A Parallel Abstract Machine for the Thread-Based Concurrent Language Oz

Konstantin Popov

PS Lab, University of Saarland
Geb. 45, Postfach 15 11 50, D–66041 Saarbrücken, Germany
popow@ps.uni-sb.de

September 22, 1997

Abstract

A parallel abstract machine for Oz (PAMOz) is presented in this paper. Oz is a thread-based concurrent constraint programming language with state. Oz is convenient for concurrent programming like modelling multi-agent systems, as well as for solving combinatoric problems. PAMOz models the execution of a sublanguage of Oz without its constraint solving facilities. PAMOz has been implemented in the parallel Oz system, which is derived from the sequential Oz system and inherits its optimizations. PAMOz is targeted to shared-memory multiprocessors. PAMOz executes Oz threads in parallel. PAMOz is derived from AMOz, a sequential abstract machine for Oz. There are two principal differences between PAMOz and AMOz: the architecture of the abstract machine, and the implementation of operations on stateful data. PAMOz can be conservatively extended for full Oz; there is an interface between PAMOz and its constraint solving extension.

1 Introduction

PAMOz (**P**arallel **A**bstract **M**achine for **Oz**) is an abstract machine for a parallel implementation of Oz. This paper is dedicated to motivating it, explaining it, and sketching both its usage in the parallel Oz system and its extensions.

Parallelizing Oz. PAMOz is implemented in the parallel Oz system [Pop97] with the goal to achieve parallel speedups, compared to the existing sequential implementation of Oz [OzS]. Parallelizing Oz is promising because:

- Oz is convenient and efficient programming language for concurrent programming (e.g. modelling multi-agent systems) and constraint solving [HW95, Wür96b, Wür96a].
- There is an efficient sequential implementation of Oz. The parallel system is derived from it and preserves its optimizations.

- A parallel implementation of Oz is arguably less complex than implementations of similar concurrent constraint languages like AKL [JH94].

Oz [Smo95a, Smo95b] is a concurrent constraint (henceforth `cc`, see [SR90]) programming language with state. Computation in Oz is organized into sequential threads. Threads are created explicitly. Oz provides for convenient functional and object-oriented programming. It provides also for constraint solving based on its constraint systems (rational tree, finite domains and finite sets) and innovative control facilities as first-class computation spaces [Sch97b, SS94].

Concurrent threads and sequential composition within threads in Oz allow to exploit concurrency where it is necessary and avoid it where it is not desirable [Smo95a]. This makes it easy to model, for instance, a multi-agent system whose agents' behaviors are specified by sequential algorithms. In problem solving, a next node to be explored in a search tree is chosen by a sequential algorithm too. Such an algorithm consequently collects information about the current node, applies a heuristic to that information, and finally decides which node to take as a next one [Smo95a, MPSW94, MM97].

On the contrary, implicit concurrency in concurrent constraint languages like AKL makes it difficult to encode sequential algorithms. In general, implicit concurrency tends to be fine-grained. This has also performance disadvantages: as the amount of concurrency increases, lower parallel speedups can be obtained because of the scheduling overhead.

The parallel Oz system is derived from the sequential one [OzS]. The parallel system inherits optimizations of the sequential one. The parallel system running on a single processor approaches the speed of the sequential system: currently it is about 70% as fast, but evaluating and optimizing it is not yet completed.

The Oz sub-language for PAMOz. PAMOz models a parallel implementation of a subset of Oz without constraint solving facilities. In the following, this subset is called just Oz, while Oz with constraint solving facilities is referenced as full Oz.

Concurrent threads in Oz communicate and synchronize over a shared *store* by means of *logic variables*. The Oz store contains information in the form of *variable bindings*, produced by the binding operation. The matching conditional allows to synchronize on the store: a thread executing a conditional is blocked until a variable being matched is bound. One can view variable bindings in the store as constraints, and binding of a variable and synchronizing on a variable as instances of `ask/tell` operations in the `cc` languages.

Object-oriented programming in Oz is supported by Oz cells. An Oz cell is a stateful data structure that holds a variable or a value. A cell's content can be exchanged using the *cell exchange* operation.

This subset of Oz is choosen for PAMOz because it supports functional and object-oriented styles of programming, it makes PAMOz simple, and, finally, PAMOz is conservatively extendible for full Oz. Full Oz contains yet unification (known from Prolog), conditionals that check logical entailment of a constraint by the store, further constraint systems (finite domains and finite sets), and Oz first-class computation spaces.

PAMOz essentials. PAMOz is targeted for shared-memory multiprocessor (SMP in the following) computers. The Oz computation model is mapped naturally on SMP computers: Oz threads are executed by working in parallel processors, and the shared Oz store lies in the computer's shared memory.

PAMOz is derived from AMOz [MSS95], an abstract machine for the sequential implementation of Oz [OzS]. Both abstract machines are emulator-based, as opposed to native-code compilation systems. They inherit ideas for data representation and code generation from the WAM [AK91], and the representation scheme for first-class procedures and compilation of pattern matching from functional languages [PJ87].

PAMOz executes Oz threads in parallel. Threads are executed by running parallel *workers*. Workers are the only active entities in PAMOz. Runnable threads are taken from the *thread pool* which is shared among workers. The thread pool also implements a scheduling strategy, for instance – FIFO. Since the thread pool is shared among workers, access to it in the parallel Oz system is serialized in order to guarantee the internal consistency of the pool.

The PAMOz worker looks like a sequential Oz system: this is a procedure that executes Oz programs. Workers are executed in parallel by the operating system of an underlying SMP computer. The worker is a *stateful* entity; its state contains at least the current thread. The state is allocated explicitly and passed to a worker procedure when a worker is created. The state cannot be put into a static, globally visible region as it is done in the sequential Oz system, since there are multiple workers, each with its own state.

