects the template X_i in a template declaration $X_1(\tilde{x}_1) = P_1\ \texttt{and} \ldots\ X_k(\tilde{x}_k) = P_k$; the result is the process P_i where the variables in $\tilde{x}_i$ are replaced by the values of the expressions in $\tilde{e}$. This form of reduction is called *instantiation*.

We let the scope of variables, introduced with `new`, extend as far to the right as possible, i.e., up to the end of the current composition of processes. We single out a label — `val` — to be used in objects with a single method. This allows us to abbreviate the syntax of messages and objects. Single branch conditionals are also defined from common conditionals with an `inaction` in the else branch.

$$
\begin{array}{ll}
x![\tilde{e}] & \equiv x!\texttt{val}[\tilde{e}] \\
x?(\tilde{y}) = P & \equiv x?\{\texttt{val}(\tilde{y}) = P\} \\
\texttt{if}\ e\ \texttt{then}\ P & \equiv \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else inaction}
\end{array}
$$

To illustrate the programming style and syntax of the language we sketch a simple example: a single element polymorphic cell. We define a template object

with two attributes: the `self` channel and the value `u` itself. The object has two methods one for reading the current cell value and another to change it. The recursion keeps the cell alive.

```
def Cell( self, u ) =
self ? {
    read( r ) = r![u] | Cell[self, u],
    write( v ) = Cell[self, v]
}
in  new x Cell[x,9] | new y Cell[y,true]
```

The continuation of the definition instantiates an integer cell and a boolean cell at the channels `x` and `y`, respectively.

3 The Virtual Machine

The implementation of the virtual machine is supported by a formal specification of an abstract machine for TyCO[15]. This abstract machine grows from Turner's abstract machine for Pict [22], but modifies it in the following major ways:

1. objects are first class entities and substitute input processes. Objects are more efficient than Pict's encoding in π [26] both in reduction and heap usage;
2. we use recursion instead of replication for persistence. This allows a cleaner design of the abstract machine – no need for distinct `?` and `?*` rules, and allows a more rational heap usage;
3. we introduce a new syntactic category – the *thread* – that represents the basic schedulable and runnable block in the abstract machine. Threads are identified as bodies of template definitions or method implementations;
4. threads cannot be suspended. With this property, our objects are very akin to actors and provide a good model for object oriented concurrent languages [4, 5]. This choice, along with the previous item, also simplifies the treatment of local bindings, introduced with `new` statements, and the management of environments.

The abstract machine is sound, i.e., every state transition in the abstract machine can be viewed as a reduction or a congruence between their process encodings in the base calculus [15]. It also features important run-time properties such as: a) at any time during a computation the queues associated with names are either empty or either have communications or method-closures [22]; b) for well-typed programs the abstract machine does not deadlock. This property is linked intimately to the ability of the type system to guarantee that no run-time protocol errors will occur, and; c) the machine is fair, in the sense that every runnable thread will be executed in a constant time after its creation.

The virtual machine closely maps the formal specification and executes TyCO programs quite efficiently.

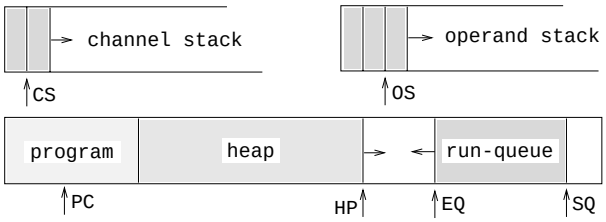

Fig. 1. Memory Layout of the Virtual Machine

The Memory Layout The virtual machine uses five logically distinct memory areas (figure.1) to compute.

The *program* area keeps the byte-code instructions to be executed. The byte-code is composed of instruction blocks and method tables (sequences of pointers to byte-code blocks).

Dynamic data-structures such as objects, messages, channels and builtin values are allocated in the *heap*. The basic building block of the heap is a machine *word*. The basic allocation unit in the heap is the *frame* and consists of one or more contiguous heap words with a descriptor for garbage collection.

When a reduction (either communication or instantiation) occurs, a new virtual machine thread (*vm_thread*) is created. The new vm_thread is simply a frame with a pointer to the byte-code and a set of bindings, and is allocated in the *run-queue* where it waits to be scheduled for execution. Using a run-queue to store vm_threads ready for execution provides fairness. The heap and the run-queue are allocated in the bottommost and topmost areas, respectively, of a single memory block. They grow in opposite directions and garbage collection is triggered when a collision is predicted.

Local variables, introduced with `new` statements, are bound to fresh channels allocated in the heap and the bindings (pointers) are kept in the *channel stack*. These bindings are discarded after a vm_thread finishes but the channels, in the heap, may remain active outside the scope of the current vm_thread through scope extrusion.

Finally, expressions with builtin data-types are evaluated in the *operand stack*. Simple values do not require evaluation and are copied directly in the heap. Using an operand stack to perform built-in operations enables the generation of more compact byte-codes since many otherwise explicit arguments (e.g., registers for the arguments and result of an operation) are implicitly located at the top of the stack.

Heap Representation of Processes and Channels TyCO manipulates three basic kinds of processes at runtime: messages, objects and instantiations (figure 2). Messages and objects are located in shared communication channels. Internally, the virtual machine sees all these abstractions as simple frames, although their internal structure is distinct. A message frame holds the label of the method it is invoking plus a variable number of arguments. An object

frame, on the other hand, holds a pointer to the byte-code (the location of its method table) plus a variable number of bindings for variables occurring free in its methods. An instance frame has a pointer to the byte-code for the template and a variable number of arguments.

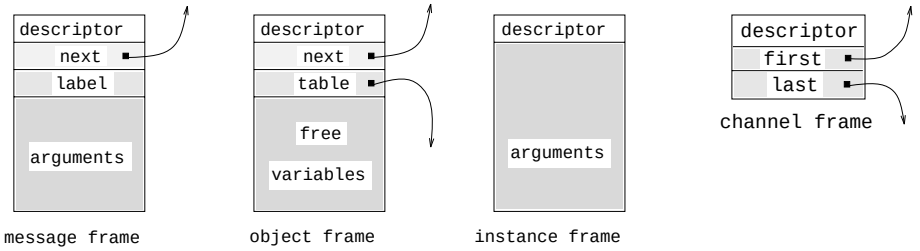

Fig. 2. Message, object, instance and channel frames.

Channel frames hold communication queues which at run-time have either only objects or only messages or are empty. The first word of the three that compose a channel is the descriptor for the frame which in this case also carries the state of the channel. This state indicates the internal configuration and composition of the queue. The other two words hold pointers to, respectively, the first and last message (object) frame in the queue.

The Machine Registers The virtual machine uses a small set of global registers to control the program flow and to manipulate machine and user data-structures. Register `PC` (Program Counter) points to the next instruction to be executed. Register `HP` (Heap Pointer) points to the next available position in the heap. Registers `SQ` (Start Queue) and `EQ` (End Queue) point to the limits of the run-queue. Finally, registers `OS` (Operand Stack) and `CS` (Channel Stack) point to the last used position in each area.

When a program starts, register `PC` is loaded with the address of the first instruction. Register `CC` (Current Channel) points to the channel which is currently being used to try a reduction. Register `CF` (Current Frame) holds frames temporarily until they are either enqueued or used in a reduction. If a reduction takes place, the frame for the other redex's component is kept in the register `OF` (Other Frame). Registers `FV` (Free Variable bindings) and `PM` (ParaMeter bindings) are used to hold the free variable and parameter bindings, respectively.

The Instruction Set The virtual machine instruction set was intentionally designed to be minimal in size and to have a very simple layout. Instructions are identified by an unique opcode held in the word pointed to by the program counter `PC`. For most instructions the opcode of an instruction determines the number of arguments that follow in contiguous words. Alternatively, the first argument indicates the number of remaining arguments as in `switch`. In the

sequel we let n,k range over the natural numbers, l over code labels and w over machine words (representing constants or heap references).