The Oz store is implemented in PAMOz as an abstract data type (in the following – just object). The store is shared among workers. It encapsulates an area to keep representations of Oz variables and data, and provides for methods for accessing that data. The store holds *store objects*. There are variable objects, value objects and *reference* objects. A variable is bound by replacing the variable object with a reference object. Blocking a thread on a variable (synchronizing with the conditional) results in preempting the thread and storing it in the variable object; a worker executed the thread goes for a next runnable one. Binding a variable that keeps blocked threads follows by scheduling the blocked threads for execution.

Maintaining stateful data in PAMOz, as well as in every parallel system, requires special care compared to a sequential abstract machine. This can be illustrated abstractly as follows. A new state of an object depends on a previous one, so if the state becomes changed after reading an old one but before writing a new one, that change will be overwritten and therefore just dissapears.

There are two stateful objects in PAMOz – variable objects and cell objects. Let us consider first executing a matching conditional that blocks its thread. It proceeds as follows: first it checks whether the variable being matched is bound (reads its state), and if it is not – the thread is put into the variable object. If the variable becomes bound between these two steps, trying to put the thread into the variable object will end up with the error: there is no variable object anymore, but the reference object.

A naive implementation of the cell exchange operation that first reads a cell's content and then stores a new one suffers from the same problem. Here, if two exchanges of the same cell interleave in such a way that a first one does its "read" and "write" between "read" and "write" of another one, the new content put by the first exchange will disappear.

Compared to Oz variables and cells, Oz values are stateless and accessed as in a sequential system.

Correctness of the bind and block operations on variables is guaranteed by serializing them for each variable. This is achieved by means of so-called "spin locks", known from [Cra88, LMT95, MA95]. A spin lock transforms temporarily the variable object into the self-reference object. The effect of the lock is that the original object becomes inaccessible to other thread(s): a worker executing it falls into an endless loop until the object is unlocked. Note that no deadlock is possible: there are no PAMOz instructions that require to lock more than one variable object at a time.

The PAMOz implementation of the cell exchange operation performs *atomic swap* of the previous and new contexts of the cell, which combines reading and writing a cell into an atomic operation. Both taking a spin lock of a variable and atomic swap of a cell exploit a special machine instruction (e.g. Sparc's SWAP) provided by an SMP computer.

PAMOz versus AMOz. At the glance, PAMOz principally differs from AMOz in two respects: the PAMOz architecture and the implementation of synchronizing primitives. The same set of changes can be applied to other `cc` languages since they all contain `ask/tell` – based operations[1], while data itself is stateless and accessed as in a sequential abstract machine. Additionally, the PAMOz implementation of the cell exchange needs special care compared to a sequential abstract machine (AMOz does not provide for cells).

Extending PAMOz for constraint solving. PAMOz can be extended nearly conservatively for full Oz. This proceeds as follows: first, one adds unification like it is done in [MA95]. Second, PAMOz is extended for first-class spaces through a dedicated interface, and finally it is extended for other constraint systems like finite domains, sets, etc. by means of a parallel implementation of the constraint programming interface (CPI) [MW97].

PAMOz and the parallel Oz system. PAMOz is implemented in the parallel Oz system [Pop97]. The parallel implementation of Oz is concerned with the full language except (by now) its finite domains and sets constraint systems.

The main implementation detail is that methods of PAMOz abstract objects whose implementation are not considered here are atomic. An example of such objects is the PAMOz thread pool. Atomic methods allow to neither see nor modify intermediate states of objects. This is ensured by mutex locks over methods that can violate this property. The implementation contains also a lot of low-level optimizations, primarily dealing with enhancing the memory usage (such as using more compact data representations, and caching information in a worker's registers), and increasing the speed of method applications in the Oz object system. The implementation has reached a stable state, but has not been fully evaluated yet.

Related work. The most direct sibling of PAMOz is AMOz, which deals with the sequential execution of programs written in a larger subset of Oz than the PAMOz's one.

KL1 is a fine-grained concurrent programming language with annotations for controlling parallelism. KL1 does not contain constraint solving facilities. Its parallel implementaion KLIC [FCRN94] is a native-code system. Synchronizing between executed in parallel activ-

[1] While, of course, constraint systems and synchronized entities are in general different.

ities has been added to the original sequential system through a special construct, instead of re-implementing basic primitives in PAMOz. Another implementation of KL1 – the Super Monaco system [LMT95] is also a native-code system, which was specially designed for parallel execution.

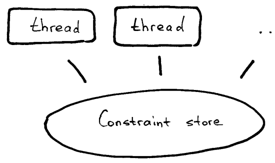
AKL is a fine-grained concurrent constraint programming language, suitable for both concurrent programming and constraint solving. Its parallel implementation Penny [MA95] constitutes a major redesign of the original sequential, emulation-based implementation of AKL. The abstract machine for Penny deals with full AKL; there are no obvious components dealing with concurrent programming and constraint solving.

The contributions of the paper are:

- PAMOz models a parallel implementation of the subset of Oz, suitable for concurrent and object-oriented programming. PAMOz executes Oz threads in parallel. PAMOz ideas are exploited in the implemented parallel Oz system.
- PAMOz approach for deriving a parallel abstract machine from a sequential one is promising for other concurrent constraint programming languages.
- PAMOz is a minimal model for an Oz system without constraint solving facilities. It is simpler than AMOz, since it does not contain such things like unification or local computation spaces, which are necessary for constraint solving only.

Plan of the paper. The rest of the paper is organized as follows. The relevant subset of Oz is introduced in Section 2. The architecture of PAMOz is presented in Section 3. Section 4 is devoted to the representation scheme of Oz data. The background for the discussion of the PAMOz primitives on stateful data is given in Section 5. The extended version of this paper (located at `http://www.ps.uni-sb.de/~popow/pamoz-ext.ps`) presents also the compilation and execution of Oz by PAMOz. The PAMOz implementation of Oz operations on stateful data is discussed in Section 6. The interface between PAMOz and its extension providing for problem solving is sketched in Section 7. Implementation details of the parallel Oz system are considered in Section 8. Related works are listed in Section 9. Finally, conclusions and directions for further work are given in Section 10.