The first set of instructions is used to allocate heap space for dynamic data-structures.

`msgf` n,k `objf` n,l `instf` n `newc` k

`msgf` n,k allocates a frame for a message with label k and with n words for the arguments. `objf` n,l does the same for an object with a method table at l and n words for free variable bindings. `instf` n allocates a frame for a template instance with n words for arguments. `newc` k allocates a channel in the heap and keeps a reference for it in the stack position k.

Next we have instructions that move data, one word at a time, within the heap and between the heap and the operand and channel stacks.

`put` k, w `push` w `pop` k

`put` k, w copies the word w directly to the position k in the frame currently being assembled. `push` w pushes the data in w to the top of the operand stack. `pop` k moves the result of evaluating an expression from the top of the operand stack to the position k in the frame currently being assembled.

We also need the following basic control flow instructions.

`if` l `switch` n,l_1,,...,l_n `jump` l `ret`

`if` l jumps to label l if the value at the top of the operand stack is (boolean) false. `switch` n,l_1,,...,l_n jumps to label l_k, where k is taken from the top of the operand stack. `jump` l jumps unconditionally to the code at label l. Finally, `ret` checks the halt condition and exits if it holds; otherwise it loads another vm_thread for execution from the run-queue.

For communication queues we need instructions to check and update their state and insert and remove items from the queues.

`state` w `update` k `reset` `enqueue` `dequeue`

`state` w takes the state of the channel in word w and places it at the top of the operand stack. `update` k changes the state of the current channel to k. In a unoptimized setting the state of a channel is 0 if it is empty, 1 if it has messages or 2 if it has objects. `reset` sets the state to 0 if the current channel is empty. `enqueue` enqueues the current frame (at `CF`) in the current channel (at `CC`) whereas `dequeue` dequeues a frame from the current channel (at `CC`) and prepares it for reduction (placing it at `OF`).

Finally, we require instructions that handle reductions: both communication and instantiation.

`redobj` w `redmsg` w `instof` l

`redobj` w reduces the current object frame with a message. `redmsg` w is similar, reducing the current message frame with an object. Finally, `instof` l creates a

new vm_thread, an instance of the byte-code at label l, with the arguments in the current frame.

In addition to this basic set there is a set of operations on builtin data-types and also specialized instructions that implement some optimizations. For example, in case a sequence of words is copied preserving its order, the `put` and `pop` operations may be replaced by optimized versions where the k argument is dropped.

The Emulator Before running a byte-code file, the emulator links the opcodes in the byte-code to the actual addresses of the instructions (C functions). It also translates integer offsets within the byte-code into absolute hardware addresses. This *link* operation avoids an extra indirection each time a new instruction is fetched and also avoids the computation of addresses on-the-fly. The emulator loop is a very small `while` loop adapted from the STG machine [11]. Each instruction is implemented as a parameterless C function that returns the address of the following instruction. The emulation loop starts with the first instruction in the byte-code and ends when a NULL pointer is returned. The emulator halts whenever a vm_thread ends and the run-queue is empty.

Garbage Collection A major concern of the implementation is to make the emulator run programs in as small a heap space as possible. Efficient garbage collection is essential for such a goal. The emulator triggers garbage collection whenever the gap between the top of the heap, pointed to by `HP`, and the end of the run-queue, pointed to by `EQ`, is smaller than the required number of words to execute a new vm_thread.

Making this test for every instruction that uses the heap before actually executing it would be very costly. Instead, when a byte-code file is assembled the maximum number of heap words required for the execution of each vm_thread is computed and placed in the word immediately preceding the first instruction of the vm_thread. At run-time, before starting executing a new vm_thread, the emulator checks whether there is enough space in the heap to safely run it. If not then garbage collection is triggered. The emulator aborts if the space reclaimed by the garbage collector is less than `required`.

If the space between the heap limit and `SQ` is enough, then the garbage collector just shifts the run-queue upwards and returns; if not, it must perform a full garbage collection. We use a *copying* garbage collection algorithm. The algorithm performs one pass through the run-queue to copy the active frames. Active frames are those that can still be accessed by taking each item in the run-queue as the root of a tree and recursively following the links in the heap frames. The garbage collector does not use any knowledge about the internal structure of the frames (e.g., if they represent objects or messages).

Compilation TyCO programs are compiled into the virtual machine instruction set by the language compiler. This instruction set maps almost one-to-one with the byte-code representation. The syntax of the intermediate representation reflects exactly the way the corresponding byte-code is structured. It is important

that the compilation preserves the nested structure of the source program in the final byte-code. This provides a very efficient way of extracting byte-code blocks at run-time when considering code mobility in distributed computations [14, 24]. To illustrate the compilation we present a skeletal version of the unoptimized, intermediate code for the `Cell` example presented in section 2. The run-time environment of a vm_thread is distributed into three distinct locations: the parameter and free variable bindings pointed to by registers `PM` and `FV`, respectively, and the bindings for local variables held in the channel stack `CS`. In the machine instructions the words `PM`$[k]$, `FV`$[k]$ and `CS`$[k]$ are represented by `p`k, `f`k and `c`k, respectively.

```
main  = {
   def Cell = {
      objf     2                  % self located object with parameters
      put      p0                 % p0=self
      put      p1                 % p1=u
      trobj    p0 = {
         { read, write }
         read = {                 % the method 'read', p0=r,f0=self,f1=u
            msgf     1,0          % message r![u]
            put      f1
            trmsg    p0
            instof   2,Cell       % instantiation Cell[self,u]
            put      f0
            put      f1
         }
         write = {                % the method 'write', p0=v,f0=self,f1=u
            instof   2,Cell       % instantiation Cell[self,v]
            put      f0
            put      p0
         }
      }
   }
   newc     c0                    % creation of x
   instof   2,Cell                % instantiation Cell[x,9]
   put      c0
   put      9
   newc     c1                    % creation of y
   instof   2,Cell                % instantiation Cell[y,true]
   put      c1
   put      true
}
```

4 Optimizations

This section describes a sequence of optimizations that we applied to the emulator with support from the compiler to improve performance. Some optimizations rely on type information, namely channel usage properties, gathered at compile time by the type inference system [6, 12, 13, 23].

Compacted messages and objects This optimization is due to Turner [22]. Given that at any time a channel is very likely to hold at most a single object or message frame in its queue before a communication takes place, we optimize the channel layout for this case. The idea is to avoid the overhead of queuing and dequeuing frames and instead to access the frame contents directly.

Optimizing frame sizes We minimize the size of the frames required by each process frame. For example, messages or objects in a synchronization channel do not require a `next` field. This minimizes heap consumption and improves performance.

Single method objects These objects do not require a method table. The compiler generates the code for an object with an offset for the byte-code of the method, instead of an offset for a method table. This avoids one extra indirection before each method invocation.

Fast reduction This optimization can be performed in cases where we can assure that a given channel will have exactly one object. Two important cases are: uniform receptors [21] where a persistent object is placed in a channel and receives an arbitrary number of messages, and; linear synchronization channels [13] where an ephemeral object and a message meet once in a channel for synchronization and the arrival order is unknown. The main point of this optimization is that we never allocate the channel in the heap to hold the object (figure 3a). We just create a frame for the object in the heap and use its pointer directly as a binding (figure 3b).

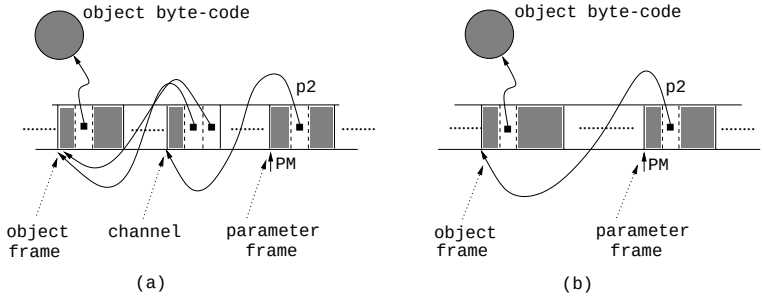

Fig. 3. Fast reduction

In the case of uniform receptors, when a message arrives for this binding (`p2` in the figure) we can reduce at once since the binding already holds a pointer to the object frame. For linear channels, if we have, say, a message for `p2`, we first check the value at `p2`. If it is null we assign it the pointer for the message frame, otherwise the binding must point to an object frame and reduction is immediate. What distinguishes persistent from linear synchronization channels in our model