2 Oz



Computation Model. Oz is a concurrent constraint programming language with state. Computation in Oz is organized into concurrent *threads* which contain computation tasks composed sequentially. Threads process data from the shared *store*. Data in the store describe values of *logic variables*. A logic variable denotes a certain value, like does a variable in mathematic. Information cannot be retrieved from the store. A thread is *blocked* until the store contains enough information for performing the next computation task. Concurrent computation in Oz has interleaving semantics. This means in particular that operations on the store are sequentialized.

Data Types. In this paper we consider *records*, *procedures* and *cells*. An Oz record consists of a symbolic *label* and zero or more *components*, each of them is an Oz value again. Record's components have symbolic names. A record having no components is called *atom*.

Oz procedures are primitive data structures which encapsulate "closures". A closure consist of code to be executed and references to variables that are lexically non-local to a procedure definition but are referenced inside it. This is similar to other languages with lexical scoping and first-class procedures, e.g. Common Lisp.

Cells are similar to records, but have only one component, and this component can be *exchanged* by means of a dedicated cell operation.

Storage Model. The Oz storage model defines how Oz data is stored in memory from the point of view of an Oz programmer. In fact, the PAMOz data representation model is a refinement of the storage model presented here.

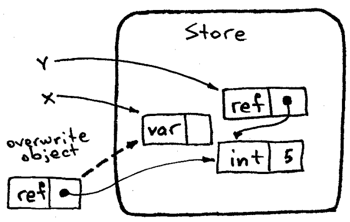


Figure 1: Binding a Variable.

The Oz store contains *store objects*. A store object is either a *primitive* value object (like integers), a *compound* one (like e.g. structures in C), a *variable* object or a *reference* object which points to another object. Primitive objects represent their values, compound ones contain additionally one or more references to other objects, variable objects represent Oz variables that values are not known yet, and reference objects are auxiliary structures explained below.

Variable identifiers in a running Oz program are mapped to store locations, which originally contain variable objects. Store objects have indefinite extent, that is, they are kept alive until they cannot be reached by a running program anymore.

Information about values of variables is represented by means of binding variables to data objects. A variable is bound by replacing its object with a reference object, pointing to another object. Thus, one can think about information in the store as a graph whose nodes are objects and edges are references to objects.

The example on Figure 1 shows how a variable x whose location is known to a thread is bound to the integer 5. The result of binding is shown on the example of another variable y. Accessing a value of a bound variable involves skipping reference objects on a path from the original variable's location up to a non-reference data object. This is called *dereferencing*.

Program Structure. Oz is a statically scoped language with the block structure. An Oz program is a sequence of expressions. Thus, Oz programs look very similar to block-structured imperative programming languages.

Expressions. There are the following expressions[2], where x, y, z are variable identifiers and

[2]We don't follow the actual Oz syntax here, because it is designed with respect to many other requirements and language features that are not considered here. Neither do we claim that this subset is "minimal".

a, b are symbolic constants:

V	::=	$f(\overline{a}:\overline{y})$ | a | **proc** $\{\overline{y}\}\ E$ **end** | **cell**$[y]$	*values*
E, F	::=	**skip**	*skip*
	|	$E\,F$	*sequential composition*
	|	**local** x **in** E **end**	*declaration*
	|	$x := V$ | $x := y$	*binding*
	|	$\{x\,\overline{y}\}$	*procedure application*
	|	**exch** x **set** y **get** z **in** E **end**	*cell exchange*
	|	**case** x **of** $f(\overline{a}:\overline{y})$ **then** E **else** F **end**	*conditional*
	|	**thread** E **end**	*thread creation*

Records are specified as $f(\overline{a}:\overline{y})$ terms, where f is a label, $\overline{a}$ are components' names, and y are the components themselves. Atoms are given by their symbolic names a. A procedure is created as **proc** $\{\overline{y}\}\,E$ **end**, where $\overline{y}$ are formal arguments and E is a body. Cells are specified as **cell**$[y]$, where y is an initial cell's content.

Two expressions written one after another are composed *sequentially.* A new block with the variable x in the scope of E is introduced by means of the expression **local** x **in** E **end**. Procedure bodies, cell exchange expressions and threads are other types of program blocks.

A variable x can be bound either to a value V or another variable y by means of $x := V$ and $x := y$ respectively. A variable can be bound only once. Note that executing both $x := y$ and $y := x$ is a programming error in the Oz subset considered here.

Procedure application $\{x\,\overline{y}\}$ of a procedure **proc** $\{\overline{z}\}$ E **end** proceeds by executing the expression $E[y_1/z_1, \ldots, y_n/z_n]$. Thus, actual parameters $\overline{y}$ are passed by name. Argument passing can be seen also as "call-by-reference" since only variables are allowed at argument positions. Applying a non-procedure is a programming error.

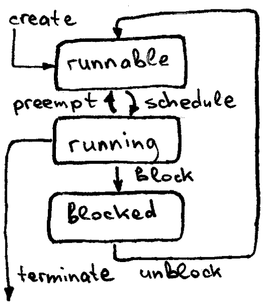
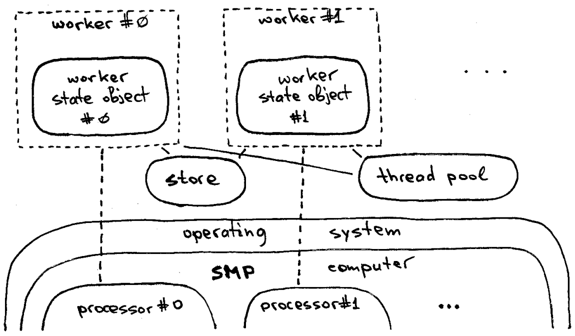


Figure 2: Thread States.

A cell's content is exchanged by means of the **exch** expression. In this expression x is expected to be a cell, y is a new cell's content, and z is a variable within the scope of E. **exch** puts a reference to y into a cell and continues with $E[y'/z]$, where y' references a previous cell's content. Applying **exch** to a non-cell is a programming error.

A conditional **case** x **of** $f(\overline{a}:\overline{y})$ **then** E **else** F **end** proceeds by executing either E or F, depending on whether a value bound to the variable x *matches* the linear pattern $f(\overline{a}:\overline{y})$ or not. The matching procedure is similar to e.g. the SML's one [MTH90]. Records match if they have the same label and the same components' names. Variables $\overline{y}$ from a pattern have the scope of E, and are bound to corresponding components of a record being matched. Without the loss of generality we omit in this paper nested patterns, token matching of procedures and cells, and multi-pattern conditionals which are present in Oz.

Procedure application, cell exchange and conditionals are the *synchronizing* operations: they wait until a variable x is bound to a value. Other primitives are *unsynchronizing* ones.




```
void worker(WorkerState *w) {
  ...
  while(running) {
    op = getInstruction();
    switch(op) {
    case Bind(VarNum,RegNum):
      ...
    }
  }
}
```

Figure 3: PAMOz Architecture.

A thread is created by means of **thread** E **end**. A thread can be either *runnable*, *running* or *blocked* (see Figure 2).

3 PAMOz Architecture

PAMOz executes threads in parallel. Threads are executed by *workers*. The architecture of PAMOz (see Figure 3) allows existence of multiple workers. Workers are the only active entities in the system. Runnable threads are kept in a shared thread pool. Workers can access data in the shared store.

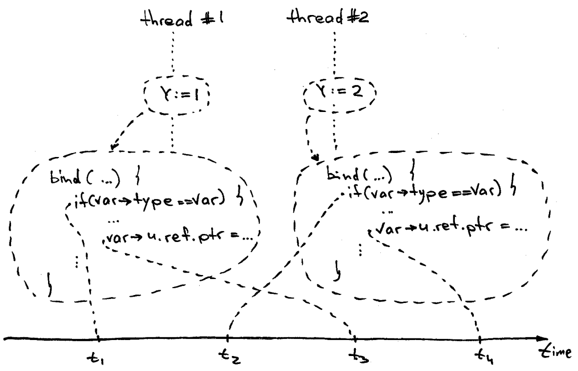
The PAMOz worker is a procedure that executes PAMOz code, introduced in Section 5. Workers are executed in parallel by the operating system of the underlying SMP computer. The worker is a *stateful* entity; its state contains at least the current thread.

Compared with AMOz, PAMOz abstracts the state of the worker. It is encapsulated in the *worker state object*. Thus, one speaks about components of the state object rather than about global registers and memory locations keeping global data. PAMOz operations that require access to the worker state take the state object as an additional argument.

The thread pool encapsulates a scheduling strategy which is not specified by PAMOz. It is required only that scheduling is *fair* (due to the language specificiation). A naive implementation of the pool contains a queue. A more elaborated one could try e.g. to avoid moving threads between workers (while preserving fairness), if it would help to improve the performance of the memory subsystem of the computer (see e.g. [ML92a, ML92b]).

4 Data Representation

The PAMOz data representation model is a specialization of the Oz storage model. It is similar to the AMOz one, which, in turn, is based on ideas from WAM [AK91].




```
Bool StoreObject::bind(StoreObject *var,
                       StoreObject *to) {
  if (var→type == Var) {
    var→type = Ref;
    var→u.ref.ptr = to;
    return (TRUE);
  } else {
    return (FALSE);
  }
}
```

Figure 4: Erroneous Behavior of the AMOz BIND in the Context of PAMOz.

A store object contains a *type tag* and further data depending on its type (the C++ like notation is used here):

```
typedef enum {Atom, Record, Cell, Proc, Var, Ref} Type;
class StoreObject {
  Type type;
  union { struct { char *name; } atom;
          struct { char *name; Signature *sig; StoreObject components[]; } record;
          struct { StoreObject *content; } cell;
          struct { int arity; ProgramCounter pc; int varNumber; StoreObject vars[]; } proc;
          struct { ThreadList *blockedThreads; } var;
          struct { StoreObject *ptr; } ref; } u;
};
```

Here the `sig` records' substructure keeps a mapping from symbolic components' names to their indexes in the array `components`. The structure of procedures is considered in Section 5. The `blockedThreads` substructure holds a list of threads; its purpose is discussed in Section 6.

PAMOz assumes that an underlying SMP computer is sequentially consistent. That is [Lam79], "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"[3].

There are two basic operations on the store: *binding* a variable and *dereferencing* a chain of references. The BIND operation checks consequently whether a store object is a variable one, and if it is – replaces it with the reference object. Dereferencing skips reference objects on the path to a non-reference object. This operation allows to access an object a variable has been bound to, as discussed in Section 2.

[3] It is not considered yet which weakest consistency model is sufficient for PAMOz. The real parallel Oz system runs on SPARC multiprocessors running in the so-called "total store order" mode, which relaxes the order constraints in the case of a write followed by a read to a different location [SPA92].

A straightforward encoding of the BIND operation (see the code on Figure 4) is sufficient for sequential AMOz [MSS95], but needs improvement when put into a parallel system. This is illustrated on Figure 4. There, the first thread binds the variable Y (at t_3) between checking the variable (at t_2) and binding it (at t_4) by the second thread. Clearly, both threads believe they have bound Y, and no exception is raised.

The problem with the BIND from AMOz when it is used in PAMOz is fixed by serializing access to each variable. This is achieved by means of locking of store objects. The idea of the PAMOz lock is to convert temporarily a store object to a reference object pointing to itself, so other workers will "spin" while dereferencing it. A content of a succefully locked object is saved in another store object, which location is known only to the worker that has locked the object. A locked object is unlocked by restoring its content back:

```
StoreObject* lock(StoreObject *obj) {
  StoreObject *lock = new StoreObject;
  lock->type = Ref;
  lock->u.ref.ptr = obj;
  swapObjects(obj, lock);
  return (lock);
}
```

```
void unlock(StoreObject *obj,
            StoreObject *lock) {
  swapObjects(obj, lock);
  delete lock;
}
```

The SWAPOBJECTS procedure that exchanges contents of two objects is *atomic*. That is, objects are exchanged in a single step from the point of view of all workers. SWAPOBJECTS requires hardware support, like the Sparc's SWAP instruction that exchanges two machine words. A non-atomic implementation of SWAPOBJECTS lets other workers to observe partially overwritten objects, as well as to lock an object by two workers simultaneously. The later problem is similar to the erroneous behavior of the AMOz BIND in the context of PAMOz.