is the fact that, since we use recursion to model persistence, a persistent object must always have a self referencing pointer in its closure when it goes to the run-queue. This preserves the frame if garbage collection is triggered. In the case of a linear channel this link cannot exist and so, the object frame only lasts until the resulting vm_thread ends.

Merging instructions Certain instructions occur in patterns that are very common, sometimes pervasive, in the intermediate assembly. Since an instruction-by-instruction execution is expensive (one unconditional jump per instruction) we create new instructions that merge a few basic ones to optimize for the common case. One such optimization is shown for macros `trobj` and `trmsg`, used to try to reduce objects and messages immediately. For example, `trobj` w checks the state of channel w. If it has messages then it dequeues a message frame and creates a new vm_thread in the run-queue from the reduction with the current object. If the channel state is either empty or already has objects the current frame is enqueued. The case for `trmsg` w is the dual.

```
trobj w:
       state     w
       switch    3,empty,msg,obj
empty: enqueue
       update    2
       jump      end
msg:   dequeue
       redobj    w
       reset
       jump      end
obj:   enqueue
end:
```

```
trmsg w:
       state     w
       switch    3,empty,msg,obj
empty: enqueue
       update    1
       jump      end
msg:   enqueue
       jump      end
obj:   dequeue
       redmsg    w
       reset
end:
```

Inline arguments in the run-queue Each time a template instantiation or a fast reduction occurs we copy the arguments directly to the run-queue. The advantage of the optimization is that the space used for vm_threads in the run-queue is reclaimed immediately after the vm_thread is selected for execution. This increases the number of *fast* (and decreases the number of *full*) garbage collections. Fast garbage collections are very light weight involving just a shift of the run-queue.

5 System Performance

Currently, we have an implementation of the TyCO programming language which includes a source to assembly compiler and a byte-code assembler. We have chosen to separate the intermediate assembly code generation from the byte-code generation to allow us more flexibility namely in programming directly in assembly. The emulator is a very compact implementation of the virtual

machine with about 4000 lines of C code. Figure 4 illustrates the architecture of the TyCO system.

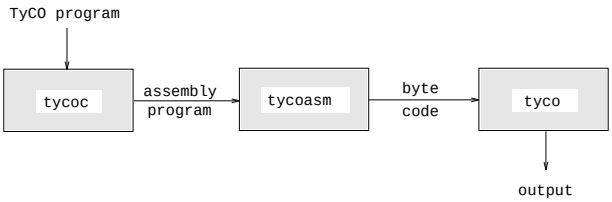

Fig. 4. The TyCO system

We used a set of small programs to measure the efficiency of the current implementation relative to the concurrent programming languages Pict [19], Oz [16] and the JoCaml implementation of the Join calculus [8]. Both JoCaml and Oz may use byte-codes whereas Pict generates binary files from intermediate C code.

The values presented in table 1 are just indicative of the system's performance in terms of speed. These helped us in fine-tuning the implementation. A full performance evaluation will have to address larger, *real world* applications [18].

The benchmark programs used include some standard programs such as `tak`, `sieve` and `queens`, and three other larger programs, `mirror`, `graph` and `fourier`, that best illustrate the potential of the language. `mirror` takes a tree with 10k nodes, leafs and buds, and builds another one which is its mirror image. It uses objects for pattern matching and is deeply recursive. `graph` takes a connected graph with 128 nodes and traverses it mapping a function on each node's attribute. Each node of the graph is an object with an integer attribute. The computation time for each node is exponential on the integer attribute. `fourier` takes a list of complex numbers (implemented as objects) and computes its Discrete Fourier Transform. We have implemented all programs in TyCO 0.2, Pict 4.1 [19], Oz 2.0.4 [16] and JoCaml [2] to compare the performance of these systems.