The BIND operation locks the object that is supposed to be a variable, and works with its content from the copy:

```
Bool StoreObject::bind(StoreObject *var,
                       StoreObject *to) {
  StoreObject *lock = lock(var);
  if (lock->type == Var) {
    lock->type = Ref;
    lock->u.ref.ptr = to;
    unlock(var, lock); return (TRUE);
  } else {
    unlock(var, lock); return (FALSE);
  }
}
```

```
StoreObject* deref(StoreObject *obj) {
  while (obj->type == Ref)
    obj = obj->u.ref.ptr;
  return (obj);
}
```

5 Executing Oz Programs

This small section is intended to prepare the reader to Section 6, where differences between AMOz and PAMOz are presented. Implementations of the Oz operations on stateless data are inherited from AMOz without changes. The complete presentation of the compilation

```
case Wait(nValue):  // ``nValue'' is an index in the environment;
  StoreObject *var = w→currentThread→E[nValue]; // ``w'' is the worker's state;
  loop {
    var = var→deref(); StoreObject *lock = lock(var);
    if (lock→type == Var) {
      w→currentThread→push(w→currentThread→pc,
                            w→currentThread→E);
      lock→addBlockedThread(w→currentThread);
      unlock(var, lock); goto getThread;
    } else if (lock→type == Ref) {
      unlock(var, lock); continue; // bound after deref();
    } else {
      unlock(var, lock); DISPATCH; // a non-variable;
    }
  }
case Exch(nCell,nNewValue,nOldValue):
  StoreObject *cell = w→currentThread→E[nCell]→deref();
  w→currentThread→E[nOldValue] =
    cell→exchange(w→currentThread→E[nNewValue]);
  DISPATCH; // fetch the next instruction;
```

Figure 5: The PAMOz Wait and Exch instructions.

and execution of Oz programs is given in the extended version of the paper, located at `http://www.ps.uni-sb.de/~popow/pamoz-ext.ps`.

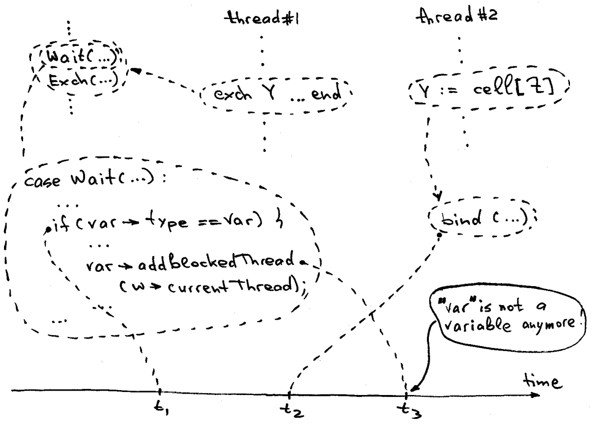
Oz programs are compiled to PAMOz byte-code. The PAMOz code is interpreted sequentially by the PAMOz *emulator loop*. Each thread is equipped with the program counter (`pc`).

The PAMOz thread keeps an *enviroment* E mapping variable identifiers (variables for short) $\overline{x}$ to store locations. PAMOz instructions access environment cells by their indexes.

6 Synchronizing On The Store

The idea of the implementation of Oz synchronizing operations is shared with AMOz: a thread missing a value of a variable is preempted and saved in the variable object. A worker executed the thread that has been blocked picks up from the thread pool a next runnable one. A thread blocked on a variable is pushed back into the thread pool first when this variable becomes bound. A continued thread retries the operation that has caused it to block.

The implementation of Oz synchronizing operations is shown in this section on the example of the cell exchange. It is compiled into a sequence of PAMOz instructions, the core of which consists of the Wait and Exch instructions presented on Figure 5. The Wait




```
case Wait(n):
  StoreObject *var = w→currentThread→E[n];
  loop {
    var = var→deref();
    if (var→type == Var) {
      w→currentThread→push(w→currentThread→pc,
                           w→currentThread→E);
      var→addBlockedThread(w→currentThread);
      goto getThread;
    } else if (lock→type == Ref) {
      continue; // bound after deref() - retry;
    } else {
      DISPATCH; // a non-variable;
    }
  }
```

Figure 6: Erroneous Behavior of the WAIT Instruction Without Locking in PAMOZ.

instruction checks whether the variable denoting the cell is bound. If it is (checking the type of a value object is omitted here for simplicity), then the execution proceeds with the EXCH instruction which exchanges the cell's content; otherwise WAIT blocks the thread.

WAIT puts the thread blocked on the variable into the `blockedThreads` list in the variable object. WAIT in PAMOZ needs to lock the variable similarly to BIND. This is illustrated on Figure 6. There, a version of WAIT without locking is given on the right side. On the left side the first thread starts the cell exchange of a yet unbound variable, and the second thread binds that variable to a cell. In the scenario shown on the figure the variable `Y` becomes bound (at t_2) when WAIT has found `Y` to be a variable (at t_1) but before the thread is put into the `blockedThreads` list (at t_3). The problem is that at t_3 the variable object has been overwritten already, so the method `addBlockedThread` cannot be executed.

When a variable is bound by BIND, threads from that list are put into the thread pool:

```
Bool StoreObject::bind(StoreObject *var, StoreObject *to) {
  ...
  if (lock→isVar()) {
    threadPool→insert(var→blockedThreads); // new code;
    var→type = Ref;
  ...
}
```

The EXCH instruction also needs to be atomic. It cannot be implemented as a couple of READ and STORE operations: an error happens when such "read&store" exchange is executed between READing and STOREing the same cell by another worker. In this case there are two consequent reads followed by two consequent stores, so both cell exchanges return the same values, and the first value stored just disappears.

7 Towards PAMOz for Full Oz

Full Oz is characterized by its constraint solving facilities [SSW94, MW97]:

- There is *unification* known from Prolog, and the matching conditional is extended to conditionals **if** ψ **then** E **else** F **fi**, where ψ is a constraint which needs to be *entailed* by the constraint store when taking the "**then**" branch.

- It offers a control mechanism suitable for programming search – *first-class computation spaces* [SSW94, Sch97c]. A computation space encapsulate threads and constraints produced by them. Spaces allow to explicitly exploit search alternatives. Oz as a concurrent constraint language does not provide for the builtin search mechanism like the backtracking-based one from Prolog; instead, the Oz synchronizing primitives wait until sufficient information is available in the store.

Oz unification and general conditional can be seen as constructs orthogonal to binding and matching conditional. The AMOz unification algorithm is described in [MSS95]. PAMOz unification exploits a different scheme for keeping track of rational (cyclic) trees. Another point is that the variable-variable unification may not neither produce endless reference chains nor allow workers to deadlock. Techniques used in PAMOz are similar to those described in e.g. [Mon97].

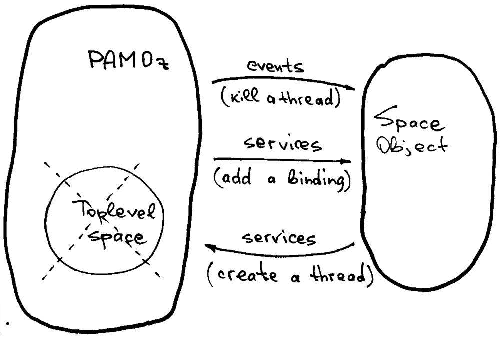


Figure 7: PAMOz and Local Spaces.

The first-class spaces and general conditionals are implemented on the top of the PAMOz *local computation spaces*. The PAMOz implementation of the Oz store together with the threads' management routines can be seen as a primitive version of computation spaces. The PAMOz local space can manage yet constraints on non-local variables and keep track of local threads.

A local computation space is represented by an object (see Figure 7). There is the interface between PAMOz and space objects. It consists of two parts:

- Services provided by PAMOz, like creating a new thread. In order to provide it, certain PAMOz objects are conservatively extended. An example is that thread objects and variable store objects carry a *space pointer* – a pointer to the space they belong to.

- Services and event handlers provided by space objects. For example, the space object provides for binding variables and creating threads. Events in PAMOz like terminating a thread in the current space are reported to the space object through method applications.

General conditionals mentioned above can be expressed by means of spaces. The idea is to put a constraint ψ into a computation space: if it becomes stable, then the constraint is entailed and the "**then**" expression is executed. If the space fails, then the constraint is disentailed, and the "**else**" expression is choosen. When a thread blocks on a space, then there is not yet enough information to determine entailment, the the conditional blocks.

PAMOz for full Oz can be extended for the constraint programming interface (CPI) [MW97]. CPI is an interface between an abstract machine for Oz and *propagators* written in C++ which are used to express non-trivial constraints. Propagators are sequential programs that try to amplify information in the store. For example, if a finite domain propagator $x = y + z$ sees that y and z can be either 1 or 2, then it adds the constraint saying that x must be in the range of 2..4. CPI provides propagators for reading and writing information to the store, and controls their invocation.

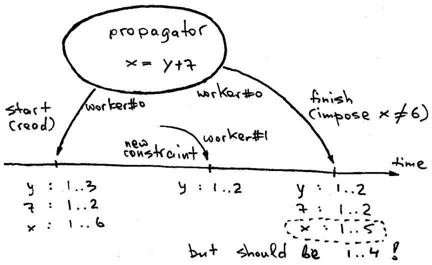


Figure 8: Erroneous Behavior of the Sequential CPI in PAMOz.

The problem with executing propagators in a parallel system is that the store can be amplified during the execution of a propagator. Consider the example on Figure 8: the propagator fails to shrink the domain of x to 2..4 because the additional constraint on y arrives *after* reading parameters by the propagator.

The PAMOz implementation of CPI solves the problem by locking the propagator's parameters during its execution. Note that after the modification of the CPI implementation the propagators originally written for the sequential Oz system are reused in the parallel one.

8 Parallel Oz Implementation

PAMOz has been implemented in the parallel Oz system [Pop97]. The origin of this implementation is the (sequential) DFKI Oz System [OzS]. The emulator which is implemented in C++ has been converted to the parallel one, while the Oz Compiler remained unchanged. The parallel system runs currently on SPARC multiprocessors under Solaris 5.*.

Two questions have been considered when implementing PAMOz: how to implement it correctly, and what could be done in order to speed up the implementation.

Correctness. Complex shared objects in PAMOz are held consistent. For instance, if a thread object has the allocated task stack, then the corresponding flag in the object is set, and vice versa. In order to guarantee consistency methods that either (a) temporarily bring an object into an inconsistent state and/or (b) allow to observe that inconsistency are serialized. This is done by means of mutex locks.

Shared objects that relate to each other are maintained consistenly too. This is achieved by serializing transactions on them. A standard deadlock-free locking scheme is exploited, namely, each set of objects being locked together has a total order relation; objects are locked starting from a smallest one.

Compiler optimizations in the implementation language (C++) concerning the caching of data in machine registers are prohibited when compiling methods protected by locks. This is because otherwise a compiler may move out actual memory operations from regions protected by locks, as well as reorder them. These optimizations are switched off by declaring C data structures to be `volatile`. Note that memory cells that hold busy-waiting locks are volatile for the same reason as well.

Efficiency. The parallel system inherits all optimizations from the sequential Oz system. Though this required a substantial engineering effort, no design changes were necessary. In general, optimizations that are applicable to AMOz will be also applicable to PAMOz, possibly excepting those dealing with synchronizing primitives.