Table 1 shows, for each program, the smallest execution time of 10 consecutive runs in seconds. All the values observed were obtained with a default heap space of 256k words (when possible), in a 233MHz Pentium II machine with 128Mb RAM, running Linux. With Oz we used the switches `+optimize` and `-threadedqueries` to get full performance from the Oz code. Both Pict and the TyCO emulator were compiled with `-O3 -fommit-frame-pointer` optimization flags.

The initial results show that TyCO's speed is clearly in the same order of magnitude as Pict and Oz, and indeed compares favorably considering it is emulated code. TyCO is clearly faster than JoCaml as can be seen in the rightmost portion of the table. These results were obtained for distinct problem sizes, relative to the first set, since for some programs JoCaml ran into some problems

Program	Pict	Oz	TyCO	Program	JoCaml	TyCO
`tak*10` 22,16,8	4.40	11.7	24.31	`tak` 22,16,8	6.98	3.02
`queens*10` 10	17.0	33.1	58.09	`queens` 8	2.46	0.39
`sieve*10` 10k	9.90	19.2	20.92	`lsieve` 4k	62.37	8.33
`mirror*10` 10k	3.13	4.20	1.21	`mirror` 10k	2.51	0.24
`graph*10` 128	9.62	7.10	7.24	`graph` 128	—	—
`fourier*10` 64	2.30	2.60	0.91	`fourier` 64	0.95	0.21

Table 1. Execution times

either compiling (e.g., `graph`) or running them (e.g., `queens` 10). `lsieve` uses non-builtin lists to implement the sieve of primes while `sieve` uses a chain of objects. The performance gap is higher than average in the case of functional programs a fact that is explained by the optimized code generated by both Pict and Oz for functions. Further optimization, namely with information from the type system may allow this gap to be diminished. The performance ratio for applications that manipulate large numbers of objects (with more than one method), on the other hand, clearly favors TyCO. For example `mirror`, which uses objects to implement a large tree and to encode pattern matching, performs nearly three times faster than Pict and even more for Oz and JoCaml. Also notice that all the Oz programs required an increase in the heap size up to 3M words and once (`queens`) to 6M to terminate. Compare this with the very conservative 256k used by both in TyCO and Pict. The exception for Pict is `fourier` where there is a lot of parallelism and method invocations. Pict required 1.5M words to run the program, as opposed to 256k words in TyCO, and was about 2.3 times slower than TyCO. `fourier` shows that objects in both Pict and Oz are clearly less efficient than in TyCO. Pict showed lower performance on the object based benchmarks. This is due to the fact that the encoding of objects in the π-calculus is rather inefficient both in speed and heap usage.

	TyCO			**Pict**		
Program	Heap	shift-gc	full-gc	Heap	shift-gc	full-gc
`tak` 22,16,8	13487	406	18	11496	225	32
`queens` 10	12975	591	13	13933	390	55
`sieve` 10k	11110	218	31	11941	253	36
`mirror` 10k	505	4	0	1094	27	5
`graph` 128	3336	101	5	5769	139	25
`fourier` 64	538	18	1	4870	2	0

Table 2. Total heap usage and number of garbage collections.

Table 2 shows that TyCO uses more heap space than Pict in functional applications such as `tak`. The situation changes completely when we switch to programs with object based data-structures. TyCO performs more *shift* garbage collections since it uses the run-queue to store the arguments of instances and some messages directly and, on average, TyCO uses one extra word per heap frame. This increases the number of collisions with the top of the heap. On the other hand the number of *full* garbage collections performed is substantially smaller in TyCO. This is mostly due to the combined effect of the inlining of arguments in the run-queue and to the fact that TyCO does not produce run-time heap garbage in the form of unused channels or process closures. Pict produces significant amounts of heap garbage in applications where objects are pervasive since each object is modeled with: a) one channel for the object, one channel for each method and one process closure per method. Most of the times only a subset of these will actually be used. On the other hand an object in TyCO just requires a channel (*self*) and one closure for the method collection and the free variables. This effect is plainly visible in `mirror` for example.