Both sequential and parallel systems feature space- and speed- efficient representations of data. Primarily, they exploit *tagged pointers* instead of tagged data (as described e.g. in [Gud93]). Both systems exploit threaded code made possible by the GNU C compiler [Sta96], which reduces the overhead of emulation by up to 50% [Sch97a]. Oz object-oriented capabilities are supported directly by the emulator: method applications, first-class methods and attribute accesses are as fast as procedure applications and accessing record components. The speed of the sequential system is comparable to "state-of-the-art" implementations of Prolog like SICStus. Detailed analysis and evaluation of optimizations can be found in [Meh97, Sch97a, Hen97].

The parallel system is optimized on its own. Optimizations serve for better sequential (i.e. on a single worker) speed and/or help to obtain better parallel speedups. The most essential one among the first group is the locking scheme that does not consume additional memory. The idea of the scheme is to reuse temporarily a memory cell of the object's representation for the lock, while the content of that cell is kept as a local data in the PAMOz procedure that has taken the lock.

I believe that the most effective optimization among those that serve for better speedups is the per-worker memory management scheme, as implemented in [Mon97], and to some extent in [FCRN94]. Each worker has a local memory pool that is used to allocate locally created store objects. This eliminates the contention on memory manager that takes place in the case of a shared memory pool. Apparently, this helps also to reduce the so-called false sharing of cache lines [TLH94][4].

9 Related Work

Oz/AMOz [MSS95]. The most direct sibling for PAMOz is the sequential abstract machine for Oz. Both use similar data representation schemes and ideas for translating and executing (P)AMOz code. They differ in the implementation of operations crucial for parallel execution, and PAMOz offers an interface between a "pure" concurrent sub-language and its constraint solving extension, while AMOz does both parts in a rather monolithic fashion.

[4]False sharing takes place when a processor memory access results in prefetching a cache line that contains also some irrelevant data being processed by other processor. As a result, two processors interactively invalidate that cache line. Eliminating false sharing could result in 10% speedups [TLH94].

KL1/KLIC [FCRN94]. KL1 is a fine-grain concurrent programming language with logic variables. The KLIC system exploits annotations to control the amount of parallelism. KLIC is a native code system: KL1 programs are translated into C. The system contains both local and shared heaps. Accessing a shared variable is implemented with the help of generic objects which serve for synchronization between concurrent goals. Methods of a generic object are applied whenever a variable is bound or a concurrent activity blocks on it. The generic objects mechanism allows to preserve the sequential core unchanged at the expense of an additional object type.

KL1/Super Monaco [LMT95]. Super Monaco is also a native code system. The native code is derived by macro-expansion from the intermediate code. Native code calls functions from a run-time library with implementations of complex operations like memory management and unification. Operations crucial for parallel execution are encapsulated in the run-time library; their implementations are not described in [LMT95]. Variable objects in the system do not contain references to goals blocked on them; instead, suspensions are accessed through a global "hook" table. Authors motivate this solution by measurements saying that goals block rather seldom (which is also true for Oz). However, the impact of contention on a shared global hook table is not measured.

AKL/Penny [MA95]. AKL (Agents Kernel Language [JH94]) is a concurrent constraint programming language with fine-grain implicit concurrency. Penny exploits a store model and compilation techniques similar to (P)AMOz. Since AKL does not contain a sub-language without constraint solving facilities, the Penny abstract machine implements the whole language. Penny originated from the sequential AKL system, similar to PAMOz, but changes made to the sequential system are not listed explicitly in publications on Penny.

PAMOz among KLIC, Super Monaco and Penny. The parallel implementation of AKL (Penny) has faced the major problem of managing fine-grain concurrency. The problem is solved in part by exploiting various scheduling heuristics. The impact of the scheduling overhead is reduced also by careful encoding of benchmark programs. Since threads in Oz are created explicitly, and concurrency in a typical Oz program appeared to be coarse-grained, this problem is less of an issue for the parallel Oz system.

PAMOz is inherited from AMOz by modifying some of its primitives. It seems to be that similar changes are applicable in cases of other `cc` programing languages like KL1 and AKL. The KLIC approach of conservative extensions of the sequential core system is an alternative way for building a parallel system. On the contrary, the Super Monaco and Penny systems are presented in such a way like they were designed "from scratch".

Oz is a language with constraint solving facilities, which appear as a conservative extension of the base language. As mentioned before, PAMOz can be extended for that facilities. AKL is the only language among mentioned here that supports constraint solving. However, both AKL and its parallel implementation are monolithic in the sense that they do not separate concurrent programming and constraint solving.

10 Conclusions and Further Work

In this paper a parallel abstract machine for Oz (PAMOz) is presented. PAMOz models an implementation of a subset of Oz without constraint solving facilities. PAMOz is derived from AMOz – a sequential abstract machine for Oz, which, in turn, has its origins in works on implementations of Prolog [AK91], concurrent constraint languages [JH94], and functional programming languages [PJ87].

It turns out that PAMOz can be derived from a sequential abstract machine like AMOz only with a few changes. The first one is the implementation of primitives dealing with stateful data structures – variables and cells. Accessing stateless data is not changed. Operations on variables are serialized for each variable, which is achieved by means of spin locks. The cell exchange is made atomic, so cells are inaccessible between reading a previous value and storing a new one. The second change is the PAMOz architecture: the state of the PAMOz worker is a dynamically allocated first-class entity, compared to AMOz's global registers.

Another PAMOz contribution is that there is an interface to its extension providing for Oz constraint solving facilities. This interface allows to add further constraint systems convenient for problem solving, and to extend a sequential or parallel implementation for search facilities (first-class computation spaces).

PAMOz has been implemented in the parallel Oz system, which is now in a stable state. This system is derived from the sequential Oz system (DFKI Oz 2.0) and has inherited all its optimizations. In the future I plan the implementation of a parallel garbage collector, and a complete evaluation of the system simultaneously with its profiling and further optimizations. Profiling is performed using both standard profiling techniques (like GNU `gprof`), as well as memory simulators like SimICS and WARTS [MW95, HLL+95].