We tried to measure the amount of overhead generated by our emulation strategy using a small test program that has a similar emulation cycle but always invokes the same function. An artificial addition was introduced in the body of the function as a way to simulate the overhead of adjusting the program counter for the next instruction, as is done in the actual machine. Running this small program with the optimizations `-O3 -fomit-frame-pointer`, we found that the emulation overhead accounts for 15 to 28% of the total execution time, with the higher limit observed for functional programs.

The run-time system for TyCO is very lightweight. It uses byte-codes to provide a small, architecture independent, representation of programs. The engine of the system is a compact (the binary occupies 39k) and efficient emulator with light system requirements, namely it features a rather conservative use of the heap.

This system architecture provides in our opinion the ideal starting point for the introduction of distribution and code mobility, which is the focus of an ongoing project.

6 Related Work

We briefly describe the main features of some concurrent process-based programming languages that relate to our work.

Pict is a pure concurrent programming language based on the asynchronous π-calculus [19]. The run-time system is based on Turner's abstract machine specification and the implementation borrows from the C and OCaml programming languages [22]. The basic programming abstractions are processes and names (channels). Objects in Pict are persistent with each method implemented as an input process held in a distinct channel. The execution of methods in concurrent objects by a client process is achieved by first interacting with a server process (that serves requests to the object and acts like a *lock* ensuring mutual exclu-

sion), followed by the method invocation proper [26]. This protocol for method invocation involves two synchronizations as opposed to one in TyCO that uses branching structures. Moreover, this encoding of objects produces large amounts of computational garbage in the form of unused channels and process closures. Also, Pict uses replication to model recursion whereas TyCO uses recursion. Recursion not only provides a more natural programming model but also allows replication only when it is strictly needed, avoiding the generation of unused process.

Oz is based on the γ-calculus and combines the functional, object-oriented and constraint logic programming paradigms in a single system [16]. The Oz abstract machine is fully self contained and its implementation is inspired in the AKL machine [10]. The basic abstractions are names, logical variables, procedural abstraction and cells. Constraints are built over logical variables by first-order logic equations, using a set of predefined predicates. Cells are primitive entities that maintain state and provide atomic read-write operations. Channels are modeled through first class entities called *ports*. They are explicitly manipulated queues that can be shared among threads and can be used for asynchronous communication. Oz procedures have encapsulated state so that objects can be defined directly as sets of methods (procedures) acting over their state (represented as a set of logical variables). This representation is close to objects in TyCO if we view the state held in logical variables as template parameter bindings.

Join implements the Join-calculus [7, 8]. The JoCaml implementation integrates Join into the OCaml language. Join collapses the creation of new names, reception and replication into a single construct called a *join pattern*. Channels, both synchronous and asynchronous, expressions and processes are the basic abstractions. Programs are made of processes, communicating asynchronously and producing no values, and expressions evaluated synchronously and producing values. Processes communicate by sending messages on channels. Join patterns describe the way multiple processes (molecules) may interact with each other (the reactions) when receiving certain messages (molecules) producing other processes (molecules) plus eventual variable bindings. The JoCaml implementation has some fairly advanced tools for modular software development inherited from its development language OCaml and supports mobile computing [1].

7 Conclusions and Future Work

We presented a virtual machine for a programming language based on Typed Concurrent Objects, a process-calculus [25]. The virtual machine emulates byte-code programs generated by a compiler and an assembler. The performance of the byte-code is enhanced with optimizations based on type information gathered at compile-time. Preliminary results are promising and there is scope for plenty of optimizations. The current implementation performs close to Pict and Oz on average and clearly surpasses JoCaml. TyCO is faster in applications using objects and persistent data structures, despite being emulated. TyCO consistently

runs in very small heap sizes, and performs significantly less garbage collection than either Pict or Oz.