11 Acknowledgments

I am thankful to all members of Programming Systems Lab at the University of Saarland, Germany for fruitful discussions during my work on the parallel system and this presentation of it. Special thanks are to Seif Haridi, Joachim Niehren, Tobias Mueller, Christian Schulte, and Denys Duchier. The work is supported by the "Graduiertenkolleg Informatik" at the University of Saarland.

References

[AK91] Hassan Aït-Kaci. *The WAM: A Real Tutorial.* Logic Programming. MIT Press, 1991.

[Cra88] J. A. Crammond. *Implementation of committed choice logic languages on shared memory multiprocessors.* PhD thesis, Heriot-Watt University, May 1988.

[FCRN94] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A portable implementation of KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 66–79, ICOT, Tokyo, Japan, December 1994. ACM Press.

[Gud93] David Gudeman. Representing type information in dynamically typed languages. TR 93-27, Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA, October 1993.

[Hen97] Martin Henz. *Objects in Oz.* Doctoral dissertation, Technische Fakultät der Universität des Saarlandes, Universität des Saarlandes, 66041 Saarbrücken, Germany, May 1997.

[HLL+95] Mark D. Hill, James R. Larus, Alvin R. Lebeck, Madhusudhan Talluri, and David A. Wood. *Wisconsin Architectural Research Tool Set*. Computer Science Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, WI 53706, May 4 1995.

[HW95] M. Henz and J. Würtz. Using Oz for college time tabling. In *International Conference on the Practice and Theory of Automated Time Tabling*, Edinburgh, Scotland, August/September 1995.

[JH94] Sverker Janson and Seif Haridi. An introduction to AKL, a multi-paradigm programming language. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, NATO Advanced Science Institute Series, pages 411–443. Springer, 1994.

[Lam79] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LMT95] J.S. Larson, B.C. Massey, and E. Tick. Super Monaco: Its portable and efficient parallel runtime system. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *Proceedings of the First International EURO-PAR Conference*, number 966 in Lecture Notes in Computer Science, pages 527–538, Stockholm, Sweden, August 29–31 1995. Springer-Verlag.

[MA95] Johan Montelius and Khayri A. M. Ali. An And/Or-parallel implementation of AKL. *New Generation Computing*, 13(4), December 1995.

[Meh97] Michael Mehl. *The Architecture and Implementation of an Abstract Machine for Oz*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1997. In preparation (preliminary title).

[ML92a] Evangelos P. Markatos and Thomas J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessoss. In *1992 International Conference on Parallel Processing*, volume I, pages 258–267, St. Charles, Ill., August 1992.

[ML92b] Evangelos P. Markatos and Thomas J. LeBlanc. Using processor affinity on loop scheduling on shared-memory multiprocessors. Technical Report 410, The University of Rochester, Computer Science Department, Rochester, New York 14627, March 1992.

[MM97] Tobias Müller and Martin Müller. Finite set constraints in Oz, 1997. To appear at WLP'97.

[Mon97] Johan Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. Dissertation, SICS – Swedish Institute of Computer Science, and Uppsala University, SICS, Box 1263, S-164 28 Kista, Sweden, 1997. Uppsala Theses in Computing Science 28, and SICS Dissertation Series 25.

[MPSW94] Tobias Müller, Konstantin Popow, Christian Schulte, and Jörg Würtz. Constraint programming in Oz. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1994.

[MSS95] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An abstract machine for Oz. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics and Programs, 7^{th} International Symposium, PLILP'95*, volume 982 of *Lecture Notes in Computer Science*, pages 151–168, Utrecht, The Netherlands, September 1995. Springer-Verlag.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[MW95] Peter Magnusson and Bengt Werner. Efficient memory simulation in SimICS. In *Proceedings of the 28^{th} Annual Simulation Symposium*, April 1995.

[MW97] Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In *International Logic Programming Symposium*, 1997. To appear.

[OzS] The Oz 2.0 programming system. Check http://www.ps.uni-sb.de/oz2.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[Pop97] Konstantin Popov. *Exploiting Coarse-grain Concurrency in Constraint-Based Programming Languages*. Doctoral dissertation, Technische Fakultät der Universitä des Saarlandes, Universität des Saarlandes, 66041 Saarbrücken, Germany, 1997. In Preparation (preliminary title).

[Sch97a] Ralf Scheidhauer. *Effiziente Implementierung von nebenläufigem Constraint-Programmieren mit höheren Prozeduren*. Doctoral dissertation, DFKI GmbH, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany, 1997. In german. In Preparation.

[Sch97b] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the 14^{th} International Conference on Logic Programming*, Leuven, Belgium, 8–11July 1997. The MIT Press.

[Sch97c] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP'97)*, Schloss Hagenberg, Linz, Austria, October 1997. To Appear.

[Smo95a] Gert Smolka. An Oz primer. DFKI Oz documentation series, DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany, 1995.

[Smo95b] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, volume 1000 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1995.

[SPA92] SPARC International, Inc. *The SPARC Architecture Manual. Version 8*. Prentice Hall, Englewood Cliffs, NJ 07632, 1992.

[SR90] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 7^{th} Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1990.

[SS94] Christian Schulte and Gert Smolka. Encapsulated search in higher-order concurrent constraint programming. In Maurice Bruynooghe, editor, *Logic Programming: Proceedings of the 1994 International Symposium*, pages 505–520, Ithaca, New York, USA, November 1994. MIT-Press.

[SSW94] Christian Schulte, Gert Smolka, and Jörg Würtz. Encapsulated search and constraint programming in Oz. In A.H. Borning, editor, *Second Workshop on Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 134–150, Orcas Island, Washington, USA, 2–4 May 1994. Springer-Verlag.

[Sta96] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, Inc., June 27 1996.

[TLH94] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[Wür96a] Jörg Würtz. Constraint-based scheduling in Oz. In *Proceedings of Symposium on Operations Research*, pages 218–223, Braunschweig, Germany, September 1996. Springer Verlag.

[Wür96b] Jörg Würtz. Oz Scheduler: A workbench for scheduling problems. In *Proceedings of the 8th IEEE International Conference on Tools with Artificial Intelligence*, pages 132–139, Toulouse, France, November16–19 1996. IEEE Computer Society Press.