Future work will focus on performance evaluation and fine tuning of the system using larger, *real world* applications [18]. Channel usage information from type systems such as those described can dramatically optimize the assembly and byte-code [13, 12]. An ongoing project is introducing support for mobile computing in this framework as proposed in [24]. A multi-threaded/parallel implementation of the current virtual machine is also being considered since it will provide an interesting model for parallel data-flow computations.

The TyCO system, version 0.2 (*alpha* release) may be obtained from the web site: `http://www.ncc.up.pt/~lblopes/tyco.html`.

Acknowledgments We would like to thank the anonymous referees for their valuable comments and suggestions. The authors are partially supported by projects Dolphin (contract PRAXIS/2/2.1/TIT/1577/95), and DiCoMo (contract PRAXIS/P/EEI/12059/98).

References

1. Objective CAML Home Page. `http://pauillac.inria.fr/ocaml`.
2. The JoCaml Home Page. `http://pauillac.inria.fr/jocaml`.
3. M. Abadi and L. Cardelli. *A Theory Of Objects*. Springer-Verlag, 1996.
4. G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.
5. G. Agha and C. Hewitt. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. *Research Directions on Object-Oriented Programming*, 1981. Shiver and Wegner, editors. MIT Press.
6. Roberto M. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In COORDINATION'97, volume 1282 of LNCS, pages 374–391. Springer-Verlag, 1997.
7. C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM, 1996.
8. C. Fournet and L. Maranget. *The Join-Calculus Language (release 1.02)*. Institute National de Recherche en Informatique et Automatique, June 1997.
9. K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *5th European Conference on Object-Oriented Programming*, volume 512 of *LNCS, Springer-Verlag*, pages 141–162, 1991.
10. S. Janson. *AKL - A Multiparadigm Programming Language*. PhD thesis, SICS Swedish Institute of Computer Science, Uppsala University, 1994.
11. S. Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, July 92.
12. N. Kobayashi. Quasi-Linear Types. In *26^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 29-42, January 1999.
13. N. Kobayashi, B. Pierce, and D. Turner. Linearity and the π-calculus. In *ACM Symposium on Principles of Programming Languages*, 1996.

14. L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An Experiment in Code Mobility from the Realm of Process Calculi. In *5^{th} Mobile Object Systems Workshop*, 1999. part of ECOOP'99.
15. L. Lopes and V. Vasconcelos. An Abstract Machine for an Object-Calculus. Technical report, DCC-FC & LIACC, Universidade do Porto, May 1997.
16. M. Mehl, R. Scheidhauer, and C. Schulte. An Abstract Machine for Oz. Technical report, German Research Center for Artificial Intelligence (DFKI), June 1995.
17. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
18. W. Partain. The nofib Benchmark Suite of Haskell Programs. In J. Launchbury and P.M. Sansom, editors, *Proceedings of Functional Programming Workshop*, Workshops in Computing, pages 195–202. Springer Verlag, 1992.
19. B. Pierce and D. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in H onour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Toft e, editors, MIT Press, 1999.
20. António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Euro-Par'97*, volume 1300 of *LNCS*, pages 554–561. Springer Verlag, 1997.
21. D. Sangiorgi. The name discipline of receptiveness. In *24th ICALP*, volume 1256 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.
22. D. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.
23. V. Vasconcelos and R. Bastos. Core-TyCO - The Language Definition. Technical Report TR-98-3, DI / FCUL, March 1998.
24. V. Vasconcelos, L. Lopes, and F. Silva. Distribution and mobility with lexical scoping in process calculi. In *3rd International Workshop on High-Level Concurrent Languages*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
25. V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *1st International Symposium on Object Technologies for Advanced Software, LNCS*, volume 742, pages 460–474. Springer-Verlag, November 1993.
26. D. Walker. Objects in the π-calculus. *Journal of Information and Computation*, 116(2):253–271, 1995.